Using VoltDB

Abstract

This book explains how to use VoltDB to design, build, and run high performance applications.

V15

Using VoltDB

V15

Copyright © 2008-2025 Volt Active Data, Inc.

This document is published under copyright by Volt Active Data, Inc. All Rights Reserved.

The software described in this document is furnished under a license by Volt Active Data, Inc. Your rights to access and use VoltDB features are defined by the license you received when you acquired the software.

The VoltDB client libraries, for accessing VoltDB databases programmatically, are licensed separately under the MIT license.

Volt Active Data, VoltDB, and Active(N) are registered trademarks of Volt Active Data, Inc.

VoltDB software is protected by U.S. Patent Nos. 9,600,514, 9,639,571, 10,067,999, 10,176,240, and 10,268,707. Other patents pending.

This document was generated on March 22, 2026.

Table of Contents

About This Book xiv
1. Overview 1
    1.1. What is VoltDB? 1
    1.2. Who Should Use VoltDB 1
    1.3. How VoltDB Works 2
        1.3.1. Partitioning 2
        1.3.2. Serialized (Single-Threaded) Processing 2
        1.3.3. Partitioned vs. Replicated Tables 3
        1.3.4. Ease of Scaling to Meet Application Needs 4
    1.4. Working with VoltDB Effectively 4
2. Installing VoltDB 5
    2.1. Operating System and Software Requirements 5
    2.2. Installing VoltDB 5
        2.2.1. Upgrading From Older Versions 6
    2.3. Setting Up Your Environment 7
    2.4. What is Included in the VoltDB Distribution 7
3. Starting the Database 9
    3.1. Initializing and Starting a VoltDB Database 9
    3.2. Initializing and Starting a VoltDB Database on a Cluster 10
    3.3. Stopping a VoltDB Database 11
    3.4. Saving the Data 12
    3.5. Restarting a VoltDB Database 12
    3.6. Updating Nodes on the Cluster 12
    3.7. Defining the Cluster Configuration 13
        3.7.1. Determining How Many Sites per Host 14
        3.7.2. Configuring Paths for Runtime Features 14
        3.7.3. Verifying your Hardware Configuration 15
4. Designing the Database Schema 16
    4.1. How to Enter DDL Statements 17
    4.2. Creating Tables and Primary Keys 18
    4.3. Analyzing Data Volume and Workload 19
    4.4. Partitioning Database Tables 20
        4.4.1. Choosing a Column on which to Partition Table Rows 20
        4.4.2. Specifying Partitioned Tables 21
        4.4.3. Design Rules for Partitioning Tables 21
    4.5. Replicating Database Tables 21
        4.5.1. Choosing Replicated Tables 22
        4.5.2. Specifying Replicated Tables 22
    4.6. Modifying the Schema 22
        4.6.1. Effects of Schema Changes on Data and Clients 23
        4.6.2. Viewing the Schema 24
        4.6.3. Modifying Tables 24
        4.6.4. Adding and Dropping Indexes 26
        4.6.5. Modifying Partitioning for Tables and Stored Procedures 27
5. Designing Stored Procedures to Access the Database 31
    5.1. How Stored Procedures Work 31
        5.1.1. VoltDB Stored Procedures are Transactional 31
        5.1.2. VoltDB Stored Procedures are Deterministic 31
    5.2. The Anatomy of a VoltDB Stored Procedure 33
        5.2.1. The Structure of the Stored Procedure 33
        5.2.2. Passing Arguments to a Stored Procedure 35

5.2.3. Creating and Executing SQL Queries in Stored Procedures 36
5.2.4. Interpreting the Results of SQL Queries 37
5.2.5. Returning Results from a Stored Procedure 40
5.2.6. Rolling Back a Transaction 41
5.3. Installing Stored Procedures into the Database 41
5.3.1. Compiling, Packaging, and Loading Stored Procedures 42
5.3.2. Declaring Stored Procedures in the Schema 43
5.3.3. Partitioning Stored Procedures in the Schema 43
6. Designing VoltDB Client Applications 47
6.1. Connecting to the VoltDB Database 47
6.1.1. How Connections To The Cluster Work 48
6.1.2. Specifying Multiple Servers 48
6.2. Invoking Stored Procedures 49
6.3. Invoking Stored Procedures Asynchronously 49
6.4. Defining Options on a Per-Call Basis 51
6.5. Closing the Connection 51
6.6. Handling Errors 51
6.6.1. Interpreting Execution Errors 52
6.6.2. Handling Timeouts 53
6.6.3. Writing Handlers to Interpret Other Errors 54
6.7. Compiling and Running Client Applications 56
6.7.1. Starting the Client Application 57
6.7.2. Running Clients from Outside the Cluster 57
7. Simplifying Application Development 58
7.1. Using Default Procedures 58
7.2. Shortcut for Defining Simple Stored Procedures 59
7.3. Verifying Expected Query Results 60
7.4. Scheduling Stored Procedures as Tasks 61
7.5. Directed Procedures: Distributing Transactions to Every Partition 62
8. Using VoltDB with Other Programming Languages 64
8.1. C++ Client Interface 64
8.1.1. Writing VoltDB Client Applications in C++ 64
8.1.2. Creating a Connection to the Database Cluster 65
8.1.3. Invoking Stored Procedures 65
8.1.4. Invoking Stored Procedures Asynchronously 66
8.1.5. Interpreting the Results 67
8.2. JSON HTTP Interface 67
8.2.1. How the JSON Interface Works 67
8.2.2. Using the JSON Interface from Client Applications 69
8.2.3. How Parameters Are Interpreted 71
8.2.4. Interpreting the JSON Results 72
8.2.5. Error Handling using the JSON Interface 73
8.2.6. Managing Client Sessions 74
8.3. JDBC Interface 74
8.3.1. Using JDBC to Connect to a VoltDB Database 74
8.3.2. Using JDBC to Query a VoltDB Database 75
9. Using VoltDB in a Cluster 77
9.1. Starting a Database Cluster 77
9.2. Defining the Cluster Configuration 77
9.3. Elastic Scaling to Resize the Cluster 78
9.3.1. Adding Nodes with Elastic Scaling 79
9.3.2. Removing Nodes with Elastic Scaling 80
9.3.3. Configuring How VoltDB Rebalances Nodes During Elastic Scaling 80
10. Availability 82

10.1. How K-Safety Works 82
10.2. Enabling K-Safety 83
10.2.1. What Happens When You Enable K-Safety 84
10.2.2. Calculating the Appropriate Number of Nodes for K-Safety 84
10.3. Recovering from System Failures 85
10.3.1. What Happens When a Node Rejoins the Cluster 85
10.3.2. Where and When Recovery May Fail 86
10.4. Avoiding Network Partitions 86
10.4.1. K-Safety and Network Partitions 86
10.4.2. Network Fault Protection 87
11. Active(N) Database Replication 89
11.1. How Database Replication Works 90
11.1.1. Starting Database Replication 90
11.1.2. Database Replication, Availability, and Disaster Recovery 91
11.1.3. Database Replication and Completeness 92
11.2. Using Active(N) Cross Datacenter Replication 93
11.2.1. Designing Your Schema for Active Replication 93
11.2.2. Configuring the Database Clusters 94
11.2.3. Starting the Database Clusters 96
11.2.4. Loading a Matching Schema and Starting Replication 97
11.2.5. Stopping Replication 97
11.2.6. Example XDCR Configurations 98
11.2.7. Understanding Conflict Resolution 99
11.3. Updating the Schema During Replication 106
11.3.1. Safely Updating the Schema While the Clusters are Paused 106
11.3.2. Adding and Removing Tables Without Pausing 107
11.3.3. Using Dynamic Schema Change to Add, Remove, and Modify Table Columns Without Pausing 107
11.4. Monitoring Database Replication 111
11.5. Tuning Active(N) to Meet Your Application Needs 111
11.5.1. The XDCR Workflow 111
11.5.2. Detecting and Correcting Bottlenecks in the XDCR Workflow 113
12. Security 116
12.1. How Security Works in VoltDB 116
12.2. Enabling Authentication and Authorization 116
12.3. Defining Users and Roles 117
12.4. Assigning Access to Stored Procedures 118
12.5. Assigning Access by Function (System Procedures, SQL Queries, and Default Procedures) 118
12.6. Using Built-in Roles 119
12.7. Encrypting VoltDB Communication Using TLS/SSL 120
12.7.1. Creating the TLS/SSL Certificates 120
12.7.2. Using Certificate Revocation Lists (CRLs) 121
12.7.3. Configuring TLS/SSL on the VoltDB Server 121
12.7.4. Configuring Mutual TLS/SSL on the Server and Clients 123
12.7.5. Using the VoltDB Command Line Utilities with TLS/SSL 123
12.7.6. Implementing TLS/SSL in the Java Client Applications 124
12.7.7. Configuring Database Replication (XDCR) With TLS/SSL 125
12.7.8. Updating TLS/SSL Certificates and Certificate Revocation Lists (CRLs) 126
12.8. Integrating LDAP Security with VoltDB 126
12.8.1. Configuring LDAP Security in VoltDB 127
12.8.2. Configuring VoltDB Security in LDAP 129
12.9. Integrating Kerberos Security with VoltDB 129
12.9.1. Installing and Configuring Kerberos 130

12.9.2. Configuring Java Security 130
12.9.3. Configuring the VoltDB Servers and Clients 131
12.9.4. Accessing the Database from the Command Line and the Web 133
13. Saving & Restoring a VoltDB Database 134
13.1. Performing a Manual Save and Restore of a VoltDB Cluster 134
13.1.1. How to Save the Contents of a VoltDB Database 135
13.1.2. How to Restore the Contents of a VoltDB Database Manually 135
13.1.3. Changing the Cluster Configuration Using Save and Restore 136
13.2. Scheduling Automated Snapshots 138
13.3. Tuning the Snapshot Process 138
13.4. Managing Snapshot Files 139
13.5. Special Notes Concerning Save and Restore 140
14. Command Logging and Recovery 142
14.1. How Command Logging Works 142
14.2. Controlling Command Logging 143
14.3. Configuring Command Logging for Optimal Performance 143
14.3.1. Log Size 144
14.3.2. Log Frequency 144
14.3.3. Synchronous vs. Asynchronous Logging 144
14.3.4. Hardware Considerations 145
15. Streaming Data: Import, Export, and Migration 147
15.1. How Data Streaming Works in VoltDB 148
15.1.1. Understanding Import 150
15.1.2. Understanding Export 150
15.1.3. Understanding Migration 151
15.1.4. Understanding Topics 152
15.2. The Business Case for Streaming Data 154
15.2.1. Extract, Transform, Load (ETL) 154
15.2.2. Change Data Capture 154
15.2.3. Streaming Data Validation 155
15.2.4. Caching 156
15.2.5. Archiving 157
15.3. VoltDB Export Connectors 158
15.3.1. How Export Works 159
15.3.2. The File Export Connector 160
15.3.3. The HTTP Export Connector 163
15.3.4. The JDBC Export Connector 167
15.3.5. The Kafka Export Connector 168
15.3.6. The Elasticsearch Export Connector 171
15.4. VoltDB Import Connectors 172
15.4.1. Bulk Loading Data Using VoltDB Standalone Utilities 172
15.4.2. Streaming Import Using Built-in Import Features 173
15.4.3. The Kafka Importer 174
15.5. VoltDB Import Formatters 176
15.6. VoltDB Topics 177
15.6.1. Types of VoltDB Topics 178
15.6.2. Declaring VoltDB Topics 180
15.6.3. Configuring and Managing Topics 181
15.6.4. Configuring the Topic Server 184
15.6.5. Calling Topics from Consumers and Producers 185
15.6.6. Using Opaque Topics 186
A. Supported SQL DDL Statements 188
ALTER STREAM 189
ALTER TABLE 190

ALTER TASK 193
ALTER VIEW 194
CREATE AGGREGATE FUNCTION 195
CREATE FUNCTION 197
CREATE INDEX 199
CREATE PROCEDURE AS 201
CREATE PROCEDURE FROM CLASS 203
CREATE ROLE 206
CREATE STREAM 208
CREATE TABLE 212
CREATE TASK 219
CREATE VIEW 222
DR TABLE 226
DROP FUNCTION 227
DROP INDEX 228
DROP PROCEDURE 229
DROP ROLE 230
DROP STREAM 231
DROP TABLE 232
DROP TASK 233
DROP VIEW 234
PARTITION TABLE 235
B. Supported SQL Statements 236
DELETE 237
INSERT 239
MIGRATE 241
SELECT 242
TRUNCATE TABLE 251
UPDATE 252
UPSERT 253
C. SQL Functions 255
ABS() 258
APPROX_COUNT_DISTINCT() 259
AREA() 260
ARRAY_ELEMENT() 261
ARRAY_LENGTH() 262
ASTEXT() 263
AVG() 264
BIN() 265
BIT_SHIFT_LEFT() 266
BIT_SHIFT_RIGHT() 267
BITAND() 268
BITNOT() 269
BITOR() 270
BITXOR() 271
CAST() 272
CEILING() 273
CENTROID() 274
CHAR() 275
CHAR_LENGTH() 276
COALESCE() 277
CONCAT() 278
CONTAINS() 279
COS() 280

COT() 281
COUNT() 282
CSC() 283
CURRENT_DATE() 284
CURRENT_TIMESTAMP() 285
DATEADD() 286
DATEDIFF() 287
DAY(), DAYOFMONTH() 288
DAYOFWEEK() 289
DAYOFYEAR() 290
DECODE() 291
DEGREES() 292
DISTANCE() 293
DWITHIN() 294
EXP() 295
EXTRACT() 296
FIELD() 298
FLOOR() 300
FORMAT() 301
FORMAT_CURRENCY() 302
FORMAT_TIMESTAMP() 303
FROM_UNIXTIME() 304
HEX() 305
HOUR() 306
INET6_ATON() 307
INET6_NTOA() 308
INET_ATON() 309
INET_NTOA() 310
ISINVALIDREASON() 311
ISVALID() 312
IS_VALID_TIMESTAMP() 314
LATITUDE() 315
LEFT() 316
LN(), LOG() 317
LOG10() 318
LONGITUDE() 319
LOWER() 320
MAKEVALIDPOLYGON() 321
MAX() 322
MAX_VALID_TIMESTAMP() 323
MIGRATING() 324
MIN() 325
MIN_VALID_TIMESTAMP() 326
MINUTE() 327
MOD() 328
MONTH() 329
NOW() 330
NUMINTERIORRINGS() 331
NUMPOINTS() 332
OCTET_LENGTH() 333
OVERLAY() 334
PI() 335
POINTFROMTEXT() 336
POLYGONFROMTEXT() 337

POSITION() 338
POWER() 339
QUARTER() 340
RADIANS() 341
REGEXP_POSITION() 342
REPEAT() 343
REPLACE() 344
RIGHT() 345
ROUND() 346
SEC() 347
SECOND() 348
SET_FIELD() 349
SIGN() 351
SIN() 352
SINCE_EPOCH() 353
SPACE() 354
SQRT() 355
STR() 356
SUBSTRING() 357
SUM() 358
TAN() 359
TIME_WINDOW() 360
TO_TIMESTAMP() 361
TRIM() 362
TRUNCATE() 363
UPPER() 364
VALIDPOLYGONFROMTEXT() 365
WEEK(), WEEKOFYEAR() 366
WEEKDAY() 367
YEAR() 368
D. VoltDB CLI Commands 369
csvloader 370
jdbcloader 375
kafkaloader 379
sqlcmd 384
voltadmin 390
voltdb 401
E. YAML Configuration Properties 410
E.1. Understanding YAML Syntax 410
E.1.1. YAML Files 410
E.1.2. Dot Notation 411
E.2. YAML Properties for Configuring Volt Databases on Bare Metal 413
F. VoltDB Datatype Compatibility 420
F.1. Java and VoltDB Datatype Compatibility 420
G. System Procedures 423
@AdHoc 424
@AdjustLogging 426
@Explain 427
@ExplainProc 428
@ExplainView 429
@GetPartitionKeys 431
@GetYamlConfiguration 433
@Metrics 435
@Note 436

@Pause 437
@Ping 438
@Quiesce 439
@Resume 441
@SetYamlConfiguration 442
@Shutdown 444
@SnapshotDelete 445
@SnapshotRestore 447
@SnapshotSave 450
@SnapshotScan 454
@Statistics 457
@StopNode 493
@SwapTables 495
@SystemCatalog 496
@SystemInformation 502
@UpdateClasses 507
@UpdateLogging 509

List of Figures

1.1. Partitioning Tables 2
1.2. Serialized Processing 3
1.3. Replicating Tables 4
4.1. Components of a Database Schema 16
4.2. Partitions Distribute Table Data and Stored Procedure Processing 17
4.3. Diagram Representing the Flight Reservation System 19
5.1. Array of VoltTable Structures 37
5.2. One VoltTable Structure is returned for each Queued SQL Statement 38
5.3. Stored Procedures Execute in the Appropriate Partition Based on the Partitioned Parameter Value 44
8.1. The Structure of the VoltDB JSON Response 72
10.1. K-Safety in Action 83
10.2. Network Partition 87
10.3. Network Fault Protection in Action 88
11.1. Active(N) Cross Datacenter Replication 89
11.2. Replicating an Existing Database 91
11.3. Disaster Recovery 92
11.4. Standard XDCR Configuration 98
11.5. XDCR Configuration with Read-Only Replicas 98
11.6. Transaction Order and Conflict Resolution 99
11.7. The XDCR Producer/Consumer Workflow 112
14.1. Command Logging in Action 142
14.2. Recovery in Action 143
15.1. Overview of Data Streaming 149
15.2. Overview of Topics 149

List of Tables

2.1. Operating System and Software Requirements 5
2.2. Components Installed by VoltDB 7
4.1. Example Application Workload 19
5.1. Methods of the VoltTable Classes 39
8.1. Datatypes in the JSON Interface 71
11.1. Structure of the XDCR Conflict Logs 104
12.1. Named Security Permissions 119
15.1. File Export Properties 161
15.2. Export Metadata 162
15.3. HTTP Export Properties 164
15.4. JDBC Export Properties 167
15.5. Kafka Export Properties 170
15.6. Elasticsearch Export Properties 172
15.7. Kafka Import Properties 175
15.8. CSV and TSV Formatter Properties 176
15.9. Topic Formatting Properties 183
A.1. Supported SQL Datatypes 212
C.1. Selectable Values for the EXTRACT Function 296
E.1. Complete List of Volt Configuration Properties 413
F.1. Java and VoltDB Datatype Compatibility 420
G.1. @SnapshotRestoreOptions 447
G.2. @SnapshotSave Options 451

List of Examples

4.1. DDL Example of a Reservation Schema 18
5.1. Components of a VoltDB Java Stored Procedure 34
5.2. Cycles of Queue and Execute in a Stored Procedure 37
5.3. Displaying the Contents of VoltTable Arrays 40
12.1. Creating a TLS Certificate for the Database Servers 121
12.2. Creating Mutual TLS Certificates for Both Servers and Clients 121

About This Book

This book is a complete guide to VoltDB. It describes what VoltDB is, how it works, and — more importantly — how to use it to build high performance, data intensive applications. The book is divided into five parts:

Part 1: Getting Started	Explains what VoltDB is, how it works, how to install it, and how to start using VoltDB. The chapters in this section are: • Chapter 1, *Overview* • Chapter 2, *Installing VoltDB* • Chapter 3, *Starting the Database*
Part 2: Developing VoltDB Database Applications	Describes how to design and develop applications using VoltDB. The chapters in this section are: • Chapter 4, *Designing the Database Schema* • Chapter 5, *Designing Stored Procedures to Access the Database* • Chapter 6, *Designing VoltDB Client Applications* • Chapter 7, *Simplifying Application Development* • Chapter 8, *Using VoltDB with Other Programming Languages*
Part 3: Running VoltDB in a Cluster	Describes additional features useful for running a database in a cluster. The chapters in this section are: • Chapter 9, *Using VoltDB in a Cluster* • Chapter 10, *Availability* • Chapter 11, *Active(N) Database Replication* • Chapter 12, *Security*
Part 4: Managing the Data	Provides techniques for ensuring data durability and integrity. The chapters in this section are: • Chapter 13, *Saving & Restoring a VoltDB Database* • Chapter 14, *Command Logging and Recovery* • Chapter 15, *Streaming Data: Import, Export, and Migration*
Part 5: Reference Material	Provides reference information about the languages and interfaces used by VoltDB, including: • Appendix A, *Supported SQL DDL Statements* • Appendix B, *Supported SQL Statements* • Appendix C, *SQL Functions* • Appendix D, *VoltDB CLI Commands*

	• Appendix E, *YAML Configuration Properties* • Appendix F, *VoltDB Datatype Compatibility* • Appendix G, *System Procedures*

This book provides the most complete description of the VoltDB product. It describes both core features from the commercial product plus separately licensed features that extend the durability, availability, and maintainability of VoltDB. In general, the features described in Parts 1 and 2 are available in all versions of the product. Some features in Parts 3 and 4 require additional licenses.

If you are new to VoltDB, the *VoltDB Tutorial* provides an introduction to the product and its features. The tutorial, and other books, are available on the web from https://docs.voltactivedata.com/.

Chapter 1. Overview

1.1. What is VoltDB?

VoltDB is a revolutionary new database product. Designed from the ground up to be the best solution for high performance business-critical applications, the VoltDB architecture is able to achieve 45 times higher throughput than current database products. The architecture also allows VoltDB databases to scale easily by adding processors to the cluster as the data volume and transaction requirements grow.

Current commercial database products are designed as general-purpose data management solutions. They can be tweaked for specific application requirements. However, the one-size-fits-all architecture of traditional databases limits the extent to which they can be optimized.

Although the basic architecture of databases has not changed significantly in 30 years, computing has. As have the demands and expectations of business applications and the corporations that depend on them.

VoltDB is designed to take full advantage of the modern computing environment:

- VoltDB uses in-memory storage to maximize throughput, avoiding costly disk access.
- Further performance gains are achieved by serializing all data access, avoiding many of the time-consuming functions of traditional databases such as locking, latching, and maintaining transaction logs.
- Scalability, reliability, and high availability are achieved through clustering and replication across multiple servers and server farms.

VoltDB is a fully ACID-compliant transactional database, relieving the application developer from having to develop code to perform transactions and manage rollbacks within their own application. By using ANSI standard SQL for the schema definition and data access, VoltDB also reduces the learning curve for experienced database designers.

1.2. Who Should Use VoltDB

VoltDB is not intended to solve all database problems. It is targeted at a specific segment of business computing.

VoltDB focuses specifically on *fast data*. That is, applications that must process large streams of data quickly. This includes financial applications, social media applications, and the burgeoning field of the Internet of Things. The key requirements for these applications are scalability, reliability, high availability, and outstanding throughput.

VoltDB is used today for traditional high performance applications such as capital markets data feeds, financial trade, telco record streams and sensor-based distribution systems. It's also used in emerging applications like wireless, online gaming, fraud detection, digital ad exchanges and micro transaction systems. Any application requiring high database throughput, linear scaling and uncompromising data accuracy will benefit immediately from VoltDB.

However, VoltDB is *not* optimized for all types of queries. For example, VoltDB is not the optimal choice for collecting and collating extremely large historical data sets which must be queried across multiple tables. This sort of activity is commonly found in business intelligence and data warehousing solutions, for which other database products are better suited.

To aid businesses that require *both* exceptional transaction performance *and* ad hoc reporting, VoltDB includes integration functions so that historical data can be exported to an analytic database for larger scale data mining.

1.3. How VoltDB Works

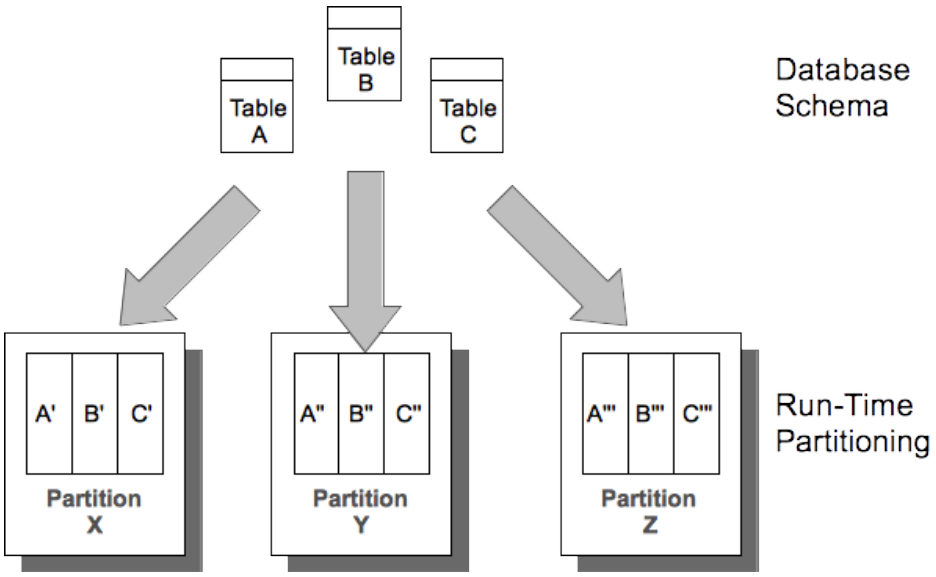
VoltDB is not like traditional database products. Each VoltDB database is optimized for a specific application by partitioning the database tables and the stored procedures that access those tables across multiple "sites" or partitions on one or more host machines to create the distributed database. Because both the data and the work is partitioned, multiple queries can be run in parallel. At the same time, because each site operates independently, each transaction can run to completion without the overhead of locking individual records that consumes much of the processing time of traditional databases. Finally, VoltDB balances the requirements of maximum performance with the flexibility to accommodate less intense but equally important queries that cross partitions. The following sections describe these concepts in more detail.

1.3.1. Partitioning

In VoltDB, each stored procedure is defined as a transaction. The stored procedure (i.e. transaction) succeeds or rolls back as a whole, ensuring database consistency.

By analyzing and precompiling the data access logic in the stored procedures, VoltDB can distribute both the data and the processing associated with it to the individual partitions on the cluster. In this way, each partition contains a unique "slice" of the data and the data processing. Each node in the cluster can support multiple partitions.

Figure 1.1. Partitioning Tables



1.3.2. Serialized (Single-Threaded) Processing

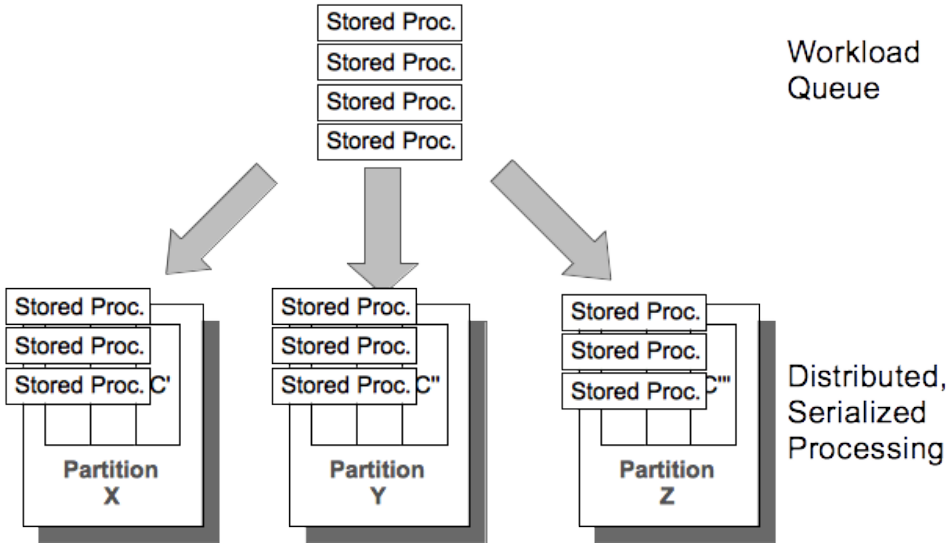
At run-time, calls to the stored procedures are passed to the appropriate partition. When procedures are "single-partitioned" (meaning they operate on data within a single partition) the server process executes the procedure by itself, freeing the rest of the cluster to handle other requests in parallel.

By using serialized processing, VoltDB ensures transactional consistency without the overhead of locking, latching, and transaction logs, while partitioning lets the database handle multiple requests at a time. As

a general rule of thumb, the more processors (and therefore the more partitions) in the cluster, the more transactions VoltDB completes per second, providing an easy, almost linear path for scaling an application's capacity and performance.

When a procedure does require data from multiple partitions, one node acts as a coordinator and hands out the necessary work to the other nodes, collects the results and completes the task. This coordination makes multi-partitioned transactions slightly slower than single-partitioned transactions. However, transactional integrity is maintained and the architecture of multiple parallel partitions ensures throughput is kept at a maximum.

Figure 1.2. Serialized Processing



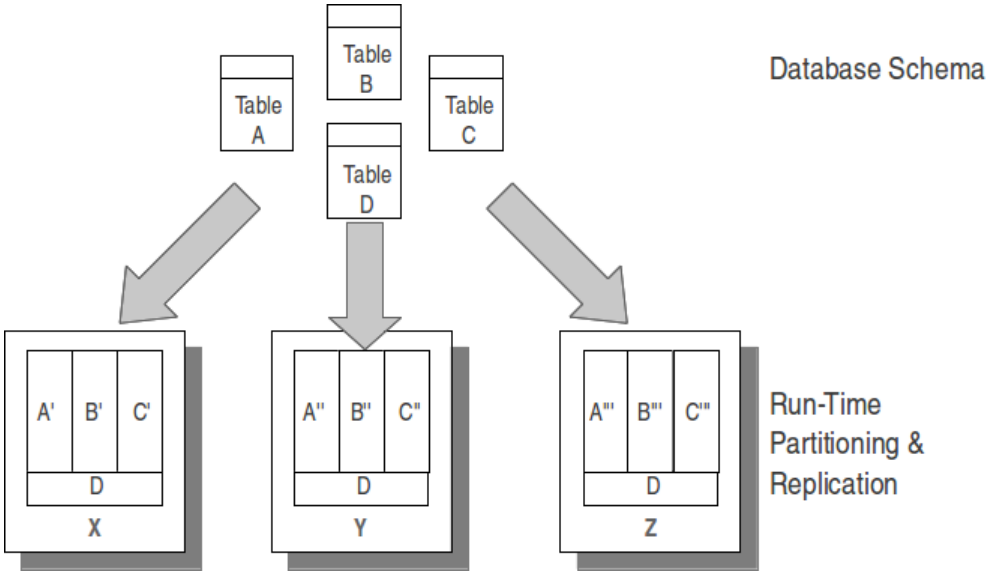
It is important to note that the VoltDB architecture is optimized for total throughput. Each transaction runs uninterrupted in its own thread, minimizing the individual latency per transaction (the time from when the transaction begins until processing ends). This also eliminates the overhead needed for locking, latching, and other administrative tasks, reducing the amount of time requests sit in the queue waiting to be executed. The result is that for a suitably partitioned schema, the number of transactions that can be completed in a second (i.e. throughput) is orders of magnitude higher than traditional databases.

1.3.3. Partitioned vs. Replicated Tables

Tables are partitioned in VoltDB based on a column that you, the developer or designer, specify. When you choose partitioning columns that match the way the data is accessed by the stored procedures, it optimizes execution at runtime.

To further optimize performance, VoltDB allows certain database tables to be replicated to all nodes of the cluster. For small tables that are largely read-only, this allows stored procedures to create joins between this table and another larger table while remaining a single-partitioned transaction. For example, a retail merchandising database that uses product codes as the primary key may have one table that simply correlates the product code with the product's category and full name, Since this table is relatively small and does not change frequently (unlike inventory and orders) it can be replicated for access by all partitions. This way stored procedures can retrieve and return user-friendly product information when searching by product code without impacting the performance of order and inventory updates and searches.

Figure 1.3. Replicating Tables



1.3.4. Ease of Scaling to Meet Application Needs

The VoltDB architecture is designed to simplify the process of scaling the database to meet the changing needs of your application. Increasing the number of nodes in a VoltDB cluster both increases throughput (by increasing the number of simultaneous queues in operation) and increases the data capacity (by increasing the number of partitions used for each table).

Scaling up a VoltDB database is a simple process that doesn't require any changes to the database schema or application code. You can either:

- Save the database (using a snapshot), then restart the database specifying the new number of nodes for the resized cluster and using restore to reload the schema and data.
- Add nodes "on the fly" while the database is running.

1.4. Working with VoltDB Effectively

It is possible to use VoltDB like any other SQL database, creating tables and performing ad hoc SQL queries using standard SQL statements. However, to take full advantage of VoltDB's capabilities, it is best to design your schema and your stored procedures to maximize the use of partitioned tables and procedures. There are also additional features of VoltDB to increase the availability and durability of your data. The following sections explain how to work effectively with VoltDB, including:

- Chapters 2 and 3 explain how to install VoltDB and create a new database.
- Chapters 4 through 8 explain how to design your database, stored procedures, and client applications to maximize performance.
- Chapters 9 through 12 explain how to create and use VoltDB clusters to increase scalability and availability.
- Chapters 13 through 15 explain how VoltDB ensures the durability of your data and how you can integrate VoltDB with other data sources using export for complete business solutions

Chapter 2. Installing VoltDB

VoltDB provides all the transactional consistency and durability of a SQL database, plus best-in-class performance and throughput due to its innovative in-memory design and architecture. The VoltDB software comes as pre-built distributions. This chapter explains the system requirements for running VoltDB, how to install and upgrade the software, and what resources are provided in the kit.

2.1. Operating System and Software Requirements

The following are the requirements for developing and running VoltDB applications.

Table 2.1. Operating System and Software Requirements

Operating System	VoltDB requires a 64-bit Linux-based operating system. Kits are built and qualified on the following platforms: • Debian 12.1 or later • Red Hat (RHEL) version 8.8 or later, including 9.0., 10.0 and subsequent releases • Rocky Linux version 8.8 or later, including 9.0, 10.0 and subsequent releases • Ubuntu versions 20.04, 22.04, 24.04 and subsequent releases. • Macintosh OS X 13.0 and later (for development only)
CPU	• Dual core[1] x86_64 processor • 64 bit • 1.6 GHz
Memory	4 Gbytes[2]
Java[3]	Java 17 or 21
Required Software	Time synchronization service, such as NTP or chrony[4] Python 3.9 or later
Recommended Software	Eclipse 3.x (or other Java IDE)

Footnotes:

1. Dual core processors are a minimum requirement. Four or eight physical cores are recommended for optimal performance.

2. Memory requirements are very specific to the storage needs of the application and the number of nodes in the cluster. However, 4 Gigabytes should be considered a minimum configuration.

3. VoltDB supports JDKs from OpenJDK or Oracle.

4. Time synchronization services minimize the time difference between nodes in a database cluster, which is critical for VoltDB. All nodes of the cluster should be configured to synchronize against the same time server. Using a single local server is recommended, but not required.

2.2. Installing VoltDB

VoltDB is distributed as a compressed tar archive. The file name identifies the version number and hardware architecture. The best way to install VoltDB is to unpack the distribution kit as a folder in the home directory of your personal account, like so:

```
$ tar -zxvf voltdb-ent-14.0.0-x86_64.tar.gz -C $HOME/
```

Installing into your personal directory gives you full access to the software and is most useful for development.

If you are installing VoltDB on a production server where the database will be run, you may want to install the software into a standard system location so that the database cluster can be started with the same commands on all nodes. The following shell commands install the VoltDB software in the folder `/opt/voltdb`:

```
$ sudo tar -zxvf voltdb-ent-14.0.0-x86_64.tar.gz -C /opt
$ cd /opt
$ sudo mv voltdb-ent-14.0.0-x86_64 voltdb
```

Note that installing as root using the **sudo** command makes the installation folders read-only for non-privileged accounts. Which is why installing in $HOME is recommended for development activities.

2.2.1. Upgrading From Older Versions

When upgrading an existing database from a recent version of VoltDB, the simplest way to upgrade is as follows:

1. Perform an orderly shutdown of the database, saving a final snapshot (**voltadmin shutdown --save**)
2. Upgrade the VoltDB software
3. Restart the database (**voltdb start**)

VoltDB automatically restores the final snapshot taken before the upgrade. This process allows you to upgrade a single database cluster from any two supported versions of Volt Active Data. However, for production environments Volt offers several methods for upgrading databases without incurring any downtime, including upgrades utilizing multiple Active(n) clusters and performing in-service upgrades. See the section on "Upgrading Existing VoltDB Installations" in the *VoltDB Administrator's Guide* or the section on "Upgrading the VoltDB Software and Helm Charts" in the *Kubernetes Administrator's Guide* for more information.

Finally, if you are upgrading from a version before V6.8, you need to save and restore the snapshot manually. In which case, the recommended steps for upgrading an existing database are:

1. Place the database in admin mode (**voltadmin pause --wait**).
2. Perform a manual snapshot of the database (**voltadmin save --blocking**).
3. Shutdown the database (**voltadmin shutdown**).
4. Upgrade VoltDB.
5. Initialize a new database root directory (**voltdb init**)
6. Start the new database in admin mode (**voltdb start --pause**).
7. Restore the snapshot created in Step #2 (**voltadmin restore**).
8. Return the database to normal operations (**voltadmin resume**).

2.3. Setting Up Your Environment

VoltDB comes with shell command scripts that simplify the process of developing and deploying VoltDB applications. These scripts are in the /bin folder under the installation root and define short-cut commands for executing many VoltDB actions. To make the commands available to your session, you must include the /bin directory as part of your PATH environment variable.

You can add the `/bin` directory to your PATH variable by redefining PATH. For example, the following shell command adds `/bin` to the end of the environment PATH, assuming you installed the VoltDB Enterprise Edition as `/voltdb-ent-n.n.n-xxxx` in your $HOME directory:

```
$ export PATH="$PATH:$HOME/voltdb-ent-n.n.n-xxxx/bin"
```

To avoid having to redefine PATH every time you create a new session, you can add the preceding command to your shell login script. For example, if you are using the bash shell, you would add the preceding command to the `$HOME/.bashrc` file.

2.4. What is Included in the VoltDB Distribution

Table 2.2 lists the components that are provided as part of the VoltDB distribution.

Table 2.2. Components Installed by VoltDB

Component	Description
VoltDB Software & Runtime	The VoltDB software comes as Java archives (.JAR files) and a callable library that can be found in the `/voltdb` subfolder. Other software libraries that VoltDB depends on are included in a separate `/lib` subfolder.
Volt Management Center	Volt Management Center is a browser-based management tool for monitoring, examining, and querying a running VoltDB database. The Management Center is bundled with the VoltDB server software and runs as a separate process, either on one of the database servers or a separate server. See the *Volt Administrator's Guide* or *Volt Kubernetes Administrator's Guide* for instructions on configuring and starting the VMC service. You start the management console interface by connecting to the HTTP port of the VMC server. For example, `http://voltvmc:8080/`. Note that the VMC service must be running to use the Management Center or the HTTP programming interface.
Shell Commands	The `/bin` subfolder contains executable scripts to perform common VoltDB tasks, such as starting the VoltDB server process and issuing database queries from the command line using sqlcmd, Add the `/bin` subfolder to your PATH environment variable to use the following shell commands:

Component	Description
	csvloader jdbcloader kafkaloader sqlcmd voltadmin voltdb
Documentation	Online documentation, including the full manuals and javadoc describing the Java programming interface, is available in the `/doc` subfolder.

Chapter 3. Starting the Database

This chapter describes the procedures for starting and stopping a VoltDB database and includes details about configuring the database. The chapter contains the following sections:

- Section 3.1, “Initializing and Starting a VoltDB Database”
- Section 3.2, “Initializing and Starting a VoltDB Database on a Cluster”
- Section 3.3, “Stopping a VoltDB Database”
- Section 3.5, “Restarting a VoltDB Database”
- Section 3.6, “Updating Nodes on the Cluster”
- Section 3.7, “Defining the Cluster Configuration”

3.1. Initializing and Starting a VoltDB Database

Before you start a VoltDB database, you must initialize the root directory where VoltDB stores its configuration data, logs, and other disk-based information. Once you initialize the root directory, you can start the database. For example, you can accept the defaults for the **voltdb init** and **start** commands to initialize and start a new, single-node database suitable for developing and testing a database and application.

```
$ voltdb init
$ voltdb start
```

This creates a VoltDB root directory as a subfolder of your current working directory and starts a database with all default options. You only need to initialize the root directory once and can then start and stop the database as often as you like.

```
$ voltadmin shutdown
$ voltdb start
```

If you are using command logging, which is enabled by default in the VoltDB Enterprise Edition, VoltDB automatically saves and recovers your database between any stoppage and a restart. If you are *not* using command logging, you will want to save a snapshot before shutting down. The easiest way to do this is by adding the --save argument to the **shutdown** command.

The snapshot is automatically restored when the database restarts:

```
$ voltadmin shutdown --save
$ voltdb start
```

If you want to create a new database, you can reinitialize the root directory. However, you must use the --force flag if the database has already been used; VoltDB will not clear the root directory of existing data unless you explicitly "force" it to.

```
$ voltdb init --force
$ voltdb start
```

Also, you can specify an alternate location for the root directory using the `--dir` or `-D` flag. Of course, you must specify the same location for the root directory when both initializing and starting the database. You cannot start a database in a directory that has not been initialized.

```
$ voltdb init  --dir=~/mydb
$ voltdb start --dir=~/mydb
```

In most cases, you will want to use additional arguments to configure the server and database options. But the preceding commands are sufficient to get you started in a test environment. The rest of this chapter explains how to use other arguments and how to start, stop, and recover a database when using a cluster.

Finally, when using the VoltDB Enterprise Edition, you must provide a license file when initializing the database. VoltDB looks for the license as a file named `license.xml` in three possible locations, in the following order:

1. The current working directory
2. The directory where the VoltDB image files are installed (usually in the `/voltdb` subfolder of the installation directory)
3. The current user's home directory

If the license file is not in any of these locations, you must explicitly identify it when you issue the **voltdb init** command using the `--license` or `-l` flag. For example, the command might be:

```
$ voltdb init -l /usr/share/voltdb-license.xml
```

The examples in this manual assume that the license file is in one of the default locations and therefore do not show the `--license` flag for simplicity's sake.

3.2. Initializing and Starting a VoltDB Database on a Cluster

You initialize and start a cluster the same way you start a single node: with the **voltdb init** and **start** commands. The only difference is that when starting the cluster, you must tell the cluster nodes how big the cluster is and which nodes to use as potential hosts for the startup.

You initialize a root directory on each server using the **voltdb init** command. You can accept the default configuration as shown in the previous section. However, when setting up a cluster you often want to make some configuration adjustments (for example, enabling K-safety). So it is a good idea to get into the habit of specifying a configuration file.

You can specify one or more configuration files with the `--config` or `-C` flag when you initialize the root directory. All nodes must use the same configuration files. For example:

```
$ voltdb init -D ~/mydb --config=myconfig.yaml
```

Once the nodes are initialized, you start the cluster by issuing the **voltdb start** command on all nodes specifying the following information:

- **Number of nodes in the cluster:** When you start the cluster, you specify how many servers will make up the cluster using the `--count` flag.
- **Host names:** You specify the hostnames or IP addresses of one or more servers from the cluster that are potential "hosts" for coordinating the formation of the cluster. You specify the list of hosts with the `--host` or `-H` flag. You must specify at least one node as a host.

For each node of the cluster, log in and start the server process using the same **voltdb start** command. For example, the following example starts a five-node database cluster specifying voltsvr1 as the host node.

Be sure the number of nodes on which you run the command match the number of nodes specified in the `--count` argument.

```
$ voltdb start --count=5 -–host=voltsvr1
```

Or you can also use shortened forms for the argument flags:

```
$ voltdb start -c 5 -H voltsvr1
```

Although you only need to specify one potential host, it is a good idea to specify multiple hosts. This way, you can use the exact same command for both starting and rejoining nodes in a highly-available cluster. Even if the rejoining node is in the host list another, running node can be chosen to facilitate the rejoin.

To simplify even further, you can specify *all* of the servers in the `--host` argument. If you do this, you can skip the `--count` argument. If `--count` is missing, VoltDB assumes the number of servers in the `--host` list is complete and sets the server count to match. For example, the following command — issued on all three servers — starts a three node cluster:

```
$ voltdb start --host=svrA,svrB,svrC
```

When starting a VoltDB database on a cluster, the VoltDB server process performs the following actions:

1. If you are starting the database process on the node selected as the host node, it waits for initialization messages from the remaining nodes. The host is selected from the list of hosts on the command line and plays a special role during startup by managing the cluster initiation process. It is important that all nodes in the cluster can resolve the hostnames or IP addresses of the host nodes you specify.

2. If you are starting the database on a non-host node, it sends an initialization message to the host indicating that it is ready. The database is not operational until the correct number of nodes (as specified on the command line) have connected.

3. Once all the nodes have sent initialization messages, the host sends out a message to the other nodes that the cluster is complete. Once the startup procedure is complete, the host's role is over and it becomes a peer like every other node in the cluster. It performs no further special functions.

Manually logging on to each node of the cluster every time you want to start the database can be tedious. Instead, you can use secure shell (ssh) to execute shell commands remotely. By creating an ssh script (with the appropriate permissions) you can copy files and/or start the database on each node in the cluster from a single script. Or you can use distributed system management tools such as Chef and Puppet to automate the startup procedures.

3.3. Stopping a VoltDB Database

Once the VoltDB database is up and running, you can shut it down by stopping the VoltDB server processes on each cluster node. However, it is easier to stop the database as a whole with a single command. You do this with the **voltadmin shutdown** command, which pauses database activity, completes all current transactions, and empties any queued data (such as export or database replication) before shutting down. For example, entering the following command without specifying a host server will perform an orderly shut down the database cluster the current system is part of.

```
$ voltadmin shutdown
```

If you are not using command logging, which automatically saves all progress, be sure to add the --save argument to save a final snapshot before shutting down:

```
$ voltadmin shutdown --save
```

To shutdown a database running on another system, use the `--host` argument to access the remote database. For example, the following command shuts down the VoltDB database that includes the server zeus:

```
$ voltadmin shutdown --host=zeus
```

You can pause the database using the **voltadmin pause** command to restrict clients from accessing it while you perform changes in administration mode. You resume the database using the **voltadmin resume** command. See the *VoltDB Administrator's Guide* for more about modes of operation.

3.4. Saving the Data

Because VoltDB is an in-memory database, once the database server process stops, the database schema and the data itself are removed from memory. However, VoltDB can save this information to disk through the use of command logs and snapshots, so use of these features is strongly encouraged.

- *Command logging* provides the most complete data durability for VoltDB and is enabled by default in the VoltDB Enterprise Edition. Command logging works automatically by saving a record of every transaction. These logs can then be replayed if the database stops for any reason.

- *Snapshots*, on the other hand, provide a point-in-time copy of the database contents written to disk. You can create snapshots manually with the **voltadmin save** command, you can enable periodic (also known as automatic) snapshots, or you can save a final snapshot when you shutdown the database using the **voltadmin shutdown --save** command. Snapshots are restored when the database restarts, but only take you back to the state of the database at the time the last snapshot was saved.

To learn more about using command logging see Chapter 14, *Command Logging and Recovery*. To learn more about how to save and restore snapshots of the database, see Chapter 13, *Saving & Restoring a VoltDB Database*.

3.5. Restarting a VoltDB Database

Once a database stops, you can restart it using the same **voltdb start** command used to start the database the first time. Once the database starts, any command logs or snapshots are restored. In the VoltDB Enterprise Edition, command logs automatically restore the last state of the database. If no command log exist but a snapshot does, the database is restored to its state when that snapshot was taken. For example, the following command restarts a single-node database:

```
$ voltdb start
```

To restart a database on a cluster, issue the same **voltdb start** command used to start that cluster, including the server count and list of host nodes. For example:

```
$ voltdb start --count=5 --–host=voltsvr1
```

3.6. Updating Nodes on the Cluster

A cluster is a dynamic system in which nodes might be stopped either deliberately or by unforeseen circumstances, or nodes might be added to the cluster on-the-fly to scale the database for improved performance. The **voltdb start** command provides the following additional functions, described later in this book, for rejoining and adding nodes to a running VoltDB database:

- Section 10.3, “Recovering from System Failures” — Use the same **voltdb start** command to start the cluster or rejoin a failed node.

- Section 9.3.1, “Adding Nodes with Elastic Scaling” — Use **voltdb start** with the **--add** flag to add a new node to the running database cluster.

3.7. Defining the Cluster Configuration

Two important aspects of a VoltDB database are the physical layout of the cluster that runs the database and the database features you choose to use. You define the physical cluster layout on the **voltdb start** command using the `--count` and `--host` arguments. You enable and disable specific database features in configuration files when you initialize the database root directory with the **voltdb init** command.

In the simplest case — when running on a single node with no configuration specified — VoltDB defaults to eight execution sites per host, and a K-safety value of zero. You can customize the database by specifying options in one or more YAML configuration files when you initialize the database with the **voltdb init** command and the `--config` (or `-C`) qualifier. You can put all of the configuration properties in a single file, or you can modularize the configuration into separate files for individual topics. For example, the following command customizes the database using separate YAML files for common properties, directory paths, and security:

```
$ voltdb init --config=common.yaml,paths.yaml,security.yaml
```

Configuration files are in YAML format, where options are specified as a hierarchy of properties with each element of the hierarchy indented on a separate line and terminated by a colon. The actual property values follow the colon. For example:

```
deployment:
  cluster:
    sitesperhost: 12
    kfactor: 1
```

In the preceding example, the child properties of the `deployment.cluster` element define the layout of the database partitions, including:

- **sitesperhost** — specifies the number of partitions created on each server in the cluster. The `sites-perhost` value times the number of servers gives you the total number of partitions in the cluster. See Section 3.7.1, “Determining How Many Sites per Host” for more information about partition count.

- **kfactor** — specifies the K-safety value to use for durability when creating the database. The K-safety value controls the duplication of database partitions. See Chapter 10, *Availability* for more information about K-safety.

Configuration files also enable and configure many runtime options related to the database, which are described later in this book. For example, the configuration file can specify:

- Whether security is enabled and what users and passwords are needed to authenticate clients at runtime. See Chapter 12, *Security* for more information.

- A schedule for saving automatic snapshots of the database. See Section 13.2, “Scheduling Automated Snapshots”.

- Properties for exporting and importing data to other data sources. See Chapter 15, *Streaming Data: Import, Export, and Migration*.

For the complete list of properties and YAML file syntax, see Appendix E, *YAML Configuration Properties*.

3.7.1. Determining How Many Sites per Host

There is very little penalty for allocating more sites than needed for the partitions the database will use (except for incremental memory usage). Consequently, VoltDB defaults to eight sites per node to provide reasonable performance on most modern system configurations. This default does not normally need to be changed. However, for systems with a large number of available processors (16 or more) or older machines with fewer than 8 processors and limited memory, you may wish to tune the `sitesperhost` property.

The number of sites needed per node is related to the number of processor cores each system has, the optimal number being approximately 3/4 of the number of CPUs reported by the operating system. For example, if you are using a cluster of dual quad-core processors (in other words, 8 cores per node), the optimal number of partitions is likely to be 6 or 7 sites per node.

```
deployment:
  cluster:
    sitesperhost: 6
```

For systems that support hyperthreading (where the number of physical cores support twice as many threads), the operating system reports twice the number of physical cores. In other words, a dual quad-core system would report 16 virtual CPUs. However, each partition is not quite as efficient as on non-hyperthreading systems. So the optimal number of sites is more likely to be between 10 and 12 per node in this situation.

Because there are no hard and set rules, the optimal number of sites per node is best calculated by actually benchmarking the application to see what combination of cores and sites produces the best results. However, it is important to remember that all nodes in the cluster will use the same number of sites. So the best performance is achieved by using a cluster with all nodes having the same physical architecture (i.e. cores).

3.7.2. Configuring Paths for Runtime Features

An important aspect of some runtime features is that they make use of disk resources for persistent storage across sessions. For example, automatic snapshots need a directory for storing snapshots of the database contents. Similarly, export uses disk storage for writing overflow data if the export connector cannot keep up with the export queue.

You can specify individual paths for each feature in the configuration. If not, VoltDB creates subfolders for each feature in the database root directory as needed, which can be useful for testing. However, in production, it is useful to direct certain high volume features, such as command logging, to separate devices to avoid disk I/O affecting database performance.

You can identify specific path locations for the following features using the `paths` property:

- Command logging (`deployment.paths.commandlog`)
- Command log snapshots(`deployment.paths.commandlogsnapshot`)
- Export overflow (`deployment.paths.exportoverflow`)
- Snapshots (`deployment.paths.snapshots`)

If you specify a relative rather than an absolute path, it is relative to the database root directory. If you name a specific feature path and it does not exist, VoltDB attempts to create it for you. For example, the export overflow path contains temporary data which can be deleted periodically. The following configuration file specifies `/opt/overflow` as the directory for export overflow.

```
deployment:
  paths:
    exportoverflow:
      path: "/opt/overflow"
```

3.7.3. Verifying your Hardware Configuration

The configuration files and start command options define the desired configuration of your database cluster. However, there are several important aspects of the physical hardware and operating system configuration that you should be aware of before running VoltDB:

- VoltDB can operate on heterogeneous clusters. However, best performance is achieved by running the cluster on similar hardware with the same type of processors, number of processors, and amount of memory on each node.
- All nodes must be able to resolve the IP addresses and host names of the other nodes in the cluster. That means they must all have valid DNS entries or have the appropriate entries in their local hosts file.
- You must run a time synchronization service such as Network Time Protocol (NTP) or chrony on all of the cluster nodes, preferably synchronizing against the same local time server. If the time skew between nodes in the cluster is greater than 200 milliseconds, VoltDB cannot start the database.
- It is strongly recommended that you configure your time service to avoid adjusting time backwards. For example, in NTP this is done using the `-x` argument. If the server time moves backward, VoltDB must pause and wait for time to catch up.

Chapter 4. Designing the Database Schema

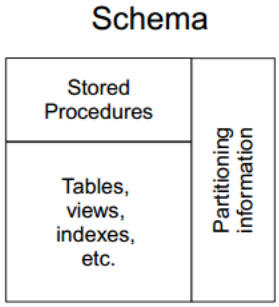
VoltDB is a relational database product. Relational databases consist of tables and columns, with constraints, indexes, and views. VoltDB uses standard SQL database definition language (DDL) statements to specify the database schema. So designing the schema for a VoltDB database uses the same skills and knowledge as designing a database for Oracle, MySQL, or any other relational database product.

This guide describes the stages of application design by dividing the work into three chapters:

- **Design the schema** in DDL to define the database structure. Schema design is covered in this chapter.
- **Design stored procedures** to access data in the database. Stored procedures provide client applications an application programming interface (API) to the database. Stored procedures are covered in Chapter 5, *Designing Stored Procedures to Access the Database*.
- **Design clients** to provide business logic and also connect to the database to access data. Client application design is covered in Chapter 6, *Designing VoltDB Client Applications*.

The database schema is a specification that describes the structure of the VoltDB database such as tables and indexes, identifies the stored procedures that access data in the database, and defines the way tables and stored procedures are partitioned for fast data access. When designing client applications to use the database, the schema specifies the details needed about data types, tables, columns, and so on.

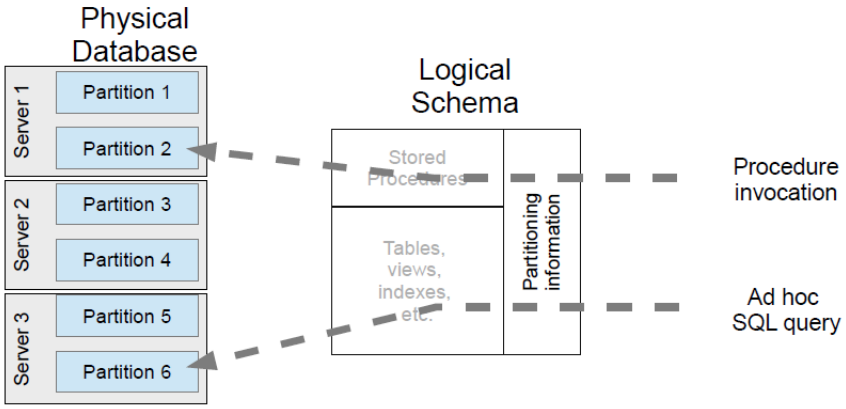
Figure 4.1. Components of a Database Schema



Along with designing your database tables, an important aspect of VoltDB database design is partitioning, which provides much more efficient access to data and processing. Partitioning distributes the rows of a table and the processing to access the table across several, independent partitions instead of one. Your design requires coordinating the partitioning of both database tables and the stored procedures that access the tables. At design time you choose a column on which to partition a table's rows. You also partition stored procedures on the same column if they use the column to identify which rows to operate on in the table.

At runtime, VoltDB decides which cluster nodes and partitions to use for the table partitions and consistently allocates rows to the appropriate partition. Figure 4.2, “Partitions Distribute Table Data and Stored Procedure Processing” shows how when data is inserted into a partitioned table, VoltDB automatically allocates the data to the correct partition. Also, when a partitioned stored procedure is invoked, VoltDB automatically executes the stored procedure in the single partition that has the data requested.

Figure 4.2. Partitions Distribute Table Data and Stored Procedure Processing



The following sections of this chapter provide guidelines for designing VoltDB database schemas. Although gathering business requirements is a typical first step in database application design, it is outside the scope of this guide.

4.1. How to Enter DDL Statements

You use standard SQL DDL statements to design your schema. For a full list of valid VoltDB DDL, see Appendix A, *Supported SQL DDL Statements*. The easiest way to enter your DDL statements is using VoltDB's command line utility, sqlcmd. Using sqlcmd you can input DDL statements in several ways:

- Redirect standard input from a file when you start sqlcmd:

```
$ sqlcmd < myschema.sql
```

- Import from a file using the sqlcmd file directive:

```
$ sqlcmd
1> file myschema.sql;
```

- Enter DDL directly at the sqlcmd prompt:

```
$ sqlcmd
1>
2> CREATE TABLE Customer (
3>   CustomerID INTEGER UNIQUE NOT NULL,
4>   FirstName VARCHAR(15),
5>   LastName VARCHAR (15),
6>   PRIMARY KEY(CustomerID)
7> );
```

- Copy DDL from another application and paste it into the sqlcmd prompt:

```
$ sqlcmd
1> CREATE TABLE Flight (
2>   FlightID INTEGER UNIQUE NOT NULL,
3>   DepartTime TIMESTAMP NOT NULL,
4>   Origin VARCHAR(3) NOT NULL,
5>   Destination VARCHAR(3) NOT NULL,
6>   NumberOfSeats INTEGER NOT NULL,
```

```
  7>   PRIMARY KEY(FlightID)
  8> );
```

The following sections show how to design and create schema objects. DDL statements and techniques for changing a schema are described later in Section 4.6, “Modifying the Schema”.

4.2. Creating Tables and Primary Keys

The schema in this section is referred to throughout the design chapters of this guide. Let's assume you are designing a flight reservation system. At its simplest, the application requires database tables for the flights, the customers, and the reservations. Example 4.1, “DDL Example of a Reservation Schema” shows how the schema looks as defined in standard SQL DDL. For the VoltDB-specific details for creating tables, see CREATE TABLE. When defining the data types for table columns, refer to Table A.1, “Supported SQL Datatypes”.

Example 4.1. DDL Example of a Reservation Schema

```
CREATE TABLE Flight (
   FlightID INTEGER UNIQUE NOT NULL,
   DepartTime TIMESTAMP NOT NULL,
   Origin VARCHAR(3) NOT NULL,
   Destination VARCHAR(3) NOT NULL,
   NumberOfSeats INTEGER NOT NULL,
   PRIMARY KEY(FlightID)
);

CREATE TABLE Reservation (
   ReserveID INTEGER NOT NULL,
   FlightID INTEGER NOT NULL,
   CustomerID INTEGER NOT NULL,
   Seat VARCHAR(5) DEFAULT NULL,
   Confirmed TINYINT DEFAULT '0'
);

CREATE TABLE Customer (
   CustomerID INTEGER UNIQUE NOT NULL,
   FirstName VARCHAR(15),
   LastName VARCHAR (15),
   PRIMARY KEY(CustomerID)
);
```

To satisfy entity integrity you can specify a table's primary key by providing the usual PRIMARY KEY constraint on one or more of the table’s columns. To create a simple key, apply the PRIMARY KEY constraint to one of the table's existing columns whose values are unique and not null, as shown in Example 4.1, “DDL Example of a Reservation Schema”.

To create a composite primary key from a combination of columns in a table, apply the PRIMARY KEY constraint to multiple columns with typical DDL such as the following:

```
$ sqlcmd
1> CREATE TABLE Customer (
2>   FirstName VARCHAR(15),
3>   LastName VARCHAR (15),
4>   CONSTRAINT pkey PRIMARY KEY (FirstName, LastName)
5> );
```

4.3. Analyzing Data Volume and Workload

A schema is not all you need to define the database effectively. You also need to know the expected volume and workload on the database. For our example, let's assume that we expect the following volume of data at any given time:

- Flights: 2,000
- Reservations: 200,000
- Customers: 1,000,000

This additional information about the volume and workload affects the design of both the database and the client application, because it impacts what SQL queries need to be written for accessing the data and what attributes (columns) to share between tables. Table 4.1, “Example Application Workload” defines a set of procedures the application must perform. The table also shows the estimated workload as expected frequency of each procedure. Procedures in bold modify the database.

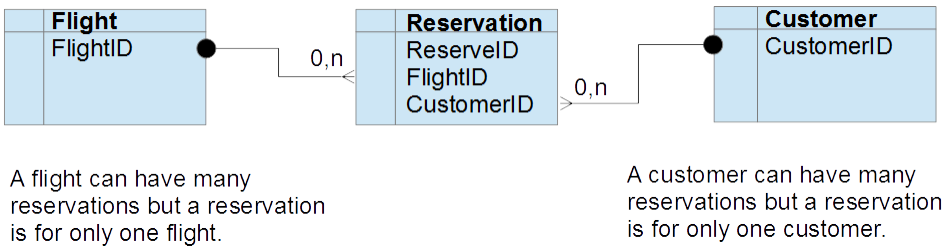
Table 4.1. Example Application Workload

Use Case	Frequency
Look up a flight (by origin and destination)	10,000/sec
See if a flight is available	5,000/sec
Make a reservation	1,000/sec
Cancel a reservation	200/sec
Look up a reservation (by reservation ID)	200/sec
Look up a reservation (by customer ID)	100/sec
Update flight info	1/sec
Take off (close reservations and archive associated records)	1/sec

You can make your procedures that access the database transactional by defining them as VoltDB stored procedures. This means each stored procedure call completes or rolls back if necessary, thus maintaining data integrity. Stored procedures are described in detail in Chapter 5, *Designing Stored Procedures to Access the Database*.

In our analysis we also need to consider referential integrity, where relationships are maintained between tables with shared columns that link tables together. For example, Figure 4.3, “Diagram Representing the Flight Reservation System” shows that the Flight table links to the Reservation table where FlightID is the shared column. Similarly, the Customer table links to the Reservation table where CustomerID is the common column.

Figure 4.3. Diagram Representing the Flight Reservation System



Since VoltDB stored procedures are transactional, you can use stored procedures to maintain referential integrity between tables as data is added or removed. For example, if a customer record is removed from the Customer table, all reservations for that customer need to be removed from the Reservations table as well.

With VoltDB, you use all this additional information about volume and workload to configure the database and optimize performance. Specifically, you want to partition the individual tables to ensure efficiency. Partitioning is described next.

4.4. Partitioning Database Tables

This section discusses how to partition a database to maximize throughput, using the flight reservation case study as an example. To partition a table, you choose a column of the table that VoltDB can use to uniquely identify and distribute the rows into partitions. The goal of partitioning a database table is to ensure that the most frequent transactions on the table execute in the same partition as the data accessed. We call this a *single-partitioned transaction*. Thus the stored procedure must uniquely identify a row by the partitioning column value. This is particularly important for queries that modify the data, such as INSERT, UPDATE, and DELETE statements.

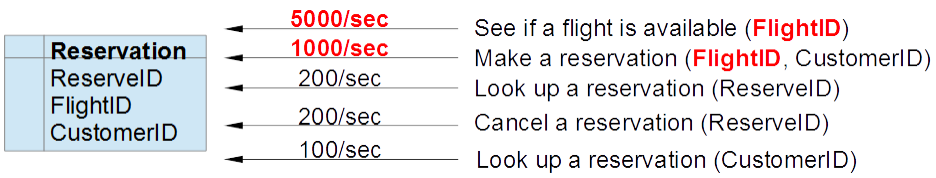
Looking at the workload for the reservation system in the previous section, the important transactions to focus on are:

- Look up a flight
- See if a flight is available
- Look up a reservation
- **Make a reservation**

Of these transactions, only the last modifies the database.

4.4.1. Choosing a Column on which to Partition Table Rows

We will discuss the Flight table later, but first let's look at the Reservation table. Looking at the schema alone (Example 4.1), ReserveID might look like a good attribute to use to partition the table rows. However, looking at the workload, there are only two transactions that are keyed to the ReserveID (“Cancel a reservation” and “Look up a reservation (by reservation ID)”), each of which occur only 200 times a second. Whereas, “See if a flight is available” , which requires looking up reservations by the FlightID, occurs 5,000 times a second, or 25 times as frequently. Therefore, the Reservation table is best partitioned on the FlightID column.



Moving to the Customer table, CustomerID is used for most data access. Although customers might need to look up their record by name, the first and last names are not guaranteed to be unique. Therefore, CustomerID is the best column to use for partitioning the Customer table.

```
CREATE TABLE Customer (
```

```
   CustomerID INTEGER UNIQUE NOT NULL,
   FirstName VARCHAR(15),
   LastName VARCHAR (15),
   PRIMARY KEY(CustomerID)
);
```

4.4.2. Specifying Partitioned Tables

Once you choose the column to use for partitioning a database table, you define your partitioning choices in the database schema. Specifying the partitioning along with the schema DDL helps keep all of the database structural information in one place.

You define the partitioning scheme using VoltDB's PARTITION TABLE statement, specifying the partitioning column for each table. For example, to specify FlightID and CustomerID as the partitioning columns for the Reservation and Customer tables, respectively, your database schema must include the following statements:

```
$ sqlcmd
1> PARTITION TABLE Reservation ON COLUMN FlightID;
2> PARTITION TABLE Customer ON COLUMN CustomerID;
```

4.4.3. Design Rules for Partitioning Tables

The following are the rules to keep in mind when choosing a column by which to partition table rows:

- **There can be only one partition column per table.** If you need to partition a table on two columns (for example first and last name), add an additional column (fullname) that combines the values of the two columns and use this new column to partition the table.
- **If the table has a primary key, the partitioning column must be included in the primary key.**
- **Any integer, string, or byte array column can identify the partition.** VoltDB can partition rows on any column that is an integer (TINYINT, SMALLINT, INTEGER, or BIGINT), string (VARCHAR), or byte array (VARBINARY) datatype. (See also Table A.1, “Supported SQL Datatypes”.)
- **Partition column values cannot be null.** The partition columns do not need to have unique values, but you must specify NOT NULL in the schema for the partition column. Numeric fields can be zero and string or character fields can be empty, but the column cannot contain a null value.

The following are some additional recommendations:

- Choose a column with a reasonable distribution of values so that rows of data will be evenly partitioned.
- Choose a column that maximizes use of single-partitioned stored procedures. If one procedure uses column A to lookup data and two procedures use column B to lookup data, partition on column B. The goal of partitioning is to make the most frequent transactions single-partitioned.
- If you partition more than one table on the same column attribute, VoltDB will partition them together.

4.5. Replicating Database Tables

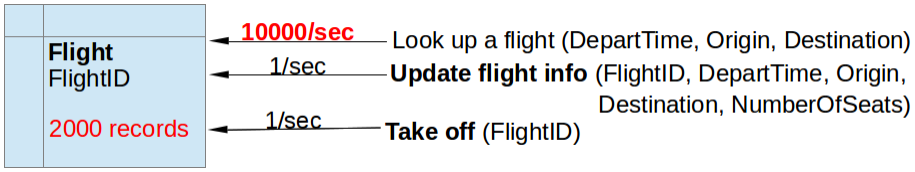
With VoltDB, tables are either partitioned or replicated across all nodes and sites of a VoltDB database. Smaller, mostly read-only tables are good candidates for replication. Note also that if a table needs to be

accessed frequently by columns other than the partitioning column, the table should be replicated instead because there is no guarantee that a particular partition includes the data that the query seeks.

The previous section describes how to partition the Reservation and Customer tables as examples, but what about the Flight table? It is possible to partition the Flight table (for example, on the FlightID column). However, not all tables benefit from partitioning.

4.5.1. Choosing Replicated Tables

Looking at the workload of the flight reservation example, the Flight table has the most frequent accesses (at 10,000 a second). However, these transactions are read-only and may involve any combination of three columns: the departure time, the point of origin, and the destination. This makes it hard to partition the table in a way that would make the transaction single-partitioned because the lookup is not restricted to one table column.



Fortunately, the number of flights available for booking at any given time is limited (estimated at 2,000) and so the size of the table is relatively small (approximately 36 megabytes). In addition, the vast majority of the transactions involving the Flight table are read-only except when new flights are added and at take-off (when the records are deleted). Therefore, Flight is a good candidate for replication.

Note that the Customer table is also largely read-only. However, because of the volume of data in the Customer table (a million records), it is not a good candidate for replication, which is why it is partitioned.

4.5.2. Specifying Replicated Tables

In VoltDB, you do not explicitly state that a table is replicated. If you do not specify a partitioning column in the database schema, the table will by default be replicated.

So, in our flight reservation example, there is no explicit action required to replicate the Flight table. However, it is very important to specify partitioning information for tables that you want to partition. If not, they will be replicated by default, significantly changing the performance characteristics of your application.

4.6. Modifying the Schema

You can use DDL to add, modify, or remove schema objects as the database is running. For a list of all valid DDL you can use, see Appendix A, *Supported SQL DDL Statements*. You can do the following types of schema changes:

- Modifying Tables — You can add, modify (alter), and remove (drop) table columns. You can also add and drop table constraints. Finally, you can drop entire tables.
- Adding and Dropping Indexes — You can add and remove (drop) named indexes.
- Modifying Partitioning for Tables and Stored Procedures — You can un-partition stored procedures and re-partition stored procedures on a different column, For tables you can change a table between partitioned and replicated, and repartition a table on a different column,

- Modify roles and users — To learn about modifying roles and users, see Chapter 12, *Security*.

VoltDB safely handles sqlcmd DDL entered by different users on different nodes of the cluster because it manages sqlcmd commands as transactions, just like stored procedures. To demonstrate the DDL statements to modify the schema, the following sections use a new table, Airport, added to the fight reservation as shown below:

```
CREATE TABLE Airport (
   AirportID integer NOT NULL,
   Name varchar(15) NOT NULL,
   City varchar(25),
   Country varchar(15),
   PRIMARY KEY (AirportID)
);
```

4.6.1. Effects of Schema Changes on Data and Clients

You can make many schema changes on empty tables with few restrictions. However, be aware that if a table has data, some schema changes are not allowed and other schema changes may modify or even remove data. When working with test data in your database, you can use TRUNCATE TABLE to empty the data from a table you are working on. Note that all DDL examples in this chapter assume the tables are empty.

We can think of the effects of schema changes on data in three severity levels:

- Schema change completes without damage to data
- Schema change fails to complete to avoid damage to data
- Schema change destroys data

VoltDB error messages and the documentation can help you avoid schema change attempts that fail to complete. For example, you cannot drop a table that has referencing procedures or views.

Obviously you need to be most aware of which schema changes cause data to be destroyed. In particular, removing objects from the schema will also remove the data they contain. Note that schema objects cannot be renamed with DDL, but objects can be replaced by performing a DROP and then ADD. However, it is important to realize that as a result of a DROP operation, such as DROP TABLE, the data associated with that table will be deleted before the new definition is added.

Plan and coordinate changes with client development. Stored procedures and ad hoc queries provide an API that clients use to access the database correctly. Changes to the schema can break the stored procedure calls client applications have developed, so use well-planned schedules to communicate database schema changes to others. Client applications depend on many schema definition features including (but not limited to):

- Table names
- Column names
- Column data types
- Primary key definitions
- Table partitions

- Stored procedure names
- Stored procedure partitioning

Plan and test carefully before making schema changes to a production database. Be aware that clients may experience connection issues during schema changes, especially for changes that take longer to complete, such as view or index changes.

Schema changes not only affect data, but the existence of data in the database affects the time it takes to process schema changes. For example, when there are large amounts of data, some DDL statements can block processing, resulting in a noticeable delay for other pending transactions. Examples include adding indexes, creating new table columns, and modifying views.

4.6.2. Viewing the Schema

The Volt Management Center provides a web browser view of database information, including the DDL schema source. Use a web browser to view the Volt Management Center (VMC) service on port 8080 (for example, http://vmcsvc:8080).

You can also use the sqlcmd **show** directive to see a list of the current database tables and all procedures. For additional details about the schema, execute the @SystemCatalog system procedure. Use any of the following arguments to @SystemCatalog to obtain details about a component of the database schema:

- TABLES
- COLUMNS
- INDEXINFO
- PRIMARYKEYS
- PROCEDURES
- PROCEDURECOLUMNS

For example:

```
$ sqlcmd
1> SHOW TABLES;
2> SHOW PROCEDURES;
3> EXEC @SystemCatalog COLUMNS;
```

4.6.3. Modifying Tables

After creating a table in a database with CREATE TABLE, you can use ALTER TABLE to make the following types of table changes:

- Altering a Table Column's Data Definition
- Adding and Dropping Table Columns
- Adding and Dropping Table Constraints

To drop an entire table, use the DROP TABLE DDL statement.

4.6.3.1. Altering a Table Column's Data Definition

You can make the following types of alterations to a table column's data definition:

```
$ sqlcmd
1> ALTER TABLE Airport ALTER COLUMN Name VARCHAR(25);              ❶
2> ALTER TABLE Airport ALTER COLUMN Country SET DEFAULT 'USA';     ❷
3> ALTER TABLE Airport ALTER COLUMN Name SET NOT NULL;             ❸
```

The examples are described as follows:

❶ Change a column's data type. In our example we decided we needed more than 15 characters for the Airport Name so we changed it to 25 characters.

If the table has no existing data, you can make any data type changes. However, if the table already contains data, the new type must be larger than the old one. This restriction prevents corrupting existing data values that might be larger than the size of the new data type (See also Table A.1, “Supported SQL Datatypes”.)

❷ Set or drop the column's DEFAULT value. In our example we assume the application is to be used mostly for US domestic travel so we can set a default value for the Airport Country of 'USA'.

To remove a default, redefine the column data definition, for example:

```
ALTER TABLE Airport ALTER COLUMN Country VARCHAR(15);
```

❸ Change whether the column is NULL or NOT NULL. In our example we set the AirportID to be not null because this is a required field.

If the table has existing data, you cannot change a column to not null.

4.6.3.2. Adding and Dropping Table Columns

```
$ sqlcmd
1> ALTER TABLE Airport ADD COLUMN AirportCode VARCHAR(3)        ❶
2> BEFORE AirportID;
3> ALTER TABLE Airport DROP COLUMN AirportID;                   ❷
```

The examples are described as follows:

❶ Add table columns. In our example, we have decided not to use the integer AirportID for airport identification but to instead add an AirportCode, which uses a unique three-letter code for any airport as defined by the International Air Transport Association's airport codes.

You cannot rename or overwrite a column but you can drop and add columns. When adding a column, you must include the new column name and the data type. Options you may include are:

- DEFAULT value — If a table contains data, the values for the new column will be automatically filled in with the default value.
- NOT NULL — If the table contains data, you must include a default value if you specify a NOT NULL column.
- One of the following index type constraints including PRIMARY KEY, UNIQUE, or ASSUME-UNIQUE.

  Note, we recommend that you not define the UNIQUE or ASSUMEUNIQUE constraint directly on a column definition when adding a column or creating a table. If you do, the constraint has no name

so you cannot drop the constraint without dropping the entire column. Instead, we recommend you apply UNIQUE or ASSUMEUNIQUE by adding the constraint (see Section 4.6.3.3, “Adding and Dropping Table Constraints”) or by adding an index with the constraint (see Section 4.6.4, “Adding and Dropping Indexes”). Defining these constraints this way names the constraint, which makes it easier to drop later if necessary.

- BEFORE column-name — Table columns cannot be reordered but the BEFORE clause allows you to place a new column in a specific position with respect to the existing columns of the table.

❷ Drop table columns. In our example we drop the AirportID column because we are replacing it with the AirportCode column.

You cannot remove a column that has a reference to it. You have to remove all references to the column first. References to a column may include:

- A stored procedure
- An index
- A view

4.6.3.3. Adding and Dropping Table Constraints

You cannot alter a table constraint but you can add and drop table constraints. If the table contains existing data, you cannot add UNIQUE, ASSUMEUNIQUE, or PRIMARY KEY constraints.

```
$ sqlcmd
1> ALTER TABLE Airport ADD CONSTRAINT                          ❶
2>    uniquecode UNIQUE (Airportcode);
3> ALTER TABLE Airport ADD PRIMARY KEY (AirportCode);          ❷
```

The examples are described as follows:

❶ Add named constraints UNIQUE or ASSUMEUNIQUE. In our example, we add the UNIQUE constraint to the AirportCode column. To drop a named constraint, include the name using the format in the following example:

```
ALTER TABLE Airport DROP CONSTRAINT uniquecode;
```

❷ Add unnamed constraint PRIMARY KEY. In our example, we add the PRIMARY KEY constraint to the new AirportCode column.

When adding a table constraint, it must not conflict with the other columns of the table. For example, only one primary key is allowed for a table so you cannot add the PRIMARY KEY constraint to an additional column.

To drop the PRIMARY KEY, include the type of constraint using the following format:

```
ALTER TABLE Airport DROP PRIMARY KEY;
```

4.6.4. Adding and Dropping Indexes

Use CREATE INDEX to create an index on one or more columns of a table. Use DROP INDEX to remove an index from the schema. The following example modifies the flight reservation schema by adding an index to the Flight table to improve performance when looking up flights.

```
$ sqlcmd
1> CREATE INDEX flightTimeIdx ON Flight (departtime);
```

The CREATE INDEX statement explicitly creates an index. VoltDB creates an index implicitly when you specify the table constraints UNIQUE, PRIMARY KEY, or ASSUMEUNIQUE. Use the ALTER TABLE statement to add or drop these table constraints along with their associated indexes, as shown in Section 4.6.3, “Modifying Tables”.

4.6.5. Modifying Partitioning for Tables and Stored Procedures

Any changes to the schema must be carefully coordinated with the design and development of stored procedures. This not only applies to column names, data types, and so on, but also to the partition plan.

How to partition tables and stored procedures using the PARTITION TABLE and CREATE PROCEDURE PARTITION ON statements is explained in Section 4.4, “Partitioning Database Tables” and Section 5.3.3, “Partitioning Stored Procedures in the Schema”.

You can change the partitioning of stored procedures, and you can change a table to a replicated table or repartition it on a different column. However, because of the intricate dependencies of partitioned tables and stored procedures, this can only be done by dropping and re-adding the tables and procedures. Also, you must pay close attention to the order in which objects are dropped and added.

The following DDL examples demonstrate some partitioning modifications to a table and stored procedures.

- Un-partitioning a Stored Procedure
- Changing a Partitioned Table to a Replicated Table
- Re-partitioning a Table to a Different Column
- Updating a Stored Procedure
- Removing a Stored Procedure from the Database

The following DDL is added to the Flight reservation schema to help demonstrate the DDL partition changes described in this section.

```
$ sqlcmd
1> PARTITION TABLE Airport ON COLUMN Name;
2> CREATE PROCEDURE FindAirportCodeByName
3>    PARTITION ON TABLE Airport COLUMN Name
4>    AS SELECT TOP 1 AirportCode FROM Airport WHERE Name=?;
5>
6> CREATE PROCEDURE FindAirportCodeByCity AS
7>    SELECT TOP 1 AirportCode FROM Airport WHERE City=?;
```

The stored procedures are tested with the following sqlcmd directives:

```
$ sqlcmd
1> exec FindAirportCodeByName 'Logan Airport';
2> exec FindAirportCodeByCity 'Boston';
```

4.6.5.1. Un-partitioning a Stored Procedure

In the simplest case, you can un-partition a single-partitioned stored procedure by dropping and re-creating that procedure without including the PARTITION ON clause. You would do this if you are planning to

change the partitioning of the table itself, or if you are modifying the SQL statements within the procedure to require access to multiple partitions. In this example we drop the single-partitioned FindAirportCode-ByName procedure and re-create it as multi-partitioned assuming it will need to search all partitions to find an airport code by name.

```
$ sqlcmd
1> DROP PROCEDURE FindAirportCodeByName;
2> CREATE PROCEDURE FindAirportCodeByName AS
3>    SELECT TOP 1 AirportCode FROM Airport WHERE Name=?;
```

4.6.5.2. Changing a Partitioned Table to a Replicated Table

Important

You cannot change the partitioning of a table that has data in it. To change a partitioned table to a replicated one, you drop and re-create the table, which deletes any data that might be in the table.

Before executing the following steps, save the existing schema so you can easily re-create the table. The Volt Management Center provides a view of the existing database schema DDL source, which you can download and save.

```
$ sqlcmd
1> DROP PROCEDURE FindAirportCodeByName;                    ❶
2> DROP PROCEDURE FindAirportCodeByCity;
3> DROP TABLE Airport IF EXISTS CASCADE;                    ❷
4> CREATE TABLE AIRPORT (                                   ❸
5>    AIRPORTCODE varchar(3) NOT NULL,
6>    NAME varchar(25),
7>    CITY varchar(25),
8>    COUNTRY varchar(15) DEFAULT 'USA',
9>    CONSTRAINT UNIQUECODE UNIQUE (AIRPORTCODE),
10>    PRIMARY KEY (AIRPORTCODE)
11> );
12> CREATE PROCEDURE FindAirportCodeByName AS               ❹
13>    SELECT TOP 1 AirportCode FROM Airport WHERE Name=?;
14> CREATE PROCEDURE FindAirportCodeByCity AS
15>    SELECT TOP 1 AirportCode FROM Airport WHERE City=?;
```

The example is described as follows:

❶ Drop all stored procedures that reference the table. You cannot drop a table if stored procedures reference it.

❷ Drop the table. Options you may include are:

- IF EXISTS — Use the IF EXISTS option to avoid command errors if the named table is already removed.
- CASCADE — A table cannot be removed if it has index or view references. You can remove the references explicitly first or use the CASCADE option to have VoltDB remove the references along with the table.

❸ Re-create the table. By default, a newly created table is a replicated table.

❹ Re-create the stored procedures that access the table. If the stored procedure is implemented with Java and changes are required, modify and reload the code before re-creating the stored procedures. For more, see Section 5.3, “Installing Stored Procedures into the Database”.

4.6.5.3. Re-partitioning a Table to a Different Column

Important

You cannot change the partitioning of a table that has data in it. In order to re-partition a table you have to drop and re-create the table, which deletes any data that might be in the table.

Follow these steps to re-partition a table:

1. Un-partition the table by following the instructions in Section 4.6.5.2, “Changing a Partitioned Table to a Replicated Table”. The sub-steps are summarized as follows:

   a. Drop all stored procedures that reference the table.

   b. Drop the table.

   c. Re-create the table.

   d. Re-create the stored procedures that access the table.

2. Partition the table on the new column. In our example, it makes sense to partition the Airport table on the AirportCode column, where each row must be unique and non null.

   ```
   $ sqlcmd
   1> PARTITION TABLE Airport ON COLUMN AirportCode;
   ```

3. Re-partition stored procedures that should be single-partitioned. See Section 4.6.5.4, “Updating a Stored Procedure”.

4.6.5.4. Updating a Stored Procedure

This section describes how to update a stored procedure that has already been declared in the database with the CREATE PROCEDURE statement. The steps to update a stored procedure are summarized as follows:

1. If the procedure is implemented in Java, update the procedure's code, recompile, and repackage the jar file. For details, see Section 5.3, “Installing Stored Procedures into the Database”.

2. Ensure all tables and columns the procedure accesses are in the database schema.

3. Update the procedure in the database.

   - If the procedure is implemented in Java, use the sqlcmd **load classes** directive to update the class in the database. For example:

     ```
     $ sqlcmd
     1> load classes GetAirport.jar;
     ```

   - If the procedure is implemented with SQL, use the CREATE PROCEDURE AS command to update the SQL.

4. If required, re-partition the stored procedure. You partition procedures using the PARTITION ON clause in the CREATE PROCEDURE statement. If you need to re-partition the procedure, either changing the partitioning column or switching from replicated to partitioned or vice versa, perform the following steps:

   a. Use DROP PROCEDURE to remove the stored procedure.

b. Use CREATE PROCEDURE to re-declare the stored procedure, including the new partitioning scheme.

In our example so far, we have three stored procedures that are adequate to access the Airport table, so no additional procedures need to be partitioned:

- VoltDB automatically defined a default select stored procedure, which is partitioned on the AirportCode column. It takes an AirportCode as input and returns a table structure containing the AirportCode, Name, City, and Country.
- The FindAirportCodeByName stored procedure should remain multi-partitioned because it needs to search in all partitions.
- The FindAirportCodeByCity stored procedure should also remain multi-partitioned because it needs to search in all partitions.

4.6.5.5. Removing a Stored Procedure from the Database

If you've decided a stored procedure is no longer needed, use the following steps to remove it from the database:

1. Drop the stored procedure from the database.

```
$ sqlcmd
1> DROP PROCEDURE GetAirport;
```

2. Remove the code from the database. If the procedure is implemented with Java, use the sqlcmd **remove classes** directive to remove the procedure's class from the database.

```
2> remove classes myapp.procedures.GetAirport;
```

Chapter 5. Designing Stored Procedures to Access the Database

As you can see from Chapter 4, *Designing the Database Schema*, defining the database schema and the partitioning plan go hand in hand with understanding how the data is accessed. The two must be coordinated to ensure optimum performance. Your stored procedures must use the same attribute for partitioning as the table being accessed. Proper partitioning ensures that the table rows the stored procedure requests are in the same partition in which the procedure executes, thereby ensuring maximum efficiency.

It doesn't matter whether you design the partitioning first or the data access first, as long as in the end they work together. However, for the sake of example, we will use the schema and partitioning outlined in Chapter 4, *Designing the Database Schema* when discussing how to design the data access.

5.1. How Stored Procedures Work

The key to designing the data access for VoltDB applications is that complex or performance sensitive access to the database should be done through stored procedures. It is possible to perform ad hoc queries on a VoltDB database. However, ad hoc queries do not benefit as fully from the performance optimizations VoltDB specializes in and therefore should not be used for frequent, repetitive, or complex transactions.

Within the stored procedure, you access the database using standard SQL syntax, with statements such as SELECT, UPDATE, INSERT, and DELETE. You can also include your own code within the stored procedure to perform calculations on the returned values, to evaluate and execute conditional statements, or to perform many other functions your applications may need.

5.1.1. VoltDB Stored Procedures are Transactional

In VoltDB, a stored procedure and a transaction are one and the same. Thus when you define a stored procedure, VoltDB automatically provides ACID transaction guarantees for the stored procedure. This means that stored procedures fully succeed or automatically roll back as a whole if an error occurs (atomic). When stored procedures change the data, the database is guaranteed to remain consistent. Stored procedures execute and access the database completely isolated from each other, including when they execute concurrently. Finally, stored procedure changes to the database are guaranteed to be saved and available for subsequent database access (durable).

Because the transaction is defined in advance as a stored procedure, there is no need for your application to manage transactions using specific transaction commands such as BEGIN, ROLLBACK, COMMIT or END.[1]

5.1.2. VoltDB Stored Procedures are Deterministic

To ensure data consistency and durability, VoltDB procedures must be deterministic. That is, given specific input values, the outcome of the procedure is consistent and predictable. Determinism is critical because it allows the same stored procedure to run in multiple locations and give the same results. It is determinism that makes it possible to run redundant copies of the database partitions without impacting performance. (See Chapter 10, *Availability* for more information on redundancy and availability.)

The following sections discuss three potential causes of non-determinism that you should avoid:

[1]One side effect of transactions being precompiled as stored procedures is that external transaction management frameworks, such as Spring or JEE, are not supported by VoltDB.

- Using system-specific functions such as system time or file or network I/O
- Misusing static variables
- Altering mutable parameters

5.1.2.1. Avoid Introducing Non-deterministic Values from External Functions

One key to deterministic behavior is avoiding calls within your stored procedures to external functions or procedures that can introduce arbitrary data. External functions include file and network I/O (which should be avoided any way because they can impact latency), as well as many common system-specific procedures such as Date and Time.

If a stored procedure *does* introduce arbitrary data and causes different results on different copies of a partition, VoltDB detects the mismatch, reports it as a potential source of data corruption, and shuts down all but one copy of each partition. By switching to reduced K-safety mode, VoltDB avoids the threat of data corruption due to non-determinism. However, it also means that the cluster is no longer K-safe; there is only one copy of each partition and any node failure will crash the database. So, although the database continues to operate after a mismatch, it is critically important you determine the cause of the non-deterministic behavior, correct the affected procedures, take a final snapshot, and restart the database to restore full K-safety.

However, this constraint does not mean you cannot use arbitrary data in VoltDB stored procedures. It just means you must either generate the arbitrary data before the stored procedure call and pass it in as input parameters or generate it in a deterministic way. For example, if you need to load a set of records from a file, you can open the file in your application and pass each row of data to a stored procedure that loads the data into the VoltDB database. This is the best method when retrieving arbitrary data from sources (such as files or network resources) that would impact latency.

The other alternative is to use data that can be generated deterministically. For two of the most common cases, timestamps and random values, VoltDB provides methods for this:

- `VoltProcedure.getTransactionTime()` returns a timestamp that can be used in place of the Java Date or Time classes.
- `VoltProcedure.getSeededRandomNumberGenerator()` returns a pseudo random number that can be used in place of the Java Util.Random class.

These procedures use the current transaction ID to generate a deterministic value for the timestamp and the random number. See the VoltDB Java Stored Procedure API for more.

5.1.2.2. Stored Procedures have no Persistence

Even seemingly harmless programming techniques, such as static variables can introduce nondeterministic behavior. VoltDB provides no guarantees concerning the state of the stored procedure class instance across invocations. Any information that you want to persist across invocations must either be stored in the database itself or passed into the stored procedure as a parameter.

5.1.2.3. Be Careful with Mutable Parameters

You can pass mutable parameters — most notably arrays — to stored procedures and those arrays can be used as parameters to SQL statements. To protect you against non-deterministic behavior from the contents of the mutable parameter being changed, VoltDB makes a copy of the array before passing it

to any SQL statements. If you call such procedures frequently with large arrays, the copy operation can consume significant amounts of memory, impacting your application.

The alternative, if you are sure that your procedures do not modify the mutable parameters, is to configure the database not to copy such parameters. You do this by setting the `deployment.systemsettings.procedure.copyparameters` property to "false". However, there is a significant risk associated with this setting. If you disable copying and a stored procedure *does* modify an array parameter, it can result in unpredictable behavior including run-time errors, database crashes, or even data corruption. So this feature should be used with extreme caution.

5.2. The Anatomy of a VoltDB Stored Procedure

You can write VoltDB stored procedures as Java classes. The following code sample illustrates the basic structure of a VoltDB java stored procedure.

```
import org.voltdb.*;

public class Procedure-name extends VoltProcedure {

                 // Declare SQL statements ...

    public datatype run ( arguments ) throws VoltAbortException {

                // Body of the Stored Procedure ...

    }
}
```

The key points to remember are to:

1. Import the VoltDB classes from `org.voltdb.*`
2. Include the class definition, which extends the abstract class `VoltProcedure`
3. Define the method `run()`, which performs the SQL queries and processing that make up the transaction

It is important to understand the details of how to design and develop stored procedures for your application as described in the following sections. However, for simple data access, the following techniques may suffice for some of your stored procedures:

- VoltDB defines default stored procedures to perform the most common table access such as inserting, selecting, updating, and deleting records based on a specific key value. See Section 7.1, “Using Default Procedures” for more.
- You can create stored procedures without writing any Java code by using the DDL statement CREATE PROCEDURE AS, where you define a single SQL query as a stored procedure. See Section 7.2, “Shortcut for Defining Simple Stored Procedures”.

The following sections describe the components of a stored procedure in more detail.

5.2.1. The Structure of the Stored Procedure

The stored procedures themselves are written as Java classes, each procedure being a separate class. Example 5.1, “Components of a VoltDB Java Stored Procedure” shows the stored procedure that looks up a flight to see if there are any available seats. The callouts identify the key components of a VoltDB stored procedure.

Example 5.1. Components of a VoltDB Java Stored Procedure

```
package fadvisor.procedures;

import org.voltdb.*;                                                    ❶

public class HowManySeats extends VoltProcedure {                      ❷

   public final SQLStmt GetSeatCount = new SQLStmt(                     ❸
        "SELECT NumberOfSeats, COUNT(ReserveID) " +
        "FROM Flight AS F, Reservation AS R " +
        "WHERE F.FlightID=R.FlightID AND R.FlightID=? " +
        "GROUP BY NumberOfSeats;");

    public long run( int flightid)                                      ❹
        throws VoltAbortException {

        long numofseats;
        long seatsinuse;
        VoltTable[] queryresults;

        voltQueueSQL( GetSeatCount, flightid);                          ❺

        queryresults = voltExecuteSQL();                                ❻

        VoltTable result = queryresults[0];                             ❼
        if (result.getRowCount() < 1) { return -1; }
        numofseats = result.fetchRow(0).getLong(0);                     ❽
        seatsinuse = result.fetchRow(0).getLong(1);
        numofseats = numofseats - seatsinuse;
        return numofseats; // Return available seats                    ❾
    }
}
```

❶ Stored procedures are written as Java classes. To access the VoltDB classes and methods, be sure to `import org.voltdb.*`.

Although VoltDB stored procedures must be written in Java and the primary client interface is Java (as described in Chapter 6, *Designing VoltDB Client Applications*), it is possible to write client applications using other programming languages. See Chapter 8, *Using VoltDB with Other Programming Languages* for more information on alternate client interfaces.

❷ Each stored procedure extends the generic class `VoltProcedure`.

❸ Within the stored procedure you access the database using ANSI-standard SQL statements. To do this, you declare the statement as a special Java type called `SQLStmt`, which must be declared as *final*.

In the SQL statement, you insert a question mark (?) everywhere you want to replace a value by a variable at runtime. In this example, the query GetSeatCount has one input variable, FlightID. (See Appendix B, *Supported SQL Statements* for details on the supported SQL statements.)

To ensure the stored procedure code is single partitioned, queries must filter on the partitioning column for a single value (using equal, =). Filtering for a range of values will not be single-partitioned because the code will have to look up in all the partitions to ensure the entire range is found.

❹ The bulk of the stored procedure is the `run()` method, whose input specifies the input arguments for the stored procedure. See Section 5.2.2, "Passing Arguments to a Stored Procedure" next for details.

Note that the `run()` method throws the exception `VoltAbortException` if any exceptions are not caught. `VoltAbortException` causes the stored procedure transaction to rollback. (See Section 5.2.6, "Rolling Back a Transaction" for more information about rollback.)

❺ To perform database queries, you queue SQL statements, specifying both the SQL statement and the variables it requires, using the `voltQueueSQL()` method. More details are described in Section 5.2.3, "Creating and Executing SQL Queries in Stored Procedures".

❻ After you queue all of the SQL statements you want to perform, use `voltExecuteSQL()` to execute the statements in the queue.

❼ Each statement returns its results in a `VoltTable` structure. Because the queue can contain multiple queries, `voltExecuteSQL()` returns an array of `VoltTable` structures, one array element for each query. More details are described in Section 5.2.4, "Interpreting the Results of SQL Queries".

❽ In addition to queueing and executing queries, stored procedures can contain custom code. However, you should limit the amount of custom code in stored procedures to only that processing that is necessary to complete the transaction, so as not to delay subsequent transactions.

❾ Stored procedures can return a long integer, a `VoltTable` structure, or an array of `VoltTable` structures. For more details, see Section 5.2.5, "Returning Results from a Stored Procedure".

5.2.2. Passing Arguments to a Stored Procedure

You specify the number and type of the arguments that the stored procedure accepts in the `run()` method. For example, the following is the declaration of the `run()` method for an `Initialize()` stored procedure from the voter sample application found in the `/doc/tutorials/voter` folder where Volt is installed. This procedure accepts two arguments: an integer and a string.

```
public long run(int maxContestants, String contestants) { . . .
```

VoltDB stored procedures can accept parameters of any of the following Java and VoltDB datatypes:

Integer types	byte, short, int, long, Byte, Short, Integer, and Long
Floating point types	float, double, Float, Double
Fixed decimal types	BigDecimal
String and binary types	String and byte[]
Timestamp and Date types	org.voltdb.types.TimestampType, org.voltdb.types.DateType java.util.Date, java.sql.Date, java.sql.Timestamp
Boolean type	org.voltdb.types.BooleanType
VoltDB type	VoltTable

The arguments can be scalar objects or arrays of any of the preceding types. For example, the following `run()` method defines three arguments: a scalar long and two arrays, one array of timestamps and one array of Strings:

```
import org.voltdb.*;
public class LogMessagesByEvent extends VoltProcedure {

     public long run (
           long eventType,
          org.voltdb.types.TimestampType[] eventTimeStamps,
          String[] eventMessages
    ) throws VoltAbortException {
```

The calling client application can use any of the preceding datatypes when invoking the `callProcedure()` method and, where necessary, VoltDB makes the appropriate type conversions (for example, from int to String or from String to Double). See Section 6.2, “Invoking Stored Procedures” for more on using the `callProcedure()` method.

5.2.3. Creating and Executing SQL Queries in Stored Procedures

The main function of the stored procedure is to perform database queries. In VoltDB this is done in two steps:

1. Queue the queries using the `voltQueueSQL()` function
2. Execute the queue and return the results using the `voltExecuteSQL()` function

Queuing SQL Statements The first argument to `voltQueueSQL()` is the SQL statement to be executed. The SQL statement is declared using a special class, `SQLStmt`, with question marks as placeholders for values that will be inserted at runtime.

The SQL statements must be declared as `final` and initialized at compile time, either when declared or within a constructor or static initializer. This allows the VoltDB planner to determine the optimal execution plan for each statement when the procedure is loaded and declared in the schema. To allow for code reuse, SQLStmt objects can be inherited from parent classes or constructed from other compile-time constants.

The remaining arguments to v`oltQueueSQL()` are the actual values that VoltDB inserts into the placeholders. For example, if you want to perform a SELECT of a table using two columns in the WHERE clause, your SQL statement might look something like this:

```
SELECT CustomerID FROM Customer WHERE FirstName=? AND LastName=?;
```

At runtime, you want the questions marks replaced by values passed in as arguments from the calling application. So the actual `voltQueueSQL()` invocation might look like this:

```
public final SQLStmt getcustid = new SQLStmt(
                                  "SELECT CustomerID FROM Customer " +
                                  "WHERE FirstName=? AND LastName=?;");

     ...

voltQueueSQL(getcustid, firstnm, lastnm);
```

Your stored procedure can call `voltQueueSQL()` more than once to queue up multiple SQL statements before they are executed. Queuing multiple SQL statements improves performance when the SQL queries execute because it minimizes the amount of network traffic within the cluster. Once you have queued all of the SQL statements you want to execute together, you then process the queue using the `voltExecuteSQL()` function.

```
VoltTable[] queryresults = voltExecuteSQL();
```

Cycles of Queue and Execute

Your procedure can queue and execute SQL statements in as many cycles as necessary to complete the transaction. For example, if you want to make a flight reservation, you may need to access the database and verify that the flight exists before creating the reservation in the database. One way to do this is to look up the flight, verify that a valid row was returned, then insert the reservation, like so:

Example 5.2. Cycles of Queue and Execute in a Stored Procedure

```
final String getflight = "SELECT FlightID FROM Flight WHERE FlightID=?;";     ❶
final String makeres = "INSERT INTO Reservation (?,?,?,?,?);";

public final SQLStmt getflightsql = new SQLStmt(getflight);
public final SQLStmt makeressql = new SQLStmt(makeres);

public VoltTable[] run( int reservenum, int flightnum, int customernum )      ❷
        throws VoltAbortException {

        // Verify flight ID
  voltQueueSQL(getflightsql, flightnum);                                      ❸
  VoltTable[] queryresults = voltExecuteSQL();

        // If there is no matching record, rollback
  if (queryresults[0].getRowCount() == 0 ) throw new VoltAbortException();    ❹

        // Make reservation
  voltQueueSQL(makeressql, reservenum, flightnum, customernum,0,0);           ❺
  return voltExecuteSQL();
}
```

This stored procedure code to make a reservation is described as follows:

❶ Define the SQL statements to use. The *getflight* string contains an SQL statement that verifies the flight ID, and the *makeres* string contains the SQL statement that makes the reservation.

❷ Define the `run()` method for the stored procedure. This stored procedure takes as input arguments the reservation number, the flight number, and the customer number.

❸ Queue and execute an SQL statement. In this example the `voltExecuteSQL()` method processes the single `getflightsql()` function, which executes the SQL statement specified in the *getflight* string.

❹ Process results. If the flight is not available, the exception `VoltAbortException` aborts the stored procedure and rolls back the transaction.

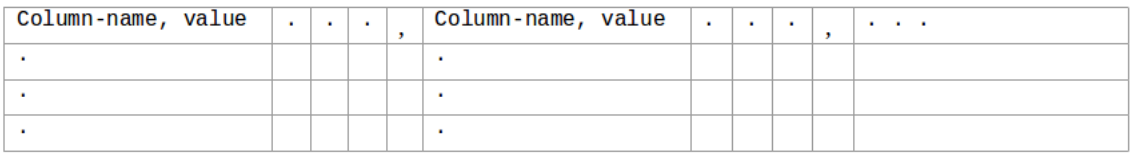
❺ The second SQL statement to make the reservation is then queued and executed. The `voltExecuteSQL()` method processes the single `makeressql()` function, which executes the SQL statement specified in the *makeres* string.

5.2.4. Interpreting the Results of SQL Queries

With the `voltExecuteSQL()` call, the results of all the queued SQL statements are returned in an array of `VoltTable` structures. The array contains one `VoltTable` for each SQL statement in the queue. The `VoltTable` structures are returned in the same order as the respective SQL statements in the queue.

The `VoltTable` itself consists of rows, where each row contains columns, and each column has the column name and a value of a fixed datatype. The number of rows and columns per row depends on the specific query.

Figure 5.1. Array of VoltTable Structures



For example, if you queue two SQL SELECT statements, one looking for the destination of a specific flight and the second looking up the ReserveID and Customer name (first and last) of reservations for that flight, the code for the stored procedure might look like the following:

```
public final SQLStmt getdestsql = new SQLStmt(
              "SELECT Destination FROM Flight WHERE FlightID=?;");
public final SQLStmt getressql = new SQLStmt(
             "SELECT r.ReserveID, c.FirstName, c.LastName " +
             "FROM Reservation AS r, Customer AS c " +
             "WHERE r.FlightID=? AND r.CustomerID=c.CustomerID;");

         ...

   voltQueueSQL(getdestsql,flightnum);
   voltQueueSQL(getressql,flightnum);
   VoltTable[] results = voltExecuteSQL();
```

The array returned by `voltExecuteSQL()` will have two elements:

- The first array element is a `VoltTable` with one row (FlightID is defined as unique) containing one column, because the SELECT statement returns only one value.
- The second array element is a `VoltTable` with as many rows as there are reservations for the specific flight, each row containing three columns: ReserveID, FirstName, and LastName.

Assuming the stored procedure call input was a FlightID value of 134, the data returned for the second array element might be represented as follows:

Figure 5.2. One VoltTable Structure is returned for each Queued SQL Statement

FlightID, 134	,	ReserveID, 4747	FirstName, Will	LastName, Poger
		ReserveID, 9879	FirstName, Janice	LastName, Josly
		ReserveID, 3456	FirstName, Holly	LastName, Eagan
		ReserveID, 1098	FirstName, Ralph	LastName, Finess

VoltDB provides a set of convenience methods for accessing the contents of the `VoltTable` array. Table 5.1, “Methods of the VoltTable Classes” lists some of the most common methods. (See also Java Stored Procedure API.)

Table 5.1. Methods of the VoltTable Classes

Method	Description
int fetchRow(int index)	Returns an instance of the VoltTableRow class for the row specified by index.
int getRowCount()	Returns the number of rows in the table.
int getColumnCount()	Returns the number of columns for each row in the table.
Type getColumnType(int index)	Returns the datatype of the column at the specified index. Type is an enumerated type with the following possible values: BIGINT BOOLEAN DATE DECIMAL FLOAT GEOGRAPHY GEOGRAPHY_POINT INTEGER INVALID NULL NUMERIC SMALLINT STRING TIMESTAMP TINYINT VARBINARY VOLTTABLE
String getColumnName(int index)	Returns the name of the column at the specified index.
double getDouble(int index) long getLong(int index) String getString(int index) BigDecimal getDecimalAsBigDecimal(int index) double getDecimalAsDouble(int index) Date getTimestampAsTimestamp(int index) long getTimestampAsLong(int index) Date getDateAsDate(int index) Date getDateAsLocalDate(int index) Date getDateAsSqlDate(int index) Date getDateAsUtilDate(int index) boolean getBoolean(int index) byte[] getVarbinary(int index)	Methods of VoltTable.Row Return the value of the column at the specified index in the appropriate datatype. Because the datatype of the columns vary depending on the SQL query, there is no generic method for returning the value. You must specify what datatype to use when fetching the value.

It is also possible to retrieve the column values by name. You can invoke any of the getDatatype() methods and pass a string argument specifying the name of the column, rather than the numeric index. Accessing the columns by name can make code easier to read and less susceptible to errors due to changes in the SQL schema (such as changing the order of the columns). On the other hand, accessing column values by numeric index is potentially more efficient under heavy load conditions.

Example 5.3, “Displaying the Contents of VoltTable Arrays” shows a generic routine for “walking” through the return results of a stored procedure. In this example, the contents of the `VoltTable` array are written to standard output.

Example 5.3. Displaying the Contents of VoltTable Arrays

```
public void displayResults(VoltTable[] results) {
  int table = 1;
     for (VoltTable result : results) {
      System.out.printf("*** Table %d ***\n",table++);
      displayTable(result);
   }
}

public void displayTable(VoltTable t) {

   final int colCount = t.getColumnCount();
   int rowCount = 1;
   t.resetRowPosition();
   while (t.advanceRow()) {
      System.out.printf("--- Row %d ---\n",rowCount++);

      for (int col=0; col<colCount; col++) {
         System.out.printf("%s: ",t.getColumnName(col));
         switch(t.getColumnType(col)) {
            case TINYINT: case SMALLINT: case BIGINT: case INTEGER:
               System.out.printf("%d\n", t.getLong(col));
               break;
            case STRING:
               System.out.printf("%s\n", t.getString(col));
               break;
            case DECIMAL:
               System.out.printf("%f\n", t.getDecimalAsBigDecimal(col));
               break;
            case FLOAT:
               System.out.printf("%f\n", t.getDouble(col));
               break;
         }
      }
   }
}
```

For further details on interpreting the VoltTable structure, see the Java documentation that is provided online in the `doc/` subfolder for your VoltDB installation.

5.2.5. Returning Results from a Stored Procedure

Stored procedures can return the following types:

- Long integer
- Single VoltTable
- Array of VoltTable structures

You can return all of the query results by returning the `VoltTable` array, or you can return a scalar value that is the logical result of the transaction. (For example, the stored procedure in Example 5.1, "Components of a VoltDB Java Stored Procedure" returns a long integer representing the number of remaining seats available in the flight.)

Whatever value the stored procedure returns, make sure the `run()` method includes the appropriate datatype in its definition. For example, the following two definitions specify different return datatypes; the first returns a long integer and the second returns the results of a SQL query as a `VoltTable` array.

```
public long run( int flightid)

public VoltTable[] run ( String lastname, String firstname)
```

Note that you can interpret the results of SQL queries either in the stored procedure or in the client application. However, for performance reasons, it is best to limit the amount of additional processing done by the stored procedure to ensure it executes quickly and frees the queue for the next stored procedure. So unless the processing is necessary for subsequent SQL queries, it is usually best to return the query results (in other words, the `VoltTable` array) directly to the calling application and interpret them there.

5.2.6. Rolling Back a Transaction

Finally, if a problem arises while a stored procedure is executing, whether the problem is anticipated or unexpected, it is important that the transaction rolls back. *Rollback* means that any changes made during the transaction are undone and the database is left in the same state it was in before the transaction started.

VoltDB is a fully transactional database, which means that if a transaction (stored procedure) fails, the transaction is automatically rolled back and the appropriate exception is returned to the calling application. Exceptions that can cause a rollback include the following:

- Runtime errors in the stored procedure code, such as division by zero or datatype overflow.
- Violating database constraints in SQL queries, such as inserting a duplicate value into a column defined as unique.

The atomicity of the stored procedure depends on VoltDB being able to roll back incomplete database changes. VoltDB relies on Java exception handling outside the stored procedure to perform the roll back. Therefore, you should not attempt to alter any exceptions thrown by the voltExecuteSql method. If your procedure code *does* catch exceptions thrown as a result of executing SQL statements, make sure that the exception handler re-throws the exception to allow VoltDB to perform the necessary roll back activities before the stored procedure returns to the calling program.

On the other hand, there may be situations where an exception occurs in the program logic. The issue might not be one that is caught by Java or VoltDB, but still there is no practical way for the transaction logic to complete. In these situations, you can force a rollback by explicitly throwing the `VoltAbortException` exception. For example, if a flight ID does not exist, you do not want to create a reservation so the stored procedure can force a rollback like so:

```
if (!flightid) { throw new VoltAbortException(); }
```

See Section 7.3, "Verifying Expected Query Results" for another way to roll back procedures when queries do not meet necessary conditions.

5.3. Installing Stored Procedures into the Database

When your stored procedure code is ready, you need to get the procedures into the database and ready to use. You first compile the procedure code, create a jar file, and load the resulting jar file into the database.

Then you need to declare in the schema which procedures are stored procedures. Finally, depending on which table each stored procedure accesses, you need to partition each procedure to match the table partitioning. These processes are covered in the following sections:

- Compiling, Packaging, and Loading Stored Procedures
- Declaring Stored Procedures in the Schema
- Partitioning Stored Procedures in the Schema

These sections show how to use DDL to declare and partition stored procedures in the database schema. If you find you need to modify the schema, see Section 4.6, “Modifying the Schema”.

5.3.1. Compiling, Packaging, and Loading Stored Procedures

The VoltDB stored procedures are written as Java classes, so you compile them using the Java compiler. Anytime you update your stored procedure code, remember to recompile, package, and reload it into the database using the following steps:

```
$ javac -classpath "./:/opt/voltdb/voltdb/*" \          ❶
        -d ./obj  \
         *.java
$ jar  cvf  myproc.jar -C obj .                         ❷
$ sqlcmd                                                ❸
1> load classes myproc.jar;
2> show classes;
```

The steps are described as follows:

❶ Use the **javac** command to compile the procedure Java code.

You include libraries by using the `-classpath` argument on the command line or by defining the environment variable CLASSPATH. You must include the VoltDB libraries in the classpath so Java can resolve references to the VoltDB classes and methods. This example assumes that the VoltDB software has been installed in the folder `/opt/voltdb`. If you installed VoltDB in a different directory, you need to include your installation path. Also, if your client application depends on other libraries, they need to be included in the classpath as well.

Use the `-d` flag to specify an output directory in which to create the resulting class files.

❷ Use the **jar** command to package your Java classes into a Java archive, or JAR file.

The JAR file must have the same Java package structure as the classes in the JAR file. For example, if a class has a structure such as `myapp.procedures.ProcedureFoo`, then the JAR file has to have `myapp/procedures/ProcedureFoo.class` as the class structure for this file.

The JAR file must include any inner classes or other dependent classes used by the stored procedures. It can also be used to load any resource files, such as XML or other data files, that the procedures need. Any additional resources in the JAR file are loaded into the server as long as they are in a subfolder. (Resources in the root directory of the JAR file are ignored.)

❸ Use the sqlcmd **load classes** directive to load the stored procedure classes into the database.

You can use the **show classes** command to display information about the classes installed in the cluster.

Before a stored procedure can be called by a client application, you need to declare it in the schema, which is described next.

5.3.2. Declaring Stored Procedures in the Schema

To make your stored procedures accessible in the database, you must declare them in the schema using the CREATE PROCEDURE statement. Be sure to identify all of your stored procedures or they will not be available to the client applications at runtime. Also, before you declare a procedure, ensure the tables and columns the procedure accesses are in the schema.

The following DDL statements declare five stored procedures, identifying them by their class name:

```
$ sqlcmd
1> CREATE PROCEDURE FROM CLASS fadvisor.procedures.LookupFlight;
2> CREATE PROCEDURE FROM CLASS fadvisor.procedures.HowManySeats;
3> CREATE PROCEDURE FROM CLASS fadvisor.procedures.MakeReservation;
4> CREATE PROCEDURE FROM CLASS fadvisor.procedures.CancelReservation;
5> CREATE PROCEDURE FROM CLASS fadvisor.procedures.RemoveFlight;
```

For some situations, you can create stored procedures directly in the schema using SQL instead of loading Java code. See how to use the CREATE PROCEDURE AS statement in Section 7.2, “Shortcut for Defining Simple Stored Procedures”.

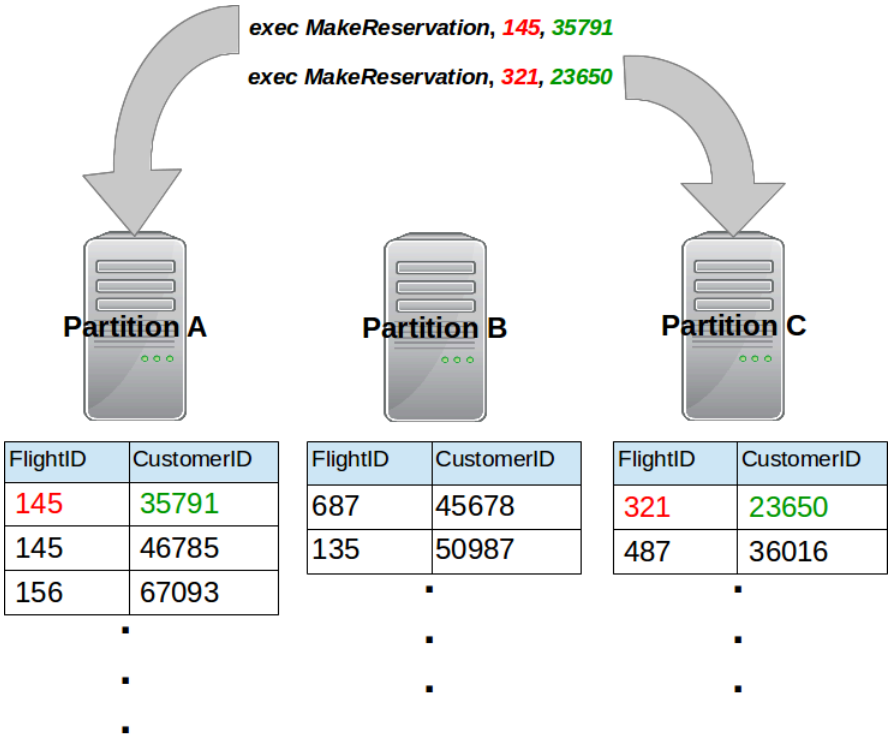
For more about modifying a schema with DDL, see Section 4.6, “Modifying the Schema”.

5.3.3. Partitioning Stored Procedures in the Schema

We want the most frequently used stored procedures to be single-partitioned. This means that the procedure executes in the one partition that also has the data it needs. Single-partitioned stored procedures do not have the overhead of processing across multiple partitions and servers, wasting time searching through the data of the entire table. To ensure single-partitioned efficiency, the parameter the stored procedure uses to identify its required data must be the same as the column on which the table rows are partitioned.

Remember that in our sample application the RESERVATION table is partitioned on FLIGHTID. Let's say you create a stored procedure, *MakeReservation()*, with two arguments, *flight_id* and *customer_id*. The following figure shows how the stored procedure will automatically execute in the partition that has the requested row.

Figure 5.3. Stored Procedures Execute in the Appropriate Partition Based on the Partitioned Parameter Value



If you do not declare a procedure as single-partitioned, it is assumed to be multi-partitioned by default. The advantage of multi-partitioned stored procedures is that they have full access to all of the data in the database, across all partitions. However, the real focus of VoltDB, and the way to achieve maximum throughput for your application, is through the use of single-partitioned stored procedures.

5.3.3.1. How to Declare Single-Partition Procedures

Before declaring a single-partitioned procedure, ensure the following prerequisites:

1. The table that the stored procedure accesses has been partitioned in the schema. See Section 4.4, “Partitioning Database Tables”.

2. If the procedure is implemented with Java code, it is loaded into the database. See Section 5.3.1, “Compiling, Packaging, and Loading Stored Procedures”.

When you declare a stored procedure as single-partitioned, you must specify both the associated table and the column on which it is partitioned using the PARTITION ON clause in the CREATE PROCEDURE statement. The following example uses the RESERVATION table and the FLIGHTID column as the partitioning column. For example:

```
CREATE PROCEDURE
   PARTITION ON
      TABLE Reservation COLUMN FlightID
   FROM CLASS fadvisor.procedures.MakeReservation;
```

The PARTITION ON clause assumes that the partitioning column value is also the first parameter to the stored procedure. Suppose you wish to partition a stored procedure on the third parameter such as the

procedure `GetCustomerDetails()`, where the third parameter is a customer_id. You must specify the partitioning parameter using the PARAMETER clause and an index for the parameter position. The index is zero-based so the third parameter would be "2" and the CREATE PROCEDURE statement would be as follows:

```
CREATE PROCEDURE
    PARTITION ON
       TABLE Customer COLUMN CustomerID PARAMETER 2
    FROM CLASS fadvisor.procedures. GetCustomerDetails;
```

5.3.3.2. Queries in Single-Partitioned Stored Procedures

Single-partitioned stored procedures are special because they operate independently of other partitions, which is why they are so fast. At the same time, single-partitioned stored procedures operate on only a subset of the entire data, that is, only the data within the specified partition.

Caution

It is the application developer's responsibility to ensure that the queries in a single-partitioned stored procedure are truly single-partitioned. VoltDB *does not* warn you about SELECT or DELETE statements that might return incomplete results. For example, if your single-partitioned procedure attempts to operate on a range of values for the partitioning column, the range is incomplete and includes only a subset of the table data that is in the current partition.

VoltDB does generate a runtime error if you attempt to INSERT a row that does not belong in the current partition.

After you partition a procedure, your stored procedure can operate on only those records in the partitioned table that are identified by the partitioning column, in this example the RESERVATION table identified by a FLIGHTID. Your stored procedure can access records in replicated tables because the entire table is available to every partition. However, for other partitioned tables, the stored procedure can only operate on those records *if both tables are partitioned on the same attribute*. In this example that would be FLIGHTID.

In other words, the following rules apply:

- Any SELECT, UPDATE, or DELETE queries must use the constraint, `WHERE identifier=?`

  The question mark is replaced at runtime by the input value that identifies the row of data in the table. In our example, queries on the RESERVATION table must use the constraint, `WHERE FLIGHTID=?`

- SELECT statements can join the partitioned table to replicated tables, as long as the preceding WHERE constraint is also applied.

- SELECT statements can join the partitioned table to other partitioned tables as long as the following are true:

  - The two tables are partitioned on the same attribute or column (in our example, FLIGHTID).

  - The tables are joined on the shared partitioning column.

  - The following WHERE constraint is also used: `WHERE partitioned-table. identifier=?` In this example, `WHERE RESERVATION.FLIGHTID=?`

For example, the RESERVATION table can be joined with the FLIGHT table (which is replicated). However, the RESERVATION table *cannot* be joined with the CUSTOMER table in a single-partitioned stored

procedure because the two tables use different partitioning columns. (CUSTOMER is partitioned on the CUSTOMERID column.)

The following are examples of invalid SQL queries for a single-partitioned stored procedure partitioned on FLIGHTID:

- **INVALID:** `SELECT * FROM reservation WHERE reservationid=?`

  The RESERVATION table is being constrained by a column (RESERVATIONID) which is not the partitioning column.

- **INVALID:** `SELECT c.lastname FROM reservation AS r, customer AS c WHERE r.flightid=? AND c.customerid = r.customerid`

  The correct partitioning column is being used in the WHERE clause, but the tables are being joined on a different column. As a result, not all CUSTOMER rows are available to the stored procedure since the CUSTOMER table is partitioned on a different column than RESERVATION.

Chapter 6. Designing VoltDB Client Applications

After you design and partition your database schema (Chapter 4, *Designing the Database Schema*), and after you design the necessary stored procedures (Chapter 5, *Designing Stored Procedures to Access the Database*), you are ready to write the client application logic. The client code contains all the business-specific logic required for the application, including business rule validation and keeping track of constraints such as proper data ranges for arguments entered in stored procedure calls.

The three steps to using VoltDB from a client application are:

1. Creating a connection to the database
2. Calling stored procedures
3. Closing the client connection

The following sections explain how to perform these functions using the latest VoltDB Java client interface, known as `Client2`. (See VoltDB Java Client API.) The VoltDB Java Client is a thread-safe class library that provides runtime access to VoltDB databases and functions.

It is possible to call VoltDB stored procedures from programming languages other than Java. However, reading this chapter is still recommended to understand the process for invoking and interpreting the results of a VoltDB stored procedure. See Chapter 8, *Using VoltDB with Other Programming Languages* for more information about using VoltDB from client applications written in other languages.

Client2 uses modern Java programming techniques and smarter defaults to provide a structured and extensible interface, including:

- A "builder" pattern for the Client2Config, with individual methods that can be chained to set configuration options.
- No mandatory settings, all attributes have default values.
- Smarter default behavior to simplify application development.
- Individually configured notifications, set as callbacks on the configuration object, that let the application decide which events it wants to be notified about and how.
- Use of Java CompletableFutures to handle asynchronous procedure calls, giving the application developer more structured and precise handling of invocation events.

6.1. Connecting to the VoltDB Database

The first task for the calling program is to create a connection to the VoltDB database. You do this with the following steps:

```
Client2 client = null;
Client2Config config = null;
try {
    config = new Client2Config().username("advent").password("xyzzy");  ❶

    client = ClientFactory.createClient(config);                        ❷
```

```
    client.connectSync("myserver.xyz.net");                    ❸
} catch (java.io.IOException ex) {
    ex.printStackTrace();
    System.exit(-1);
}
```

❶ Define the configuration for your connections. The `Client2Config` class uses the "builder" pattern to set up up the configuration for all connections from a single client. The configuration object is always required, but the attributes you need to set can vary. As shown here, the `username` and `password` attributes are set. It is also possible to define additional characteristics of the client connections as part of the configuration, such as the timeout period for procedure invocations, or handlers for various notifications. (See Section 6.6, “Handling Errors”.)

❷ Create an instance of the VoltDB `Client2` class.

❸ Call the `connectSync()` method. After you instantiate your client object, the argument to `connectSync()` specifies the database node to connect to. You can specify the server node as a hostname (as in the preceding example) or as an IP address. You can also add a second argument if you want to connect to a port other than the default. For example, the following `connectSync()` call attempts to connect to the admin port, 21211:

```
client.connectSync("myserver.xyz.net",21211);
```

If security is enabled, and the username and password in the `Client2Config()` calls do not match a user defined in the configuration, the call to `connectSync()` will throw an exception. See Chapter 12, *Security* for more information about the use of security with VoltDB databases.

When you are done with the connection, you should make sure your application calls the `close()` method to clean up any memory allocated for the connections. See Section 6.5, “Closing the Connection”.

6.1.1. How Connections To The Cluster Work

A call to `connectSync()` sets up a connection to a single node in the database cluster before it returns. The VoltDB client software (by default) then automatically establishes connections to all available cluster nodes.

This automatic management of connections has these major benefits:

- Multiple connections distribute the stored procedure requests around the cluster, avoiding a bottleneck where all requests are queued through a single host. This is particularly important when using asynchronous procedure calls or multiple clients.
- For Java applications, the VoltDB Java client library uses client affinity. That is, the client knows which server to send each request to based on the partitioning, thereby eliminating unnecessary network hops.
- Finally, if a server fails for any reason, when using K-safety the client can continue to submit requests through connections to the remaining nodes. This avoids a single point of failure between client and database cluster. See Chapter 10, *Availability* for more.
- The client will automatically reconnect whenever the topology changes. That is, if a server fails and then rejoins the cluster, or new nodes are added to the cluster, the client will automatically create connections to the newly available servers.

6.1.2. Specifying Multiple Servers

You can specify more than one hostname or IP address in the `connectSync` call. The VoltDB client software will then connect to the first available host in the list and then return to your application code. From that point, the automatic management of connections will take over. The advantage of specifying

more than one host is that it builds in some resiliency against temporary failure of some hosts. The `connectSync()` call will only fail in the case that all of the given hosts are unavailable.

For example, the following Java code creates the client object specifying three alternative nodes for the initial connection to the cluster. In this case, security is not enabled, so username and password are not needed:

```
try {
    client = ClientFactory.createClient(new Client2Config());
    client.connectSync("server1.xyz.net,server2.xyz.net,server3.xyz.net");
} catch (java.io.IOException ex) {
    ex.printStackTrace();
    System.exit(-1);
}
```

6.2. Invoking Stored Procedures

After your client creates the connection to the database, it is ready to call the stored procedures. You invoke a stored procedure using the `callProcedureSync()` method, passing the procedure name and variables as arguments. For example:

```
VoltTable[] results;

try {
    results = client.callProcedureSync("LookupFlight",                ❶
                                       origin,
                                       dest,
                                       departtime).getResults();      ❷
} catch (Exception ex) {                                              ❸
     ex.printStackTrace();
     System.exit(-1);
}
```

❶ The `callProcedureSync()` method takes the procedure name and the procedure's variables as arguments. The `LookupFlight()` stored procedure requires three variables: the originating airport, the destination, and the departure time.

❷ Once a synchronous call completes, you can evaluate the results of the stored procedure. The `callProcedureSync()` method returns a `ClientResponse` object, which includes information about the success or failure of the stored procedure. To retrieve the actual return values you use the `getResults()` method. See Section 5.2.4, “Interpreting the Results of SQL Queries” for more information about interpreting the results of VoltDB stored procedures.

❸ Note that since `callProcedureSync()` can throw an exception (such as `ProcCallException`) it is a good practice to perform error handling and catch known exceptions.

6.3. Invoking Stored Procedures Asynchronously

Calling stored procedures synchronously simplifies the program logic because your client application waits for the procedure to complete before continuing. However, for high performance applications looking to maximize throughput, it is better to queue stored procedure invocations asynchronously.

Asynchronous Invocation

To invoke stored procedures asynchronously, use the `callProcedureAsync()` method. The procedure call will be queued internally for transmission. The immediate return value is a standard Java `Com-`

pletableFuture object, which will be "completed" when the procedure completes (or an error occurs). For example, to invoke a NewCustomer() stored procedure asynchronously, the call to callProcedureAsync() might look like the following:

```
CompletableFuture<ClientResponse> future =
    client.callProcedureAsync("NewCustomer",
                              firstname,
                              lastname,
                              custID};
```

To handle the eventual ClientResponse, you can use any of the features of CompletableFuture that Java provides. These include awaiting completion with get(), declaring a handler for the eventual completion with handle(), and so on.

The following are other important points to note when making asynchronous invocations of stored procedures:

- Calls to callProcedureAsync() return control to the calling application as soon as the procedure call is locally queued.
- Errors that occur before the procedure call is queued may be reported via a Java exception. The calling application should include appropriate handling.
- Once the procedure is queued, any subsequent error (such as an exception in the stored procedure itself or loss of connection to the database) is returned via the CompletableFuture.
- There is a limit on the local queue size, after which calls will be rejected. The default queue size is 1000 calls, but this can be changed with Client2Config. Robust applications should either ensure they can never exceed the local queue size, or implement appropriate handling. You can configure a handler to receive notifications when the queue is approaching capacity; see the requestBackpressureHandler() method of Client2Config.
- If the database server queue is full, transmission is temporarily suspended. This condition, known as network backpressure (distinct from request-queue backpressure), is handled internally to the Client2 API. This situation does not normally happen unless the database cluster is not scaled sufficiently for the workload, or there are abnormal spikes in the workload. See Section 6.6.3, “Writing Handlers to Interpret Other Errors” for more information.

Using CompletableFuture with Asynchronous Calls

The CompletableFuture is "completed normally" when the called procedure has been executed on the cluster, and has returned a response. It can also be completed, in this case "exceptionally", when an error or timeout occurs.

Normal completion allows access to a ClientResponse structure, the same structure that is returned in a synchronous invocation. The ClientResponse contains information about the results of execution. In particular, the methods getStatus() and getResults() let you determine whether the stored procedure was successful and evaluate the results of the procedure.

Exceptional completion does not have a ClientResponse structure; the exception itself conveys the error information.

Completions themselves can be processed synchronously or asynchronously, using the standard facilities of CompletableFuture. The VoltDB Java client is single threaded, so synchronous completions are processed one at a time. Consequently, it is good practice to keep synchronous completion processing to a

minimum, returning control to the main thread as soon as possible. If more complex processing is required, use one of the available methods for handling the completion asynchronously.

6.4. Defining Options on a Per-Call Basis

Some invocation attributes defined in the configuration by Client2Config can be overridden on a per-call basis by means of an *options* object. You create a new option object with Client2CallOptions, then apply it to a procedure call as the first argument. For example, the following code sets a call-specific timeout of 1,234 milliseconds and a priority of 3:

```
Client2CallOptions opts = new Client2CallOptions()
    .clientTimeout(1234, TimeUnit.MILLISECONDS)
    .requestPriority(3);
addUserFuture = callProcedureAsync(opts, "AddUser", username, password);
```

Options that are not specified in Client2CallOptions are defined by the configuration settings (or their defaults), so the interface is extensible without forcing application code changes.

6.5. Closing the Connection

When the client application is done interacting with the VoltDB database, it is a good practice to close the connection. This ensures that any pending transactions are completed in an orderly way. The following example demonstrates how to close the client connection:

```
try {
    client.drain();
    client.close();
} catch (InterruptedException ex) {
    ex.printStackTrace();
}
```

There are two steps to closing the connection:

1. Call `drain()` to make sure all asynchronous calls have completed. The `drain()` method pauses the current thread until all outstanding asynchronous calls (and their callback procedures) complete. This call is not necessary if the application only makes synchronous procedure calls. However, there is no penalty for calling `drain()` and so it can be included for completeness in all applications.

2. Call `close()` to close all of the connections and release any resources associated with the client.

6.6. Handling Errors

A special situation to consider when calling VoltDB stored procedures is error handling. The VoltDB client interface catches most exceptions, including connection errors, errors thrown by the stored procedures themselves, and even exceptions that occur in asynchronous callbacks. These error conditions are not returned to the client application as exceptions. However, the application can still receive notification and interpret these conditions using the client interface.

The following sections explain how to identify and interpret errors that occur when executing stored procedures and in asynchronous callbacks. These include:

- Interpreting Execution Errors

- Handling Timeouts
- Writing Handlers to Interpret Other Errors

6.6.1. Interpreting Execution Errors

If an error occurs in a stored procedure (such as an SQL constraint violation), VoltDB catches the error and returns information about it to the calling application as part of the `ClientResponse` class. The `ClientResponse` class provides several methods to help the calling application determine whether the stored procedure completed successfully and, if not, what caused the failure. The two most important methods are `getStatus()` and `getStatusString()`.

If you are using synchronous procedure calls in your application, a response whose status shows a failure will be turned into a `ProcCallException`. Call the exception's `getClientResponse()` method to retrieve the `ClientResponse`.

If you are using asynchronous procedure calls in your application, the response is always obtained from the completed `CompletableFuture`.

For both synchronous and asynchronous calls, errors that occur locally in the Client2 library may result in an exception being thrown, for example an `IOException`. You should establish appropriate handlers so that your application is robust.

```
callProcedureAsync("MyProc", "someArg")
        .thenAccept((resp) -> processResponse(resp))       ❶
        .exceptionally((th) -> processException(th));
          ...

void processResponse(ClientResponse resp) {
    final byte AppCodeWarm = 1;
    final byte AppCodeFuzzy = 2;
    if (resp.getStatus() != ClientResponse.SUCCESS) {      ❷
        System.err.println(resp.getStatusString());        ❸
    } else {
        if (resp.getAppStatus() == AppCodeFuzzy) {         ❹
           System.err.println(resp.getAppStatusString());
        }
        myEvaluateResultsProc(resp.getResults());
    }
}

void processException(Throwable th) {
    System.err.println("Exception: " + th);
}
```

❶ `thenAccept` and `exceptionally` are just two examples of the many ways of establishing handlers that will be executed when the asynchronous procedure call completes. See the Java documentation for `CompletableFuture`.

❷ The `getStatus()` method tells you whether the stored procedure completed successfully and, if not, what type of error occurred. It is good practice to always check the status of the `ClientResponse` before evaluating the results of a procedure call, because if the status is anything but SUCCESS, there will not be any results returned. Some possible values of `getStatus()` are:

- **CLIENT_REQUEST_TIMEOUT** — The request timed out in the client before it could be sent to the cluster.

- **CLIENT_RESPONSE_TIMEOUT** — The request timed out in the client after it had been sent to the cluster. The stored procedure may or may not have been executed. It may or may not have completed execution. See Section 6.6.2, "Handling Timeouts" for more information about handling this condition.

- **CONNECTION_LOST** — The network connection was lost before the stored procedure returned status information to the calling application. The stored procedure may or may not have been executed. It may or may not have completed execution.

- **GRACEFUL_FAILURE** — An error occurred and the stored procedure was gracefully rolled back.

- **RESPONSE_UNKNOWN** — This is a rare error that occurs if the coordinating node for the transaction fails before returning a response. The node to which your application is connected cannot determine if the transaction failed or succeeded before the coordinator was lost. The best course of action, if you receive this error, is to use a new query to determine if the transaction failed or succeeded and then take action based on that knowledge.

- **SUCCESS** — The stored procedure completed successfully.

- **UNEXPECTED_FAILURE** — An unexpected error occurred on the server and the procedure failed.

- **USER_ABORT** — The code of the stored procedure intentionally threw a UserAbort exception and the stored procedure was rolled back.

❸ If a `getStatus()` call identifies an error status other than SUCCESS, you can use the `getStatusString()` method to return a text message providing more information about the specific error that occurred.

❹ If you want the stored procedure to provide additional information to the calling application, there are two more methods to the `ClientResponse` that you can use. The methods `getAppStatus()` and `getAppStatusString()` act like `getStatus()` and `getStatusString()`, but rather than returning information set by VoltDB, `getAppStatus()` and `getAppStatusString()` return information set in the stored procedure code itself.

In the stored procedure, you can use the methods `setAppStatusCode()` and `setAppStatusString()` to set the values returned to the calling application by the stored procedure. For example:

```
/* stored procedure code */
final byte AppCodeWarm = 1;
final byte AppCodeFuzzy = 2;
                . . .
setAppStatusCode(AppCodeFuzzy);
setAppStatusString("I'm not sure about that...");
                . . .
```

For more information on the getStatus method, its possible return values, or any other class or method in the Java API, see the javadoc available online.

6.6.2. Handling Timeouts

One particular error that needs special handling is if a connection or a stored procedure call times out after is has been sent to the cluster. By default, the client interface only waits a specified amount of time (two minutes) for a stored procedure to complete. If no response is received from the server before the timeout

period expires, the client interface returns control to your application, notifying it of the error. The status in the `ClientResponse` object will be CLIENT_RESPONSE_TIMEOUT.

It is important to note that CLIENT_RESPONSE_TIMEOUT does not necessarily mean the procedure call failed. In fact, it is very possible that the procedure may complete and return information after the timeout error is reported. The timeout is provided to avoid locking up the client application when procedures are delayed or the connection to the cluster hangs for any reason.

Similarly, if no response of any kind is returned on a connection (even if no transactions are pending) within the specified timeout period, the client connection will time out. When this happens, the connection is closed, and any open stored procedures on that connection are closed with a return status of CONNECTION_LOST. Unlike a procedure timeout, when the connection times out, the connection no longer exists, so your client application will receive no further notifications concerning pending procedures, whether they succeed or fail.

CONNECTION_LOST does not necessarily mean a pending procedure call failed. It is possible that the procedure completed but was unable to return its status due to a connection failure. The goal of the connection timeout is to notify the client application of a lost connection in a timely manner, even if there are no outstanding procedures using the connection.

There are several things you can do to address potential timeouts in your application:

- Change the timeout period by calling either or both the methods `procedureCallTimeout()` and `connectionResponseTimeout()` on the `Client2Config` object. The default timeout period is 2 minutes for both procedures and connections. A value of zero or less disables the timeout altogether. For example, the following client code resets the procedure timeout to 90 seconds and the connection timeout period to 3 minutes, or 180 seconds:

  ```
  config = new Client2Config()
          .procedureCallTimeout(90, TimeUnit.SECONDS)
          .connectionResponseTimeout(3, TimeUnit.MINUTES);
  client = ClientFactory.createClient(config);
  ```

- Respond to the timeout error as part of the response to a procedure call. For example, the following code reports the error to the console.

  ```
  void processResponse(ClientResponse response) {
      if (response.getStatus() == ClientResponse.RESPONSE_CONNECTION_TIMEOUT) {
          System.out.println("A procedure invocation has timed out.");
          return;
      }
      if (response.getStatus() == ClientResponse.CONNECTION_LOST) {
          System.out.println("Connection lost before procedure response.");
          return;
      }
      // ... further non-timeout processing here ...
  }
  ```

- Set up a handler to receive the results of any procedure invocations that complete after the client interface times out. See the following Section 6.6.3, “Writing Handlers to Interpret Other Errors” for an example of creating a handler for delayed procedure responses.

6.6.3. Writing Handlers to Interpret Other Errors

Certain types of errors can occur that the `ClientResponse` class cannot notify you about immediately. In these cases, an error happens and is caught by the client interface outside of the normal stored procedure

execution cycle. If you want your application to address these situations, you need to create handlers, which are each a special type of asynchronous callback that the client interface will notify whenever such errors occur.

Handlers are individually defined for each type of event, by a specific call to a `Client2Config` method. The application handler must implement the corresponding interface, as defined in the `Client2Notification` class. The types of event that handlers address include:

Connect Failure

Notification that an attempt to connect or reconnect to a cluster member has failed. This applies to both initial connection requests from the application, and to automatic reconnection attempts by the Client2 implementation.

Connection Up

Notification that a connection has been established to a particular cluster member.

Connection Down

Notification that the connection to a particular cluster member has become disconnected. Outstanding procedure calls may or may not have completed successfully.

Late Response

Procedure invocations that time out in the client may later complete on the server and return results. Since the client application can no longer react to this response inline, the client may want a way to process the returned results. A ClientResponse object will be provided as an argument when this notification is called.

Request Backpressure

Indicates that the procedure-call queue in the Client2 API is nearing its limit. The queue has two limits: a warning limit and a hard limit. This notification is triggered when the queue length exceeds the warning limit. If the hard limit is exceeded, further procedure calls are rejected until the queue length falls sufficiently. The limits can be set in the `Client2Config>` with `clientRequestBackpressureLevel()` and `ClientRequestLimit`, for warning and hard limits respectively.

Error Log

Certain unlikely error situations in the Client2 API are not directly associated with a procedure call. By default, a message is printed on standard error. The application can choose to handle the message instead, perhaps writing it to its own log. The application should not attempt to interpret the message; the wording may change without notice.

For the sake of example, the following late-response handler does little more than display a message on standard output. However, in real world applications the handler would take appropriate actions based on the circumstances.

```
/*
 * Define a late response handler.
 */
void myLateResponseHandler(ClientResponse resp, String host, int port) {  ❶
    System.out.printf("A procedure that timed out on host %s port %d"
        + " has now responded with status %d\n", host, port, resp.getStatus());
}
```

```
/*
 *  Declare the client configuration, specifying
 *  a username, a password, and the handler.
 */
Client2Config myconfig = new Client2Config()                              ❷
        .username("username")
        .password("password")
        .lateResponseHandler(this::myLateResponseHandler);

/*
 *  Create the client using the specified configuration.
 */
Client2 myclient = ClientFactory.createClient(myconfig);                  ❸
```

By performing the operations in the order as described here, you ensure that all connections to the VoltDB database cluster use the same credentials for authentication and will notify the status listener of any error conditions outside of normal procedure execution.

❶ Declare a late-response handler. Define the handler before you define the VoltDB client or open a connection. The handler signature must conform to the interface defined for this handler in `Client2Notification`, with the three formal arguments shown.

❷ Define the client configuration `Client2Config` object. After you declare your `ClientStatusListenerExt`, you create a `Client2Config` object to use for all connections, which includes the username, password, and late-response handler. This configuration is then used to define the client.

❸ Create a client with the specified configuration.

6.7. Compiling and Running Client Applications

VoltDB client applications written in Java compile and run like other Java applications. (See Chapter 8, *Using VoltDB with Other Programming Languages* for more on writing client applications using other languages.) To compile, you must include the VoltDB libraries in the classpath so Java can resolve references to the VoltDB classes and methods. It is possible to do this manually by defining the environment variable `CLASSPATH` or by using the `-classpath` argument on the command line. If your client application depends on other libraries, they need to be included in the classpath as well. You can also specify where to create the resulting class files using the `-d` flag to specify an output directory, as in the following example:

```
$ javac -classpath "./:/opt/voltdb/voltdb/*" \
        -d ./obj  \
         *.java
```

The preceding example assumes that the VoltDB software has been installed in the folder `/opt/voltdb`. If you installed VoltDB in a different directory, you need to include your installation path in the `-classpath` argument.

If you are using Apache Maven to manage your application development, the VoltDB Java client library is available from the central Maven repository. So rather than installing VoltDB locally, you can simply include it as a dependency in your Maven project object model, or pom.xml, like so:

```
<dependency>
    <groupId>org.voltdb</groupId>
    <artifactId>voltdbclient</artifactId>
    <version>5.1</version>
```

```
</dependency>
```

6.7.1. Starting the Client Application

Before you start your client application, the VoltDB database must be running. When you start your client application, you must ensure that the VoltDB library JAR file is in the classpath. For example:

```
$ java -classpath "./:/opt/voltdb/voltdb/*"  MyClientApp
```

6.7.2. Running Clients from Outside the Cluster

If you are running the database on a cluster and the client applications on separate machines, you do not need to include all of the VoltDB software with your client application. The VoltDB distribution comes with two separate libraries: `voltdb-`*`n.n.nn`*`.jar` and `voltdbclient-`*`n.n.nn`*`.jar` (where *n.n.nn* is the VoltDB version number). The first file is a complete library that is required for building and running a VoltDB database server.

The second file, `voltdbclient-`*`n.n.nn`*`.jar`, is a smaller library containing only those components needed to run a client application. If you are distributing your client applications, you only need to distribute the client classes and the VoltDB client library. You do not need to install all of the VoltDB software distribution on the client nodes.

Chapter 7. Simplifying Application Development

The previous chapter (Chapter 6, *Designing VoltDB Client Applications*) explains how to develop your VoltDB database application using the full power and flexibility of the Java client interface. However, some database tasks — such as inserting records into a table or retrieving a specific column value — do not need all of the capabilities that the Java API provides. In other cases, there are automation techniques that can reduce the amount of application code you need to write and maintain.

Now that you know how the VoltDB programming interface works, VoltDB has features to simplify common tasks and make your application development easier. Those features include:

- Using Default Procedures
- Shortcut for Defining Simple Stored Procedures
- Verifying Expected Query Results
- Scheduling Stored Procedures as Tasks
- Directed Procedures: Distributing Transactions to Every Partition

The following sections describe each of these features separately.

7.1. Using Default Procedures

Although it is possible to define quite complex SQL queries, often the simplest are also the most common. Inserting, selecting, updating, and deleting records based on a specific key value are the most basic operations for a database. Another common practice is upsert, where if a row matching the primary key already exists, the record is updated — if not, a new record is inserted. To simplify these operations, VoltDB defines these default stored procedures for tables.

The default stored procedures use a standard naming scheme, where the name of the procedure is composed of the name of the table (in all uppercase), a period, and the name of the query in lowercase. For example, the Hello World tutorial (`doc/tutorials/helloworld`) contains a single table, HELLOWORLD, with three columns and the partitioning column, DIALECT, as the primary key. As a result, five default stored procedures are included in addition to any user-defined procedures declared in the schema. The parameters to the procedures differ based on the procedure.

VoltDB defines a default insert stored procedure when any table is defined:

HELLOWORLD.insert	The parameters are the table columns, in the same order as defined in the schema.

VoltDB defines default update, upsert, and delete stored procedures if the table has a primary key:

HELLOWORLD.update	The parameters are the new column values, in the order defined by the schema, followed by the primary key column values. This means the primary key column values are specified twice: once as their corresponding new column values and once as the primary key value.
HELLOWORLD.upsert	The parameters are the table columns, in the same order as defined in the schema.

HELLOWORLD.delete	The parameters are the primary key column values, listed in the order they appear in the primary key definition.

VoltDB defines a default select stored procedure if the table has a primary key and the table is partitioned:

HELLOWORLD.select	The parameters are the primary key column values, listed in the order they appear in the primary key definition.

Use the sqlcmd command **show procedures** to list all the stored procedures available including the number and type of parameters required. Use `@SystemCatalog with the PROCEDURECOLUMNS` selector to show more details about the order and meaning of each procedure's parameters.

The following code example uses the default procedures for the HELLOWORLD table to insert, retrieve (select), update, and delete a new record with the key value "American":

```
VoltTable[] results;
client.callProcedure("HELLOWORLD.insert",
                     "American","Howdy","Earth");
results = client.callProcedure("HELLOWORLD.select",
                     "American").getResults();
client.callProcedure("HELLOWORLD.update",
                     "American","Yo","Biosphere",
                     "American");
client.callProcedure("HELLOWORLD.delete",
                     "American");
```

7.2. Shortcut for Defining Simple Stored Procedures

Sometimes all you want is to execute a single SQL query and return the results to the calling application. In these simple cases, writing the necessary Java code to create a stored procedure can be tedious, so VoltDB provides a shortcut. For very simple stored procedures that execute a single SQL query and return the results, you can define the entire stored procedure as part of the database schema.

Recall from Section 5.3.2, “Declaring Stored Procedures in the Schema”, that normally you use the CREATE PROCEDURE statement to specify the class name of the Java procedure you coded, for example:

```
CREATE PROCEDURE FROM CLASS MakeReservation;
CREATE PROCEDURE FROM CLASS CancelReservation;
```

However, to create procedures without writing any Java, you can simply insert a SQL query in the AS clause:

```
CREATE PROCEDURE CountReservations AS
    SELECT COUNT(*) FROM RESERVATION;
```

VoltDB creates the procedure when you include the SQL query in the CREATE PROCEDURE AS statement. Note that you must specify a unique class name for the procedure, which is unique among all stored procedures, including both those declared in the schema and those created as Java classes. (You can use the sqlcmd command **show procedures** to display a list of all stored procedures.)

It is also possible to pass arguments to the SQL query in simple stored procedures. If you use the question mark placeholder in the SQL, any additional arguments you pass in client applications through the

`callProcedure()` method are used to replace the placeholders, in their respective order. For example, the following simple stored procedure expects to receive three additional parameters:

```
CREATE PROCEDURE MyReservationsByTrip AS
        SELECT R.RESERVEID, F.FLIGHTID, F.DEPARTTIME
        FROM RESERVATION AS R, FLIGHT AS F
        WHERE R.CUSTOMERID = ?
        AND R.FLIGHTID = F.FLIGHTID
        AND F.ORIGIN=? AND F.DESTINATION=?;
```

You can also specify whether the simple procedure is single-partitioned or not. By default, stored procedures are assumed to be multi-partitioned. But if your procedure should be single-partitioned, specify its partitioning in the PARTITION ON clause. In the following example, the stored procedure is partitioned on the FLIGHTID column of the RESERVATION table using the first parameter as the partitioning key.

```
CREATE PROCEDURE FetchReservations
   PARTITION ON
      TABLE Reservation COLUMN flightid
   AS
      SELECT * FROM RESERVATION WHERE FLIGHTID=?;
```

Finally, if you want to execute multiple SQL statements within a simple procedure, you must enclose the SQL in a BEGIN-END clause. For example, the following CREATE PROCEDURE AS statement fetches separate records from the CUSTOMER and ORDER tables:

```
CREATE PROCEDURE OpenOrders
   AS BEGIN
      SELECT fullname FROM CUSTOMER WHERE CUSTOMERID=?;
      SELECT * FROM ORDER WHERE CUSTOMERID=?;
   END;
```

Some important points to note concerning multi-statement simple procedures:

- The END statement and all of the enclosed SQL statements, must be terminated with a semi-colon.
- The procedure returns an array of VoltTables, one for each statement in the procedure.
- Each placeholder represents one parameter to the stored procedure. Parameters cannot be reused. So in the previous example, the customer ID would need to be entered twice as separate parameters to the stored procedure, one parameter for the first statement and one parameter for the second statement.

7.3. Verifying Expected Query Results

The automated default and simple stored procedures reduce the coding needed to perform simple queries. However, another substantial chunk of stored procedure and client application code is often required to verify the correctness of the results returned by the queries. Did you get the right number of records? Does the query return the correct value?

Rather than you having to write the code to validate the query results manually, VoltDB provides a way to perform several common validations as part of the query itself. The Java client interface includes an `Expectation` object that you can use to define the expected results of a query. Then, if the query does not meet those expectations, the executing stored procedure automatically throws a `VoltAbortException` and rolls back.

You specify the expectation as the second parameter (after the SQL statement but before any arguments) when queuing the query. For example, when making a reservation in the Flight application, the procedure

must make sure there are seats available. To do this, the procedure must determine how many seats the flight has. This query can also be used to verify that the flight itself exists, because there should be one and only one record for every flight ID.

The following code fragment uses the EXPECT_ONE_ROW expectation to both fetch the number of seats and verify that the flight itself exists and is unique.

```
import org.voltdb.Expectation;
           .
           .
           .
public final SQLStmt GetSeats = new SQLStmt(
        "SELECT numberofseats FROM Flight WHERE flightid=?;");

voltQueueSQL(GetSeats, EXPECT_ONE_ROW, flightid);
VoltTable[] recordset = voltExecuteSQL();
Long numofseats = recordset[0].asScalarLong();
```

By using the expectation, the stored procedure code does not need to do additional error checking to verify that there is one and only one row in the result set. The following table describes all of the expectations that are available to use in stored procedures.

Expectation	Description
EXPECT_EMPTY	The query must return no rows.
EXPECT_ONE_ROW	The query must return one and only one row.
EXPECT_ZERO_OR_ONE_ROW	The query must return no more than one row.
EXPECT_NON_EMPTY	The query must return at least one row.
EXPECT_SCALAR	The query must return a single value (that is, one row with one column).
EXPECT_SCALAR_LONG	The query must return a single value with a datatype of Long.
EXPECT_SCALAR_MATCH(long)	The query must return a single value equal to the specified Long value.

7.4. Scheduling Stored Procedures as Tasks

There are often repetitive tasks you want to perform on the database that can be scheduled at regular intervals. These tasks may include general cleanup, pruning, or periodic data validation. Rather than write a separate application and scheduler to do this, VoltDB lets you automate tasks at intervals ranging from milliseconds to days.

A *task* is a stored procedure that you schedule using the CREATE TASK statement. The statement specifies what procedure to run and when to run it and what arguments to use. In the simplest case, you can schedule a multi-partition procedure at specific times of day (using cron notation), at a regular interval (using EVERY), or with a regular pause between iterations (using DELAY). For example, The following statements define a procedure called OrphanedRecords that deletes reservations from a specific airline with no associated flight number and a task called RemoveOrphans that uses that procedure to delete orphaned records for FlyByNight airlines every two hours.

```
CREATE PROCEDURE OrphanedRecords
  AS DELETE FROM reservations
     WHERE aireline=? AND flight_id IS NULL;
```

```
CREATE TASK RemoveOrphans
  ON SCHEDULE EVERY 2 HOURS
  PROCEDURE OrphanedRecords WITH ('FlyByNight');
```

Since the task definition is part of the schema, VoltDB automates starting and stopping the tasks with the database. Other clauses to the CREATE TASK statement let you further refine how the task is run including what user account runs it and what to do in case of errors. There are also corresponding ALTER TASK and DROP TASK statements for managing your task definitions. See the description of the CREATE TASK statement for details.

7.5. Directed Procedures: Distributing Transactions to Every Partition

As useful as scheduling regular stored procedures is in simplifying application development, it can be disruptive to ongoing workflow if multi-partition procedures take too long or run too frequently. It would be nice to be able to schedule some partitioned activities as well to do piecemeal work on each partition without tying up all of the partitions at once. This is exactly what directed procedures are designed to do.

A *directed procedure* is a special type of stored procedure, declared using the DIRECTED clause instead of PARTITION ON. You write a directed procedure the same way you write a regular stored procedure: either as a simple procedure of one or more SQL statements or as a Java class extending voltProcedure, using the voltQueueSQL method to queue SQL statements. Since it is transactional, the procedure must also be deterministic.

However, if you declare the procedure as DIRECTED, when you invoke it a separate instance of the procedure is queued on every partition in the database. Each instance is its own transaction and acts like a partitioned procedure. So the separate transactions do not block the other partitions. However, because they are separate, there is no coordination between the transactions and no guarantee that they are executed at the same time.

This makes directed procedures particularly useful for non-critical procedures that need to access data across the database but do not need to be coordinated as a single, atomic transaction. Because of the special nature of directed procedures, you cannot invoke them the way you would normal partitioned or multi-partitioned procedures. Instead, the primary way to invoke them is as a scheduled task.

To schedule a directed procedure as a task, you use the same syntax for the CREATE TASK statement as for a multi-partitioned procedure, except you add the RUN ON PARTITIONS clause. The RUN ON PARTITIONS clause specifies that the task is scheduled separately for each and every partition. For example, if you want to run the RemoveOrphans task defined in the previous section as a directed procedure so it will not block the ongoing database workload, you would add the DIRECTED clause to the CREATE PROCEDURE statement and the RUN ON PARTITIONS clause to the CREATE TASK statement, like so:

```
CREATE PROCEDURE OrphanedRecords DIRECTED
  AS DELETE FROM reservations
     WHERE airline=? AND flight_id IS NULL;
CREATE TASK RemoveOrphans
  ON SCHEDULE EVERY 2 HOURS
  PROCEDURE OrphanedRecords WITH ('FlyByNight')
  RUN ON PARTITIONS;
```

Although scheduled tasks are the easiest way to invoke directed procedures, you can also invoke them directly from your Java applications. You cannot call them with the `callProcedure` method, but you can using the `callAllPartitionProcedure` method where the results from all of the partitions are

returned as an array of VoltTables, one per partition. See the descriptions of the CREATE PROCEDURE AS, CREATE PROCEDURE FROM CLASS, and CREATE TASK statements for more information about using directed procedures.

Chapter 8. Using VoltDB with Other Programming Languages

VoltDB stored procedures are written in Java and the primary client interface also uses Java. However, that is not the only programming language you can use with VoltDB.

It is possible to have client interfaces written in almost any language. These client interfaces allow programs written in different programming languages to interact with a VoltDB database using native functions of the language. The client interface then takes responsibility for translating those requests into a standard communication protocol with the database server as described in the VoltDB wire protocol.

Some client interfaces are developed and packaged as part of the standard VoltDB distribution kit while others are compiled and distributed as separate client kits. As of this writing, the following client interfaces are available for VoltDB:

- C#
- C++
- Erlang
- Go
- Java (packaged with VoltDB)
- JDBC (packaged with VoltDB)
- JSON (packaged with the VMC service)
- Node.js
- PHP
- Python (packaged with VoltDB)

The following sections explain how to use the C++, JSON, and JDBC client interfaces.

8.1. C++ Client Interface

VoltDB provides a client interface for programs written in C++. The C++ client interface is available precompiled as a separate kit from the VoltDB web site, or in source format from the VoltDB github repository (http://github.com/VoltDB/voltdb-client-cpp). The following sections describe how to write VoltDB client applications in C++.

8.1.1. Writing VoltDB Client Applications in C++

When using the VoltDB client library, as with any C++ library, it is important to include all of the necessary definitions at the beginning of your source code. For VoltDB client applications, this includes definitions for the VoltDB methods, structures, and datatypes as well as the libraries that VoltDB depends on (specifically, boost shared pointers). For example:

```
#define __STDC_CONSTANT_MACROS
```

```
#define __STDC_LIMIT_MACROS

#include <vector>
#include <boost/shared_ptr.hpp>
#include "Client.h"
#include "Table.h"
#include "TableIterator.h"
#include "Row.hpp"
#include "WireType.h"
#include "Parameter.hpp"
#include "ParameterSet.hpp"
#include "ProcedureCallback.hpp"
```

Once you have included all of the necessary declarations, there are three steps to using the interface to interact with VoltDB:

1. Create and open a client connection

2. Invoke stored procedures

3. Interpret the results

The following sections explain how to perform each of these functions.

8.1.2. Creating a Connection to the Database Cluster

Before you can call VoltDB stored procedures, you must create a client instance and connect to the database cluster. For example:

```
voltdb::ClientConfig config("myusername", "mypassword");
voltdb::Client client = voltdb::Client::create(config);
client.createConnection("myserver");
```

As with the Java client interface, you can create connections to multiple nodes in the cluster by making multiple calls to the **createConnection** method specifying a different IP address for each connection.

8.1.3. Invoking Stored Procedures

The C++ client library provides both a synchronous and asynchronous interface. To make a synchronous stored procedure call, you must declare objects for the parameter types, the procedure call itself, the parameters, and the response. Note that the datatypes, the procedure, and the parameters need to be declared in a specific order. For example:

```
/* Declare the number and type of parameters */
std::vector<voltdb::Parameter> parameterTypes(3);
parameterTypes[0] = voltdb::Parameter(voltdb::WIRE_TYPE_BIGINT);
parameterTypes[1] = voltdb::Parameter(voltdb::WIRE_TYPE_STRING);
parameterTypes[2] = voltdb::Parameter(voltdb::WIRE_TYPE_STRING);

/* Declare the procedure and parameter structures */
voltdb::Procedure procedure("AddCustomer", parameterTypes);
voltdb::ParameterSet* params = procedure.params();

/* Declare a client response to receive the status and return values */
```

```
voltdb::InvocationResponse response;
```

Once you instantiate these objects, you can reuse them for multiple calls to the stored procedure, inserting different values into *params* each time. For example:

```
params->addInt64(13505).addString("William").addString("Smith");
response = client.invoke(procedure);
params->addInt64(13506).addString("Mary").addString("Williams");
response = client.invoke(procedure);
params->addInt64(13507).addString("Bill").addString("Smythe");
response = client.invoke(procedure);
```

8.1.4. Invoking Stored Procedures Asynchronously

To make asynchronous procedure calls, you must also declare a callback structure and method that will be used when the procedure call completes.

```
class AsyncCallback : public voltdb::ProcedureCallback
{
public:
    bool callback
         (voltdb::InvocationResponse response)
         throw (voltdb::Exception)
    {
           /*
           * The work of your callback goes here...
           */
    }
};
```

Then, when you go to make the actual stored procedure invocation, you declare an callback instance and invoke the procedure, using both the procedure structure and the callback instance:

```
boost::shared_ptr<AsyncCallback> callback(new AsyncCallback());
client.invoke(procedure, callback);
```

Note that the C++ interface is single-threaded. The interface is not thread-safe and you should not use instances of the client, client response, or other client interface structures from within multiple concurrent threads. Also, the application must release control occasionally to give the client interface an opportunity to issue network requests and retrieve responses. You can do this by calling either the run() or runOnce() methods.

The run() method waits for and processes network requests, responses, and callbacks until told not to. (That is, until a callback returns a value of false.)

The runOnce() method processes any outstanding work and then returns control to the client application.

In most applications, you will want to create a loop that makes asynchronous requests and then calls runOnce(). This allows the application to queue stored procedure requests as quickly as possible while also processing any incoming responses in a timely manner.

Another important difference when making stored procedure calls asynchronously is that you must make sure all of the procedure calls complete before the client connection is closed. The client objects destructor automatically closes the connection when your application leaves the context or scope within which the client is defined. Therefore, to make sure all asynchronous calls have completed, be sure to call the *drain* method until it returns true before leaving your client context:

```
while (!client.drain()) {}
```

8.1.5. Interpreting the Results

Both the synchronous and asynchronous invocations return a client response object that contains both the status of the call and the return values. You can use the status information to report problems encountered while running the stored procedure. For example:

```
if (response.failure())
{
    std::cout << "Stored procedure failed. " << response.toString();
    exit(-1);
}
```

If the stored procedure is successful, you can use the client response to retrieve the results. The results are returned as an array of VoltTable structures. Within each VoltTable object you can use an iterator to walk through the rows. There are also methods for retrieving each datatype from the row. For example, the following example displays the results of a single VoltTable containing two strings in each row:

```
/* Retrieve the results and an iterator for the first volttable */
std::vector<voltdb::Table> results = response.results();
voltdb::TableIterator iterator = results[0].iterator();

/* Iterate through the rows */
while (iterator.hasNext())
{
    voltdb::Row row = iterator.next();
    std::cout << row.getString(0) << ", " << row.getString(1) << std::endl;
}
```

8.2. JSON HTTP Interface

JSON (JavaScript Object Notation) is not a programming language; it is a data format. The JSON "interface" to VoltDB is actually a web interface that the Volt Management Center service makes available for processing requests and returning data in JSON format.

The JSON interface lets you invoke VoltDB stored procedures and receive their results through HTTP requests. To invoke a stored procedure, you pass VoltDB the procedure name and parameters as a querystring to the HTTP request, using either the GET or POST method.

Although many programming languages provide methods to simplify the encoding and decoding of JSON strings, you still need to understand the data structures that are created. So if you are not familiar with JSON encoding, you may want to read more about it at `http://www.json.org`.

8.2.1. How the JSON Interface Works

When you start the Volt Management Center (VMC) service and point it at a VoltDB cluster, it opens port 8080 as a simple web server. You can customize this service through the VMC configuration options, including:

- Enabling TLS encryption on the port
- Changing the port number

See the section on the "Web Interface Port" in the *VoltDB Administrator's Guide* for more information on configuring the HTTP port.

This section assumes the VMC service is using the default httpd configuration. In which case, any HTTP requests sent to the location /api/2.0/ on that port are interpreted as JSON requests to run a stored procedure. The structure of the request is:

URL	http://<server>:8080/api/2.0/
Arguments	Procedure=<procedure-name> Parameters=<procedure-parameters> User=<username for authentication> Password=<password for authentication> Hashedpassword=<Hashed password for authentication> admin=<true|false>

The arguments can be passed either using the GET or the POST method. For example, the following URL uses the GET method (where the arguments are appended to the URL) to execute the system procedure @SystemInformation on the VoltDB database cluster connected to the VMC service at vmc.mycompany.com:

```
http://vmc.mycompany.com:8080/api/2.0/?Procedure=@SystemInformation
```

Note that only the `Procedure` argument is required. You can authenticate using the `User` and `Password` (or `Hashedpassword`) arguments if security is enabled for the database. Use `Password` to send the password as plain text or `Hashedpassword` to send the password as an encoded string. (The hashed password must be a 64-byte hex-encoding of the 32-byte SHA-256 hash.)[1]

You can also include the parameters on the request. However, it is important to note that the parameters — and the response returned by the stored procedure — are JSON encoded. The parameters are an array (even if there is only one element to that array) and therefore must be enclosed in square brackets. Also, although there is an upper limit of 2 megabytes for the entire length of the parameter string, large parameter sets must be sent using POST to avoid stricter limitations on allowable URL lengths.

The `admin` argument specifies whether the request is submitted on the standard client port (the default) or the admin port (when you specify `admin=true`). When the database is in admin mode, the client port is read-only; so you must submit write requests with `admin=true` or else the request is rejected by the server.

Perhaps the best way to understand the JSON interface is to see it in action. If you build and start the Hello World example application that is provided in the VoltDB distribution kit (including the client that loads data into the database and the VMC service), you can then open a web browser and connect to the local system through port 8080, to retrieve the French translation of "Hello World". For example:

```
http://localhost:8080/api/2.0/?Procedure=Select&Parameters=["French"]
```

The query returns the following results:

```
{"status":1,"appstatus":-128,"statusstring":null,"appstatusstring":null,
"results":{"0":[{ "HELLO":"Bonjour","WORLD":"Monde"}]}}
```

As you can see, the JSON-encoded results are not particularly easy to read. But they do provide a simple, generic interface accessible from almost any programming language, most of which provide methods for encoding and decoding JSON strings and interpreting their results.

[1] Hashing the password stops the text of your password from being detectable from network traffic. However, it does not make the database access any more secure. To secure the transmission of credentials and data between client applications and VoltDB, enable TLS encryption for the HTTP port using the configuration file.

8.2.2. Using the JSON Interface from Client Applications

The general process for using the JSON interface from within a program is:

1. Encode the parameters for the stored procedure as a JSON-encoded string
2. Instantiate and execute an HTTP request, passing the name of the procedure and the parameters as arguments using either GET or POST.
3. Decode the resulting JSON string into a language-specific data structure and interpret the results.

The following are examples of invoking the Hello World Insert stored procedure from several different languages. In each case, the three arguments (the name of the language and the words for "Hello" and "World") are encoded as a JSON string.

PHP

```
<?php
// Construct the procedure name, parameter list, and URL.

   $voltdbserver = "http://myserver:8080/api/2.0/";
   $proc = "Insert";
   $a = array("Croatian","Pozdrav","Svijet");
   $params = json_encode($a);
   $params = urlencode($params);
   $querystring = "Procedure=$proc&Parameters=$params";

// create a new cURL resource and set options
   $ch = curl_init();
   curl_setopt($ch, CURLOPT_URL, $voltdbserver);
   curl_setopt($ch, CURLOPT_HEADER, 0);
   curl_setopt($ch, CURLOPT_FAILONERROR, 1);
   curl_setopt($ch, CURLOPT_POST, 1);
   curl_setopt($ch, CURLOPT_POSTFIELDS, $querystring);
   curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);

// Execute the request
   $resultstring = curl_exec($ch);
?>
```

Python

```
import urllib
import urllib2
import json

# Construct the procedure name, parameter list, and URL.
url = 'http://myserver:8080/api/2.0/'
voltparams = json.dumps(["Croatian","Pozdrav","Svijet"])
httpparams = urllib.urlencode({
    'Procedure': 'Insert',
    'Parameters' : voltparams
})
print httpparams
```

```
# Execute the request
data = urllib2.urlopen(url, httpparams).read()

# Decode the results
result = json.loads(data)
```

Perl

```
use LWP::Simple;

my $server = 'http://myserver:8080/api/2.0/';

# Insert "Hello World" in Croatian
my $proc = 'Insert';
my $params = '["Croatian","Pozdrav","Svijet"]';
my $url = $server . "?Procedure=$proc&Parameters=$params";
my $content = get $url;
die "Couldn't get $url" unless defined $content;
```

C#

```
using System;
using System.Text;
using System.Net;
using System.IO;

namespace hellovolt

{
  class Program
  {
    static void Main(string[] args)
    {
      string VoltDBServer = "http://myserver:8080/api/2.0/";
      string VoltDBProc = "Insert";
      string VoltDBParams = "[\"Croatian\",\"Pozdrav\",\"Svijet\"]";
      string Url = VoltDBServer + "?Procedure=" + VoltDBProc
                  + "&Parameters=" + VoltDBParams;

      string result = null;
      WebResponse response = null;
      StreamReader reader = null;

      try
      {
        HttpWebRequest request = (HttpWebRequest)WebRequest.Create(Url);
        request.Method = "GET";
        response = request.GetResponse();
        reader = new StreamReader(response.GetResponseStream(),Encoding.UTF8 );
        result = reader.ReadToEnd();

      }
      catch (Exception ex)
```

```
        {     // handle error
          Console.WriteLine( ex.Message );
        }
        finally
        {
          if (reader != null)reader.Close();
          if (response != null) response.Close();

        }
      }
    }
}
```

8.2.3. How Parameters Are Interpreted

When you pass arguments to the stored procedure through the JSON interface, VoltDB does its best to map the data to the datatype required by the stored procedure. This is important to make sure partitioning values are interpreted correctly.

For integer values, the JSON interface maps the parameter to the smallest possible integer type capable of holding the value. (For example, BYTE for values less than 128). Any values containing a decimal point are interpreted as DOUBLE.

String values (those that are quoted) are handled in several different ways:

- If the stored procedure is expecting a BIGDECIMAL, the JSON interface will try to interpret the quoted string as a decimal value.
- If the stored procedure is expecting a TIMESTAMP, the JSON interface will try to interpret the quoted string as a JDBC-encoded timestamp value. (You can alternately pass the argument as an integer value representing the number of microseconds from the epoch.)
- If the stored procedure is expecting a DATE, the JSON interface will try to interpret the quoted string as a JDBC-encoded date value.
- Otherwise, quoted strings are interpreted as a string datatype.

Table 8.1, “Datatypes in the JSON Interface” summarizes how to pass different datatypes in the JSON interface.

Table 8.1. Datatypes in the JSON Interface

Datatype	How to Pass	Example
Integers (Byte, Short, Integer, Long)	An integer value	12345
DOUBLE	A value with a decimal point	123.45
BIGDECIMAL	A quoted string containing a value with a decimal point	"123.45"
TIMESTAMP	Either an integer value or a quoted string containing a JDBC-encoded date and time	12345 "2010-07-01 12:30:21"
DATE	A quoted string containing a JDBC-encoded date	"2010-07-01"

Datatype	How to Pass	Example
String	A quoted string	"I am a string"
BOOLEAN	A boolean value of true or false	true

8.2.4. Interpreting the JSON Results

Making the request and decoding the result string are only the first steps. Once the request is completed, your application needs to interpret the results.

When you decode a JSON string, it is converted into a language-specific structure within your application, composed of objects and arrays. If your request is successful, VoltDB returns a JSON-encoded string that represents the same ClientResponse object returned by calls to the callProcedure method in the Java client interface. Figure 8.1, “The Structure of the VoltDB JSON Response” shows the structure of the object returned by the JSON interface.

Figure 8.1. The Structure of the VoltDB JSON Response

```
{  status                  (integer)
   appstatus               (integer)
   statusstring            (string)
   appstatusstring         (string)
   results                 (list)
     {  result-index       (array)
          [
             { column-name  (any type)  ,...
             }
          ]
     }
}
```

The key components of the JSON response are the following:

status — Indicates the success or failure of the stored procedure. If *status* is false, *statusstring* contains the text of the status message..

appstatus — Returns additional information, provided by the application developer, about the success or failure of the stored procedure. The values of *appstatus* and *appstatusstring* can be set programmatically in the stored procedure. (See Section 6.6.1, “Interpreting Execution Errors” for details.)

results — A list of objects representing the VoltTables returned by the stored procedure. Each element of the list is one set of results, identified by an index value ("0", "1", "2" and so on). Within each set is an array of rows. And within each row is a list of columns represented by the column name and value. If the stored procedure does not return any results (i.e. is void or null), then the *results* object will be null.

It is possible to create a generic procedure for testing and evaluating the result values from any VoltDB stored procedure. However, in most cases it is far more expedient to evaluate the values that you know the individual procedures return.

For example, again using the Hello World example that is provided with the VoltDB software, it is possible to use the JSON interface to call the Select stored procedure and return the values for "Hello" and "World" in a specific language. Rather than evaluate the entire results array (including the name and type fields),

we know we are only receiving one result object with two column values. So we can simplify the code, as in the following python example:

```
import urllib
import urllib2
import json
import pprint

# Construct the procedure name, parameter list, and URL.
url = 'http://localhost:8080/api/2.0/'
voltparams = json.dumps(["French"])
httpparams = urllib.urlencode({
    'Procedure': 'Select',
    'Parameters' : voltparams
})

# Execute the request
data = urllib2.urlopen(url, httpparams).read()

# Decode the results
results = json.loads(data)[u'results']
volttable = results[u'0']
row = volttable[0]

# Get the data by column name and display them
hello = row[u'HELLO']
world = row[u'WORLD']

print hello, world
```

8.2.5. Error Handling using the JSON Interface

There are a number of different reasons why a stored procedure request using the JSON interface may fail: the VoltDB server may be unreachable, the database may not be started yet, the stored procedure name may be misspelled, the stored procedure itself may fail... When using the standard Java client interface, these different situations are handled at different times. (For example, server and database access issues are addressed when instantiating the client, whereas stored procedure errors can be handled when the procedures themselves are called.) The JSON interface simplifies the programming by rolling all of these activities into a single call. But you must be more organized in how you handle errors as a consequence.

When using the JSON interface, you should check for errors in the following order:

1. First check to see that the HTTP request was submitted without errors. How this is done depends on what language-specific methods you use for submitting the request. In most cases, you can use the appropriate programming language error handlers (such as try-catch) to catch and interpret HTTP request errors.

2. Next check to see if VoltDB successfully invoked the stored procedure. You can do this by verifying that the HTTP request returned a valid JSON-encoded string and that its *status* is set to true.

3. If the VoltDB server successfully invoked the stored procedure, then check to see if the stored procedure itself succeeded, by checking to see if *appstatus* is true.

4. Finally, check to see that the results are what you expect. (For example, that the *data* array is non-empty and contains the values you need.)

8.2.6. Managing Client Sessions

For intermittent use, the JSON Interface can be very effective and easy to use. The API server creates a new client connection for each request which, for occasional ad hoc queries, is simple and performs sufficiently well. However, if you intend to make frequent repeated requests, the cumulative cost of opening and closing separate client connections for each request can be significant — both in terms of latency for your application and memory usage on the server to manage so many transient client connections. The better approach is to have all of your requests share the same client connection.

The JSON interface allows for this through the use of cookies. When your application first makes a successful request to the server, the server response includes a cookie that identifies the client session. The cookie, JSESSIONID_*nnnn*, consists of a name (JSESSIONID) plus the client port number of the server. For all subsequent calls to the JSON interface, your application can return that cookie to the server and the server will reuse the initial client connection. This makes subsequent calls much faster, since they do not have to bear the cost of creating a new session. Every few minutes, the server will close the connection and open a new client. At which point it passes a new value for the cookie and your application can then return this new cookie value for subsequent calls.

8.3. JDBC Interface

JDBC (Java Database Connectivity) is a programming interface for Java programmers that abstracts database specifics from the methods used to access the data. JDBC provides standard methods and classes for accessing a relational database and vendors then provide JDBC drivers to implement the abstracted methods on their specific software.

VoltDB provides a JDBC driver for those who would prefer to use JDBC as the data access interface. The VoltDB JDBC driver supports ad hoc queries, prepared statements, calling stored procedures, and methods for examining the metadata that describes the database schema.

8.3.1. Using JDBC to Connect to a VoltDB Database

The VoltDB driver is a standard class within the VoltDB software jar. To load the driver you use the Class.forName method to load the class org.voltdb.jdbc.Driver.

Once the driver is loaded, you create a connection to a running VoltDB database server by constructing a JDBC url using the "jdbc:" protocol, followed by "voltdb://", the server name, a colon, and the port number. In other words, the complete JDBC connection url is "jdbc:voltdb://{server}:{port}". To connect to multiple nodes in the cluster, use a comma separated list of server names and port numbers after the "jdbc:voltdb://" prefix.

For example, the following code loads the VoltDB JDBC driver and connects to the servers svr1 and svr2 using the default client port:

```
Class.forName("org.voltdb.jdbc.Driver");
Connection c = DriverManager.getConnection(
     "jdbc:voltdb://svr1:21212,svr2:21212");
```

If, after the connection is made, the connection to one or more of the servers is lost due to a network issue or server failure, the VoltDB JDBC client does not automatically reconnect the broken connection by default. However, you can have the JDBC driver reconnect lost connections, as well as make other adjustments as the cluster topology changes, by adding the `topologychangeaware` argument to the connection string. For example:

```
Class.forName("org.voltdb.jdbc.Driver");
```

```
Connection c = DriverManager.getConnection(
     "jdbc:voltdb://svr1:21212,svr2:21212?topologychangeaware=true");
```

When topology awareness is enabled and a server goes offline, the JDBC driver periodically attempts to reconnect to the missing server until it comes back online and the connection is reestablished. If a different server replaces a lost node, topology awareness also identifies this change and makes the necessary connections.

If security is enabled for the database, you must also provide a username and password. Set these as properties using the setProperty method before creating the connection and then pass the properties as a second argument to getConnection. For example, the following code uses the username/password pair of "Hemingway" and "KeyWest" to authenticate to the VoltDB database:

```
Class.forName("org.voltdb.jdbc.Driver");
Properties props = new Properties();
props.setProperty("user", “Hemingway");
props.setProperty("password", “KeyWest");
Connection c = DriverManager.getConnection(
     "jdbc:voltdb://svr1:21212,svr2:21212", props);
```

8.3.2. Using JDBC to Query a VoltDB Database

Once the connection is made, you use the standard JDBC classes and methods to access the database. (See the JDBC documentation at `http://download.oracle.com/javase/8/docs/technotes/guides/jdbc` for details.) Note, however, when running the JDBC application, you must make sure both the VoltDB software jar and the Guava library are in the Java classpath. Guava is a third party library that is shipped as part of the VoltDB kit in the /lib directory. Unless you include both components in the classpath, your application will not be able to find and load the necessary driver class.

The following is a complete example that uses JDBC to access the Hello World tutorial that comes with the VoltDB software in the subdirectory `/doc/tutorials/helloworld`. The JDBC demo program executes both an ad hoc query and a call to the VoltDB stored procedure, Select.

```
import java.sql.*;
import java.io.*;

public class JdbcDemo {

    public static void main(String[] args) {

        String driver = "org.voltdb.jdbc.Driver";
        String url = "jdbc:voltdb://localhost:21212";
        String sql = "SELECT dialect FROM helloworld";

        try {
                // Load driver. Create connection.
            Class.forName(driver);
            Connection conn = DriverManager.getConnection(url);

                // create a statement
            Statement query = conn.createStatement();
            ResultSet results = query.executeQuery(sql);
            while (results.next()) {
                System.out.println("Language is " + results.getString(1));
```

```
                }

                    // call a stored procedure
                CallableStatement proc = conn.prepareCall("{call Select(?)}");
                proc.setString(1, "French");
                results = proc.executeQuery();
                while (results.next()) {
                    System.out.printf("%s, %s!\n", results.getString(1),
                                                    results.getString(2));
                }

                     //Close statements, connections, etc.
                query.close();
                proc.close();
                results.close();
                conn.close();

            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

Chapter 9. Using VoltDB in a Cluster

It is possible to run VoltDB on a single server and still get all the advantages of parallelism because VoltDB creates multiple partitions on each server. However, there are practical limits to how much memory or processing power any one server can sustain.

One of the key advantages of VoltDB is its ease of expansion. You can increase both capacity and processing (i.e. the total number of partitions) simply by adding servers to the cluster to achieve almost linear scalability. Using VoltDB in a cluster also gives you the ability to increase the availability of the database — protecting it against possible server failures or network glitches.

This chapter explains how to create a cluster of VoltDB servers running a single database. It also explains how to expand the cluster when additional capacity or processing power is needed. The following chapters explain how to increase the availability of your database through the use of K-safety and database replication, as well as how to enable security to limit access to the data.

9.1. Starting a Database Cluster

As described in Chapter 3, *Starting the Database*, starting a VoltDB cluster is similar to starting VoltDB on a single server — you use the same commands. To start a single server database, you use the **voltdb start** command by itself. To customize database features, you specify or more configuration files when you initialize the root directory with **voltdb init**.

To start a cluster, you also use the **voltdb start** command. In addition, you must:

- Specify the number of nodes in the cluster using the --count argument.
- Choose one or more nodes as the potential lead or "host" node and specify those nodes using the --host argument on the start command
- Issue the same **voltdb start** command on all nodes of the cluster

For example, if you are creating a new five node cluster and choose nodes server2 and server3 as the hosts, you would issue a command like the following on all five nodes:

```
$ voltdb start --host=server2,server3 --count=5
```

To restart a cluster using command logs or automatic snapshots, you repeat the same command. Alternately, you can specify all nodes in the cluster in the --host argument and skip the server count:

```
$ voltdb start --host=server1,server2,server3,server4,server5
```

No matter which approach you choose, you must specify the same list of potential hosts on all nodes of the cluster. Once the database cluster is running the leader's special role is complete and all nodes become peers.

9.2. Defining the Cluster Configuration

Before you start the cluster, you choose what database features to use by specifying one or more YAML configuration files when you initialize the database root directory on each node using the **voltdb init** command. You must specify the same configuration files on every node of the cluster. For example:

```
$ voltdb init --config=configuration.yaml
```

If you choose to change database options, many of the features can be adjusted while the database is running by either:

- Using the web-based Volt Management Center to make changes interactively in the Admin tab
- Editing the original configuration file(s) and applying the modifications with the **voltadmin update** command
- Setting or editing individual configuration properties using the **voltadmin set** command

For example, you can change security settings, import and export configurations, and resource limits dynamically. The changes you make are saved by VoltDB in the database root directory.

However, there are some changes that cannot be made while the database is running. For example, changing the K-safety value or the number of partitions per server require shutting down, re-initializing, and restarting the database. To change these static aspects of your cluster, you must save the database contents, reconfigure the root directory, then restart and restore the database. The steps for changing static configuration options are:

1. Pause the database (**voltadmin pause**)
2. Save a snapshot of the contents (**voltadmin save {path} {file-prefix}**)
3. Shutdown the database (**voltadmin shutdown**)
4. Re-initialize the root directory with the new configuration file and the `--force` argument (**voltdb init --force --config=*file***)
5. Restart the database in admin mode (**voltdb start --pause**)
6. Restore the snapshot (**voltadmin restore {path} {file-prefix}**)
7. Resume normal operations (**voltadmin resume**)

See Table E.1, “Complete List of Volt Configuration Properties” for a list of which properties can and cannot be modified dynamically. See Chapter 13, *Saving & Restoring a VoltDB Database* for information on using **save** and **restore**. When doing benchmarking, where you need to change the number of partitions or adjust other static configuration options, this is the recommended approach. However, if you are adjusting the size of the cluster to increase or decrease capacity or performance, you can perform these operations while the database is running. Adding and removing nodes "on the fly" is known as *elastic* scaling and is described in the next section.

9.3. Elastic Scaling to Resize the Cluster

Elastic scaling is the ability to resize the cluster as needed, without having to shutdown the database. Elastic scaling supports both increasing and decreasing the size of the cluster. For example, you might want to increase the size of the cluster ahead of an important announcement that will drive additional traffic — and subsequently require additional capacity. Similarly, you may want to reduce the size for the cluster during slow periods to limit the number of resources that would be under utilized.

Adding and removing nodes using elastic scaling are each handled separately because increasing the size of the cluster requires adding new nodes to the cluster first. While when decreasing the size of the cluster, the nodes are already part of the cluster and VoltDB decides which node are most advantageous to remove based on the distribution of partitions within the cluster.

To add nodes to the cluster you start the additional nodes using the **voltdb start --add** command. To remove nodes from the cluster, you use the **voltadmin resize** command and the cluster decides which nodes to remove.

But in both cases, the correct number of nodes must be added or removed at the same time. The number of nodes added or removed must result in the resized cluster meeting the requirements for a K-safe cluster based on the K-safety value and number of sites per host (as described in Section 10.2.2, “Calculating the Appropriate Number of Nodes for K-Safety”). So for a cluster with no K-safety (K=0), nodes can be added and removed individually. For K-safe clusters, K+1 nodes must be added or removed at a time. For example, with K=1 two nodes must be added at a time. While in the case of reducing the size of the cluster, two nodes must be removed but the resulting cluster must also meet the requirement that the total number of partitions (sites per host X number of nodes) is divisible by K+1.

Finally, resizing the cluster "on the fly" does require both time and some amount of resources while the data and partitions are rebalanced. The length of time required to complete the rebalancing depends on the amount of data present and the current workload. Similarly, the performance impact of resizing on the ongoing operation of the cluster depends on how much additional capacity the cluster has to assign to rebalance tasks.

The following sections describe how to:

- Add nodes using elastic scaling
- Remove nodes using elastic scaling
- Control the time and performance impact of elastic scaling by configuring the rebalance workoad

9.3.1. Adding Nodes with Elastic Scaling

When you are ready to extend the cluster by adding one or more nodes, you simply initialize and start the VoltDB database process on the new nodes using the **voltdb init** command to initialize and the **voltdb start** command to start with the `--add` argument, specifying the name of one or more of the existing cluster nodes as the hosts. For example, if you are adding node ServerX to a cluster where ServerA is already a member, you can execute the following commands on ServerX:

```
$ voltdb init --config=configuration.yaml
$ voltdb start --add --host=ServerA
```

Once the elastic add action is initiated, the cluster performs the following tasks:

1. The cluster acknowledges the presence of a new server.
2. Copies of the current schema and configuration settings are sent to the new node.
3. Once sufficient nodes are added, copies of all replicated tables and their share of the partitioned tables are sent to the new nodes.
4. As the data is redistributed (or *rebalanced*), the added nodes begin participating as full members of the cluster.

There are some important notes to consider when expanding the cluster using elastic scaling:

- You must add a sufficient number of nodes to create an integral K-safe unit. That is, K+1 nodes. For example, if the K-safety value for the cluster is two, you must add three nodes at a time to expand the cluster. If the cluster is not K-safe (in other words it has a K-safety value of zero), you can add one node at a time.

- When you add nodes to a K-safe cluster, the nodes added first will complete steps #1 and #2 above, but will not complete steps #3 and #4 until the correct number of nodes are added, at which point all nodes rebalance together.
- While the cluster is rebalancing (Step #3), the database continues to handle incoming requests. However, depending on the workload and amount of data in the database, rebalancing may take a significant amount of time.
- Once elastic scaling is complete, your database configuration has changed. If you shutdown the database and then restart, you must specify the new server count in the `--count` argument to the **voltdb start** command.

9.3.2. Removing Nodes with Elastic Scaling

When you want to reduce the size of your cluster, you use the **voltadmin resize** command to start the resizing process. First, as with any significant maintenance activity, it is a good idea to take a snapshot of the database contents before you begin, just in case you need to restore it later. The next step is to test to make sure the cluster can be reduced. You do this using the **voltadmin resize --test** command:

```
$ voltadmin resize --test
```

The --test qualifier verifies that there are sufficient nodes and partitions to reduce the cluster while maintaining the K-safety and sitesperhost settings. If not, the command will report that the cluster cannot be reduced in size. If resizing is possible, the command reports which nodes will be removed when resizing begins.

Once you are ready to begin the resizing process, you use the **voltadmin resize** command:

```
$ voltadmin resize
```

The command repeats the test phase, reports which nodes will be removed and starts the resizing process.

Once resizing begins, the process cannot be canceled. Even if the cluster stops, resizing will continue once the cluster restarts (and you must restart all of the original nodes so the resize operation can complete). So be sure you want to reduce the cluster size before you issue the **voltadmin resize** command.

The length of time it takes for resizing to complete depends on the amount of data in the database and the current workload. You can adjust parameters that affect resizing (as described in Section 9.3.3, “Configuring How VoltDB Rebalances Nodes During Elastic Scaling”). However, increasing the duration or throughput for resizing will likely have a corresponding inverse impact on the performance of ongoing database activities. Use the **voltadmin status** to check on the current status of the resizing operation, or use the @Statistics system procedure with the REBALANCE selector for details.

Finally, if an unexpected event causes the resize process to fail — which will be reported in the server logs — you can restart the resize operation using the **voltadmin resize --restart** command.

9.3.3. Configuring How VoltDB Rebalances Nodes During Elastic Scaling

As you add or remove nodes using elastic scaling, VoltDB rebalances the cluster by rearranging data within the partitions. During elastic expansion, as soon as you add the necessary number of nodes (based on the K-safety value), VoltDB rebalances the cluster, moving data from existing partitions to partitions on the new nodes. During elastic contraction, before the nodes are removed, VoltDB rebalances the cluster by moving data from partitions that are being removed to partitions that will remain.

During the rebalance phase, the database remains available and actively processing client requests. How long the rebalance operation takes is dependent on two factors: how often rebalance tasks are processed and how much data each transaction moves.

Rebalance tasks are fully transactional, meaning they operate within the database's ACID-compliant transactional model. Because they involve moving data between two or more partitions, they are also multi-partition transactions. This means that each rebalance work unit can incrementally add to the latency of pending client transactions.

You can control how quickly the rebalance operation completes versus how much rebalance work impacts ongoing client transactions using the subproperties of `deployment.systemsettings.elastic`:

- The `duration` property sets a target value for the length of time each rebalance transaction will take, specified in milliseconds. The default is 50 milliseconds.
- The `throughput` property sets a target value for the number of megabytes per second that will be processed by the rebalance transactions. The default is 2 megabytes.

When you change the target duration, VoltDB adjusts the amount of data that is moved in each transaction to reach the target execution time. If you increase the duration, the volume of data moved per transaction increases. Similarly, if you reduce the duration, the volume per transaction decreases.

When you change the target throughput, VoltDB adjusts the frequency of rebalance transactions to achieve the desired volume of data moved per second. If you increase the target throughout, the number of rebalance transactions per second increases. Similarly, if you decrease the target throughout, the number of transactions decreases.

For example, the following configuration file sets the target duration to 15 milliseconds and the target throughput to 1 megabyte per second before starting the database:

```
deployment:
  systemsettings:
     elastic:
       duration: 15
       throughput: 1
```

Chapter 10. Availability

Durability is one of the four key ACID attributes required to ensure the accurate and reliable operation of a transactional database. Durability refers to the ability to maintain database consistency and availability in the face of external problems, such as hardware or operating system failure. Durability is provided by four features of VoltDB: snapshots, command logging, K-safety, and disaster recovery through database replication.

- *Snapshots* are a "snapshot" of the data within the database at a given point in time written to disk. You can use these snapshot files to restore the database to a previous, known state after a failure which brings down the database. The snapshots are guaranteed to be transactionally consistent at the point at which the snapshot was taken. Chapter 13, *Saving & Restoring a VoltDB Database* describes how to create and restore database snapshots.

- *Command Logging* is a feature where, in addition to periodic snapshots, the system keeps a log of every stored procedure (or "command") as it is invoked. If, for any reason, the servers fail, they can "replay" the log on startup to reinstate the database contents completely rather than just to an arbitrary point-in-time. Chapter 14, *Command Logging and Recovery* describes how to enable, configure, and replay command logs.

- *K-safety* refers to the practice of duplicating database partitions so that the database can withstand the loss of cluster nodes without interrupting the service. For example, a K value of zero means that there is no duplication and losing any servers will result in a loss of data and database operations. If there are two copies of every partition (a K value of one), then the cluster can withstand the loss of at least one node (and possibly more) without any interruption in service.

- *Database Replication* is similar to K-safety, since it involves replicating data. However, rather than creating redundant partitions within a single database, database replication involves creating and maintaining a complete copy of the entire database. Database replication has a number of uses, but specifically in terms of durability, replication lets you maintain two copies of the database in separate geographic locations. In case of catastrophic events, such as fires, earthquakes, or large scale power outages, the replica can be used as a replacement for a disabled cluster.

Subsequent chapters describe snapshots and command logging. The next chapter describes how you can use database replication for disaster recovery. This chapter explains how K-safety works, how to configure your VoltDB database for different values of K, and how to recover in the case of a system failure.

10.1. How K-Safety Works

K-safety involves duplicating database partitions so that if a partition is lost (either due to hardware or software problems) the database can continue to function with the remaining duplicates. In the case of VoltDB, the duplicate partitions are fully functioning members of the cluster, including all read and write operations that apply to those partitions. (In other words, the duplicates function as peers rather than in a master-slave relationship.)

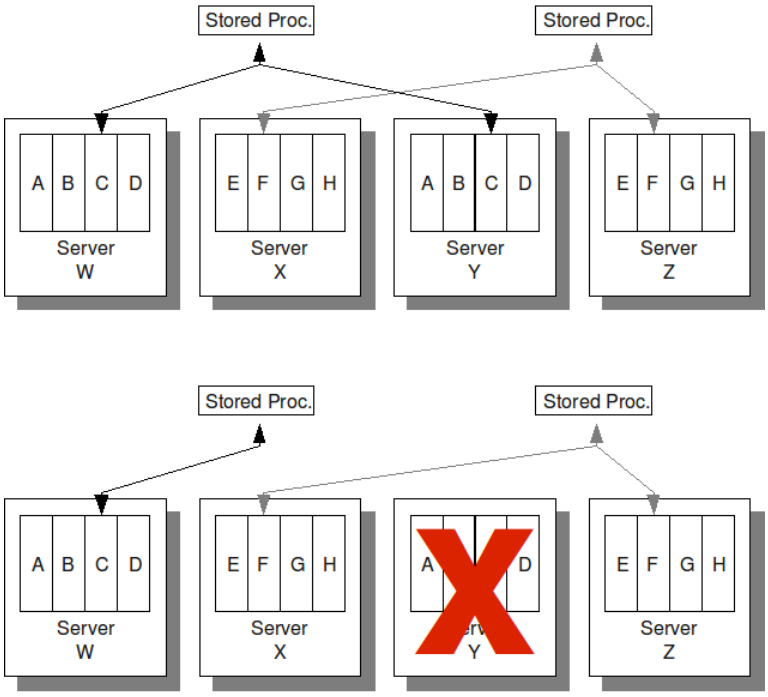
It is also important to note that K-safety is different than WAN replication. In replication the entire database cluster is replicated (usually at a remote location to provide for disaster recovery in case the entire cluster or site goes down due to catastrophic failure of some type).

In replication, the replicated cluster operates independently and cannot assist when only part of the active cluster fails. The replicate is intended to take over only when the primary database cluster fails entirely. So, in cases where the database is mission critical, it is not uncommon to use both K-safety and replication to achieve the highest levels of service.

To achieve K=1, it is necessary to duplicate all partitions. (If you don't, failure of a node that contains a non-duplicated partition would cause the database to fail.) Similarly, K=2 requires two duplicates of every partition, and so on.

What happens during normal operations is that any work assigned to a duplicated partition is sent to all copies (as shown in Figure 10.1, “K-Safety in Action”). If a node fails, the database continues to function sending the work to the unaffected copies of the partition.

Figure 10.1. K-Safety in Action



10.2. Enabling K-Safety

You specify the desired K-safety value as part of the cluster configuration when you initialize the database root directory. By default, VoltDB uses a K-safety value of zero (no duplicate partitions). You can specify a larger K-safety value using the `kfactor` property. For example, in the following configuration file, the K-safety value is set to 2:

```
deployment:
  cluster:
    kfactor: 2
```

When you start the database specifying a K-safety value greater than zero, the appropriate number of partitions out of the cluster will be assigned as duplicates. For example, if you start a cluster with 3 nodes and the default partitions per node of 8, there are a total of 24 partitions. With K=1, half of those partitions (12) will be assigned as duplicates of the other half. If K is increased to 2, the cluster would be divided into 3 copies consisting of 8 partitions each.

The important point to note when setting the K value is that, if you do not change the hardware configuration, you are dividing the available partitions among the duplicate copies. Therefore performance (and

capacity) will be proportionally decreased as K-safety is increased. So running K=1 on a 6-node cluster will be approximately equivalent to running a 3-node cluster with K=0.

If you wish to increase reliability without impacting performance, you must increase the cluster size to provide the appropriate capacity to accommodate for K-safety.

10.2.1. What Happens When You Enable K-Safety

Of course, to ensure a system failure does not impact the database, not only do the partitions need to be duplicated, but VoltDB must ensure that the duplicates are kept on separate nodes of the cluster. To achieve this, VoltDB calculates the maximum number of unique partitions that can be created, given the number of nodes, partitions per node, and the desired K-safety value.

When the number of nodes is an integral multiple of the duplicates needed, this is easy to calculate. For example, if you have a six node cluster and choose K=1, VoltDB will create two instances of three nodes each. If you choose K=2, VoltDB will create three instances of two nodes each. And so on.

If the number of nodes is not a multiple of the number of duplicates, VoltDB does its best to distribute the partitions evenly. For example, if you have a three node cluster with two partitions per node, when you ask for K=1 (in other words, two of every partition), VoltDB will duplicate three partitions, distributing the six total partitions across the three nodes.

10.2.2. Calculating the Appropriate Number of Nodes for K-Safety

By now it should be clear that there is a correlation between the K value and the number of nodes and partitions in the cluster. Ideally, the number of nodes is a multiple of the number of copies needed (in other words, the K value plus one). This is both the easiest configuration to understand and manage.

However, if the number of nodes is not an exact multiple, VoltDB distributes the duplicated partitions across the cluster using the largest number of unique partitions possible. This is the highest whole integer where the number of unique partitions is equal to the total number of partitions divided by the needed number of copies:

```
Unique partitions = (nodes * partitions/node) / (K + 1)
```

Therefore, when you specify a cluster size that is not a multiple of K+1, but where the total number of partitions is, VoltDB will use all of the partitions to achieve the required K-safety value.

Note that the total number of partitions must be a whole multiple of the number of copies (that is, K+1). If neither the number of nodes nor the total number of partitions is divisible by K+1, then VoltDB will not let the cluster start and will display an appropriate error message. For example, if the configuration specifies 3 sites per host and a K-safety value of 1 but the **voltdb start** command specifies a server count of 3, the cluster cannot start because the total number of partitions (3X3=9) is not a multiple of the number of copies (K+1=2). To start the cluster, you must either change the configuration to increase the K-safety value to 2 (so the number of copies is 3) or change the sites per host to 2 or 4 so the total number of partitions is divisible by 2.

Finally, if the configuration specifies a K value higher than the available number of nodes, it is not possible to achieve the requested K-safety. Even if there are enough partitions to create the requested duplicates, VoltDB cannot distribute the duplicates to distinct nodes. For example, if you start a 3 node cluster when the configuration specifies 4 partitions per node (12 total partitions) and a K-safety value of 3, the number of total partitions (12) is divisible by K+1 (4) but not without some duplicates residing on the same node. In this situation, VoltDB issues an error message. You must either reduce the K-safety or increase the number of nodes.

10.3. Recovering from System Failures

When running without K-safety (in other words, a K-safety value of zero) any node failure is fatal and will bring down the database (since there are no longer enough partitions to maintain operation). When running with K-safety on, if a node goes down, the remaining nodes of the database cluster log an error indicating that a node has failed.

By default, these error messages are logged to the console terminal. Since the loss of one or more nodes reduces the reliability of the cluster, you may want to increase the urgency of these messages. For example, you can configure a separate Log4J appender (such as the SMTP appender) to report node failure messages. To do this, you should configure the appender to handle messages of class HOST and severity level ERROR or greater. See the chapter on Logging in the *VoltDB Administrator's Guide* for more information about configuring logging.

When a node fails with K-safety enabled, the database continues to operate. But at the earliest possible convenience, you should repair (or replace) the failed node.

To replace a failed node to a running VoltDB cluster, you restart the VoltDB server process specifying the address of at least one of the remaining nodes of the cluster as the *host*. For example, to rejoin a node to the VoltDB cluster where server5 is one of the current member nodes, you use the following voltdb start command:

```
$ voltdb  start --host=server5
```

If you started the servers specifying multiple hosts, you can use the same **voltdb start** command used to start the cluster as a whole since, even if the failed node is in the host list, one of the other nodes in the list can service its rejoin request.

If the failed server cannot be restarted (for example, if hardware problems caused the failure) you can start a replacement server in its place. Note you will need to initialize a root directory on the replacement server before you can start the database process. You can either initialize the root with the original configuration file. Or, if you have changed the configuration, you can retrieve a copy of the current configuration from one of the other servers with the **voltdb get deployment** command and then using that file to initialize the root directory before starting:

```
$ voltdb init --config=latest-config.yaml
$ voltdb  start --host=server5
```

Note that at least one node you specify in the --host argument must be an active member of the cluster. It *does not* have to be one of the nodes identified as the host when the cluster was originally started.

10.3.1. What Happens When a Node Rejoins the Cluster

When you use **voltdb start** to bring back a server to a running cluster, the node first rejoins the cluster, then retrieves a copy of the database schema and the appropriate data for its partitions from other nodes in the cluster. Rejoining the cluster only takes seconds and once this is done and the schema is received, the node can accept and distribute stored procedure requests like any other member.

However, the new node will not actively participate in the work until a full working copy of its partition data is received. While the data is being copied, the cluster separates the rejoin process from the standard transactional workflow, allowing the database to continue operating with a minimal impact to throughput or latency. So the database remains available and responsive to client applications throughout the rejoin procedure.

It is important to remember that the cluster is not fully K-safe until the restoration is complete. For example, if the cluster was established with a K-safety value of two and one node failed, until that node rejoins and is updated, the cluster is operating with a K-safety value of one. Once the node is up to date, the cluster becomes fully operational and the original K-safety is restored.

10.3.2. Where and When Recovery May Fail

It is possible to rejoin any appropriately configured node to the cluster. It does not have to be the same physical machine that failed. This way, if a node fails for hardware reasons, it is possible to replace it in the cluster immediately with a new node, giving you time to diagnose and repair the faulty hardware without endangering the database itself.

There are a few conditions in which the rejoin operation may fail. Those situations include the following:

- Insufficient K-safety

  If the database is running without K-safety, or more nodes fail simultaneously than the cluster is capable of sustaining, the entire cluster will fail and must be restarted from scratch. (At a minimum, a VoltDB database running with K-safety can withstand at least as many simultaneous failures as the K-safety value. It may be able to withstand more node failures, depending upon the specific situation. But the K-safety value tells you the minimum number of node failures that the cluster can withstand.)

- Mismatched configuration in the root directory

  If the configuration file that you specify when initializing the root directory does not match the current configuration of the database, the cluster will refuse to let the node rejoin.

- More nodes attempt to rejoin than have failed

  If one or more nodes fail, the cluster will accept rejoin requests from as many nodes as failed. For example, if one node fails, the first node requesting to rejoin will be accepted. Once the cluster is back to the correct number of nodes, any further requests to rejoin will be rejected. (This is the same behavior as if you try to start more nodes than specified in the `--count` argument to the **voltdb start** command when starting the database.)

10.4. Avoiding Network Partitions

VoltDB achieves scalability by creating a tightly bound network of servers that distribute both data and processing. When you configure and manage your own server hardware, you can ensure that the cluster resides on a single network switch, guaranteeing the best network connection between nodes and reducing the possibility of network faults interfering with communication.

However, there are situations where this is not the case. For example, if you run VoltDB "in the cloud", you may not control or even know what is the physical configuration of your cluster.

The danger is that a network fault — between switches, for example — can interrupt communication between nodes in the cluster. The server nodes continue to run, and may even be able to communicate with others nodes on their side of the fault, but cannot "see" the rest of the cluster. In fact, both halves of the cluster think that the other half has failed. This condition is known as a *network partition*.

10.4.1. K-Safety and Network Partitions

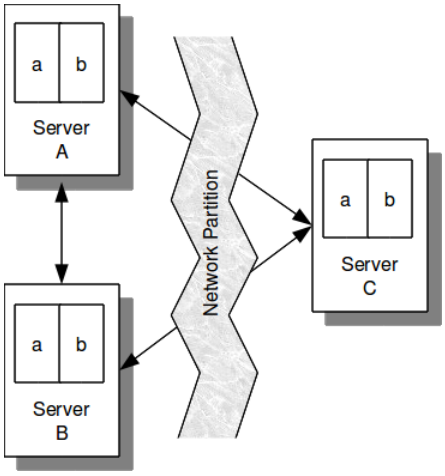
When you run a VoltDB cluster without availability (in other words, no K-safety) the danger of a network partition is simple: loss of the database. Any node failure makes the cluster incomplete and the database

will stop, You will need to reestablish network communications, restart VoltDB, and restore the database from the last snapshot.

However, if you are running a cluster with K-safety, it is possible that when a network partition occurs, the two separate segments of the cluster might have enough partitions each to continue running, each thinking the other group of nodes has failed.

For example, if you have a 3 node cluster with 2 sites per node, and a K-safety value of 2, each node is a separate, self-sustaining copy of the database, as shown in Figure 10.2, “Network Partition”. If a network partition separates nodes A and B from node C, each segment has sufficient partitions remaining to sustain the database. Nodes A and B think node C has failed; node C thinks that nodes A and B have failed.

Figure 10.2. Network Partition



The problem is that you never want two separate copies of the database continuing to operate and accepting requests thinking they are the only viable copy. If the cluster is physically on a single network switch, the threat of a network partition is reduced. But if the cluster is on multiple switches, the risk increases significantly and must be accounted for.

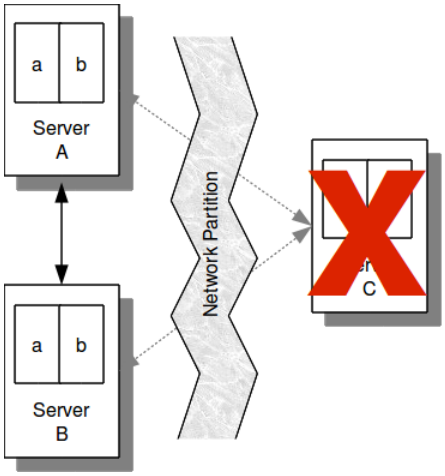
10.4.2. Network Fault Protection

VoltDB provides a mechanism for guaranteeing that a network partition does not accidentally create two separate copies of the database. The feature is called network fault protection. When a fault is detected (either due to a network fault or one or more servers failing), any viable segment of the cluster will perform the following steps:

1. Determine what nodes are missing
2. Determine if the missing nodes are also a viable self-sustained cluster. If so...
3. Determine which segment is the larger segment (that is, contains more nodes).
   - If the current segment is larger, continue to operate assuming the nodes in the smaller segment have failed.
   - If the other segment is larger, shutdown to avoid creating two separate copies of the database.

For example, in the case shown in Figure 10.2, “Network Partition”, if a network partition separates nodes A and B from C, the larger segment (nodes A and B) will continue to run and node C will shutdown (as shown in Figure 10.3, “Network Fault Protection in Action”).

Figure 10.3. Network Fault Protection in Action



If a network partition creates two viable segments of the same size (for example, if a four node cluster is split into two two-node segments), a special case is invoked where one segment is uniquely chosen to continue, based on the internal numbering of the host nodes. Thereby ensuring that only one viable segment of the partitioned database continues.

Network fault protection is a very valuable tool when running VoltDB clusters in a distributed or uncontrolled environment where network partitions may occur. The one downside is that there is no way to differentiate between network partitions and actual node failures. In the case where no network partition occurs but a large number of nodes actually fail, the remaining nodes may believe they are the smaller segment. In this case, the remaining nodes will shut themselves down to avoid partitioning.

For example, in the previous case shown in Figure 10.3, “Network Fault Protection in Action”, if rather than a network partition, nodes A and B fail, node C is the only node still running. Although node C is viable and could continue because the database was configured with K-safety set to 2, with fault protection node C will shut itself down to avoid a partition. For higher values of K-safety, the calculation of which segment stays up becomes more complex since the network topology can change dynamically if network partitions happen in sequence. However, the principle remains the same: to ensure that only one segment continues to run and it is possible all segments will shut down if there is ambiguity about which segments may still be viable. Specifically, if a node loses contact with all other nodes, even if it would normally be the surviving node, if it was previously part of a cluster with three or more nodes, it will shut itself down to avoid a potential network partition.

In the worst case, if half the nodes of a cluster fail, the remaining nodes may actually shut themselves down under the special provisions for a network partition that splits a cluster into two equal parts. For example, consider the situation where a two node cluster with a k-safety value of one has network partition detection enabled. If one of the nodes fails (half the cluster), there is only a 50/50 chance the remaining node is the "blessed" node chosen to continue under these conditions. If the remaining node is *not* the chosen node, it will shut itself down to avoid a conflict, taking the database out of service in the process.

Because this situation — a 50/50 split — could result in either a network partition or a viable cluster shutting down, VoltDB recommends always using clusters with an odd number of nodes when creating small clusters. By using an odd number of servers, you avoid even the possibility of a 50/50 split, whether caused by partitioning or node failures. For larger clusters, the chance of losing exactly half the cluster is relatively small and so an even number of nodes (such as six, eight, or larger) is rarely a problem.

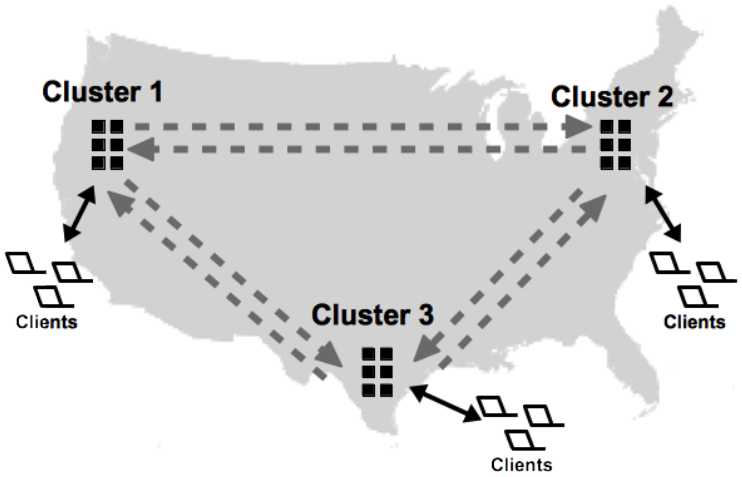
Chapter 11. Active(N) Database Replication

There are times when it is useful to create multiple copies of a database. Not just a snapshot of a moment in time, but live, constantly updated copies.

K-safety maintains redundant copies of partitions within a single VoltDB database, which helps protect the database cluster against individual node failure. Database replication also creates a copy. However, database replication creates and maintains copies in separate, often remote, databases.

Volt Active Data uses Active(N)® to replicate databases. Unlike passive, or one-way database replication where there is a master database and a read-only replica that only receives changes from the master, Active(N) creates multiple active databases (also known as cross datacenter replication, or XDCR) that are synchronized in both directions. Which means you can create an Active(N) network of two or more databases all of which can process database transactions and update the contents simultaneously, and those changes are replicated to all the participating database clusters.

Figure 11.1. Active(N) Cross Datacenter Replication



Of course, you can implement passive replication using Active(N), simply by creating a two cluster relationship and pausing the "replica" cluster to make it read-only. This can be useful for certain business applications, such as disaster recovery, where you want to maintain a read-only copy of the active database in a different geographic location in case the primary database becomes physically disabled in some way. Or if you want to maintain a read-only copy for reporting purposes and to offload these potentially complex queries from the active database.

Active(N) database replication (XDCR) solves two key business problems:

1. Supporting business operations on a global scale 24/7. XDCR lets you globally distribute and operate separate, active, but coordinated copies of a single database in multiple locations. For example, you can maintain copies of a product inventory database at two or more separate warehouses, close to the applications that need the data. This feature makes it possible to support massive numbers of clients that could not be supported by a single database instance or might result in unacceptable latency when the database and the users are geographically separated. The databases can even reside on separate continents.

2. Protecting your business and its data against catastrophic events, such as power outages or natural disasters, capable of taking down an entire cluster or data center. This is often referred to as *disaster recovery*. Because the Active(N) clusters can be in different geographic locations, XDCR allows the other clusters to continue unaffected when one becomes inoperable. XDCR also allows you to offload read-only workloads, such as reporting, from the primary database instances.

It is important to note, however, that database replication is not instantaneous. The transactions are committed locally, then copied to the other database or databases. So when using XDCR to maintain multiple active clusters you must be careful to design your applications to avoid possible conflicts when transactions change the same record in two databases at approximately the same time. See Section 11.2.7, "Understanding Conflict Resolution" for more information about conflict resolution.

The remainder of this chapter discusses the following topics:

- Section 11.1, "How Database Replication Works"
- Section 11.2, "Using Active(N) Cross Datacenter Replication"
- Section 11.3, "Updating the Schema During Replication"
- Section 11.4, "Monitoring Database Replication"
- Section 11.5, "Tuning Active(N) to Meet Your Application Needs"

11.1. How Database Replication Works

Database replication (DR) involves duplicating the contents of selected tables between two database clusters. You identify which tables to replicate in the schema, by specifying the table name in a DR TABLE statement. For example, to replicate all tables in the voter sample application, you would execute three DR TABLE statements when defining the database schema:

```
DR TABLE contestants;
DR TABLE votes;
DR TABLE area_code_state;
```

11.1.1. Starting Database Replication

You enable DR using the `dr` property in the configuration when initializing the database. The subproperties of `dr` identify three pieces of information:

- A unique cluster ID for each database. The ID is required and can be any number between 0 and 127, as long as each cluster has a different ID.
- The database role as *xdcr*.
- A connection source listing the host name or IP address of one or more nodes from the other databases.

For example:

```
deployment:
  dr:
    id: 2
    role: xdcr
    connection:
      source: serverA1,serverA2
```

Each cluster must have a unique ID. All clusters must also include the `connection` property pointing to at least one other cluster. If you are establishing an XDCR network with multiple clusters, the `connection` property can specify hosts from one or more of the other clusters. The participating clusters will coordinate establishing the correct connections, even if the `connection` property does not list them all.

Note that you must specify the cluster `id` property and the `role` before starting each cluster.

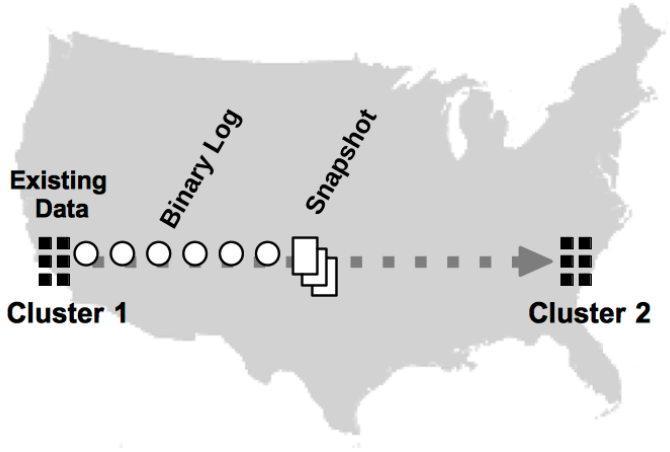
The actual replication process is performed in multiple parallel streams; each unique partition on one cluster sends a binary log of completed transactions to the other clusters. Replicating by partition has two key advantages:

- The process is faster — Because the replication process uses a binary log of the results of the transaction (rather than the transaction itself), the receiving cluster (or *consumer*) does not need to reprocess the transaction; it simply applies the results. Also, since each partition replicates autonomously, multiple streams of data are processed in parallel, significantly increasing throughout.

- The process is more durable — In a K-safe environment, if a server fails on a DR cluster, individual partition streams can be redirected to other nodes or a stream can wait for the server to rejoin — without interfering with the replication of the other partitions.

If data already exists in one of the clusters before database replication starts for the first time, that database sends a snapshot of the existing data to the other, as shown in Figure 11.2, “Replicating an Existing Database”. Once the snapshot is received and applied (and the two clusters are in sync), the partitions start sending binary logs of transaction results to keep the clusters synchronized.

Figure 11.2. Replicating an Existing Database



For XDCR, the first database that is started can have data in the DR tables. If other clusters contain data, replication cannot start. Once DR has started, the databases can stop and recover using command logging without having to restart DR from the beginning.

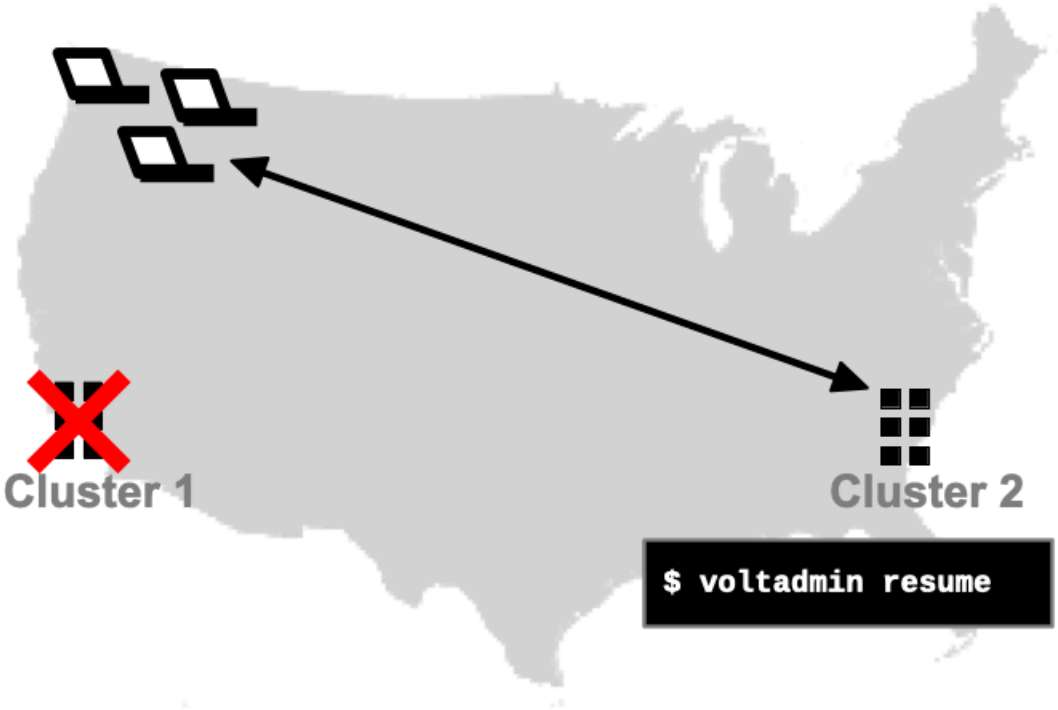
11.1.2. Database Replication, Availability, and Disaster Recovery

Once replication begins, the DR process is designed to withstand normal failures and operational downtime. When using K-safety, if a node fails on any cluster, you can rejoin the node (or a replacement) using the **voltdb start** command without breaking replication. Similarly, if a cluster shuts down, you can use **voltdb start** to restart the database and restart replication where it left off. The ability to restart DR

assumes you are using command logging. Specifically, *synchronous* command logging is recommended to ensure complete durability.

If unforeseen events occur that make a database unreachable, database replication lets you replace the missing database with its copy. This process is known as *disaster recovery*. For cross datacenter replication (XDCR), you simply need to redirect your client applications to the remaining cluster(s). If one of the databases was being used as a read-only copy, you can use the **voltadmin resume** command to switch it from read-only mode to a fully operational database.

Figure 11.3. Disaster Recovery



11.1.3. Database Replication and Completeness

It is important to note that, unlike K-safety where multiple copies of each partition are updated simultaneously, database replication involves shipping the results of completed transactions from one database to another. Because replication happens after the fact, there is no guarantee that the contents of the clusters are identical at any given point in time. Instead, the receiving database (or consumer) "catches up" with the sending database (or producer) after the binary logs are received and applied by each partition.

Also, because DR occurs on a per partition basis, changes to partitions may not occur in the same order on the consumer, since one partition may replicate faster than another. Normally this is not a problem because the results of all transactions are atomic in the binary log. However, if the producer cluster crashes, there is no guarantee that the consumer has managed to retrieve all the logs that were queued. Therefore, it is possible that some transactions that completed on the producer are not reflected on the consumer.

Fortunately, using command logging, when you restart the failed cluster, any unacknowledged transactions will be replayed from the failed cluster's disk-based DR cache, allowing the clusters to recover and resume DR where they left off. To ensure effective recovery, the use of synchronous command logging is recommended. Synchronous command logging guarantees that all transactions are recorded in the command log and no transactions are lost. If you use asynchronous command logging, there is a possibility that a binary log is applied but not captured by the command log before the cluster crashes. Then when the

database recovers, the clusters will not agree on the last acknowledged DR transaction, and DR will not be able to resume. At this point, you will need to decide which cluster to use as the authoritative source, then reinitialize and restart DR from scratch for the other cluster.

11.2. Using Active(N) Cross Datacenter Replication

The following sections provide step-by-step instructions for setting up and running XDCR between two or more VoltDB clusters. The sections describe how to:

1. Design your schema and identify the DR tables
2. Configure the database clusters, including:
   - Choosing unique cluster IDs
   - Identifying the DR connections
3. Start the databases
4. Load the schema and start replication

Later sections discuss other aspects of managing XDCR, including:

- Stopping database replication
- Resolving conflicts

Important

XDCR is a separately licensed feature. If your current VoltDB license does not include a key for XDCR you will not be able to complete the tasks described in this section. See your VoltDB sales representative for more information on licensing XDCR.

11.2.1. Designing Your Schema for Active Replication

To manage XDCR, VoltDB stores a small amount (8 bytes) of extra metadata with every row of data that is shared. This additional space is allocated automatically for any table declared as a DR TABLE on a cluster configured with the `deployment.dr.role` property set to *xdcr*. Be sure to take this additional space requirement into consideration when planning the memory usage of servers participating in an XDCR network.

Next, you must identify which tables you wish to share between the databases. Only the selected tables are copied. You identify the tables in the schema with the DR TABLE statement. For example, the following statements identify two tables to be replicated, the *Customers* and *Orders* tables:

```
CREATE TABLE customers (
  customerID INTEGER NOT NULL,
  firstname VARCHAR(128),
  LASTNAME varchar(128)
);
CREATE TABLE orders (
  orderID INTEGER NOT NULL,
  customerID INTEGER NOT NULL,
  placed TIMESTAMP
```

```
);
DR TABLE customers;
DR TABLE orders;
```

You can identify any regular table, whether partitioned or not, as a DR table, as long as the table is empty. That is, the table must have no data in it when you issue the DR TABLE statement. The important point to remember is that the schema definitions for all DR tables, including the DR TABLE statements, must be identical on all the participating clusters.

11.2.2. Configuring the Database Clusters

The next step is to configure and initialize the database root directories. The database clusters can have different physical configurations (that is, different numbers of nodes, different sites per host, or a different K factor). Identical cluster configurations guarantee the most efficient replication, because the databases do not need to repartition the incoming binary logs. Differing configurations, on the other hand, may incrementally increase the time needed to apply the binary logs.

When initializing the database root directories, you must also enable and configure DR in the configuration, including:

- Choosing a unique ID for each cluster
- Specifying the DR connections

11.2.2.1. Choosing Unique IDs

You enable DR in the configuration using the `deployment.dr` property, including a unique cluster ID for each database cluster. To manage the DR process VoltDB needs to uniquely identify the clusters. You provide this unique identifier as a number between 0 and 127 when you configure the clusters. For example, if we assign ID=1 to a cluster in New York and ID=2 to another in Chicago, their respective configuration files must contain the following properties. You must also specify that the cluster is participating in XDCR by specifying the role. For example:

New York Cluster

```
deployment:
  dr:
    id: 1
    role: xdcr
```

Chicago Cluster

```
deployment:
  dr:
    id: 2
    role: xdcr
```

11.2.2.2. Identifying the DR Connections

For each database cluster, you must also specify the source of replication in the `connection` property. You do this by pointing each cluster to at least one of the other clusters, specifying one or more servers on the remote cluster(s) i the `source` subproperty.

You only need to point each connection source at servers from one of the other clusters, even if more clusters are participating in the XDRC relationship. However, it is a good idea to include them all in the source string so the current cluster is not dependent on the order in which the clusters start.

For example, say there are two clusters. The New York cluster has nodes NYserverA, NYserverB, and NYserverC. While the Chicago cluster has CHIserverX, CHIserverY, and CHIserverZ. The configuration files for the two clusters might look like this:

New York Cluster

```
deployment:
  dr:
    id: 1
    role: xdcr
    connection:
      source: CHIserverX,CHIserverY
```

Chicago Cluster

```
deployment:
  dr:
    id: 2
    role: xdcr
    connection:
      source: NYserverA,NYserverB,NYserverC
```

Note that *both* clusters must have a connection defined for active replication to start. An alternative approach is to initialize the databases leaving the source property empty. You can then update the configuration to add source servers once the database is up and running and the appropriate schema has been applied. For example, you can initialize the cluster with an empty source property:

```
deployment:
  dr:
    id: 1
    role: xdcr
    connection:
      source: ""
```

Then update the source using **voltadmin set** once the database is running:

```
$ voltadmin set deployment.dr.connection.source="CHIserverX,CHIserverY"
```

Once the configuration files have the necessary declarations, you can initialize the root directories on all cluster nodes using the appropriate configuration files:

New York Cluster

```
$ voltdb init -D ~/nydb --config=nyconfig.yaml
```

Chicago Cluster

```
$ voltdb init -D ~/chidb --config=chiconfig.yaml
```

If you then want to add a third cluster to the XDRC relationship (say San Francisco), you can define a configuration file that points at either or both of the other clusters:

San Francisco Cluster

```
deployment:
  dr:
    id: 3
    role: xdcr
```

```
    comnection:
      source: CHIserverX,CHIserverY,NYserverA,NYserverB
```

When configuring three or more XDCR clusters, you also have the option of specifying which cluster a new instance uses as the source for downloading the initial snapshot. For example, if two of the clusters are located in the same physical location, you can specify the cluster ID of a preferred source to reduce the time needed to synchronize the clusters. Note that the preferred source attribute only applies when the database first joins the XDCR environment or if DR is restarted from scratch. When the cluster recovers existing data under normal operation the preferred source is ignored. For example, a second Chicago cluster could specify the cluster ID of the original Chicago database as the preferred source, like so:

2nd Chicago Cluster

```
deployment:
  dr:
    id: 4
    role: xdcr
    comnection:
      source: CHIserverX,CHIserverY,NYserverA,NYserverB
      preferredSource: 2
```

11.2.3. Starting the Database Clusters

Once the servers are initialized with the necessary configuration, you can start the database clusters. However, it is important to note three important points:

- Only one of the clusters can have data in the DR tables when setting up XDCR and that database must be the first in the XDCR network. In other words, start the database containing the data first. Then start and connect a second, empty database to it.
- As soon as the databases start, they automatically attempt to contact each other, verify that the DR table schema match, and start the DR process
- Only one database can join the XDCR network at a time. You must wait for each joining cluster to complete the initial synchronization before starting the next.

Often the easiest method for starting the databases is to:

1. Start one cluster
2. Load the schema (including the DR table declarations) and any pre-existing data on that cluster
3. Once the first cluster is fully configured, start the second cluster and load the schema
4. Once the second cluster finishes synchronizing with the first, start each additional cluster, one at a time.

Using this approach, DR does not start until step #3 is complete and the first two clusters are fully configured. Then any additional clusters are added separately.

You can then start and load the schema on the databases and perform any other preparatory work you require. Then use the **voltadmin set** command to update the connection property for each cluster filling in the `source` property to point at the other cluster. For example:

```
$ voltadmin set deployment.dr.connection.source="CHIserverX,CHIserverY"
```

As soon as the source property is filled out, the DR process will begin for the first pair of clusters. Once the first two clusters synchronize, you can repeat this process, one at a time, with any other participating clusters.

Note

Although the source property can be modified on a running database, the unique cluster ID *cannot* be changed after the database starts. So it is important to include the `dr` property with the unique ID and *xdcr* role when initializing the database root directories.

11.2.4. Loading a Matching Schema and Starting Replication

As soon as the databases start with DR enabled, they attempt to contact a cooperating database to start replication. Each cluster will issue warnings until the schema for the databases match. This is normal and gives you time to load a matching schema. The key point is that once matching schema are loaded on the databases, replication will begin automatically.

When replication starts, the following actions occur:

1. The clusters verify that the DR tables match on both clusters.

2. If data already exists in the DR tables of the first database, that cluster sends a snapshot of the current contents to the other cluster where it is restored into the appropriate tables.

3. Once the snapshot, if any, is restored, both databases (and any other participating clusters) start sending binary logs of changes from DR tables to the other cluster.

If any errors occur during the snapshot transmission, replication stops and must be restarted from the beginning. However, once the third step is reached, replication proceeds independently for each unique partition and, in a K safe environment, the DR process becomes durable across node failures and rejoins as well as cluster shutdowns and restarts.

11.2.5. Stopping Replication

If, for any reason, you need to break replication between the XDCR databases, you can issue the **voltadmin dr reset** command to any cluster. For example, if one of two clusters goes down and will not be brought back online for an extended period, you can issue a **voltadmin dr reset** command on the remaining cluster to tell it to stop queuing binary logs. If not, the logs will be saved on disk, waiting for the other cluster to return, until you run out of disk space.

When using multiple clusters in an XDCR environment, you must choose whether to break replication with all other clusters (**voltadmin dr reset --all**) or with one specific cluster. Breaking replication with all clusters means that all of the other clusters will need to restart DR from scratch to rejoin the XDCR environment. Breaking replication with a single cluster means the remaining clusters retain their XDCR relationship.

If you wish to remove just one, active cluster from the XDCR relationship, you can issue the **voltadmin dr drop** command to the cluster you wish to remove. This command finalizes any remaining DR logs on the cluster and tells all other clusters to break their DR connection with that cluster. If the cluster you want to remove is not currently running, you can issue the **voltadmin dr reset --cluster=*n*** to all the remaining clusters where ***n*** is the cluster ID of the cluster being removed.

However, there is a danger that if you remove a failed cluster from a multi-cluster XDCR environment, the failed cluster may not have sent the same binary logs to all of the other clusters. In which case, when you drop that cluster from the environment, the data on the remaining clusters will diverge. So, using **dr reset --cluster** is recommended only if you are sure that there were no outstanding logs to be sent from the failed cluster. For example, stopping an XDCR cluster with an orderly shutdown (**voltadmin shutdown**) ensures that all its binary logs are transmitted and therefore the other clusters are in sync.

When using the **dr reset --cluster** command, you must also include the **--force** option to verify that you understand the risks associated with this action. So, the process for removing a single, failed cluster from a multi-cluster XDCR environment is:

1. Identify the cluster ID of the cluster that has failed.

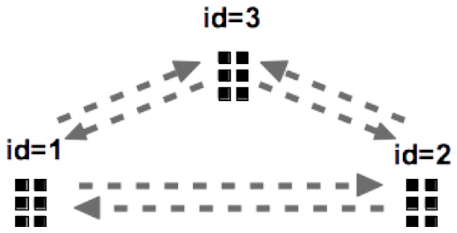
2. Issue the **voltadmin dr reset --cluster=*{failed-cluster-ID}* --force** command on all the remaining clusters to clear the binary log queues.

This way, the remaining clusters can maintain their XDCR relationship but not retain queued data for the failed cluster. If, later, you want to rejoin the failed cluster to the XDCR environment, you will need to reinitialize the failed cluster's root directories and restart its XDCR relationship from scratch.

11.2.6. Example XDCR Configurations

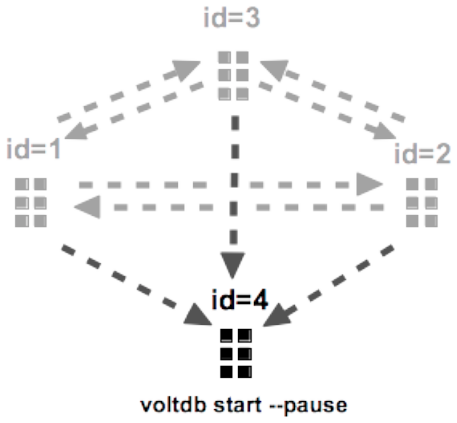
Normally, in an XDCR environment, all cluster participate equally. They can all initiate transactions and replicate those transactions among themselves, as shown in Figure 11.4, “Standard XDCR Configuration”.

Figure 11.4. Standard XDCR Configuration



If you also want to have one (or more) clusters on "standby", for example, purely for disaster recovery or to off-load read-only workloads, you can dedicate clusters from within your XDCR environment for that purpose. The easiest way to do that is to configure the extra clusters as normal XDCR clusters. That is setting their role as "XDCR" and assigning them a unique DR ID. However, rather than starting the clusters in normal operational mode, you can use the `--pause` flag on the **voltdb start** command to start them in admin mode. This way no transactions can be initiated on the cluster's client ports. However, the cluster *will* receive and process DR binary logs from the other clusters in the DR relationship. Figure 11.5, “XDCR Configuration with Read-Only Replicas” demonstrates one such configuration.

Figure 11.5. XDCR Configuration with Read-Only Replicas

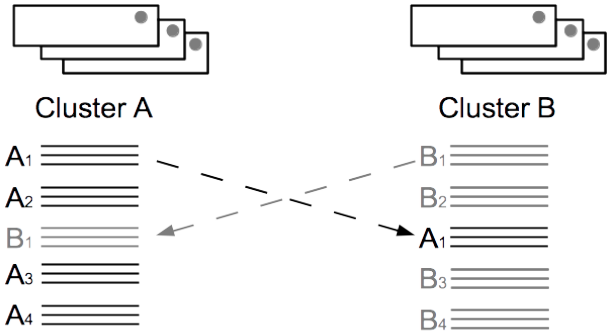


11.2.7. Understanding Conflict Resolution

One aspect of database replication that you need to prepare for and manage is conflicts between the databases. With XDCR replication occurring in both directions, it is possible for changes to be made to the same data at approximately the same time on two databases. Those changes are then sent to the other database, resulting in possible inconsistencies or invalid transactions.

For example, say clusters A and B are processing transactions as shown in Figure 11.6, "Transaction Order and Conflict Resolution". Cluster A executes a transaction that modifies a specific record and this transaction is included in the binary log A_1. By the time cluster B receives the binary log and processes A_1, cluster B has already processed its own transactions B_1 and B_2. Those transactions may have modified the same record as the transaction in A_1, or another record that would conflict with the change in A_1, such as a matching unique index entry.

Figure 11.6. Transaction Order and Conflict Resolution



Under these conditions, cluster B cannot simply apply the changes in A_1 because doing so could violate the uniqueness constraints of the schema and, more importantly, is likely to result in the content of the two database clusters diverging. Instead, cluster B must decide which change takes priority. That is, what resolution to the conflict is most likely to produce meaningful results or match the *intent* of the business application. This decision making process is called *conflict resolution*.

No matter what the resolution, it is important that the database administrators are notified of the conflict, why it occurred, and what action was taken. The following sections explain:

- How to avoid conflicts
- How VoltDB resolves conflicts when they do occur
- What types of conflicts can occur
- How those conflicts are reported

11.2.7.1. Designing Your Application to Avoid Conflicts

VoltDB uses well-defined rules for resolving conflicts. However, the best protection against conflicts and the problems they can cause is to design your application to avoid conflicts in the first place. There are at least two things you can do in your client applications to avoid conflicts:

- **Use Primary Keys**

  It is best, wherever possible, to define a primary key for all DR tables. The primary key index greatly improves performance for finding the matching row to apply the change on a consumer cluster. It is also

required if you want conflicts to be resolved using the standard rules described in the following section. Any conflicting action *without* a primary key is rejected.

- **Apply related transactions to the same cluster**

  Another tactic for avoiding conflicts is to make sure any autonomous set of transactions affecting a set of rows are all applied on the same cluster. For example, ensuring that all transactions for a single user session, or associated with a particular purchase order, are directed to the same cluster.

- **Do not use TRUNCATE TABLE**

  TRUNCATE TABLE is a convenient statement for deleting all records in a table. The statement is optimized to avoid deleting row by row. However, this optimization means that the binary log does not report which rows were deleted. As a consequence, a TRUNCATE TABLE statement can easily produce a conflict between two XDCR clusters that is not detected or reported in the conflict log.

  Therefore, do not use TRUNCATE TABLE with XDCR. Instead, explicitly delete all rows with a DELETE statement and a filter. For example, `DELETE * FROM table WHERE column=column` ensures all deleted rows are identified in the binary log and any conflicts are accurately reported.

  Note that `DELETE FROM table` is *not* sufficient, since its execution plan is optimized to equate to TRUNCATE TABLE. Also, when deleting all rows in a table, it is best to perform the delete in smaller batches to avoid overflowing the maximum size allowed for the binary log packets.

11.2.7.2. How Conflicts are Resolved

Even with the best application design possible, errors in program logic or operation may occur that result in conflicting records being written to two or more databases. When a conflict does occur, VoltDB follows specific rules for resolving the issue. The conflict resolution rules are:

- Conflicts are resolved on a per action basis. That is, resolution rules apply to the individual INSERT, UPDATE, or DELETE operation on a specific tuple. Resolutions are not applied to the transaction as a whole.
- The resolution is that the incoming action is accepted (that is, applied to the receiving database) or rejected.
- Only actions involving a table with a primary key can be accepted, all other conflicting actions are rejected.
- Accepted actions are applied as a whole — the entire record is changed to match the result on the producer cluster. That means for UPDATE actions, all columns are written not just the columns specified in the SQL statement.
- For tables with primary keys, the rules for which transaction wins are, in order:

  1. DELETE transactions always win
  2. If neither action is a DELETE, the last transaction (based on the timestamp) wins

Let's look at a simple example to see how these rules work. Assume that the database stores user records, using a numeric user ID as the primary key and containing columns for the user's name and password. A user logs on simultaneously in two locations and performs two separate updates: one on cluster A changing their name and one on cluster B changing the password. These updates are almost simultaneous. However, cluster A timestamps its transaction as occurring at 10:15.00.003 and cluster B timestamps its transaction at 10:15.00.001.

The binary logs from the two transactions include the type of action, the contents of the record before and after the change, and the timestamps — both of the last previous transaction and the timestamp of the new transaction. (Note that the timestamp includes both the time and the cluster ID where the transaction occurred.) So the two binary logs might look like the following.

Binary Log A_1:

```
Action: UPDATE
Current Timestamp:  1, 10:15.00.003
Previous Timestamp: 1, 06:30.00.000
```

Before	After
UserID: 12345 Name: Joe Smith Password: abalone	UserID: 12345 Name: **Joseph Smith** Password: abalone

Binary Log B_1:

```
Action: UPDATE
Current Timestamp:  2, 10:15.00.001
Previous Timestamp: 1, 06:30.00.000
```

Before	After
UserID: 12345 Name: Joe Smith Password: abalone	UserID: 12345 Name: Joe Smith Password: **flounder**

When the binary log A_1 arrives at cluster B, the DR process performs the following steps:

1. Uses the primary key (12345) to look up the current record in the database.
2. Compares the current timestamp in the database with the previous timestamp in the binary log.
3. Because the transaction in B_1 has already been applied on cluster B, the time stamps do not match. A conflict is recognized.
4. A primary key exists, so cluster B attempts to resolve the conflict by comparing the new timestamp, 10:15.00.003, to the current timestamp, 10:15.00.001.
5. Because the new timestamp is the later of the two, the new transaction "wins" and the change is applied to the database.
6. Finally, the conflict and resolution is logged. (See Section 11.2.7.4, “Reporting Conflicts” for more information about how conflicts are reported.)

Note that when the UPDATE from A_1 is applied, the change to the password in B_1 is overwritten and the password is reset to "abalone". Which at first looks like a problem. However, when the binary log B_1 arrives at cluster A, the same steps are followed. But when cluster A reaches steps #4 and 5, it finds that the new timestamp from B_1 is older than the current timestamp, and so the action is rejected and the record is left unchanged. As a result both databases end up with the same value for the record. Essentially, the password change is dropped.

If the transaction on cluster B had been to delete the user record rather than change the password, then the outcome would be different, but still consistent. In that case, when binary log A_1 reaches cluster B, it would not be able to find the matching record in step #1. This is recognized as a DELETE action having

occurred. Since DELETE always wins, the incoming UPDATE is rejected. Similarly, when binary log B_1 reaches cluster A, the previous timestamps do not match but, even though the incoming action in B_1 has an older timestamp than the UPDATE action in A_1, B_1 "wins" because it is a delete action and the record is deleted from cluster A. Again, the result is consistent across the two databases.

The real problem with conflicts is when there is no primary key on the database table. Primary keys uniquely identify a record. Without a primary key, there is no way for VoltDB to tell, even if there are one or more unique indexes on the table, whether two records are the same record modified or two different records with the same unique key values.

As a result, if there is a conflict between two transactions without a primary key, VoltDB has no way to resolve the conflict and simply rejects the incoming action. Going back to our example, if the user table had a unique index on the user ID rather than a primary key, and both cluster A and cluster B update the user record at approximately the same time, when binary log A_1 arrives at cluster B, it would look for the record based on all columns in the record and fail to find a match.

However, when it attempts to insert the record, it will encounter a constraint violation on the unique index. Again, since there is no primary key, VoltDB cannot resolve the conflict and rejects the incoming action, leaving the record with the changed password. On cluster A, the same process occurs and the password change in B_1 gets rejected, leaving cluster A with a changed name column and database B with a changed password column — the databases diverge.

11.2.7.3. What Types of Conflict Can Occur

The preceding section uses a simple case of conflicting UPDATE transactions to illustrate the steps involved in conflict resolution. However, there are several different types of conflict that can occur. First, there are three possible actions that the binary log can contain: INSERT, UPDATE, or DELETE. There are also three types of conflicts that can be generated:

- **Missing row** — The affected row is missing from the consumer database.
- **Timestamp mismatch** — The affected row exists in the consumer database, but has a different timestamp than expected (in other words, it has been modified).
- **Constraint violation** — Applying the incoming action would result in one or more constraint violations on unique indexes.

A missing row means that the binary log contains an UPDATE or DELETE action, but the affected row cannot be found in the consumer database. (A missing row conflict cannot occur for INSERT actions, since INSERT assumes no such row exists.) In the case of a missing row conflict, VoltDB assumes a DELETE action has removed the affected row. Since the rule is that DELETE wins, this means the incoming action is rejected.

Note that if the table does not have a primary key, the assumption that a DELETE action removed the row is not guaranteed to be true, since it is possible an UPDATE changed the row. Without a primary key, there is no way for the DR process to find the matching row when some columns may have changed, so it assumes it was deleted. As a result, an UPDATE could occur on one cluster and a DELETE on the other. This is why assigning primary keys is recommended for DR tables when using XDCR.

If the matching primary key *is* found, it is still possible that the contents of the row have been changed. In which case, the timestamps will not match and a timestamp mismatch conflict occurs. Again, this can happen for UPDATE and DELETE actions where an existing row is being modified. If the incoming action is a DELETE, it takes precedence and the row is deleted. If not, if the incoming action has the later of the two timestamps, it is accepted. If the existing record has the later timestamp, the incoming action is rejected.

Finally, whether the timestamps match or not, with an INSERT or UPDATE action, it is possible that applying the action would violate one of more unique index constraints. This can happen because another row has been updated with matching values for the unique index or another record has been inserted with similar values. Whatever the cause, VoltDB cannot apply the incoming action so it is rejected. Note that for a single action there can be more than one unique index that applies to the table, so there can be multiple constraint violations as well as a possible incorrect timestamp. When a conflict occurs, all conflicts associated with the action are included in the conflict log.

To summarize, the following chart shows the conflicts that can occur with each type of action and the result for tables with a primary key.

Action	**Possible Conflict**	**Result for Tables with Primary Key**
INSERT	Constraint violation	Rejected
UPDATE	Missing row Timestamp mismatch Constraint violation	Rejected Last transaction wins Rejected
DELETE	Missing row Timestamp mismatch	Accepted (no op) Accepted

11.2.7.4. Reporting Conflicts

VoltDB makes a record of every conflict that occurs when processing the DR binary logs. These conflict logs include:

- The intended action
- The type of conflict
- The timestamp and contents of the row before and after the action from the binary log
- The timestamp and contents of the row(s) in the consumer database that caused the conflict
- The timestamp and cluster ID of the conflict itself

By default, these logs are written as comma-separated value (CSV) files on the cluster where the conflicts occur. These files are usually written to a subfolder of the voltdbroot directory (`voltdbroot/xdcr_conflicts`) using the file prefix `LOG`. However, you can configure the logs to be written to different destinations or locations using the VoltDB export configuration settings.

The DR process writes the conflicts as export data to the export stream VOLTDB_XDCR_CONFLICTS. You do not need to explicitly configure export — the DR process automatically declares the necessary export streams, establishes a default export configuration for the file connector, and enables the export stream. However, if you want the data to be sent to a different location or using a different export connector, you can do this by configuring the export stream yourself.

For example, if you want to export the XDCR conflicts to a Kafka stream where they can be used for automatic notifications, you can change the export properties in the configuration. The following configuration writes the conflict logs to the Kafka topic *sysops* on the broker *kafkabroker.mycompany.com*:

```
deployment:
  export:
    configuration:
    - target: VOLTDB_XDCR_CONFLICTS
      type: kafka
```

```
enabled: true
property:
- name: broker
  value: kafkabroker.mycompany.com
- name: topic
  value: sysops
```

Each action in the binary log can generate one or more conflicts. When this occurs, VoltDB logs the conflict(s) as multiple rows in the conflict report. Each row is identified by the type of action (INSERT, UPDATE, DELETE) as well as the type of information the row contains:

- EXISTING (EXT) — The timestamp and contents of an existing row in the consumer database that caused a conflict. There can be multiple existing row logs, if there are multiple conflicts.
- EXPECTED (EXP) — The timestamp and contents of the row that is expected before the action is applied (from the binary log).
- NEW (NEW) — The new timestamp and contents for the row once the action is applied (from the binary log).
- DELETE (DEL) — For a DELETE conflict, the timestamp and cluster ID indicating when and where the conflict occurred.

For an INSERT action, there is no EXPECTED row. For either an INSERT or an UPDATE action there is no DELETE row. And for a DELETE action there is no NEW row. The order of the rows in the report is as follows:

1. The EXISTING row, if there is a timestamp mismatch
2. The EXPECTED row, if there is a timestamp mismatch
3. One or more EXISTING rows, if there are any constraint violations
4. The NEW row, for all actions but DELETE
5. The DELETE row, for the DELETE action only

Table 11.1, “Structure of the XDCR Conflict Logs” describes the structure and content of the conflict log records in the export stream.

Table 11.1. Structure of the XDCR Conflict Logs

Column Name	Datatype	Description
ROW_TYPE	3 Byte string	The type of row, specified as: `EXT` — existing `EXP` — expected `NEW` — new `DEL` — delete
ACTION_TYPE	1 Byte string	The type of action, specified as: `I` — insert `U` — update `D` — delete
CONFLICT_TYPE	4 Byte string	The type of conflict, specified as:

Column Name	Datatype	Description
		`MISS` — missing row `MSMT` — timestamp mismatch `CNST` — constraint violation `NONE` — no violation[a]
CONFLICTS_ON _PRIMARY_KEY	TINYINT	Whether a constraint violation is associated with the primary key. 1 for true and 0 for false.
DECISION	1 Byte string	How the conflict was resolved, specified as: A — the incoming action is accepted R — the incoming action is rejected
CLUSTER_ID	TINYINT	The DR cluster ID of the cluster that last modified the row
TIMESTAMP	BIGINT	The timestamp of the row.
DIVERGENCE	1 Byte string	Whether the resulting action could cause the two cluster to diverge, specified as: `C` — the clusters are consistent `D` — the cluster may have diverged
TABLE_NAME	String	The name of the table.
CURRENT _CLUSTER_ID	TINYINT	The DR cluster ID of the cluster reporting the conflict.
CURRENT _TIMESTAMP	BIGINT	The timestamp of the conflict.
TUPLE	JSON-encoded string	The schema and contents of the row, as a JSON-encoded string. The column is limited to 1MB in size. If the schema and contents exceeds the 1MB limit, it is truncated.

[a]Update operations are executed as two separate statements: a delete and an insert, where only one of the two statements might result in a violation. For example, the delete may trigger a missing row violation but the insert not generate a violation. In which case the EXT row of the conflict log reports the MISS conflict and the NEW row reports NONE.

11.2.7.5. Managing XDCR Conflict Logs

The XDCR conflict logs provide the information necessary to recover from unexpected conflicts in your application workflow. Of course, not all conflicts that are logged are critical. For example, if a row is deleted simultaneously by two XDCR clusters, one or both of the clusters will log a "missing row" conflict when it receives the matching delete transaction from the other cluster. It is a business decision which conflicts are acceptable and which require intervention.

It is also a business decision how long the conflict logs need to be retained, either for corrective action or as historical records. By default, VoltDB saves all conflict logs. However, over time these add up and put a strain on system resources. So it is a good idea to establish a retention policy for managing the log files.

VoltDB lets you specify a retention period for conflict logs as part of the configuration, when initializing the database. The `deployment.dr.conflictretention` property specifies a time limit, after which old conflict logs are deleted from the system. For example, the following configuration specifies that the conflict logs are kept for 14 days:

```
deployment:
  dr:
```

```
id: 1
role: xdcr
conflictretention: 14d
```

The argument to `conflictretention` is an integer followed by a single character specifying the time unit, where the time unit is *s*, *m*, *h*, *or d* representing seconds, minutes, hours, or days respectively. By default, there is no retention limit and all conflict log files are kept (except on Kubernetes, where a default retention limit of 30 days is applied). Also, `conflictretention` only applies if you *do not* customize the export connector for XDCR conflict logs, as described in Section 11.2.7.4, “Reporting Conflicts”. If you *do* customize the export connector and are using a file exporter, you can use the `retain` subproperty of `configuration` to specify a retention limit.

11.3. Updating the Schema During Replication

SQL statements that modify the database contents, such as DELETE, INSERT, and UPDATE, are transmitted through the DR binary logs; however, schema changes are not. When updating the schema for DR tables, you must make the changes to each database separately.

By default, if the schema of the tables do not match when the results of a transaction are received as binary logs, the consumer will reject the change, causing the producer (that is the partition on the cluster that sent the problematic binary log) to halt replication until the schema mismatch is resolved. In the best case, there are mismatched transactions in only one direction (that is, from cluster B to cluster A). If so, once you update the schema on the stalled consumer cluster A, replication resumes and cluster B can send the subsequent transactions it had buffered.

In XDCR, while binary logs from the producer are stalled, the consumer continues to process client transactions itself and will send those transactions as binary logs to the other cluster. That is, cluster A also acts as a producer sending binary logs to cluster B as a consumer. If there are simultaneous write transactions to the same table on the two clusters while the schema do not match, a deadlock can result. Both clusters will stall due to mismatched schema and their content will have diverged. In this situation, your only option is to choose one of the clusters as the "winner" and reinitialize the other cluster and restart XDCR from scratch.

When planning schema changes you must be careful to avoid passing incompatible changes between the databases while the schema of the two or more clusters do not match.

There are three alternatives that allow you to update the schema while the database is running:

- Pause the databases to safely modify the schema while transactions are paused
- Carefully add or remove tables without pausing
- Use dynamic schema change to add, remove, or modify columns in a table without pausing

The following sections describe each of these approaches.

11.3.1. Safely Updating the Schema While the Clusters are Paused

The safest process for changing the schema for DR tables is to:

1. Pause and drain the outstanding DR binary logs on *all* clusters using the **voltadmin pause --wait** command
2. Update the schema for the DR tables on all clusters

3. Resume all clusters using the **voltadmin resume** command

This process ensures that no transactions are processed until the schema on the clusters are updated and in sync. However, this process also means that there are no client transactions processed during the update. So this is the safest approach, but also has the largest negative impact on ongoing transactions.

11.3.2. Adding and Removing Tables Without Pausing

Because schema validation occurs on a per table, per transaction basis, it is possible to update the schema without pausing the database. However, this only works if you ensure that no client transactions attempt to modify the affected tables while the schema differ. If any transactions attempt to write to an affected table while the schema differ, the consumer will stall until the schema match.

For example, it is possible to add tables to the database schema without pausing the database. You can add the new tables to the databases in one step, then update the stored procedures and client applications in a second step. This way no client applications access the new tables until their schema exist and match on all of the XDCR databases. At the same time, ongoing transactions associated with older tables are not impacted.

You can even modify existing tables without pausing the database. But in this case you must be much more careful about avoiding operations that access the affected tables during the transition. One way to do this is to create a new table, matching the existing table but with the desired changes. Update the schema on both clusters, then update the client applications and stored procedures to use the new table. Finally, once all client applications are updated, the original table can be deleted.

11.3.3. Using Dynamic Schema Change to Add, Remove, and Modify Table Columns Without Pausing

It is possible to make additional schema changes in an XDCR environment without pausing the databases and while continuing to process transactions during the schema transition. This is referred to as dynamic schema change. However, because dynamic schema change introduces additional risk of the database contents diverging, the feature is not enabled by default. To use dynamic schema change you must configure the database server to allow schema change when you first start it. (Enabling schema change can also be done when reinitializing servers as part of the software upgrade process when upgrading the VoltDB software.)

Dynamic schema change lets you:

- Add a column to the end of an existing DR table, as long as it has a default value
- Delete the last column in the table, as long as it has a default value
- Modify the length of a VARCHAR column

The following sections describe the process for enabling and using dynamic schema change.

11.3.3.1. Configuring VoltDB Databases to Allow Dynamic Schema Changes

You enable dynamic schema change by setting the `deployment.dr.schemachang.enabled` property to *true* in the configuration. The `enabled` property takes a value of "true" or "false", with a default value of false. In other words, changing the schema and passing binary logs containing tuples with differ-

ent schema will continue to break replication as in previous versions unless you explicitly enable schema change in the configuration. The following configuration demonstrates how to enable dynamic schema change:

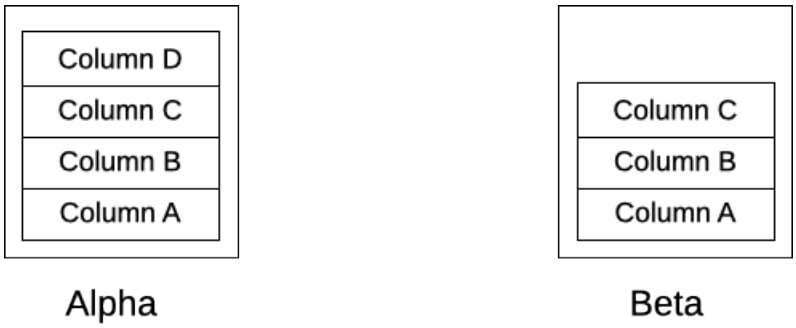
```
deployment:
  dr:
    id: 1
    role: xdcr
    schemachange:
      enabled: true
    connection:
      source: paris.mycompany.com,rome.mycompany.com
```

You must configure the XDCR schema settings when you initialize the database. You cannot change the setting once the database has started.

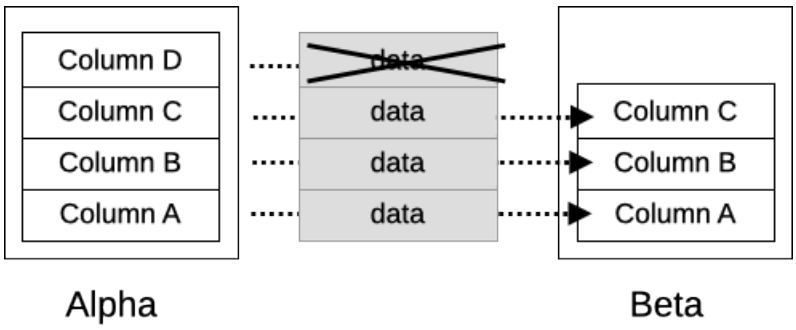
11.3.3.2. How Dynamic Schema Change Works

Normally, if you change the schema for a table in an XDCR environment, as soon as a record is passed from one cluster to another where the schema do not match, the clusters stop replication to avoid the possibility of their data diverging. When you enable dynamic schema change, the clusters do *not* stop replication if the differences are adding or removing the last column or changing the length of VARCHAR columns. Instead, when tuples are received that do not match, XDCR uses a set of rules to accommodate the changes.

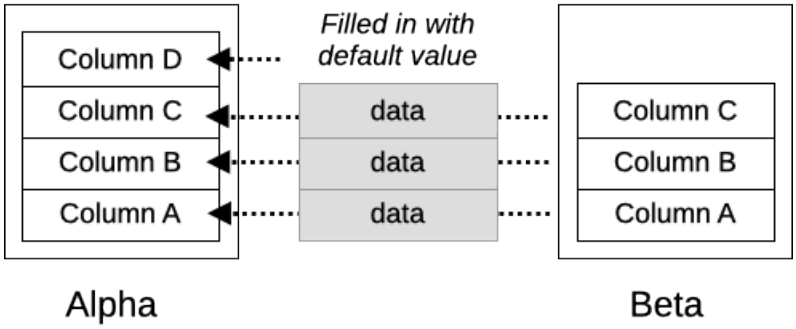
For example, if there are two XDCR clusters, Alpha and Beta, and you modify the schema to add a new column with a default value to one of the tables. When you apply the schema change to cluster Alpha, the schema on the two clusters look like this:



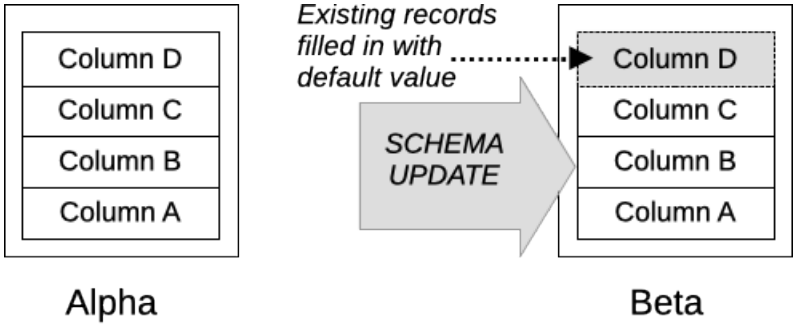
When a record is inserted into cluster Alpha, data for all four columns in the updated table are sent as a binary log to cluster Beta. But since Beta has the old schema, the data for the new column is dropped when the binary log is applied.



Similarly, when data is inserted into Alpha during the transition, the logs passed to cluster Alpha only have three columns of data. So Volt uses the default value for the new column while applying the binary log.



This means that the data exchanged between the two clusters do not match while the schema do not match. However, when you apply the schema change to cluster Beta, the new column of existing records gets filled in with the default value. (This is why a default value is required when adding or dropping columns as part of dynamic schema change.) As a result, as long as all of the records passed from Alpha to Beta during the transition period use the default value for the new column, the content of the databases match once the schema change is complete.



Schema changes that remove columns from the table work on the same principle: binary logs containing fewer columns than the current schema are filled in with the default value while logs containing more columns than the current schema have the extra column's data dropped.

Changing the length of VARCHAR columns is slightly different. First, when reducing the length of a column, you can only make the schema change when the table is empty. This is true whether the cluster is configured for XDCR or not. So the primary rule is to make sure you update the schema on all of the clusters before any data is inserted into the table. Or else you will not be able to complete the schema change process.

When lengthening a VARCHAR column, you must be careful not to insert any data that will exceed the original length of the column. If you do insert data that is too long before all of the clusters' schema are updated, a cluster using the older schema will receive a binary log that is too long for its current schema and will break replication.

11.3.3.3. The Risks Associated with Dynamic Schema Change

Allowing a limited set of schema changes that can be applied dynamically provides a way to adjust your schema without interrupting ongoing processing. It also ensures that the resulting database contents are in sync at the conclusion of the schema change *as long as you follow a set of simple rules*.

However, dynamic schema change, even under the current limitations, introduce additional risk. If client applications attempt to use the new columns for non-default values during the transition between when the database schema change starts and when all of the clusters' schema are updated, the databases will diverge without warning.

The primary risk is if, during the addition of a column, clients either insert or update the table using non-default values for the additional column, the clusters will silently diverge. For example, let's say column D is added to cluster Alpha with a default value of zero (0). Before cluster Beta is updated, an insert transaction adds a record to the table on Alpha with a value of five (5) for the new column. When that record is sent to Beta, which does not have the new column, the value is dropped. When the schema on Beta is finally updated, the new column is filled in with the default value zero, not five which is the value Alpha has for that record.

The problem is that cluster Beta has no way to tell if the data being dropped during the transition is a default or non-default value. So there are no entries in the conflict log for this event.

Note that this issue is specific to adding columns. When dropping a column, even if non-default value are inserted during the transition, ultimately all clusters will drop the affected column and its associated data, so the divergence is resolved once the schema for all clusters are updated.

When increasing the length of VARCHAR columns there is a risk that during the transition, a client may insert a record that exceeds the original limit into cluster Alpha that accepts the longer length. Fortunately, in this situation, VoltDB does not allow VARCHAR values that exceed the declared limit for the column. So when the cluster Beta receives the binary log with a value that exceeds the column's limit, it will break replication.

The good news is that VoltDB recognizes the problem and avoids any divergence. The bad news is the replication stops and you must reinitialize and restart one of the clusters to reestablish XDCR communication.

Finally, you cannot decrease the length of a VARCHAR column unless the table is empty. So, under normal circumstances, the table is empty when you start the schema change and no records should be written to it until the change is complete on all clusters.

11.3.3.4. Rules for Dynamically Updating Schema in an XDCR Environment

Although dynamic schema change does introduce potential dangers, those dangers are easily avoided by following a few very simple rules when modifying the schema of XDCR clusters:

- When adding a column to a table, do not insert data into the new column until the schema change is complete. (That is, the schema has been updated on all clusters.) This ensures that all records added or modified during the transition period receive the default value for the new field.

- There are no specific requirements when dropping a column from a table. However it is good practice to update your client applications to remove any reference to the column *before* beginning the schema update, to avoid unnecessary run-time errors when the applications attempt to select, insert, or update the column being removed.

- When extending the length of a VARCHAR column, make sure your applications do not make use of the additional space until *after* the schema update is complete.

- When reducing the length of a VARCHAR column the table must be empty, so be sure your applications do not attempt to write any records into the table until the update is complete.

11.4. Monitoring Database Replication

Database replication runs silently in the background. To ensure replication is proceeding effectively, VoltDB provides statistics on the producer and consumer clusters that help you understand the current state of the DR process. Specifically, the statistics can tell you:

- The amount of DR data waiting to be sent from the producer
- The timestamp and unique ID of the last transaction received by the consumer
- Whether any partitions are "falling behind" in processing DR data

This information is available from the @Statistics system procedure using the DRROLE, DRCONSUMER, and DRPRODUCER selectors. In XDCR, all clusters act as both producer and consumer and provide statistics on both roles:

- The @Statistics DRROLE procedure provides summary information about the database's DR role (xdcr or none), the cluster ID, and the current state of the DR process.
- On the producer database, the @Statistics DRPRODUCER procedure includes columns for the cluster IDs of the current cluster and the consumer, as well as the transaction ID and timestamp of the last queued transaction and for the last transaction ACKed by the consumer. The difference between these two events can tell you the approximate latency between the two databases.
- On the consumer database, the @Statistics DRCONSUMER procedure includes statistics, on a per partition basis, showing whether it has an identified "host" server from each producer cluster "covering" it, or in other words, providing it DR logs. The system procedure results also include columns listing the ID and timestamp of the last received transaction for each producer cluster. If a consumer partition is not covered, it means it has lost contact with the server on the producer database that was providing it logs (possibly due to a node failure). It is possible for the partition to recover, once the covering server rejoins. However, the difference between the last received timestamp of that partition and the other partitions may give you an indication of how long the interruption has persisted and how far behind that partition may be.

11.5. Tuning Active(N) to Meet Your Application Needs

Active(N) database replication is designed to accommodate different workloads and will provide outstanding performance for most active database applications. However, the default settings may not meet the needs of all business use cases. There are a number of configuration options available to help you optimize replication to meet your specific needs. Before adjusting the configuration, you need to understand how the DR consumers and producers interoperate so you can understand what effect each option has on the flow of data between the clusters.

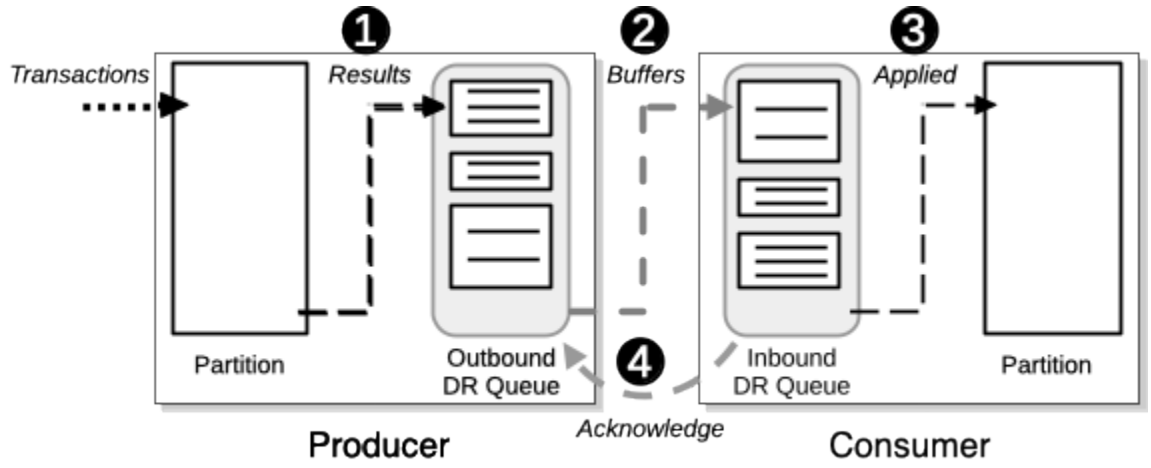
The following sections explain the XDCR workflow process and how to detect and resolve possible bottlenecks in runtime performance.

11.5.1. The XDCR Workflow

In an Active(N) environment, each unique partition within the cluster acts as both a DR producer and consumer. As a producer, the cluster sends the results of any transactions that modify the database content

to the participating clusters. At the same time, as a consumer the cluster receives transaction results from the other clusters and applies them to its copy of the database. The producer and consumer operations are performed on a per partition basis, as shown in Figure 11.7, “The XDCR Producer/Consumer Workflow”.

Figure 11.7. The XDCR Producer/Consumer Workflow



The workflow consists of four main steps:

1. As transactions are processed on the producer cluster, results of transactions that alter the database contents are added as binary logs to buffers in the producer's outbound queue.
2. The buffers are then sent to the consumer cluster (or multiple consumers if there are three or more clusters in the quorum). When the consumer receives the buffers, they are added to the consumer's inbound queue.
3. The buffers are then processed in sequential order, applying the binary logs to the consumer's database contents.
4. Once a buffer is processed, acknowledgement is sent back to the producer cluster so it knows that the data has been successfully processed and the buffer can (once all consumer clusters acknowledge it) be deleted from the queue.

The workflow is optimized to allow each component to operate independently, maximizing efficiency:

- The size of transaction results vary dramatically — from tens of bytes to as much as 37 megabytes — depending on how many records are affected, the size of the records, etc. As a result, the number of transactions in any given buffer can also vary, as can the size of the buffer itself.

- When the producer sends buffers it sends all of the available buffers not already acknowledged by the consumer.

- On the receiving end, when the consumer receives buffers from the producer it places as many buffers as it can fit on the inbound queue. If the queue limit is too low to accommodate all of the buffers, the consumer simply drops the excess buffers. This means the producer will resend the dropped buffers in the next packet it sends, which causes some redundancy. However, the additional network bandwidth is usually a minimal cost to avoid any temporary delays in the consumer causing backups in the producer.

- Even if a buffer is added to the inbound queue, it is possible the consumer does not have time to apply and acknowledge the transactions before the producer sends the next batch of buffers. When this happens, the same buffer may be sent twice. The consumer simply adds the duplicate buffer to the queue and when it comes time to apply it, recognizes that the buffer has already been applied, throws it away and moves on to the next buffer in the queue.

11.5.2. Detecting and Correcting Bottlenecks in the XDCR Workflow

Under normal conditions, there will be occasional buffers discarded by the consumer and some sent multiple times from the producer. This is not a problem because the workflow accommodates the exceptions and allows each cluster to perform its functions at maximum speed. The producer periodically sending too many buffers or duplicate buffers does not result in any noticeable performance impact.

However, if there are continual or increasingly frequent buffers rejected or resent, the condition points to bottlenecks in the workflow that can result in delays in data replication. Volt provides metrics that help you detect and correct these situations. In particular, there are three statistics that help detect issues with XDCR:

- **Buffers ignored by the consumer** — This is the number of excess buffers the consumer receives from the producer but cannot fit within the consumer's inbound queue. The value is returned in the IGNORED_BUFFERS column returned by the @Statistics DRCONSUMER selector.

- **Duplicate buffers dropped by the consumer** — This is the number of duplicate buffers that are sent to the consumer, added to the queue, then discarded when it comes time to apply them since the associated transaction results have already been applied. The value is returned in the DUPLICATE_BUFFERS column returned by the @Statistics DRCONSUMER selector

- **Latency between the producer and the consumer** — There are four columns available on the producer that provide information on the latency of DR transactions; that is, how long it takes for a transaction to be sent to the consumer, applied, and then acknowledged back to the producer. There is both a maximum and average value in milliseconds, measured both over the past minute and past five minutes. These DR_ROUNDTRIPTIME_* columns of the @Statistics DRPRODUCER selector can help you determine if there are delays in the workflow and, with the other statistics, where those delays may be.

These statistics are available from the @Statistics system procedure or from the corresponding Prometheus metrics, as described in the *Volt Administrator's Guide*. By tracking these statistics over time, you can detect potential issues in the replication workflow, as described in the following sections.

11.5.2.1. Not Enough Space in the DR Consumer Queue

If the number of ignored buffers (IGNORED_BUFFERS) is high or steadily increasing, while the number of duplicate buffers remains low, it means that the maximum queue size for the consumer is not large enough to hold the amount of buffered DR transactions being sent by the producer(s). By increasing the maximum size you can provide the space needed to hold the volume of incoming buffers.

You specify the size in bytes[1] as part of the connection configuration, using the consumerlimit.`maxsize` property. For example, the following configuration doubles the consumer queue from the default of 50MB (52428800 bytes) to 100MB :

```
deployment:
  dr:
    id: 1
    role: xdcr
    connection:
      source: chicago,boston
    consumerlimit:
      maxsize: 104857600
```

[1] It is also possible to specify the consumer queue size in number of buffers. However, that is not recommended since buffers can vary dramatically in size. Specifying the queue size in bytes rather than buffers gives you control over how much memory may be consumed at run time.

Of course, increasing the queue size can reduce the number of ignored buffers, but it also increases memory utilization across the cluster. By specifying the limit in bytes you can estimate how much of an increase that will be, since the total memory required for the consumer queues is approximately the size of the queue limit in bytes times the number of unique partitions on the cluster. The calculation is not exact because if there is any room in the queue when the consumer receives buffers from the producer, at least one buffer will be placed on the queue, even if that buffer extends the queue beyond its configured size. So be sure to allow some additional space for this overflow.

You can increase the consumer queue size on a running cluster using the **voltadmin update** or **set** command. After making the adjustment, be sure to measure the number of ignored buffers again to verify that the change has the desired effect and the count of ignored buffers has gone down.

11.5.2.2. Unbalanced Workload or Servers

If the duplicate buffer count (DUPLICATE_BUFFERS) is high, and especially if it is increasing over time, it means that the consumer cluster is not able to keep up processing the incoming buffers at the same rate they are being generated. There are several possible causes for this situation and the resolution depends on the cause.

- First and foremost, check the ignore buffers to make sure the consumer queue is large enough to handle the incoming traffic.
- Next, make sure the clusters are homogeneous; that is, they have the same physical configuration or at least the same number of unique partitions. If the clusters have a different partition count, the DR consumer will have to redistribute the transaction results it receives from the producer to the appropriate partitions within the consumer cluster, significantly increasing the time needed to apply the buffers. Heterogeneous clusters may have a significant impact on the latency of DR processing and are therefore not generally recommended for production use, except temporarily during elastic expansion or contraction.
- If the cluster are homogeneous and have a suitably sized consumer queue, the next step is to check to see if undue latency is delaying the delivery of buffers to the inbound queue, in particular, look for network delays, as described in Section 11.5.2.3, “Too Much Latency Between Clusters”.
- If no other causes can be found, check for conflicts on the consumer cluster itself that could delay DR processing. Issues to look for are:
  - Non-Volt processes competing for system resources,
  - Frequent or long-running multi-partition procedures on the consumer blocking the individual partitions from applying the inbound DR buffers
  - An imbalance in the workload across the XDCR clusters

11.5.2.3. Too Much Latency Between Clusters

The DR_ROUNDTRIPTIME_* columns returned by the @Statistics DRPRODUCER selector tell you how long it is taking for transactions to be sent from the producer to the consumer, be applied, and then acknowledged. If the consumer queue is too small or other activity on the server is delaying the processing of buffers in the DR consumer queue, this latency will be reflected in the round trip times reported by @Statistics, as well as in the IGNORED_BUFFERS and DUPLICATE_BUFFERS counts reported by the @Statistics DRCONSUMER selector.

However, if the consumer buffer is not overflowing and DR transaction results are being applied efficiently but the latency reported by the round trip time is still high, there are two other possible issues that need to be addressed. The first is to investigate whether the latency is due to slow or erratic network performance.

You can use standard networking utilities such as ping to evaluate the network performance between the clusters.

The other situation that can result in unexpected latency in DR processing is if the volume of write transactions is so low or erratic, there is not enough traffic to actually fill the outbound buffers in a timely fashion. In this case the producer waits for additional transaction results to fill the buffer before adding it to the outbound queue. Eventually, even if no additional transactions arrive, the producer will submit whatever transaction results are waiting into a buffer and add them to the queue. However, the period between when a transaction is executed and when the buffer is finally submitted will result in increased latency. For workloads of this nature, there is a setting that helps reduce the latency between transaction execution and adding the results to the outbound queue and this is the DR flush interval.

The flush interval specifies a timeout period where, even if there are not enough results to fill a buffer, the results are submitted to the queue. The flush interval defaults to one second, but if you have an low or erratic workload you can specify a shorter flush interval using the `deployment.systemsettings.flushinterval.dr.interval` property. For example, the following configuration sets the flush interval for XDCR to half a second. Note that you must make sure the global minimum flush interval (`deployment.systemsettings.flushinterval.minimum`) is less than or equal to the new DR flush interval, or else the global minimum will override your DR setting:

```
deployment:
  systemsettings:
    flushinterval:
      minimum: 400
      dr:
        interval: 500
```

Chapter 12. Security

Security is an important feature of any application. By default, VoltDB does not perform any security checks when a client application opens a connection to the database or invokes a stored procedure. This is convenient when developing and distributing an application on a private network.

However, on public or semi-private networks, it is important to make sure only known client applications are interacting with the database. VoltDB lets you control access to the database through settings in the schema and configuration. The following sections explain how to enable and configure security for your VoltDB application.

12.1. How Security Works in VoltDB

When an application creates a connection to a VoltDB database (using ClientFactory.clientCreate), it passes a username and password as part of the client configuration. These parameters identify the client to the database and are used for authenticating access.

At runtime, if security is enabled, the username and password passed in by the client application are validated by the server against the users defined in the configuration. If the client application passes in a valid username and password pair for an account that has not expired, the connection is established. When the application calls a stored procedure, permissions are checked again. If the schema identifies the user as being assigned a role having access to that stored procedure, the procedure is executed. If not, an error is returned to the calling application.

Note

VoltDB uses SHA-256 hashing rather than encryption when passing the username and password between the client and the server. The passwords are also hashed within the database. To secure the actual communication between the server and client, you can implement either Transport Layer Security (TLS) or Kerberos security. Use of TLS is described in Section 12.7, “Encrypting VoltDB Communication Using TLS/SSL” while the use of Kerberos with VoltDB is described in Section 12.9, “Integrating Kerberos Security with VoltDB”.

There are three steps to enabling security for a VoltDB application:

1. `Set the deployment.security.enabled` property to *true* in the configuration to turn on authentication and authorization.

2. Define the users and roles you need to authenticate.

3. Define which roles have access to each stored procedure.

The following sections describe each step of this process, plus how to enable access to system procedures and ad hoc queries.

12.2. Enabling Authentication and Authorization

By default VoltDB does not perform authentication and client applications have full access to the database. To enable authentication, add the `security` property and its subproperties to the configuration. You can enable security when you initialize the database root directory, or you can use **voltadmin set** to change

the security settings on the running database. (Or you can change the setting interactively through the Volt Management Center.)

```
deployment:
  security:
    enabled: true
```

12.3. Defining Users and Roles

The key to security for VoltDB applications is the users and roles defined in the schema and configuration. You define users in the configuration and roles in the schema.

This split is deliberate because it allows you to define the overall security structure globally in the schema, assigning permissions to generic roles (such as operator, dbuser, apps, and so on). You then define specific users and assign them to the generic roles as part of the database configuration. This way you can create one configuration (including cluster information and users) for development and testing, then move the database to a different configuration and a different set of users for production by changing only one file: the configuration file.

You define users within the `deployment.users` property in the configuration. The syntax for defining users is as follows.

```
deployment:
  users:
  - name: user-name
    password: password-string
    roles: role-name[,...]
    expires: expiration-date
```

Note

If you do not want to distribute the account passwords in plain text, you can use the **voltdb mask** command to hash the passwords in the configuration file.

Include a `user` list element for every username/password pair you want to define. You specify which roles a user belongs to as part of the user definition in the configuration using the `roles` property. You can assign users built-in roles, user-defined roles, or both. For user-defined roles, you define the roles in the database schema using the CREATE ROLE statement.

```
CREATE ROLE role-name;
```

You can optionally add an expiration date for the account using the `expires` property and specifying a date in ISO 8601 format (i.e. *yyyy-MM-dd*). If an expiration date is included, the account loses access to the database after midnight of the specified date, until an administrator resets the account properties in the configuration using the **voltadmin update** or **set** command.

Note that, since the user can no longer access the database once their password expires, it is strongly suggested you do *not* set an expiration date on accounts with the ADMINISTRATOR role. If the passwords expire on all accounts with the ADMINISTRATOR role, there will be no way to access the database for administrative functions and you may have to reinitialize the database from scratch to regain access.

At least one user must be assigned the built-in ADMINISTRATOR role. For example, the following code defines three users, assigning operator the built-in ADMINISTRATOR role and the user-defined OPS role, assigning developer the user-defined roles OPS and DBUSER, and assigning the user clientapp DBUSER. When a user is assigned more than one role, you specify the role names as a comma-delimited list.

```
deployment:
  users:
  - name: operator
    password: mech
    roles: administrator,ops
  - name: developer
    password: tech
    roles: ops,dbuser
    expires: 2029-02-17
  - name: clientapp
    password: xyzzy
    roles; dbuser
    expires: 2028-01-29
```

Four important notes concerning the assignment of users and roles:

- Users must be assigned at least one role, or else they have no permissions. (Permissions are assigned by role.)
- At least one user must be assigned the built-in ADMINISTRATOR role.
- At least one user with the ADMINISTRATOR role should be defined *without* an expiration date to avoid losing access to the database.
- There must be a corresponding role defined in the schema for any user-defined roles listed in the configuration.

12.4. Assigning Access to Stored Procedures

Once you define the users and roles you need, you assign them access to individual stored procedures using the ALLOW clause of the CREATE PROCEDURE statement in the schema. In the following example, users assigned the roles dbuser and ops are permitted access to both the MyProc1 and MyProc2 procedures. Only users assigned the ops role have access to the MyProc3 procedure.

```
CREATE PROCEDURE ALLOW dbuser,ops FROM CLASS MyProc1;
CREATE PROCEDURE ALLOW dbuser,ops FROM CLASS MyProc2;
CREATE PROCEDURE ALLOW ops FROM CLASS MyProc3;
```

Usually, when security is enabled, you must specify access rights for each stored procedure. If a procedure declaration does not include an ALLOW clause, no access is allowed. In other words, calling applications will not be able to invoke that procedure.

12.5. Assigning Access by Function (System Procedures, SQL Queries, and Default Procedures)

It is not always convenient to assign permissions one at a time. You might want a special role for access to all user-defined stored procedures. Also, there are special capabilities available within VoltDB that are not called out individually in the schema so cannot be assigned using the CREATE PROCEDURE statement.

For these special cases VoltDB provides named permissions that you can use to assign functions as a group. For example, the ALLPROC permission grants a role access to all user-defined stored procedures so the role does not need to be granted access to each procedure individually.

Several of the special function permissions have two versions: a full access permission and a read-only permission. So, for example, DEFAULTPROC assigns access to all default procedures while DEFAULT-

PROCREAD allows access to only the read-only default procedures; that is, the *TABLE*.select procedures. Similarly, the SQL permission allows the user to execute both read and write SQL queries interactively while SQLREAD only allows read-only (SELECT) queries to be executed.

One additional functional permission is access to the read-only system procedures, such as @Statistics and @SystemInformation. This permission is special in that it does not have a name and does not need to be assigned; all authenticated users are automatically assigned read-only access to these system procedures.

Table 12.1, "Named Security Permissions" describes the named functional permissions.

Table 12.1. Named Security Permissions

Permission	Description	Inherits
DEFAULTPROCREAD	Access to read-only default procedures (*TABLE*.select)	
DEFAULTPROC	Access to all default procedures (*TABLE*.select, *TABLE*.insert, *TABLE*.delete, *TABLE*.update, and *TABLE*.upsert)	DEFAULTPROCREAD
SQLREAD	Access to read-only ad hoc SQL queries (SELECT)	DEFAULTPROCREAD
SQL	Access to all ad hoc SQL queries and default procedures	SQLREAD, DEFAULTPROC
ALLPROC	Access to all user-defined stored procedures	
ADMIN	Full access to all system procedures, all user-defined procedures, as well as default procedures, ad hoc SQL, and DDL statements.	ALLPROC, DEFAULTPROC, SQL
Note: For backwards compatibility, the special permissions ADHOC and SYSPROC are still recognized. They are interpreted as synonyms for SQL and ADMIN, respectively.		

In the CREATE ROLE statement you enable access to these functions by including the permission name in the WITH clause. (The default, if security is enabled and the keyword is not specified, is that the role is not allowed access to the corresponding function.)

Note that the permissions are additive. So if a user is assigned one role that allows access to SQLREAD but not DEFAULTPROC, but that user is also assigned another role that allows DEFAULTPROC, the user has both permissions.

The following example assigns full access to members of the ops role, access to interactive SQL queries (and default procedures by inheritance) and all user-defined procedures to members of the developer role, and no special access beyond read-only system procedures to members of the apps role.

```
CREATE ROLE ops WITH admin;
CREATE ROLE developer WITH sql, allproc;
CREATE ROLE apps;
```

12.6. Using Built-in Roles

To simplify the development process, VoltDB predefines two roles for you when you enable security: administrator and user. *Administrator* has ADMIN permissions: access to all functions including interactive SQL queries, DDL, system procedures, and user-defined procedures. *User* has SQL and ALLPROC pemissions: access to ad hoc SQL and all default and user-defined stored procedures.

These predefined roles are important, because when you start the database there is no schema and therefore no user-defined roles available to assign to users. So you must always include at least one user who is

assigned the Administrator role when starting a database with security enabled. You can use this account to then load the schema — including additional security roles and permissions — and then update the configuration to add more users as necessary.

12.7. Encrypting VoltDB Communication Using TLS/SSL

VoltDB hashes usernames and passwords both within the database server and while passing them across the network. However, the network communication itself is not encrypted by default. You can enable Transport Layer Security (TLS) — the recommended upgrade from Secure Socket Layer (SSL) communication — for the client and admin ports, the internal interface, and the port used for cross datacenter database replication (XDCR) for more thorough security. Similarly, you can enable TLS for the Volt Management Console (VMC) and HTTP/JSON port when configuring the separate VMC service. The following sections summarize how to enable TLS for the servers in a cluster, including:

- Creating TLS/SSL certificates
- Using Certificate Revocation Lists (CRLs)
- Configuring TLS encryption on the server, including choosing which ports to encrypt
- Configuring mutual TLS encryption for both clients and servers
- Using the VoltDB command line utilities with TLS
- Implementing TLS communication in Java client applications
- Configuring Database Replication (XDCR) using TLS

12.7.1. Creating the TLS/SSL Certificates

TLS, like its predecessor SSL, uses certificates to validate the authenticity of the communication. You can either use certificates created by a commercial certificate provider (such as Digitcert, GeoTrust, or Symantec) or you can create your own certificates. If you use a commercial provider, that provider also handles the authentication of the certificate. If you create a local or self-signed certificate, you need to provide the certificate and authentication to the server and clients yourself.

If you choose to use a locally created certificate, you must first generate the certificate key store and trust store. For traditional TLS validation of the data repository, you will need a certificate for the cluster servers. (All nodes in the cluster use the same certificate.) For mutual TLS, or mTLS, you will need separate certificates for the server and the clients.

Note

The following instructions describe creating and using TLS/SSL certificates in Privacy Enhanced Mail, or PEM, format. PEM is a text-based format that is easy to work with and is accepted by all Volt functions starting with Volt version 14.2.0 and Volt Operator 3.6.0. For earlier versions of Volt, certain functions require a JKS file instead of PEM. Please see the version 13 documentation for configuring TLS for instructions on creating and using certificates acceptable to previous releases.

You can create local certificates in PEM format using the freely available OpenSSL command line utility. (See the OpenSSL web site, https://openssl.org, for more information and download instructions.) For encryption with authentication of the server, you need to create a key store and a trust store for the server. For example:

Example 12.1. Creating a TLS Certificate for the Database Servers

```
openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 \
  -noenc -subj "/CN=mydb.myorg.org" \
  -keyout keystore.pem -out truststore.pem
```

If you wish to use mutual TLS (mTLS), you need key and trust stores for both the server and the clients. The easiest way to do this is to create a common certificate to act as the root authority for both, then create key and trust stores based on the root authority. You can also append the trust store to the key store so they can easily be shared. By doing this, you can use the root authority trust store for both clients and servers. For example:

Example 12.2. Creating Mutual TLS Certificates for Both Servers and Clients

```
# define the root authority
openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 \
  -noenc -subj "/CN=mydb.myorg.org" \
  -keyout root.keystore.pem -out root.truststore.pem
cat root.truststore.pem >> root.keystore.pem


# Derive server key from root authority
openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 \
  -noenc -subj "/CN=mydb.myorg.org" \
  -CA root.keystore.pem \
  -keyout server.keystore.pem -out truststore.tmp
cat truststore.tmp >> server.keystore.pem

# Derive client key from root authority
openssl req -x509 -newkey rsa:4096 -sha256 -days 3650 \
  -noenc -subj "/CN=mydb.myorg.org" \
  -CA root.keystore.pem \
  -keyout client.keystore.pem -out truststore.tmp
cat truststore.tmp >> client.keystore.pem
```

12.7.2. Using Certificate Revocation Lists (CRLs)

Certificates specify the authenticity of the server and the truststore allows the server to authenticate the certificate provided by the client applications in mutual TLS (mTLS). But there may be times when you want to explicitly exclude certain client certificates. This is done using a certificate revocation list (CRL). You can add one or more CRLs to the Volt server by creating a directory where PEM-formatted CRLs are stored, which is then referenced in the cluster configuration.

You can create your own CRL. Or, if you purchase certificates from a commercial authority, you may receive CRLs from the vendor. CRLs received from commercial authorities are often sent in a binary format such as DER or PKCS. In this case, you need to convert the CRL to PEM format using an **openssl** command such as the following:

```
$ openssl x509 -inform der -in new-crl.der -out /etc/ssl/local/crls/new-crl.pem
```

12.7.3. Configuring TLS/SSL on the VoltDB Server

Once you create the key store and the trust store, you can reference them in the database configuration file to enable TLS when initializing the database root directory. For example, using the key store and trust

store created in Example 12.1, “Creating a TLS Certificate for the Database Servers”, the configuration might look like this:

```
deployment:
  ssl:
    enabled: true
    keystore:
      file: /etc/ssl/local/keystore.pem
    truststore:
      file: /etc/ssl/local/truststore.pem
```

If you are purchasing a commercial certificate, rather than creating a self-signed certificate, the certificate authority will provide you with a signed certificate, which you can use as the root authority when creating your trust and key store. If you are using certificate revocation lists (CRLs), you must also specify the CRL directory. For example:

```
deployment:
  ssl:
    keystore:
      file: /etc/ssl/local/keystore.pem
    truststore:
      file: /etc/ssl/local/truststore.pem
    crl:
      path: /etc/ssl/local/crls
```

Once you identify the certificates to use, you need to enable TLS for the server using the `deployment.ssl.enabled` property. You can also specify which ports to encrypt using subproperties of `deployment.ssl`:

- External ports (`deployment.ssl.external`), including the client and admin ports
- Internal ports (`deployment.ssl.internal`), used for intra-cluster communication between the nodes of the cluster
- Extranet ports (`deployment.ssl.dr`), including the replication port used for XDCR

For each type of port, you specify that the ports are either enabled ("true") or disabled ("false"). The default is false. For example, the following configuration enables TLS encryption on the external, internal, and XCDR ports:

```
deployment:
  ssl:
    enabled: true
    external: true
    internal: true
    dr: true
    keystore:
      file: /etc/ssl/local/keystore.pem
    truststore:
      file: /etc/ssl/local/truststore.pem
```

Note that if you enable TLS encryption for the XDCR port, other clusters replicating from this cluster must include the appropriate client configuration when they enable XDCR. See Section 12.7.7, “Configuring Database Replication (XDCR) With TLS/SSL” for information on setting up TLS when configuring XDCR.

Also, enabling TLS encryption on the internal port means that all intra-cluster communication must be encrypted and decrypted as it passes between nodes. Consequently, any operations that require interactions between cluster nodes (such as K-safety or multi-partition transactions) may take longer and therefore impact overall latency. Be sure to benchmark your application with and without TLS encryption before enabling internal port encryption on production systems.

Finally, it is important to note that all ports where TLS is enabled and all the servers within a single cluster use the same certificate.

12.7.4. Configuring Mutual TLS/SSL on the Server and Clients

If you choose to use mutual TLS, or mTLS, you need to configure both the servers and clients with the appropriate key stores and trust stores, as well as enabling mTLS on the server. To do this on the server, you must:

- Enable TLS/SSL
- Enable TLS/SSL for the category of ports to use (external, XDCR, or both)
- Enable mTLS (also called client authentication) for either or both of the port categories

For example, using the key stores for client and server and the shared trust store created in Example 12.2, "Creating Mutual TLS Certificates for Both Servers and Clients" and enabling mTLS, the configuration might look like the following. Note that there are separate properties — `deployment.ssl.drclientauthrequired` and `deployment.ssl.clientauthrequired` respectively — to enable mTLS for XDCR and for the external ports, so you can choose which ports require mTLS authentication. (Mutual TLS is not available for the internal ports.) In the following example, both XDCR and external ports have mTLS enabled :

```
deployment:
  ssl:
    enabled: true
    external: true
    dr: true
    clientauthrequired: true
    drclientauthrequired: true
    keystore:
      file: /etc/ssl/local/server.keystore.pem
    truststore:
      file: /etc/ssl/local/root.truststore.pem
```

12.7.5. Using the VoltDB Command Line Utilities with TLS/SSL

Once you enable TLS for the external interfaces on your database servers, you must also enable TLS on the command line utilities so they use the appropriate protocols to connect to the servers. (The **voltdb** utility is the one exception. Since it only operates on the local server it does not require a network connection.)

When invoking the command line utilities, such as **voltadmin** and **sqlcmd**, you use the `--ssl` option to activate encryption with TLS-enabled VoltDB servers. If the servers are using a commercially-provided certificate, you can specify **sqlcmd** with the `--ssl` option without an argument, or in the case of the other commands, with an empty string as an argument. For example:

```
$ sqlcmd --ssl
$ voltadmin --ssl=""
```

If the servers are using a local or self-signed certificate you must also specify a Java properties file as an argument to the `--ssl` option. For example:

```
$ sqlcmd --ssl=localcert.properties
```

The properties file must identify the filename and location of the trust store. So, using the trust store generated by the example in Section 12.7.1, “Creating the TLS/SSL Certificates”, the `localcert.properties` file might look like the following:

```
trustStore=/etc/ssl/local/truststore.pem
```

If you configure the servers for mTLS on the external ports and create a root certificate authority for the client and server keystores, the command line utilities can use the root trust store to validate the server, but must also include the key store for the client certificate. You do this by adding a line identifying the client key store in the properties file. The following example shows the use of a properties file, `mtlscert.properties`, for accessing a database configured for mutual TLS using the key and trust store files created in Example 12.2, “Creating Mutual TLS Certificates for Both Servers and Clients”:

```
$ cat mtlscert.properties
trustStore=/etc/ssl/local/root.truststore.pem
keyStore=/etc/ssl/local/client.keystore.pem
$ sqlcmd --ssl=mtlscert.properties
```

12.7.6. Implementing TLS/SSL in the Java Client Applications

Just as the command line tools must specify how to connect to an TLS-enabled server, client applications must also establish an appropriate connection. Using the VoltDB Java API, you can enable TLS by setting the appropriate attributes of the client configuration. Specifically, if you are using a self-signed certificate, you must provide the path to the trust store. You can do this using either the .trustStore() or .trustStoreFromPropertyFile() method. For example, the following two commands are equivalent, assuming the `localcert.properties` file matches the properties file described in Section 12.7.5, “Using the VoltDB Command Line Utilities with TLS/SSL”:

```
config.trustStore("/etc/ssl/local/truststore.pem", "");
config.trustStoreFromPropertyFile("localcert.properties");
```

After setting the trust store properties you can enable TLS communication using the .enableSSL() method and create the client connection. For example:

```
ClientConfig2 config = new ClientConfig2()
            .username("JDoe")
            .password("JDsPasswd")
            .trustStoreFromPropertyFile("localcert.properties")
            .enableSSL();
client = ClientFactory.createClient(config);
```

If the server enables mTLS for the external ports, the client applications must provide both a trust store and a key store, in the same way the VoltDB command line utilities do. In fact, using the .trustStoreFromPropertyFile() method, you can specify the same property file specifying both the root trust store and client key store to provide mTLS authentication with the servers. Or you can explicitly identify the trust and key store using the .trustStoreWithMutualAuth() method. So, using the trust and key stores from the preceding examples, the following two statements are equivalent:

```
config.trustStoreWithMutualAuth(
                "/etc/ssl/local/root.truststore.pem", "",
                "/etc/ssl/local/client.keystore.pem", "");
config.trustStoreFromPropertyFile("mtlscert.properties");
```

When using a commercially generated certificate, you do not need to specify the trust store and can use just the .enableSSL() method.

12.7.7. Configuring Database Replication (XDCR) With TLS/SSL

When using TLS encryption on the DR port, the DR snapshots and binary logs are encrypted as they pass from the producer cluster to the consumer cluster. Since in XDCR the clusters act as both producer and consumer, this means that the clusters must not only have TLS enabled for the DR port, but must configure TLS for their connections to the other clusters.

Section 12.7.3, "Configuring TLS/SSL on the VoltDB Server" describes how to enable TLS encryption for the DR port, which must be done before the cluster starts. To configure TLS connectivity for communication with the other clusters, you add the `ssl` property to the `connection` list entry within the DR configuration. The value of the `ssl` property is either blank — for commercial certificates — or the path to a Java properties file specifying the trust store for the remote cluster(s) when using a locally-generated certificate. These property values are the same as the `--ssl` argument you use when running the command line utilities described in Section 12.7.5, "Using the VoltDB Command Line Utilities with TLS/SSL".

The configuration might look like this:

XDCR Cluster

```
deployment:
  ssl:
    enabled: true
    dr: true
    keystore:
      file: /etc/ssl/local/keystore.pem
    truststore:
      file: /etc/ssl/local/truststore.pem
  dr:
    id: 1
    role: xdcr
    connection:
      source: NYCSvrA,NYCSvrB
      ssl: /etc/ssl/local/nyccert.properties
```

When using mutual TLS, the configuration and TLS property file should include the root trust store and server key store, since the server acts as both a server and a client in the XDCR relationship. For example:

XDCR Cluster With Mutual TLS

```
deployment:
  ssl:
    enabled: true
    dr: true
    drclientauthrequired: true
    keystore:
      file: /etc/ssl/local/server.keystore.pem
```

```
    truststore:
      file: /etc/ssl/local/root.truststore.pem
  dr:
    id: 1
    role: xdcr
    connection:
      source: NYCSvrA,NYCSvrB
      ssl: /etc/ssl/local/nycmtlscert.properties
```

When using commercially purchased certificates, the `ssl` subproperties are left blank; so each cluster can, if you choose, use a separate certificate. However, when using locally-generated certificates, there is only one properties file specified in the `ssl` property. So to use separate certificates for each cluster in the XDCR relationship, you should start by creating a root authority certificate as you would for mutual TLS as described in Example 12.2, “Creating Mutual TLS Certificates for Both Servers and Clients”, then create the certificates for each cluster off the root certificate. This way you can use the root trust store to authenticate all of the clusters.

12.7.8. Updating TLS/SSL Certificates and Certificate Revocation Lists (CRLs)

TLS certificates have an expiration date. So there will come a time when you want to update or replace the existing certificate. The following instructions explain how to update the TLS certificates and CRLs on a running Volt cluster.

To update the existing TLS certificate on a running system, you simply replace the current keystore and truststore PEM files — using the exact same file names and locations as specified in the `deployment.ssl.keystore.file` and `deployment.ssl.truststore.file` properties — then issue the **voltadmin ssl reload** command. If the configuration also specifies a CRL directory with the `deployment.ssl.crl.path` property, any CRLs in that directory will be reloaded as well.

When issuing the **voltadmin ssl reload** command, you must use a properties file pointing to the original truststore when invoking **voltadmin**, since the original certificate is still in effect until the command is completed:

```
$ voltadmin --ssl=oldcert.properties ssl reload
```

After issuing the command, you will need to use a properties file pointing to the new truststore when invoking Volt command line utilities, as described in Section 12.7.5, “Using the VoltDB Command Line Utilities with TLS/SSL”.

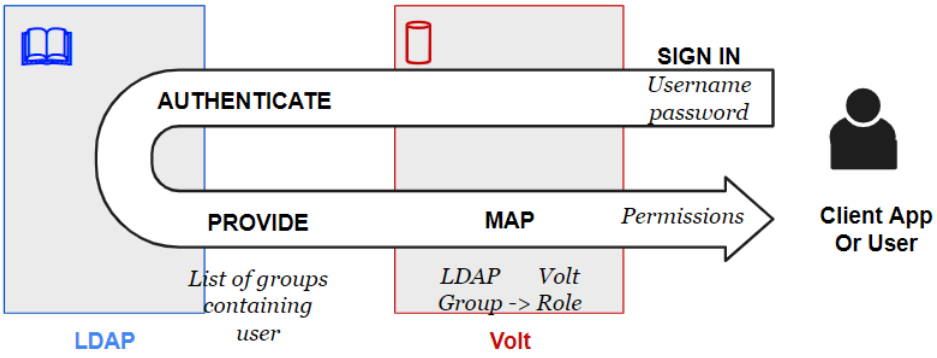
12.8. Integrating LDAP Security with VoltDB

LDAP (Lightweight Directory Access Protocol) is a directory service for managing hierarchical information. LDAP is often used within organizations to centralize the management and maintenance of security information about employees and other resources. LDAP integration in Volt Active Data allows you to use LDAP to **authenticate** Volt users and **authorize** their access to specific features in place of the builtin security described earlier in this chapter. In other words, manage users and permissions in LDAP rather than internally within Volt.

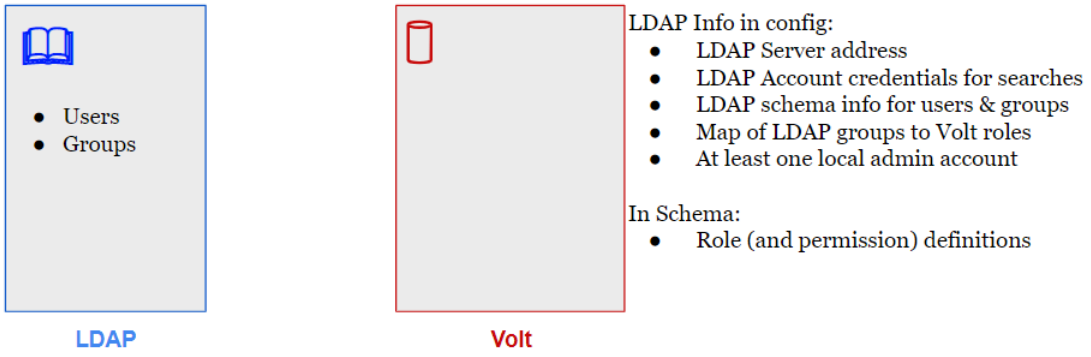
Note

LDAP integration does require at least one builtin user account with administrative permissions to ensure the database is accessible in case the LDAP server is not available.

The following diagram shows the overall workflow of LDAP integration at runtime. The user, or client application, connects to the database passing in their LDAP credentials: a username and password. VoltDB passes those credentials to LDAP for authentication. Once the user is authenticated, the VoltDB server then looks for groups in LDAP of which the user is a member. Finally, the LDAP group is mapped to one or more VoltDB roles to decide what permissions the user has.



LDAP integration allows VoltDB to use existing resources within LDAP for authentication and authorization. That means that administrators can use the existing user and group definitions within LDAP for security management of VoltDB. Of course, you can always create specific users and/or groups for managing VoltDB access, but it is not necessary. As a result, very little change is needed to an existing LDAP infrastructure; most of the set up for LDAP integration belongs to the VoltDB servers, as shown below.



The following sections explain how to configure VoltDB for LDAP integration and specific requirements for the LDAP server.

12.8.1. Configuring LDAP Security in VoltDB

You configure LDAP integration in the `deployment.security` property of the VoltDB configuration file. The following is an example configuration file that illustrates all of the available LDAP configuration options.

```
deployment:
  security:
    enabled: true
    provider: ldap                                    ❶

    ldap:
      server: ldaps://ldap.myorg.com                  ❷
```

```
    rootdn: "dc=ldaptest,dc=voltactivedata,dc=com"      ❸
    timeout: 12                                         ❹

    user: "cn=admin,dc=ldaptest,dc=volt,dc=com"         ❺
    password: secret.password

    # LDAP group to Volt Role mapping
    group:                                              ❻
    - name: "cn=heroes,dc=ldaptest,dc=volt,dc=com"
      role: admin
    - name: "cn=zeroes,dc=ldaptest,dc=volt,dc=com"
      role: user

ssl:
  truststore:                                           ❼
    file: /my/ssl/ldap/truststore.jks
    password: trustpasswd

users:
- name: houdini
  password: shazam
  roles: administrator                                  ❽
```

❶ **Enabling LDAP security** — To enable LDAP integration, you must set the properties `enabled` to "true" and `provider` to "ldap".

❷ **LDAP server address** — The location of the LDAP server, using either plain ("ldap://") or secure ("ldaps://") LDAP protocol. Secure LDAP is recommended. But if you use secure LDAP, you must also provide the TLS/SSL credentials (❼).

❸ **Root domain name** — The LDAP domain containing both the user and group objects needed for Volt access.

❹ **Timeout (OPTIONAL)** — The timeout period, in seconds, to wait for a response from the LDAP server. After the timeout expires with no response from the LDAP server, the authentication of the user will fail. The default timeout is 10 seconds.

❺ **LDAP privileged account (username and password)** — The usernasme and password of an LDAP account that has read-only access to the specified root domain. This account is used to search the domain for the specified username and the groups of which that user is a member. You can mask the password for the LDAP account so it does not appear in plain text in the configuration file using the **voltdb mask** command.

❻ **LDAP group → Volt role mapping** — The mapping of LDAP group names to Volt roles. The roles themselves are defined in the database schema. Each `group` list element maps one LDAP group name to one or more Volt roles, specifying the roles as a comma-separated list. If there is no mapping for the LDAP group, the group is ignored.

❼ **TLS/SSL credentials** — The `ssl.truststore` property specifies the TLS/SSL credentials to use when accessing the directory using secure LDAP (LDAPS://).

❽ **Local administrator's account** — There must be at least one local user account with administrative privileges defined in the configuration. This account provides access to the database in case the LDAP server is not reachable. You can define other local accounts if you wish. When a user attempts to access the server, Volt looks for the username in the local accounts first; if no local account with that username exists, it then passes the credentials on to LDAP for authentication and authorization.

Finally, by default, Volt searches for the username in the *uid* attribute of the *inetOrgPerson* class, then uses the full distinguished name to search the *uniqueMember* attribute of the *groupOfUniqueNames* class to determine what groups they belong to. If your LDAP directory uses different classes or attribute settings for users and groups, you must specify the class and attribute names as properties of `deployment.ldap`.

For example, the following configuration identifies *posixAccount* and *posixGroup* as the classes to use for users and groups, respectively:

```
deployment:
  security:
    enabled: true
    provider: ldap

    ldap:
      server: ldaps://ldap.myorg.com
      rootdn: "dc=ldaptest,dc=voltactivedata,dc=com"
      userclass: posixAccount
      useruid: uid
      groupclass: posixGroup
      groupmemberuid: memberuid"
      user: "cn=admin,dc=ldaptest,dc=volt,dc=com"
      password: secret.password
```

12.8.2. Configuring VoltDB Security in LDAP

The goal of LDAP integration in VoltDB is to adapt to the customer's existing LDAP implementation, avoiding any changes to the schema wherever possible. So, for example, there are configuration options that allow VoltDB to access users and groups records defined in non-standard LDAP classes and objects. However, there is one requirement on the LDAP user objects:

> *The LDAP user records must include the SHA-256 hash of the user's password as an alternate password.*

VoltDB never actually receives the password sent by the user at runtime in clear text. The client APIs hashes the password and sends the base-64 encoding of the SHA-256 hash instead. So Volt cannot pass on the actual password to LDAP. Instead it sends the SHA-256 hash. So, to be able to authenticate the user, LDAP must have the matching hash as one of the allowable passwords for the user. You can get the SHA-256 hash of a password using the **shasum** Linux utility. For example:

```
$ echo -n 'MyFavoritePasswrd' | shasum -a 256
238d59ddd7b0b8d54d3b8ff864855bcd332e66b1a06dad010f676a23d1928a68  -
```

12.9. Integrating Kerberos Security with VoltDB

For environments where more secure communication is required than hashed usernames and passwords, it is possible for a VoltDB database to use Kerberos to authenticate clients and servers. Kerberos is a popular network security protocol that you can use to authenticate the Java client processes when they connect to VoltDB database servers. Use of Kerberos is supported for the Java client library and JSON interface only.

To use Kerberos authentication for VoltDB security, you must perform the following steps:

1. Set up and configure Kerberos on your network, servers, and clients.

2. Install and configure the Java security extensions on your VoltDB servers and clients.

3. Configure the VoltDB cluster and client applications to use Kerberos.

The following sections describe these steps in detail.

12.9.1. Installing and Configuring Kerberos

Kerberos is a complete software solution for establishing a secure network environment. It includes network protocols and software for handling authentication and authorization in a secure, encrypted fashion. Kerberos requires one or more servers known as key distribution centers (KDC) to authenticate and authorize services and the users who access them.

To use Kerberos for VoltDB authentication you must first set up Kerberos within your network environment. If you do not already have a Kerberos KDC, you will need to create one. You will also need to install the Kerberos client libraries on all of the VoltDB servers and clients and set up the appropriate principals and services. Because Kerberos is a complete network environment rather than a single platform application, it is beyond the scope of this document to explain how to install and configure Kerberos itself. This section only provides notes specific to configuring Kerberos for use by VoltDB. For complete information about setting up and using Kerberos, please see the Kerberos documentation.

Part of the Kerberos setup is the creation of a configuration file on both the VoltDB server and client machines. By default, the configuration file is located in `/etc/krb5.conf` on Linux systems. (On Macintosh systems, the configuration file is `edu.mit.Kerberos` located either in `~/Library/Preferences/` or `/Library/Preferences/`.) Be sure this file exists and points to the correct realm and KDC.

Once a KDC exists and the nodes are configured correctly, you must create the necessary Kerberos accounts — known as "user principals" for the accounts that run the VoltDB client applications and a "service principal" for the VoltDB cluster. If you intend to use the web-based Volt Management Center or the JSON interface, you will also want to create a host and HTTP service principle for each server as well. For example, to create the service keytab file for the VoltDB database, you can issue the following commands on the Kerberos KDC:

```
$ sudo kadmin.local
kadmin.local: addprinc -randkey service/voltdb
kadmin.local: ktadd -k voltdb.keytab service/voltdb
```

Then copy the keytab file to the database servers, making sure it is only accessible by the user account that starts the database process:

```
$ scp voltdb.keytab voltadmin@voltsvr:voltdb.keytab
$ ssh voltadmin@voltsvr chmod 0600 voltdb.keytab
```

You can then create host and HTTP service principles for each server in the cluster and write them to a server-specific keytab. For example, to create a keytab file for the database node *server1*, the command would be the following:

```
$ sudo kadmin.local

kadmin.local: addprinc -randkey host/server1.mycompany.lan
kadmin.local: addprinc -randkey HTTP/server1.mycompany.lan

kadmin.local: ktadd -k server1.mycompany.lan.keytab HTTP/server1.mycompany.lan
kadmin.local: ktadd -k server1.mycompany.lan.keytab host/server1.mycompany.lan
```

12.9.2. Configuring Java Security

The next step is to create a JAAS login configuration file so Java knows how to authenticate the current process. By default, the JAAS login configuration file is `$HOME/.java.login.config`. On the data-

base servers, the configuration file must define the *VoltDBService* module and associate it with the keytab created in the previous section.

To enable Kerberos access from the web-based Volt Management Center and JSON interface, you must also include entries for the Java Generic Security Service (JGSS) declaring the VoltDB service principle and the server's HTTP service principle. For example:

Server JAAS Login Configuration File

```
VoltDBService {
    com.sun.security.auth.module.Krb5LoginModule required
        useKeyTab=true keyTab="/home/voltadmin/voltdb.keytab"
        doNotPrompt=true
        principal="service/voltdb@MYCOMPANY.LAN" storeKey=true;
};

com.sun.security.jgss.initiate {
     com.sun.security.auth.module.Krb5LoginModule required
     principal="service/voltdb@MYCOMPANY.LAN"
     keyTab="/home/voltadmin/voltdb.keytab"
     useKeyTab=true
     storeKey=true
     debug=false;
};

com.sun.security.jgss.accept {
     com.sun.security.auth.module.Krb5LoginModule required
     principal="HTTP/server1.mycompany.lan@MYCOMPANY.LAN"
     useKeyTab=true
     keyTab="/etc/krb5.keytab"
     storeKey=true
     debug=false
     isInitiator=false;
};
```

On the client nodes, the JAAS login configuration defines the *VoltDBClient* module.

Client JAAS Login Configuration File

```
VoltDBClient {
    com.sun.security.auth.module.Krb5LoginModule required
    useTicketCache=true renewTGT=true doNotPrompt=true;
};
```

12.9.3. Configuring the VoltDB Servers and Clients

Finally, once Kerberos and the Java security extensions are installed and configured, you must configure the VoltDB database cluster and client applications to use Kerberos.

On the database servers, you enable Kerberos security using security and its subproperties when you initialize the database root directory, specifying "kerberos" as the provider. For example:

```
deployment:
  security:
```

```
    enabled: True
    provider: kerberos
       . . .
```

You then assign roles to individual users as described in Section 12.3, “Defining Users and Roles”, except in place of generic usernames, you specify the Kerberos user — or "principal" — names, including their realm. Since Kerberos uses encrypted certificates, the password property is ignored and can be filled in with arbitrary text. For example:

```
deployment:
  security:
    enabled: True
    provider: kerberos
       . . .
  users:
  - name: "mtwain@MYCOMPANY.LAN"
    password: n/a
    roles=administrator
  - name: "cdickens@MYCOMPANY.LAN"
    password: n/a
    roles: dev
  - name: "hbalzac@MYCOMPANY.LAN"
    password: n/a
    roles: adhoc
```

Having configured Kerberos in a configuration file, you are ready to initialize and start the VoltDB cluster. When starting the VoltDB process, Java must know how to access the Kerberos and JAAS login configuration files created in the preceding sections. If the files are not in their default locations, you can override the default location using the VOLTDB_OPTS environment variable and setting the flags `java.security.krb5.conf` and `java.security.auth.login.config`, respectively.[1]

In Java client applications, you specify Kerberos as the security protocol when you create the client connection, using the enableKerberosAuthentication method as part of the configuration. For example:

```
import org.voltdb.client.ClientConfig;
import org.voltdb.client.ClientFactory;

ClientConfig config = new ClientConfig();
      // specify the JAAS login module
config.enableKerberosAuthentication("VoltDBClient");

VoltClient client = ClientFactory.createClient(config);
client.createConnection("voltsvr");
```

Note that the VoltDB client automatically picks up the Kerberos cached credentials of the current process, the user's Kerberos "principal". So you do not need to — and should *not* — specify a username or password as part of the VoltDB client configuration.

When using the VoltDB JDBC client interface, you can enable Kerberos by setting the `kerberos` property on the connection to match the settings in the Java API. For example, you can enable Kerberos by setting the property on the connection string as a query parameter:

```
Class.forName("org.voltdb.jdbc.Driver");
Connection c = DriverManager.getConnection(
```

[1] On Macintosh systems, you must always specify the `java.security.krb5.conf` property.

```
     "jdbc:voltdb://svr1:21212,svr2:21212?kerberos=VoltDBClient");
```

Alternately, you can supply a list of properties, including the `kerberos` property, when you initialize the connection:

```
Class.forName("org.voltdb.jdbc.Driver");
Properties props = new Properties();
props.setProperty("kerberos", “VoltDBClient");
Connection c = DriverManager.getConnection(
     "jdbc:voltdb://svr1:21212,svr2:21212", props);
```

12.9.4. Accessing the Database from the Command Line and the Web

It is also important to note that once the cluster starts using Kerberos authentication, only Java, JDBC, JSON, and Python clients can connect to the cluster and they must use Kerberos authentication to do it. The same is true for the CLI commands, such has **sqlcmd** and **voltadmin**. To authenticate to a VoltDB server with Kerberos security enabled using the Java-based utilities **sqlcmd** and **cvsloader**, you must include the `--kerberos` flag identifying the name of the Kerberos client service module. For example:

```
$ sqlcmd --kerberos=VoltDBClient
```

If the configuration files are not in the default location, you must specify their location on the command line:

```
$ sqlcmd --kerberos=VoltDBClient \
     -J-Djava.security.auth.login.config=myclient.kerberos.conf
```

To use the Python API or Python-based **voltadmin** utility, you must first make sure you have the python-gssapi package installed. Then, login to your Kerberos account using **kinit** before invoking the Python client. When using the **voltadmin** utility, you must also include `--kerberos` flag, but you do not need to specify any argument since it picks up the credentials in the Kerberos user's cache. For example:

```
$ voltadmin shutdown --kerberos
```

To use the Volt Management Center or the JSON interface to access the database, your web browser must be configured to use the Simple and Protected GSS-API Negotiation Mechanism (also known as SPNEGO). See your web browser's help for instructions on configuring SPNEGO.

Chapter 13. Saving & Restoring a VoltDB Database

There are times when it is necessary to save the contents of a VoltDB database to disk and then restore it. For example, if the cluster needs to be shut down for maintenance, you may want to save the current state of the database before shutting down the cluster and then restore the database once the cluster comes back online. Performing periodic backups of the data can also provide a fallback in case of unexpected failures — either physical failures, such as power outages, or logic errors where a client application mistakenly corrupts the database contents.

VoltDB provides shell commands, system procedures, and an automated snapshot feature that help you perform these operations. The following sections explain how to save and restore a running VoltDB cluster, either manually or automatically.

13.1. Performing a Manual Save and Restore of a VoltDB Cluster

Manually saving and restoring a VoltDB database is useful when you need to modify the database's physical structure or make schema changes that cannot be made to a running database. For example, changing the K-safety value, the number of sites per site, or changing the partitioning column of a partitioned table. The normal way to perform such a maintenance operation using save and restore is as follows:

1. Stop database activities (using **pause**).
2. Use **save** to write a snapshot of the current data to disk.
3. Shutdown the cluster.
4. Make changes to the VoltDB schema, cluster configuration, and/or configuration file as desired.
5. Reinitialize the database with the modified configuration file, using **voltdb init --force**.
6. Restart the cluster in admin mode, using **voltdb start --pause**.
7. Optionally, reload the schema and stored procedures (if you are changing the schema).
8. Restore the previous snapshot.
9. Restart client activity (using **resume**).

The key is to make sure that all database activity is stopped before the save and shutdown are performed. This ensures that no further changes to the database are made (and therefore lost) after the save and before the shutdown. Similarly, it is important that no client activity starts until the database has started and the restore operation completes.

Also note that Step #7, reloading the schema, is optional. If you are going to reuse the same schema in a new database instance, the restore operation will automatically load the schema from the snapshot itself. If you want to modify the schema in any way, such as changing indexes or tables and columns, you should load the modified schema before restoring the data from the snapshot. If the database schema is *not* empty (that is there are tables already defined), only the data is loaded from the snapshot. See Section 13.1.3.2, “Modifying the Database Schema and Stored Procedures” for more information on modifying the schema when restoring snapshots.

Save and restore operations are performed either by calling VoltDB system procedures or using the corresponding **voltadmin** shell commands. In most cases, the shell commands are simpler since they do not require program code to use. Therefore, this chapter uses **voltadmin** commands in the examples. If you are interested in programming the save and restore procedures, see Appendix G, *System Procedures* for more information about the corresponding system procedures.

When you issue a save command, you specify a path where the data will be saved and a unique identifier for tagging the files. VoltDB then saves the current data on each node of the cluster to a set of files at the specified location (using the unique identifier as a prefix to the file names). This set of files is referred to as a snapshot, since it contains a complete record of the database for a given point in time (when the save operation was performed).

The `--blocking` option lets you specify whether the save operation should block other transactions until it completes. In the case of manual saves, it is a good idea to use this option since you do not want additional changes made to the database during the save operation.

Note that every node in the cluster uses the same absolute path, so the path specified must be valid, must exist on every node, and must not already contain data from any previous saves using the same unique identifier, or the save will fail.

When you issue a restore command, you specify the same absolute path and unique identifier used when creating the snapshot. VoltDB checks to make sure the appropriate save set exists on each node, then restores the data into memory.

13.1.1. How to Save the Contents of a VoltDB Database

To save the contents of a VoltDB database, use the **voltadmin save** command. The following example creates a snapshot at the path */tmp/voltdb/backup* using the unique identifier *TestSnapshot*.

```
$ voltadmin save --blocking /tmp/voltdb/backup "TestSnapshot"
```

In this example, the command tells the save operation to block all other transactions until it completes. It is possible to save the contents without blocking other transactions (which is what automated snapshots do). However, when performing a manual save prior to shutting down, it is normal to block other transactions to ensure you save a known state of the database.

Note that it is possible for the save operation to succeed on some nodes of the cluster and not others. When you issue the **voltadmin save** command, VoltDB displays messages from each partition indicating the status of the save operation. If there are any issues that would stop the process from starting, such as a bad file path, they are displayed on the console. It is a good practice to examine these messages to make sure all partitions are saved as expected.

Note that it is also possible to issue the **voltadmin save** command without arguments. In that case the snapshot is saved to the default snapshots folder in the database root directory. This can be useful because the **voltdb start** command can automatically restore the latest snapshot in that directory as described in the next section.

13.1.2. How to Restore the Contents of a VoltDB Database Manually

The easiest way to restore a snapshot is to let VoltDB do it for you as part of the recover operation. If you are not changing the cluster configuration you can use an automated snapshot or other snapshot saved into the `voltdbroot/snapshots` directory by simply restarting the cluster nodes using the **voltdb start** command. With the start action VoltDB automatically starts and restores the most recent snapshot.

If command logging is enabled, it also replays any logs after the snapshot. This approach has the added benefit that VoltDB automatically loads the previous schema as well as part of the snapshot.

However, you cannot use **voltdb start** to restore a snapshot if the physical configuration of the cluster has changed or if you want to restore an earlier snapshot or a snapshot stored in an alternate location. In these cases you must do a manual restore.

To manually restore a VoltDB database from a snapshot previously created by a save operation, you can create a new database instance and use the **voltadmin restore** command. So, for example, if you modify the configuration, you must re-initialize the root directory with the new configuration file, using the --force flag to overwrite the previous configuration and database content:

```
$ voltdb init --config=newconfig.yaml --force
```

Then you can start the reconfigured database, which creates a new empty database. It is also a good idea to start the database in admin mode by including the --pause flag:

```
$ voltdb start --pause
```

Finally, you restore the previously saved snapshot using the same pathname and unique identifier used during the save. The following example restores the snapshot created by the example in Section 13.1.1 and resumes normal operation (that is, exits admin mode).

```
$ voltadmin restore /tmp/voltdb/backup "TestSnapshot"
$ voltadmin resume
```

As with save operations, it is always a good idea to check the status information displayed by the command to ensure the operation completed as expected.

13.1.3. Changing the Cluster Configuration Using Save and Restore

Most changes to a VoltDB database can be made "on the fly" while the database is running. Adding and removing tables, enabling and disabling database features such as import and export, and adding or updating stored procedures can all be done while the database is active. However, between a save and a restore, it is possible to make changes to the database and cluster configuration that cannot be made on a running cluster. For example, you can:

- Add or remove nodes from the cluster
- Modify the schema and/or stored procedures that:
  - Change partitioned tables to replicated and vice versa
  - Change the partitioning column on partitioned tables
  - Add unique indexes to tables with existing data
- Change the number of sites per host
- Change the K-safety value

The following sections discuss these procedures in more detail.

13.1.3.1. Adding and Removing Nodes from the Database

To add nodes to the cluster, use the following procedure:

1. Save the database with the **voltadmin save** command.

2. Shutdown and re-initialize the database root directories on each node (including initializing new root directories for the nodes you are adding).

3. Start the cluster (including the new nodes) specifying the new server count with the `--count` argument to the **voltdb start** command.

4. Restore the database with the **voltadmin restore** command..

When the snapshot is restored, the database (and partitions) are redistributed over the new cluster configuration.

It is also possible to remove nodes from the cluster using this procedure. However, to make sure that no data is lost in the process, you must copy the snapshot files from the nodes that are being removed to one of the nodes that is remaining in the cluster. This way, the restore operation can find and restore the data from partitions on the missing nodes.

13.1.3.2. Modifying the Database Schema and Stored Procedures

The easiest and recommended way to change the database schema is by sending the appropriate SQL database definition language (DDL) statements to the sqlcmd utility. Similarly you can update the stored procedures on a running database using the LOAD CLASSES and REMOVE CLASSES directives.

However, there are a few changes that cannot be made to a running database,. For example, changing the partitioning column of a table if the table contains data. For these changes, you must use save and restore to change the schema.

To modify the database schema or stored procedures between a save and restore, make the appropriate changes to the source files (that is, the database DDL and the stored procedure Java source files). If you modify the stored procedures, be sure to repackage any Java stored procedures into a JAR file. Then you can:

1. Save the database with the **voltadmin save** command.

2. Shutdown and re-initialize the database root directories on each node.

3. Start the cluster with the **voltdb start** command.

4. Load the modified schema and stored procedures using sqlcmd.

5. Restore the database contents with the **voltadmin restore** command.

Two points to note when modifying the database structure before restoring a snapshot are:

- When existing rows are restored to tables where new columns have been added, the new columns are filled with either the default value (if defined by the schema) or nulls.

- When changing the datatypes of columns, it is possible to decrease the datatype size (for example, going from an INT to an TINYINT). However, if any existing values exceed the capacity of the new datatype (such as an integer value of 5,000 where the datatype has been changed to TINYINT), the entire restore will fail.

If you remove or modify stored procedures (particularly if you change the number and/or datatype of the parameters), you must make sure the corresponding changes are made to client applications as well.

13.2. Scheduling Automated Snapshots

Save and restore are useful when planning for scheduled down times. However, these functions are also important for reducing the risk from unexpected outages. VoltDB assists in contingency planning and recovery from such worst case scenarios as power failures, fatal system errors, or data corruption due to application logic errors.

In these cases, the database stops unexpectedly or becomes unreliable. By automatically generating snapshots at set intervals, VoltDB gives you the ability to restore the database to a previous valid state.

You schedule automated snapshots of the database as part of the configuration. The `deployment.snapshot` property lets you specify:

- The frequency of the snapshots. You can specify any whole number of seconds, minutes, or hours (using the suffix "s", "m", or "h", respectively, to denote the unit of measure). For example "3600s", "60m", and "1h" are all equivalent. The default frequency is 24 hours.
- The unique identifier to use as a prefix for the snapshot files. The default prefix is "AUTOSNAP".
- The number of snapshots to retain. Snapshots are marked with a timestamp (as part of the file names), so multiple snapshots can be saved. The `retain` property lets you specify how many snapshots to keep. Older snapshots are purged once this limit is reached. The default number of snapshots retained is two.

The following example enables automated snapshots every thirty minutes using the prefix "flightsave" and keeping only the three most recent snapshots.

```
deployment:
  snapshot:
    prefix: flightsave
    frequency: 30m
    retain: 3
```

By default, automated snapshots are stored in a snapshots subfolder of the VoltDB root directory (as described in Section 3.7.2, “Configuring Paths for Runtime Features”). You can save the snapshots to a specific path by adding the `deployment.paths.snapshots.path` property. For example, the following example defines the path for automated snapshots as `/etc/voltdb/autobackup/`.

```
deployment:
  paths:
    snapshots:
      path: /etc/voltdb/autobackup/
```

13.3. Tuning the Snapshot Process

Snapshot activity — for automated as well as manual, command logging, and other operational snapshots — involves both processing and disk I/O. The snapshot as a whole is broken up into smaller snapshot tasks, each writing a small part of the snapshot to disk. These tasks are interspersed among user transactions in the transaction queue. Since both snapshots and user transactions use the same queue, snapshots can have a noticeable impact on performance (in terms of throughput and/or latency) on a very busy database.

However, there are ways you can adjust the overall impact of snapshots, by controlling the frequency and size of the individual snapshot tasks. By reducing the frequency or size of each snapshot task, the snapshot can take longer but have less impact on the latency of user transactions. There are three ways to manage snapshot activity:

- **Snapshot Priority** — Snapshot-specific priority is a simple control that increases or decreases the frequency with which snapshot tasks are added to the queue. You control the snapshot priority by setting the `deployment.systemsettings.snapshot.priority` property as an integer value. The larger the value, the longer the interval between snapshot tasks.

- **Queue Priority** — You can assign snapshots a priority for queueing the same way you can for other user and operational tasks. This means you set the priority relative to other activities such as XDCR, export, or user tasks. You can even set the priority for individual user transactions using the Client2 API. You set the queue priority for snapshots by assigning the `deployment.systemsettings.priorities.snapshot.piority` property an integer value between 1 and 8. Again, the higher the number, the lower the priority. Note the snapshot-specific priority and queue priority are mutually exclusive. If queue priorities are enabled, the snapshot-specific prioritization will be disabled.

- **Autotuning** — Finally, you can choose to let the system select the best option for the frequency and size of snapshot tasks by enabling snapshot autotuning. Rather than setting fixed values, autotuning uses the current workload, measured as the size of the transaction queue, to decide how frequently to run snapshot tasks each time. The busier the queue, the less frequently snapshot tasks will run and the smaller the tasks will be. Resulting in slower snapshots but less impact on latency. On the other hand, because the adjustments are sensitive to the current queue size, when the workload is low the snapshot tasks can be larger and run more frequently. So snapshot performance is not negatively affected during lulls in the database workload. You enable autotuning by setting the `deployment.systemsettings.snapshot.autotune.enabled` property to true, which can be done either when configuring the database or on the fly while the database is running.

If snapshots are impacting the latency of business transactions, you can try turning on autotuning to adjust the snapshot processing to match the available transaction capacity. The best way to determine if autotuning is being effective is to use transaction performance statistics to compare the results between default behavior with autotuning disabled and after autotuning is enabled.

1. First, establish baseline performance statistics for the default snapshot behavior. One method of measuring transaction performance is to collect raw statistics for the 99th percentile and maximum latency of transactions before, during, and after a snapshot, which can be done by calling the @Statistics system procedure with the LATENCY selector and examining the columns labeled P99 and MAX. This information is also available in the metrics properties `voltdb_initiator_procedure_invoked_time_seconds_bucket` and `voltdb_initiator_procedure_invoked_time_seconds_max`.

2. Update the database configuration to enable snapshot autotuning by changing the `deployment.systemsettings.snapshot.autotune.enabled` property to true.

3. After enabling autotuning, capture the same metrics as in step #1 and compare the before and after results.

4. To measure the effect on snapshot duration, you can repeat the preceding steps using the @Statistics system procedure SNAPSHOTSUMMARY selector and comparing the DURATION column (or the `voltdb_snapshot_summary_info` metric) to determine how long snapshots are taking.

Finally, you can use Prometheus and Grafana to graph the output from the metrics in Step #1 to visualize the change in latency over time.

13.4. Managing Snapshot Files

VoltDB does not delete snapshots after they are restored; the snapshot files remain on each node of the cluster. For automated snapshots, the oldest snapshot files are purged according to the settings in the

configuration. But if you create snapshots manually or if you change the directory path or the prefix for automated snapshots, the old snapshots will also be left on the cluster.

To simplify maintenance, it is a good idea to observe certain guidelines when using save and restore:

- Create dedicated directories for use as the paths for VoltDB snapshots.
- Do not store any other files in the directories used for VoltDB snapshots.
- Periodically cleanup the directories by deleting obsolete, unused snapshots.

You can delete snapshots manually. To delete a snapshot, use the unique identifier, which is applied as a filename prefix, to find all of the files in the snapshot. For example, the following commands remove the snapshot with the ID TestSave from the directory /etc/voltdb/backup/. Note that VoltDB separates the prefix from the remainder of the file name with a dash for manual snapshots:

```
$ rm /etc/voltdb/backup/TestSave-*
```

However, it is easier if you use the system procedures VoltDB provides for managing snapshots. If you delete snapshots manually, you must make sure you execute the commands on all nodes of the cluster. When you use the system procedures, VoltDB distributes the operations across the cluster automatically.

VoltDB provides several system procedures to assist with the management of snapshots:

- @Statistics "SNAPSHOTSTATUS" provides information about the most recently performed snapshots for the current database. The response from @Statistics for this selector includes information about up to ten recent snapshots, including their location, when they were created, how long the save took, whether they completed successfully, and the size of the individual files that make up the snapshot. See the reference section on @Statistics for details.
- @SnapshotScan lists all of the snapshots available in a specified directory path. You can use this system procedure to determine what snapshots exist and, as a consequence, which ought to be deleted. See the reference section on @SnapshotScan for details.
- @SnapshotDelete deletes one or more snapshots based on the paths and prefixes you provide. The parameters to the system procedure are two string arrays. The first array specifies one or more directory paths. The second array specifies one or more prefixes. The array elements are taken in pairs to determine which snapshots to delete. For example, if the first array contains paths A, B, and C and the second array contains the unique identifiers X, Y, and Z, the following three snapshots will be deleted: A/X, B/Y, and C/Z. See the reference section on @SnapshotDelete for details.

13.5. Special Notes Concerning Save and Restore

The following are special considerations concerning save and restore that are important to keep in mind:

- Save and restore do not check the cluster health (whether all nodes exist and are running) before executing. The user can find out what nodes were saved by looking at the messages displayed by the save operation.
- Both the save and restore calls do a pre-check to see if the action is likely to succeed before the actual save/restore is attempted. For save, VoltDB checks to see if the path exists, if there is any data that might be overwritten, and if it has write access to the directory. For restore, VoltDB verifies that the saved data can be restored completely.
- It is possible to provide additional protection against failure by copying the automated snapshots to remote locations. Automated snapshots are saved locally on the cluster. However, you can set up a

network process to periodically copy the snapshot files to a remote system. (Be sure to copy the files from all of the cluster nodes.) Another approach would be to save the snapshots to a SAN disk that is already set up to replicate to another location. (For example, using iSCSI.)

Chapter 14. Command Logging and Recovery

By executing transactions in memory, VoltDB, frees itself from much of the management overhead and I/O costs of traditional database products. However, accidents do happen and it is important that the contents of the database be safeguarded against loss or corruption.

Snapshots provide one mechanism for safeguarding your data, by creating a point-in-time copy of the database contents. But what happens to the transactions that occur between snapshots?

Command logging provides a more complete solution to the durability and availability of your VoltDB database. Command logging keeps a record of every transaction (that is, stored procedure) as it is executed. Then, if the servers fail for any reason, the database can restore the last snapshot and "replay" the subsequent logs to re-establish the database contents in their entirety.

The key to command logging is that it logs the invocations, not the consequences, of the transactions. A single stored procedure can include many individual SQL statements and each SQL statement can modify hundreds or thousands of table rows. By recording only the invocation, the command logs are kept to a bare minimum, limiting the impact the disk I/O will have on performance.

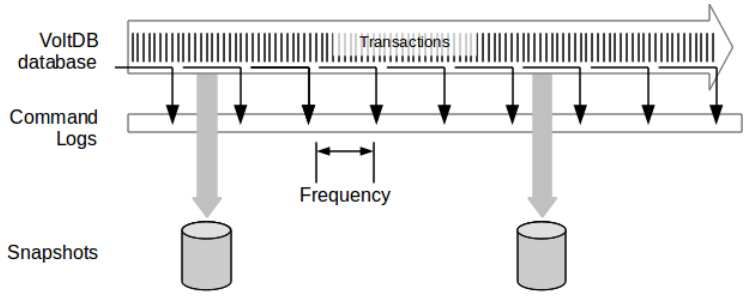
Howev'er, any additional processing can impact overall performance, especially when it involves disk I/O. So it is important to understand the tradeoffs concerning different aspects of command logging and how it interacts with the hardware and any other options you are utilizing. The following sections explain how command logging works and how to configure it to meet your specific needs.

14.1. How Command Logging Works

When command logging is enabled, VoltDB keeps a log of every transaction (that is, stored procedure) invocation. At first, the log of the invocations are held in memory. Then, at a set interval the logs are physically written to disk. Of course, at a high transaction rate, even limiting the logs to just invocations, the logs begin to fill up. So at a broader interval, the server initiates a snapshot. Once the snapshot is complete, the command logging process is able to free up — or "truncate" — the log keeping only a record of procedure invocations since the last snapshot.

This process can continue indefinitely, using snapshots as a baseline and loading and truncating the command logs for all transactions since the last snapshot.

Figure 14.1. Command Logging in Action

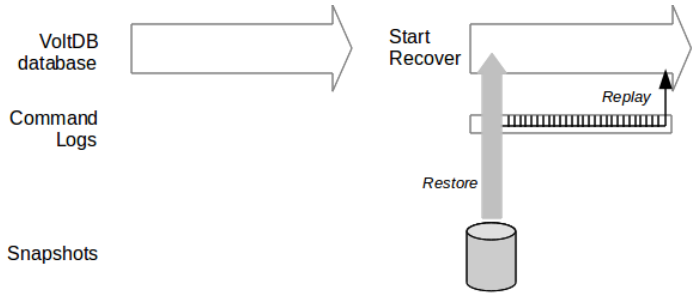


The frequency with which the transactions are written to the command log is configurable (as described in Section 14.3, “Configuring Command Logging for Optimal Performance”). By adjusting the frequency and

type of logging (synchronous or asynchronous) you can balance the performance needs of your application against the level of durability desired.

In reverse, when it is time to "replay" the logs, you start the database and the server nodes establish a quorum, the first thing the database servers do is restore the most recent snapshot. Then they replay all of the transactions in the log since that snapshot.

Figure 14.2. Recovery in Action



14.2. Controlling Command Logging

Command logging is enabled by default. Using command logging is recommended to ensure durability of your data. However, you can choose whether to have command logging enabled or not using the `deployment.commandlog` property in the configuration. For example:

```
deployment:
  commandlog:
    enabled: true
```

In its simplest form, the `commandlog` property enables or disables command logging by setting `enabled` to "true" or "false". You can also use other subproperties to control specific characteristics of command logging. The following section describes those options in detail.

14.3. Configuring Command Logging for Optimal Performance

Command logging can provide complete durability, preserving a record of every transaction that is completed before the database stops. However, the amount of durability must be balanced against the performance impact and hardware requirements to achieve effective I/O.

VoltDB provides three settings you can use to optimize command logging:

- The amount of disk space allocated to the command logs
- The frequency between writes to the command logs
- Whether logging is synchronous or asynchronous

The following sections describe these options. A fourth section discusses the impact of storage hardware on the different logging options.

14.3.1. Log Size

The command log size specifies how much disk space is preallocated for storing the logs on disk. The logs are divided into three "segments" Once a segment is full, it is written to a snapshot (as shown in Figure 14.1, “Command Logging in Action”).

For most workloads, the default log size of one gigabyte is sufficient. However, if your workload writes large volumes of data or uses large strings for queries (so the procedure invocations include large parameter values), the log segments fill up very quickly. When this happens, VoltDB can end up snapshotting continuously, because by the time one snapshot finishes, the next log segment is full.

To avoid this situation, you can increase the total log size, to reduce the frequency of snapshots. You define the log size in the configuration using the `logsize` property. Specify the desired log size as an integer number of megabytes. For example:

```
deployment:
  commandlog:
    enabled: true
    logsize: 3072
```

When increasing the log size, be aware that the larger the log, the longer it may take to recover the database since any transactions in the log since the last snapshot must be replayed before the recovery is complete. So, while reducing the frequency of snapshots, you also may be increasing the time needed to restart.

The minimum log size is three megabytes. Note that the log size specifies the *initial* size. If the existing segments are filled before a snapshot can truncate the logs, the server will allocate additional segments.

14.3.2. Log Frequency

The log frequency specifies how often transactions are written to the command log. In other words, the interval between writes, as shown in Figure 14.1, “Command Logging in Action”. You can specify the frequency in either or both time and number of transactions.

For example, you might specify that the command log is written every 200 milliseconds or every 10,000 transactions, whichever comes first. You do this by adding the `frequency` property and specifying the individual frequencies as attributes. For example:

```
deployment:
  commandlog:
    enabled: true
    frequency:
      time: 200
      transactions: 10000"
```

Time frequency is specified in milliseconds and transaction frequency is specified as the number of transactions. You can specify either or both types of frequency. If you specify both, whichever limit is reached first initiates a write.

14.3.3. Synchronous vs. Asynchronous Logging

If the command logs are being written *asynchronously* (which is the default), results are returned to the client applications as soon as the transactions are completed. This allows the transactions to execute uninterrupted.

However, with asynchronous logging there is always the possibility that a catastrophic event (such as a power failure) could cause the cluster to fail. In that case, any transactions completed since the last write and before the failure would be lost. The smaller the frequency, the less data that could be lost. This is how you "dial up" the amount of durability you want using the configuration options for command logging.

In some cases, no loss of data is acceptable. For those situations, it is best to use *synchronous logging*. When you select synchronous logging, no results are returned to the client applications until those transactions are written to the log. In other words, the results for all of the transactions since the last write are held on the server until the next write occurs.

The advantage of synchronous logging is that no transaction is "complete" and reported back to the calling application until it is guaranteed to be logged — no transactions are lost. The obvious disadvantage of synchronous logging is that the interval between writes (i.e. the frequency) while the results are held, adds to the latency of the transactions. To reduce the penalty of synchronous logging, you need to reduce the frequency.

When using synchronous logging, it is recommended that the frequency be limited to between 1 and 4 milliseconds to avoid adding undue latency to the transaction rate. A frequency of 1 or 2 milliseconds should have little or no measurable affect on overall latency. However, low frequencies can only be achieved effectively when using appropriate hardware (as discussed in the next section, Section 14.3.4, “Hardware Considerations”).

To select synchronous logging, use the `synchronous` property. For example:

```
deployment:
  commandlog:
    enabled: true
    synchronous: true
    frequency:
      time:2
```

14.3.4. Hardware Considerations

Clearly, synchronous logging is preferable since it provides complete durability. However, to avoid negatively impacting database performance you must not only use very low frequencies, but you must have storage hardware that is capable of handling frequent, small writes. Attempting to use aggressively low log frequencies with storage devices that cannot keep up will also hurt transaction throughput and latency.

Standard, uncached storage devices can quickly become overwhelmed with frequent writes. So you should not use low frequencies (and therefore synchronous logging) with slower storage devices. Similarly, if the command logs are competing for the device with other disk I/O, performance will suffer. So do not write the command logs to the same device that is being used for other I/O, such as snapshots or export overflow.

On the other hand, fast, cached devices such as disks with a battery-backed cache, are capable of handling frequent writes. So it is strongly recommended that you use such devices when using synchronous logging.

To specify where the command logs and their associated snapshots are written, you use command log sub-properties of the `deployment.paths` property. For example, the following example specifies that the logs are written to `/fastdisk/voltdblog` and the snapshots are written to `/opt/voltdb/cmdsnaps`:

```
deployment:
  paths:
    commandlog:
      path: /fastdisk/voltdblog/
```

```
commandlogsnapshot:
  path: /opt/voltdb/cmdsnaps/
```

Note that the default paths for the command logs and the command log snapshots are both subfolders of the voltdbroot directory. To avoid overloading a single device on production servers, it is recommended that you specify an explicit path for the command logs, at a minimum, and preferably for both logs and snapshots.

To summarize, the rules for balancing command logging with performance and throughput on production databases are:

- Use asynchronous logging with slower storage devices.
- Write command logs to a dedicated device. Do not write logs and snapshots to the same device.
- Use low (1-2 milisecond) frequencies when performing synchronous logging.
- Use moderate (100 millisecond or greater) frequencies when performing asynchronous logging.

Chapter 15. Streaming Data: Import, Export, and Migration

Earlier chapters discuss features of VoltDB as a standalone component of your business application. But like most technologies, VoltDB is often used within a diverse and heterogeneous computing ecosystem where it needs to "play well" with other services This chapter describes features of VoltDB that help integrate it with other databases, systems, and applications to simplify, automate, and speed up your business processes.

Just as VoltDB as a database aims to provide the optimal transaction throughput, VoltDB as a data service aims to efficiently and reliably transfer data to and from other services. Of course, you can always write custom code to integrate VoltDB into your application environment, calling stored procedures to move data in and out of the database. However, the VoltDB feature set simplifies and automates the process of streaming data into, out of, and through VoltDB allowing your application to focus on the important work of analyzing, processing, and modifying the data in flight through secure, reliable transactions. To make this possible, VoltDB introduces five key concepts:

- Streams
- Import
- Export
- Migration
- Topics

Streams operate much like regular database tables. You define them with a CREATE statement like tables, they consist of columns and you insert data into streams the same way you insert data into tables using the INSERT statement. You can define views that aggregate the data as it passes through the stream. Interactions with streams within a stored procedure are transactional just like tables. The only difference is a stream *does not store any data* in the database. This allows you to use all the consistency and reliability of a transactional database and the familiar syntax of SQL to manage data "in flight" without necessarily having to save it to persistent storage. Of course, since there is no storage associated with streams, they are for INSERT only. Any attempt to SELECT, UPDATE, or DELETE data from a stream results in an error.

Import automates the process of pulling data from external sources and inserting it into the database workflow through the same stored procedures your applications use. The import connectors are declared as part of the database configuration and stop and start with the database. The key point being that the database manages the entire import process and ensures the durability of the data while it is within VoltDB. Alternately, you can use one of the VoltDB data loading utilities to push data into the VoltDB database from a variety of sources.

Export automates the reverse process from import: it manages copying any data written to an export table or stream and sending it to the associated external target, whether it be a file, a service such as Kafka, or another database. The export targets are defined in the database configuration, while the connection of a table or stream to it specific export target is done in the data definition language (DDL) CREATE statement using the EXPORT TO TARGET clause.

Topics are similar to import and export in that topics let you stream data into and out of the VoltDB database. The differences are that a single topic can perform both import and output, there can be multiple consumers and producers for a single topic, and it is the external producers and consumers that control how

and when data is transferred rather than VoltDB pulling from and pushing to individual external targets. You identify the stream or table to use for output to the topic by specifying EXPORT TO TOPIC in the CREATE STREAM or CREATE TABLE statement. You then configure the topic, including the stored procedure to use for input, in the configuration file. Another difference between export and topics is that, because topics do not have a single output consumer, there is no single event that determines when the data transfer is complete. Instead, you must define a retention/expiration policy (based on time or size) for when data is no longer needed and can be deleted from the queue.

Migration is a special case where export is more fully integrated into the business workflow. When you define a table or view with the MIGRATE TO... clause instead of EXPORT TO..., data is not deleted from the VoltDB table or view until it is successfully written to the associated target or topic. You trigger a migration of data using an explicit MIGRATE statement or you can declare the schema object with USING TTL to schedule the migration based on a timestamp within the data records and an expiration time defined as the TTL value.

How you configure these features depends on your specific business goals. The bulk of this chapter describes how to declare and configure import, export and migration in detail. The next two sections provide an overview of how data streaming works and how to use these features to perform common business activities.

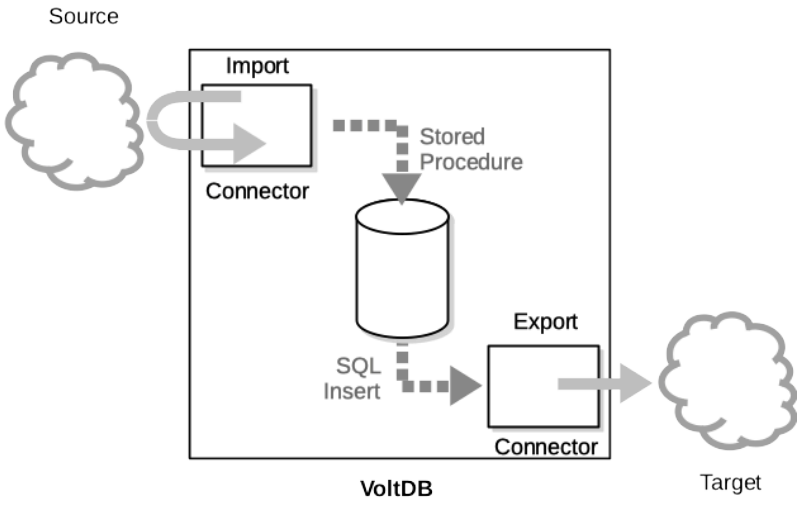
15.1. How Data Streaming Works in VoltDB

Import associates incoming data with a stored procedure that determines what is done with the data. Export associates a database object (a table or stream) with an external target, where the external target determines how the exported data is handled. But in both cases the handling of streamed data follows three key principles:

- Interaction with the VoltDB database is transactional, providing the same ACID guarantees as all other transactions.
- Interaction with the external system occurs as a separate asynchronous process, avoiding any negative impact on the latency of ongoing transactions in the VoltDB database.
- The VoltDB server takes care of starting and stopping the import and export subsystems when the database starts and stops. The server also takes responsibility for managing streaming data "in flight" — ensuring that no data is lost once it enters the subsystem and before it reaches its final destination.

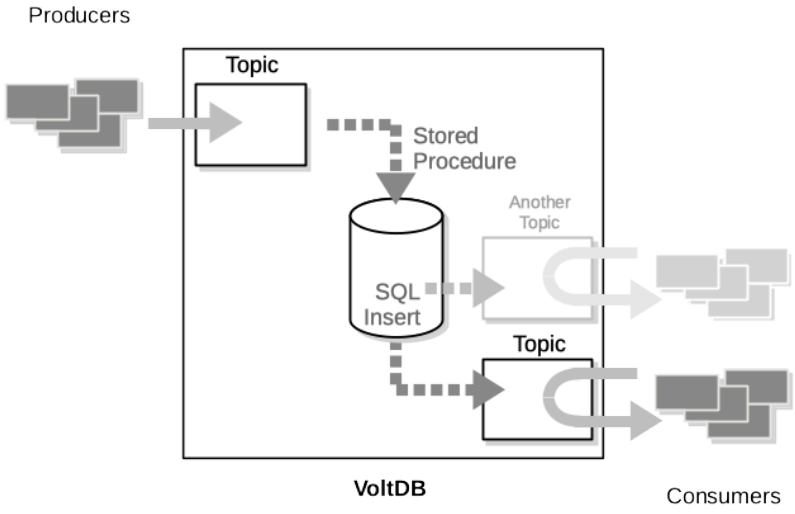
VoltDB database achieves these goals is by having separate export and import connectors handle the data as it passes from one system to the next as shown in Figure 15.1, “Overview of Data Streaming”.

Figure 15.1. Overview of Data Streaming



In the case of topics, there is no specific source or target; multiple producers and consumers can write to and read from the topic. And the stored procedure that receives the incoming data can do whatever you choose with that content: it can write it to the stream as output for the same topic, it can write into other topics, it can write into other database tables, or any combination, providing the ultimate flexibility to meet your business logic needs, as shown in Figure 15.2, “Overview of Topics”.

Figure 15.2. Overview of Topics



Which streaming features you use depend on your business requirements. The key point is that orchestrating multiple disparate systems is complex and error prone and the VoltDB streaming services free you from these complexities by ensuring that all operations start and stop automatically as part of the server process, the data in flight is made durable across database sessions, and that all data is delivered at least once or retained until delivery is possible.

The following sections provide an overview of each service. Later sections describe the services and built-in connectors in more detail. You can also define your own custom import and export connectors, as described in the *VoltDB Guide to Performance and Customization*.

15.1.1. Understanding Import

To import data into VoltDB from an external system you have two options: you can use one of the standard VoltDB data loading utilities (such as csvloader) or you can define an import connector in the database configuration that associates the external source with a stored procedure. The data loading utilities are standalone external applications that push data into the VoltDB database. VoltDB import connectors use a pull model. In other words, the connector periodically checks the data source to determine if new content is available. If so, the connector retrieves the data and passes it to the stored procedure where it can analyze the data, validate it, manipulate it, insert it into the database, or even pass it along to an export stream; whatever your application needs.

The creation of the import connector is done using the `import` property and the list elements of the `configurations` subproperty in the configuration. The child properties of the `configurations` elements specify the type of import connector to use (Kafka or custom) and, optionally, the input format (CSV by default). Within the connector definition, the `properties` list elements specify the actual data source, the stored procedure to use as a destination, and any other connector-specific attributes you wish to set.

For example, to process data from a Kafka topic, the connector definition must specify the type (Kafka), the addresses of one or more Kafka brokers as the source, the name of the topic (or topics), and the stored procedure to process the data. If the data does not need additional processing, you can use the default stored procedure that VoltDB generates for each table to insert the data directly into the database. The following configuration reads the Kafka topics *nyse* and *nasdaq* in CSV format and inserts records into the *stocks* table using the default insert procedure:

```
deployment:
  import:
    configurations:
    - type: kafka
      format: csv
      properties:
      - name: brokers
        value: kafkasvr1:9092,kafkasvr2:9092
      - name: topics
        value: nyse,nasdaq
      - name: procedure
        value: STOCKS.insert
```

Having the import connectors defined in the configuration lets VoltDB manage the entire import process, from starting and stopping the connectors to making sure the specified stored procedure exists, fetching the data in batches and ensuring nothing is lost in transit. You can even add, delete, or modify the connector definitions on the fly by updating the database configuration while the database is running.

VoltDB provides a built-in import connector for Kafka. Section 15.4, “VoltDB Import Connectors” describes the built-in connector and its required and optional properties. Section 15.5, “VoltDB Import Formatters” provides additional information about the input formatters that prepare the incoming data for the stored procedure.

15.1.2. Understanding Export

To export data from VoltDB to an external system you define a database table or stream as the export source by including the EXPORT TO TARGET clause in the DDL definition and associating that data source with a logical target name. For example, to associate the stream *alerts* with a target called *systemlog*, you would declare a stream like so:

```
CREATE STREAM alerts
   EXPORT TO TARGET systemlog
   ( {column-definition} [,...] );
```

For tables, you can also specify *when* data is queued for export. By default, data inserted into export tables with the INSERT statement (or UPSERT that results in a new record being inserted) is queued to the target, similar to streams. However, you can customize the export to write on any combination of data manipulation language (DML) statements, using the ON clause. For example, to include updates into the export steam, the CREATE TABLE statement might look like this:

```
CREATE TABLE orders
   EXPORT TO TARGET orderprocessing ON INSERT, UPDATE
   ( {column-definition} [,...] );
```

As soon as you declare a stream or table as exporting to a target, any data written to that source (or in the case of tables, the export actions you specified in the CREATE TABLE statement) is queued for the export stream. You associate the named target with a specific connector and external system in the `export` section of the database configuration. Note that you can define the target either before or after declaring the source, and you can add, remove, or modify the export configuration at any time before or after the database is started.

In the configuration you define the export connector as a list element of the `configurations` property, identifying the target name and type of connector to use. You identify the specific external target to use and any necessary connector-specific attributes as properties of the connector. For example, to write export data to files locally on the database servers, you use the file connector and specify attributes such as the file prefix, location, and roll-over frequency as properties:

```
deployment:
 export:
   configurations:
   - target: systemlog
     type: file
     properties:
     - name: type
       value: csv
     - name: nonce
       value: syslog
     - name: period
       value: 60     # roll every hour (60 minutes)
```

VoltDB supports built-in connectors for five types of external targets: file, HTTP (including Hadoop), JDBC, Kafka, and Elasticsearch. Each export connector supports different properties specific to that type of target. Section 15.3, “VoltDB Export Connectors” describes the built-in export connectors and the required and optional properties for each.

15.1.3. Understanding Migration

Migration is a special case of export that synchronizes export with the deletion of data in database tables. You can migrate data from either regular database tables or stream views. And you can migrate the data to either an export target or to a topic.

When you migrate a record, VoltDB ensures the data is successfully transmitted to (and acknowledged by) the target before the data is deleted from the database. In the case of export to a target, VoltDB waits until it receives acknowledgement that the data has reached the external system before deleting it. For export to a topic, the data is deleted as soon as it is available to topic consumers. Either way, migration ensures that the data is always available from one of the two systems — it cannot temporarily "disappear" during the move.

You define a VoltDB table as a source of migration using the MIGRATE TO... clause, the same way you define an export source with the EXPORT TO... clause. For example, the following CREATE TABLE statement defines the *orders* table as a source for migration to the *oldorders* target connector:

```
CREATE TABLE orders
   MIGRATE TO TARGET oldorders
   ( {column-definition} [,...] );
```

Migration uses the export subsystem to perform the interaction with the external data store. So you can use any of the supported connectors to configure the target of the migration; and you do so the exact same way you do for any other export target. The difference is that rather than exporting the data when it is inserted into the table, the data is exported when you initiate migration.

You trigger migration at run time using the MIGRATE SQL statement and a WHERE clause to identify the specific rows to move. For example, to migrate all of the orders for a specific customer, you could use the following MIGRATE statement:

```
MIGRATE FROM orders
   WHERE custmer_id = ? AND NOT MIGRATING;
```

Note the use of NOT MIGRATING. MIGRATING is a special function that identifies all rows that are *currently* being migrated; that is, where migration (and deletion) has not yet completed. Although not required — VoltDb will skip rows that are already migrating — adding AND NOT MIGRATING to a MIGRATE statement can improve performance by reducing the number of rows evaluated by the expression.

Once the rows are migrated and the external target acknowledges receipt, the rows are deleted from the database.

To further automate the migration of data to external targets, you can use the MIGRATE TO... clause with USING TTL. USING TTL automates the deletion of records based on a TTL value and a TIMESTAMP column in the table. For example, adding the clause `USING TTL 12 HOURS ON COLUMN created` to a table where the `created` column defaults to NOW, means that records will be deleted from the table 12 hours after they are inserted. By adding the MIGRATE TO TARGET clause, you can tell VoltDB to migrate the data to the specified target before removing it when its TTL expiration is reached.

```
CREATE TABLE sessions
   MIGRATE TO TARGET sessionlog
   ( session_id BIGINT NOT NULL,
     created TIMESTAMP DEFAULT NOW [,...]
   )
   USING TTL 12 HOURS ON COLUMN created;
```

15.1.4. Understanding Topics

Topics allow you to integrate both import and export into a single stream. They also allow multiple external producers and consumers to access the topic at the same time, keeping track of where each consumer or group of consumers is in the stream of output.

There are actually two distinct and independent components to a topic that you control separately: input and output. You declare a topic having either or both, depending on the schema and configuration. The schema associates individual streams or tables with topics and the configuration defines the properties of the topic, including what stored procedure to use for input. For example, you can declare an output-only topic by specifying the topic in the CREATE STREAM.... EXPORT TO TOPIC statement but specifying no stored procedure in the configuration. In this case, any records written to the associated stream are queued for output and available to any consumers of the topic:

```
CREATE STREAM session EXPORT TO TOPIC sessions ...
```

If, on the other hand, you specify a stored procedure in the configuration, records written to the topic by producers invoke the specified procedure passing the message contents (and, optionally, the key) as arguments:

```
deployment:
  topics:
    topic:
    - name: sessions
      procedure: ProcessSessions
```

If you include both the EXPORT TO TOPIC clause in the CREATE STEAM statement and the `procedure` property in the `topic` list element, the topic is available for both input and output. What happens to the data as it passes through VoltDB is up to you. You can simply pass it from producers to consumers by taking the data received by the input procedure and inserting it into the associated stream. Or the stored procedure can filter, modify, or redirect the content as needed. For example, the following data definitions create a topic where the input procedure uses an existing table in the database (*users*) to fill out additional fields based on the matching *username* in the incoming records while writing the data to the stream for output:

Schema

```
CREATE TABLE tempuser ( username VARCHAR(128) NOT NULL);
CREATE TABLE users ( username VARCHAR(128)  NOT NULL,
    country VARCHAR(32), userrank INTEGER);
PARTITION TABLE tempuser on column username;
PARTITION TABLE users on column username;

CREATE STREAM session
  EXPORT TO TOPIC sessions
  PARTITION ON COLUMN username (
    username VARCHAR(128) NOT NULL,
    login TIMESTAMP, country VARCHAR(32), userrank INTEGER);

CREATE PROCEDURE ProcessSessions
  PARTITION ON TABLE users COLUMN username
  AS BEGIN
   INSERT INTO tempuser VALUES(CAST(? AS VARCHAR));
   INSERT INTO session SELECT u.username,
     CAST(? AS TIMESTAMP), u.country, u.userrank
     FROM users AS u, tempuser AS t
     WHERE u.username=t.username;
   TRUNCATE TABLE tempuser;
  END;
```

Configuration

```
deployment:
  topics:
    topic:
    - name: sessions
      procedure: ProcessSessions
```

Finally, if you want to create a topic that is not processed but simply flows through VoltDB from producers to consumers, you declare the topic as "opaque" in the configuration, without either specifying a stored procedure for input or associating a stream with the topic for output.

```
deployment:
  topics:
    topic:
    - name: sysmsgs
```

```
      opaque: true
```

Opaque topics are useful if you want to have a single set of brokers for all your topics but only need to analyze and process some of the data feeds. Opaque topics let VoltDB handle the additional topics without requiring the stored procedure or stream definitions needed for processed topics.

15.2. The Business Case for Streaming Data

The streaming features of VoltDB provide a robust and flexible set of capabilities for connecting a VoltDB database to external systems. They can be configured in many different ways. At the most basic, they let you automate the import and export data from a VoltDB database. The following section demonstrate other ways these capabilities can simplify and automate common business processes, including:

- Section 15.2.1, “Extract, Transform, Load (ETL)”
- Section 15.2.2, “Change Data Capture”
- Section 15.2.3, “Streaming Data Validation”
- Section 15.2.4, “Caching”
- Section 15.2.5, “Archiving”

15.2.1. Extract, Transform, Load (ETL)

Extract, transform, load (ETL) is a common business pattern where you extract data from a database, restructure and repurpose it, then load into another system. For example, an order processing database might have separate tables for customer data, orders, and product information. When it comes time to ship the order, information from all three tables is needed: the customer ID and product SKU from the order, the name and address from the customer record, and the product name and description from the product table. This information is merged and passed to the shipping management system.

Rather than writing a separate application to perform these tasks, VoltDB lets you integrate them in a single stored procedure. By creating a stream with the appropriate columns for the transformed data and assigning it as an export source and defining a target that matches the shipping management system, you can declare single stored procedure to complete the process:

```
CREATE STREAM shipping
   EXPORT TO TARGET shipmgtsystem
   ( order_number BIGINT,
     prod_sku BIGINT,
     prod_name VARCHAR(64),
     customer_name VARCHAR(64),
     customer_address VARCHAR(128) );
CREATE PROCEDURE shiporder AS
   INSERT INTO shipping SELECT
      o.id, p.sku, p.name, c.name, c.address
      FROM orders AS o, products AS p, customers AS c
      WHERE o.id = ? AND
         o.sku = p.sku  AND o.customer_id = c.id;
```

15.2.2. Change Data Capture

Change Data Capture is the process of recording all changes to the content of a database. Those changes can then be reused by inserting into another repository for redundancy, logging to a file, merging into another database or whatever the business workflow call for.

VoltDB simplifies change data capture by allowing you to export all or any subset of data changes to a table to any of the available export targets or topics. When you declare a table as an export source with the EXPORT TO TARGET or EXPORT TO TOPIC clause you can specify which actions trigger export using ON. Possible triggers are INSERT, UPDATE, UPDATE_NEW, UPDATE_OLD, and DELETE.

INSERT and DELETE are self-explanatory. UPDATE, on the other hand, generates two export records: one for the row before the update and one for the row after the update. To select only one or these records, you can use the actions UPDATE_OLD or UPDATE_NEW.

For change data capture, you can export all changes by specifying ON INSERT, UPDATE, DELETE. For example, the following schema definitions ensure that all data changes for the tables *products* and *orders* are exported to the targets *offsiteprod* and *offsiteorder*, respectively:

```
CREATE TABLE products EXPORT TO TARGET offsiteprod
   ON INSERT, UPDATE, DELETE
   [ ... ];
CREATE TABLE orders EXPORT TO TARGET offsiteorder
   ON INSERT, UPDATE, DELETE
   [ ... ];
```

Note that the built-in export connectors include six columns of metadata at the beginning of the export data by default. For change data capture, the most important piece of metadata is the sixth column, which is a single byte value that indicates which action triggered the export. The external target can use this information to determine what to do with the record. The possible values for the operation indicator are shown in Table 15.2, “Export Metadata”.

The metadata is included by default for standard export targets. However, they are *not* included by default when exporting to topics. To include this information in the topic output, you must configure the topic to not skip the internal metadata by setting the property `consumer.skip.internals` to *false*. For example, if you are exporting the *products* table to a topic called *productstream*, the schema includes the ON clause to the EXPORT TO TOPIC statement:

```
CREATE TABLE products EXPORT TO TOPIC productstream
   ON INSERT, UPDATE, DELETE
   [ ... ];
```

But you must also configure the topic to include the change data capture metadata in the output:

```
topics:
  topic:
    name: productstream
    properties:
    - name: consumer.skip.internals
      value: false
```

15.2.3. Streaming Data Validation

VoltDB provides the necessary speed and features to implement an intelligent data pipeline — where information passing through a high performance stream is analyzed, validated and then accepted, rejected, or modified as necessary and passed on to the next stage of the pipeline. In this use case, the data in VoltDB is used as reference for comparison with the influx of data in the pipeline. VoltDB import connectors accept the incoming data, where it is submitted to a stored procedure. The stored procedure analyses the data against the reference tables, then inserts the validated content into a stream which is in turn declared as a source for an export connector that sends it along to its next target.

For example, VoltDB can be inserted into a Kafka pipeline by using:

- A Kafka import connector as the input
- A VoltDB stream and a Kafka export connector as the output
- A stored procedure analyzing the input and inserting it into the stream

The following schema and configuration illustrate a simple example that checks if the data in a Kafka stream matches an existing user account with appropriate funds. The schema uses a reference table (account), a temporary table (incoming), and an export stream (outgoing). Any data matching the requirements is written to the export target; all other incoming data is dropped.

```
CREATE TABLE account
   ( user_id BIGINT, username VARCHAR(25), balance BIGINT );
CREATE TABLE incoming
   ( trans_id BIGINT, amount BIGINT, user_id BIGINT );
CREATE STREAM outgoing EXPORT TO TARGET kafka_output
   ( trans_id BIGINT, amount BIGINT, user_id BIGINT );

CREATE PROCEDURE validate AS
  BEGIN
    INSERT INTO incoming (?,?,?);
    INSERT INTO outgoing
       SELECT i.trans_id, i.amount, i.user_id
          FROM incoming AS i, account AS a
          WHERE i.user_id = a.user_id AND a.balance + i.amount > 0;
    TRUNCATE TABLE incoming;
  END;

deployment:
  import:
    configurations:
    - type: kafka
      properties:
      - name: procedure
        value: validate
      - name: brokers
        value: kfkasrc1,kfksrc2
      - name: topics
        value: transactions
  export:
    configurations:
      - target: kafka_output
        type: kafka
        properties:
        - name: bootstrap.servers
          value: kfkdest1,kfkdest2
        - name: topic.key
          value: outgoing.transactions
        - name: skipinternals
          value: true
```

15.2.4. Caching

Because of its architecture, VoltDB is excellent at handling high volume transactions. It is not as well suited for ad hoc analytical processing of extremely large volumes of historical data. But sometimes you

need both. Caching allows current, high touch content to be accessible from a fast front-end repository while historical, less frequently accessed content is stored in slower, large back-end repositories (such as Hadoop) sometimes called *data lakes*.

Export, Time To Live (TTL), and automated tasks help automate the use of VoltDB as a hot cache. By declaring tables in VoltDB as export sources to a large back-end repository, any data added to VoltDB automatically gets added to the historical data lake. Once data in VoltDB is no longer "hot", it can be deleted but remains available from larger back-end servers.

In the simplest case, caching can be done by declaring the VoltDB tables with EXPORT TO TARGET and using ON INSERT, UPDATE_NEW so all data changes except deletes are exported to the data lake. You can then manually delete data from VoltDB when it becomes unnecessary in the cache.

```
CREATE TABLE sessions
   EXPORT TO TARGET historical ON INSERT, UPDATE_NEW
   ( id BIGINT NOT NULL,
     login TIMESTAMP, last_access TIMESTAMP [,...] );
```

To make it easier, VoltDB can automate the process of aging out old data. If the content is time sensitive, you can add USING TTL to the table declaration to automatically delete records once a column exceeds a certain time limit. You specify the reference column and the time limit in the USING TTL clause. For example, if you want to automatically delete any sessions that have not been accessed for more than two hours, you can change the *sessions* table declaration like so:

```
CREATE TABLE sessions
   EXPORT TO TARGET historical ON INSERT, UPDATE_NEW
   ( id BIGINT NOT NULL, user_id BIGINT,
     login TIMESTAMP, last_access TIMESTAMP [,...] )
   USING TTL 2 hours ON COLUMN last_access;
```

If your expiration criteria is more complex than a single column value, you can use a stored procedure to identify rows that need deleting. To automate this process, you then define a task that executes the stored procedure on a regular basis. For example, if you want to remove sessions more frequently if there is no access after the initial login, you can define a stored procedure *GhostSessions* to remove inactive sessions, then execute that procedure periodically with the task *RemoveGhosts*. Note that the actual time limit can be made adjustable by a parameter passed to the task.

```
CREATE PROCEDURE GhostSessions AS
   DELETE FROM sessions
     WHERE login = last_access AND DATEADD(MINUTE,?,login) < NOW;
CREATE TASK ON SCHEDULE EVERY 2 MINUTES
   PROCEDURE GhostSessions WITH (20); -- 20 minute limit
```

15.2.5. Archiving

Archiving is like caching in that older data is maintained in slower, large-scale repositories. The difference is that for archiving, rather than having copies of the current data in both locations, data is not moved to the archive until *after* it's usefulness in VoltDB expires.

You could simply export the data when you delete it from the VoltDB database. But since export is asynchronous, there will be a short period of time when the data is neither in VoltDB or in the archive. To avoid this situation, you can use migration rather than export, which ensures the data is not deleted from VoltDB until the export target acknowledges receipt of the migrated content.

For example, if we are archiving orders, we can include the MIGRATE TO TARGET clause in the table definition and then use the MIGRATE statement instead of DELETE to clear the records from VoltDB:

```
CREATE TABLE orders MIGRATE TO TARGET archive
     [ . . . ];
```

If you are archiving records based on age, you can use MIGRATE TO TARGET with USING TTL to automatically migrate the table rows once a specific column in the table expires. Used alone, USING TTL simply deletes records; used with MIGRATE TO TARGET it initiates a migration for the expired records:

```
CREATE TABLE orders MIGRATE TO TARGET archive
     [ . . . ]
   USING TTL 30 DAYS ON COLUMN order_completed;
```

15.3. VoltDB Export Connectors

You use the EXPORT TO TARGET or MIGRATE TO TARGET clauses to identify the sources of export and start queuing export data. To enable the actual transmission of data to an export target at runtime, you include the `export` and `configuration` properties in the configuration. You can configure the export targets when you initialize the database root directory. Or you can add or modify the export configuration while the database is running using the **voltadmin update** or **set** commands.

In the configuration file, the `configurations` list elements specify the target you are configuring and which export connector to use (with the `type` property). To export to multiple destinations, you include multiple elements in the `configurations` property, each specifying the target it is configuring. For example:

```
deployment:
  export:
    configurations:
    - target: log
      enabled: true
      type: file
    - target: archive
      enabled: true
      type: file
```

You configure each export connector by specifying properties as one or more list elements to the `properties` property. For example, the following YAML code enables export to comma-separated (CSV) text files using the file prefix "MyExport".

```
deployment:
  export:
    configurations:
    - target: log
      enabled: true
      type: file
      properties:
      - name: type
        value: csv
      - name: nonce
        value: MyExport
```

The properties that are allowed and/or required depend on the export connector you select. VoltDB comes with five export connectors:

- Export to file (type "file")

- Export to HTTP, including Hadoop (type "http")
- Export to JDBC (type "jdbc")
- Export to Kafka (type "Kafka")
- Export to Elasticsearch (type "elasticsearch")

15.3.1. How Export Works

Two important points about export to keep in mind are:

- Data is queued for export as soon you declare a stream or table with the EXPORT TO TARGET clause and write to it. Even if the export target has not been configured yet. Be careful not to declare export sources and forget to configure their targets, or else the export queues could grow and cause disk space issues. Similarly, when you drop the stream or table, its export queue is deleted, even if there is data waiting to be delivered to the configured export target.
- VoltDB will send at least one copy of every export record to the target. It is possible, when recovering command logs or rejoining nodes, that certain export records are resent. It is up to the downstream target to handle these duplicate records. For example, using unique indexes or including a unique record ID in the export stream.

All nodes in a cluster queue export data, but only one actually writes to the external target. If one or more nodes fail, responsibility for writing to the export targets is transferred to another currently active server. It is possible for gaps to appear in the export queues while servers are offline. Normally if a gap is found, it is not a problem because another node can take over responsibility for writing (and queuing) export data.

However, in unusual cases where export falls behind and nodes fail and rejoin consecutively, it is possible for gaps to occur in all the available queues. When this happens, VoltDB issues a warning to the console (and via SNMP) and waits for the missing data to be resolved. You can also use the @Statistics system procedure with the EXPORT selector to determine exactly what records are and are not present in the queues. If the gap cannot be resolved (usually by rejoining a failed server), you must use the **voltadmin export release** command to free the queue and resume export at the next available record.

15.3.1.1. Export Overflow

VoltDB uses persistent files on disk to queue export data waiting to be written to its specified target. If for any reason the export target can not keep up with the connector, VoltDB writes the excess data in the export buffer from memory to disk. This protects your database in several ways:

- If the destination target is not configured, is unreachable, or cannot keep up with the data flow, writing to disk helps VoltDB avoid consuming too much memory while waiting for the destination to accept the data.
- If the database stops, the export data is retained across sessions. When the database restarts, the connector will retrieve the overflow data and reinsert it in the export queue.

Even when the target does keep up with the flow, some amount of data is written to the overflow directory to ensure durability across database sessions. You can specify where VoltDB writes the overflow export data using the `paths` and `exportoverflow` properties in the configuration. For example:

```
deployment:
  paths:
```

```
exportoverflow:
  path: /tmp/export/
```

If you do not specify a path for export overflow, VoltDB creates a subfolder in the database root directory. See Section 3.7.2, “Configuring Paths for Runtime Features” for more information about configuring paths in the configuration.

15.3.1.2. Persistence Across Database Sessions

It is important to note that VoltDB only uses the disk storage for overflow data. However, you can force VoltDB to write all queued export data to disk using any of the following methods:

- Calling the @Quiesce system procedure
- Requesting a blocking snapshot (using **voltadmin save --blocking**)
- Performing an orderly shutdown (using **voltadmin shutdown**)

This means that if you perform an orderly shutdown with the **voltadmin shutdown** command, you can recover the database — and any pending export queue data — by simply restarting the database cluster in the same root directories.

Note that when you initialize or re-initialize a root directory, any subdirectories of the root are purged.[1] So if your configuration did *not* specify a different location for the export overflow, and you re-initialize the root directories and then restore the database from a snapshot, the database is restored but the export overflow will be lost. If both your original and new configuration use the same, explicit directory outside the root directory for export overflow, you can start a new database and restore a snapshot without losing the overflow data.

15.3.2. The File Export Connector

The file connector receives the serialized data from the export source and writes it out as text files (either comma or tab separated) to disk. The file connector writes the data out one file per source table or stream, "rolling" over to new files periodically. The filenames of the exported data are constructed from:

- A unique prefix (specified with the `nonce` property)
- A unique value identifying the current version of the database schema
- The stream or table name
- A timestamp identifying when the file was started
- Optionally, the ID of the host server writing the file

While the file is being written, the file name also contains the prefix "active-". Once the file is complete and a new file started, the "active-" prefix is removed. Therefore, any export files without the prefix are complete and can be copied, moved, deleted, or post-processed as desired.

There is only one required property that must be set when using the file connector. The `nonce` property specifies a unique prefix to identify all files that the connector writes out for this database instance. All other properties are optional and have default values.

[1] Initializing a root directory deletes any files in the command log and overflow directories. The snapshots directory is archived to a named subdirectory.

Table 15.1, “File Export Properties” describes the supported properties for the file connector.

Table 15.1. File Export Properties

Property	Allowable Values	Description
nonce*	string	A unique prefix for the output files.
type	csv, tsv	Specifies whether to create comma-separated (CSV) or tab-delimited (TSV) files. CSV is the default format.
outdir	directory path	The directory where the files are created. Relative paths are relative to the database root directory on each server. If you do not specify an output path, VoltDB writes the output files into a subfolder of the root directory itself.
period	Integer	The frequency, in minutes, for "rolling" the output file. The default frequency is 60 minutes.
retention	string	Specifies how long exported files are retained. You specify the retention period as an integer number and a time unit identifier from the following list: • s — Seconds • m — Minutes • h — Hours • d — Days For example, "31d" sets the retention limit at 31 days. After files exceed the specified time limit, they are deleted by the export subsystem. The default is to retain all files indefinitely.
binaryencoding	hex, base64	Specifies whether VARBINARY data is encoded in hexadecimal or BASE64 format. The default is hexadecimal.
charset	string	Specifies the character set encoding to use when writing VARCHAR columns to the output stream. The default character encoding is UTF-8.
dateformat	format string	The format of the date used when constructing the output file names. You specify the date format as a Java SimpleDateFormat string. The default format is "yyyyMMddHHmmss".
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java TimeZone identifier. For example, you can specify a continent and region ("America/New_York") or a time zone acronym ("EST"). The default is GMT.
delimiters	string	Specifies the delimiter characters for CSV output. The text string specifies four characters in the following order: the separator, the quote character, the escape character, and the end-of-line character. Non-printing characters must be encoded as Java literals. For example, the new line character (ASCII code 13) should be entered as "\n". Alternately, you can use Java Unicode literals, such as "\u000d". Enclose strings containing special characters such as the backslash in single quotation marks. The following property definition matches the default delimiters. That is, the comma, the double quotation character twice (as both the quote and escape delimiters) and the new line character:

Property	Allowable Values	Description

		`property:`
		`  name: delimiter`
		`  value: ',""\n'`

batched	true, false	Specifies whether to store the output files in subfolders that are "rolled" according to the frequency specified by the period property. The subfolders are named according to the nonce and the timestamp, with "active-" prefixed to the subfolder currently being written.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skipinternals as "true", the output files contain only the exported data.
uniquenames	true, false	Specifies whether to include the host ID in the file name to ensure that all files written are unique across a cluster. The export files are always unique per server. But if you plan to write all cluster files to a network drive or copy them to a single location, set this property to true to avoid any possible conflict in the file names. The default is false.
with-schema	true, false	Specifies whether to write a JSON representation of the source's schema as part of the export. The JSON schema files can be used to ensure the appropriate datatype and precision is maintained if and when the output files are imported into another system.

*Required

Whatever properties you choose, the order and representation of the content within the output files is the same. The export connector writes a separate line of data for every INSERT it receives, including the following information:

- Six columns of metadata generated by the export connector.
- The remaining columns are the columns of the database source, in the same order as they are listed in the database definition (DDL) file.

Table 15.2, “Export Metadata” describes the six columns of metadata generated by the export connector and the meaning of each column.

Table 15.2. Export Metadata

Column	Datatype	Description
Transaction ID	BIGINT	Identifier uniquely identifying the transaction that generated the export record.
Timestamp	TIMESTAMP	The time when the export record was generated.
Sequence Number	BIGINT	For internal use.
Partition ID	BIGINT	Identifies the partition that sent the record to the export target.
Site ID	BIGINT	Identifies the site that sent the record to the export target.
Export Operation	TINYINT	A single byte value identifying the type of transaction that initiated the export. Possible values include: • 1 — insert • 2 — delete • 3 — update (record before update) • 4 — update (record after update)

Column	Datatype	Description
		• 5 — migration

15.3.3. The HTTP Export Connector

The HTTP connector receives the serialized data from the export streams and writes it out via HTTP requests. The connector is designed to be flexible enough to accommodate most potential targets. For example, the connector can be configured to send out individual records using a GET request or batch multiple records using POST and PUT requests. The connector also contains optimizations to support export to Hadoop via WebHDFS.

15.3.3.1. Understanding HTTP Properties

The HTTP connector is a general purpose export utility that can export to any number of destinations from simple messaging services to more complex REST APIs. The properties work together to create a consistent export process. However, it is important to understand how the properties interact to configure your export correctly. The four key properties you need to consider are:

- **batch.mode** — whether data is exported in batches or one record at a time
- **method** — the HTTP request method used to transmit the data
- **type** — the format of the output
- **endpoint** — the target HTTP URL to which export is written

The properties are described in detail in Table 15.3, “HTTP Export Properties”. This section explains the relationship between the properties.

There are essentially two types of HTTP export: batch mode and one record at a time. Batch mode is appropriate for exporting large volumes of data to targets such as Hadoop. Exporting one record at a time is less efficient for large volumes but can be very useful for writing intermittent messages to other services.

In batch mode, the data is exported using a POST or PUT method, where multiple records are combined in either comma-separated value (CSV) or Avro format in the body of the request. When writing one record at a time, you can choose whether to submit the HTTP request as a POST, PUT or GET (that is, as a querystring attached to the URL). When exporting in batch mode, the method must be either `POST` or `PUT` and the type must be either `csv` or `avro`. When exporting one record at a time, you can use the `GET`, `POST`, or `PUT` method, but the output type must be `form`.

Finally, the endpoint property specifies the target URL where data is being sent, using either the http: or https: protocol. Again, the endpoint must be compatible with the possible settings for the other properties. In particular, if the endpoint is a WebHDFS URL, batch mode must enabled.

The URL can also contain placeholders that are filled in at runtime with metadata associated with the export data. Each placeholder consists of a percent sign (%) and a single ASCII character. The following are the valid placeholders for the HTTP endpoint property:

Placeholder	Description
%t	The name of the VoltDB export source table or stream. The source name is inserted into the endpoint in all uppercase.
%p	The VoltDB partition ID for the partition where the INSERT query to the export source is executing. The partition ID is an integer value assigned by VoltDB internally and can be used to randomly partition data. For example, when exporting to webHDFS, the partition ID can be used to direct data to different HDFS files or directories.

Placeholder	Description
%g	The export generation. The generation is an identifier assigned by VoltDB. The generation increments each time the database starts or the database schema is modified in any way.
%d	The date and hour of the current export period. Applicable to WebHDFS export only. This placeholder identifies the start of each period and the replacement value remains the same until the period ends, at which point the date and hour is reset for the new period. You can use this placeholder to "roll over" WebHDFS export destination files on a regular basis, as defined by the `period` property. The `period` property defaults to one hour.

When exporting in batch mode, the endpoint must contain at least one instance each of the %t, %p, and %g placeholders. However, beyond that requirement, it can contain as many placeholders as desired and in any order. When not in batch mode, use of the placeholders are optional.

Table 15.3, “HTTP Export Properties” describes the supported properties for the HTTP connector.

Table 15.3. HTTP Export Properties

Property	Allowable Values	Description
endpoint*	string	Specifies the target URL. The endpoint can contain placeholders for inserting the source name (%t), the partition ID (%p), the date and hour (%d), and the export generation (%g).
avro.compress	true, false	Specifies whether Avro output is compressed or not. The default is false and this property is ignored if the type is not Avro.
avro.schema.location	string	Specifies the location where the Avro schema will be written. The schema location can be either an absolute path name on the local database server or a webHDFS URL and must include at least one instance of the placeholder for the source name (%t). Optionally it can contain other instances of both %t and %g. The default location for the Avro schema is the file path `export/avro/%t_avro_schema.json` on the database server under the voltdbroot directory. This property is ignored if the type is not Avro.
batch.mode	true, false	Specifies whether to send multiple rows as a single request or send each export row separately. The default is true. Batch mode must be enabled for WebHDFS export.
httpfs.enable	true, false	Specifies that the target of WebHDFS export is an Apache HttpFS (Hadoop HDFS over HTTP) server. This property must be set to true when exporting webHDFS to HttpFS targets.
kerberos.enable	true, false	Specifies whether Kerberos authentication is used when connecting to a WebHDFS endpoint. This property is only valid when connecting to WebHDFS servers and is false by default.
method	get, post, put	Specifies the HTTP method for transmitting the export data. The default method is POST. For WebHDFS export, this property is ignored.
period	Integer	Specifies the frequency, in hours, for "rolling" the WebHDFS output date and time. The default frequency is every hour (1). For WebHDFS export only.

Property	Allowable Values	Description
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java TimeZone identifier. For example, you can specify a continent and region ("America/New_York") or a time zone acronym ("EST"). The default is the local time zone.
type	csv, avro, form	Specifies the output format. If batch.mode is true, the default type is CSV. If batch.mode is false, the default and only allowable value for type is form. Avro format is supported for WebHDFS export only (see Section 15.3.3.2, “Exporting to Hadoop via WebHDFS” for details.)

*Required

15.3.3.2. Exporting to Hadoop via WebHDFS

As mentioned earlier, the HTTP connector contains special optimizations to support exporting data to Hadoop via the WebHDFS protocol. If the endpoint property contains a WebHDFS URL (identified by the URL path component starting with the string "/webhdfs/v1/"), special rules apply.

First, for WebHDFS URLs, the batch.mode property must be enabled. Also, the endpoint must have at least one instance each of the source name (%t), the partition ID (%p), and the export generation (%g) placeholders and those placeholders must be part of the URL path, not the domain or querystring.

Next, the method property is ignored. For WebHDFS, the HTTP connector uses a combination of POST, PUT, and GET requests to perform the necessary operations using the WebHDFS REST API.

For example, The following configuration file exports stream data to WebHDFS using the HTTP connector and writing each stream to a separate directory, with separate files based on the partition ID, generation, and period timestamp, rolling over every 2 hours:

```
deployment:
  export:
    configurations:
    - target: hadoop
      enabled: true
      type: http
      properties:
      - name: endpoint
        value: "http://myhadoopsvr/webhdfs/v1/%t/data%p-%g.%d.csv"
      - name: batch.mode
        value: true
      - name: period
        value: 2
```

Note that the HTTP connector will create any directories or files in the WebHDFS endpoint path that do not currently exist and then append the data to those files, using the POST or PUT method as appropriate for the WebHDFS REST API.

You also have a choice between two formats for the export data when using WebHDFS: comma-separated values (CSV) and Apache Avro™ format. By default, data is written as CSV data with each record on a separate line and batches of records attached as the contents of the HTTP request. However, you can choose to set the output format to Avro by setting the `type` property, as in the following example:

```
deployment:
  export:
    configurations:
    - target: hadoop
      enabled: true
      type: http
      properties:
      - name: endpoint
        value: "http://myhadoopsvr/webhdfs/v1/%t/data%p-%g.%d.avro"
      - name: type
        value: avro
      - name:avro.compress
        value: true
      - name: avro.schema.location
        value: "http://myhadoopsvr/webhdfs/v1/%t/schema.json"
```

Avro is a data serialization system that includes a binary format that is used natively by Hadoop utilities such as Pig and Hive. Because it is a binary format, Avro data takes up less network bandwidth than text-based formats such as CSV. In addition, you can choose to compress the data even further by setting the `avro.compress` property to true, as in the previous example.

When you select Avro as the output format, VoltDB writes out an accompanying schema definition as a JSON document. For compatibility purposes, the source name and columns names are converted, removing underscores and changing the resulting words to lowercase with initial capital letters (sometimes called "camelcase"). The source name is given an initial capital letter, while columns names start with a lowercase letter. For example, the stream EMPLOYEE_DATA and its column named EMPLOYEE_iD would be converted to EmployeeData and employeeId in the Avro schema.

By default, the Avro schema is written to a local file on the VoltDB database server. However, you can specify an alternate location, including a webHDFS URL. So, for example, you can store the schema in the same HDFS repository as the data by setting the `avro.schema.location` property, as shown in the preceding example.

See the Apache Avro web site for more details on the Avro format.

15.3.3.3. Exporting to Hadoop Using Kerberos Security

If the WebHDFS service to which you are exporting data is configured to use Kerberos security, the VoltDB servers must be able to authenticate using Kerberos as well. To do this, you must perform the following two extra steps:

- Configure Kerberos security for the VoltDB cluster itself
- Enable Kerberos authentication in the export configuration

The first step is to configure the VoltDB servers to use Kerberos as described in Section 12.9, “Integrating Kerberos Security with VoltDB”. Because use of Kerberos authentication for VoltDB security changes how the clients connect to the database cluster, It is best to set up, enable, and test Kerberos authentication first to ensure your client applications work properly in this environment before trying to enable Kerberos export as well.

Once you have Kerberos authentication working for the VoltDB cluster, you can enable Kerberos authentication in the configuration of the WebHDFS export target as well. Enabling Kerberos authentication in the HTTP connector only requires one additional property, `kerberos.enable`, to be set. To use Kerberos authentication, set the property to "true". For example:

```
deployment:
  export:
    configurations:
    - target: hadoop
      enabled: true
      type: http
      properties:
      - name: endpoint
        value: "myhadoopsvr/webhdfs/v1/%t/data%p-%g.%d.csv"
      - name: type
        value: csv
      - name: kerberos.enable
        value: true
```

Note that Kerberos authentication is only supported for WebHDFS endpoints.

15.3.4. The JDBC Export Connector

The JDBC connector receives the serialized data from the export source and writes it, in batches, to another database through the standard JDBC (Java Database Connectivity) protocol.

By default, when the JDBC connector opens the connection to the remote database, it first attempts to create tables in the remote database to match the VoltDB export source by executing CREATE TABLE statements through JDBC. This is important to note because, it ensures there are suitable tables to receive the exported data. The tables are created using either the names from the VoltDB schema or (if you do not enable the ignoregenerations property) the name prefixed by the database generation ID.

If the target database has existing tables that match the VoltDB export sources in both name and structure (that is, the number, order, and datatype of the columns), be sure to enable the ignoregenerations property in the export configuration to ensure that VoltDB uses those tables as the export target.

It is also important to note that the JDBC connector exports data through JDBC in batches. That is, multiple INSERT instructions are passed to the target database at a time, in approximately two megabyte batches. There are two consequences of the batching of export data:

- For many databases, such as Netezza, where there is a cost for individual invocations, batching reduces the performance impact on the receiving database and avoids unnecessary latency in the export processing.
- On the other hand, no matter what the target database, if a query fails for any reason the entire batch fails.

To avoid errors causing batch inserts to fail, it is strongly recommended that the target database not use unique indexes on the receiving tables that might cause constraint violations.

If any errors *do* occur when the JDBC connector attempts to submit data to the remote database, the VoltDB disconnects and then retries the connection. This process is repeated until the connection succeeds. If the connection does not succeed, VoltDB eventually reduces the retry rate to approximately every eight seconds.

Table 15.4, “JDBC Export Properties” describes the supported properties for the JDBC connector.

Table 15.4. JDBC Export Properties

Property	Allowable Values	Description
jdbcurl*	connection string	The JDBC connection string, also known as the URL.
jdbcuser*	string	The username for accessing the target database.

Property	Allowable Values	Description
jdbcpassword	string	The password for accessing the target database.
jdbcdriver	string	The class name of the JDBC driver. The JDBC driver class must be accessible to the VoltDB process for the JDBC export process to work. Place the driver JAR files in the `lib/extension/` directory where VoltDB is installed to ensure they are accessible at runtime. You do not need to specify the driver as a property value for several popular databases, including MySQL, Netezza, Oracle, PostgreSQL, and Vertica. However, you still must provide the driver JAR file.
schema	string	The schema name for the target database. The use of the schema name is database specific. In some cases you must specify the database name as the schema. In other cases, the schema name is not needed and the connection string contains all the information necessary. See the documentation for the JDBC driver you are using for more information.
minpoolsize	integer	The minimum number of connections in the pool of connections to the target database. The default value is 10.
maxpoolsize	integer	The maximum number of connections in the pool. The default value is 100.
maxidletime	integer	The number of milliseconds a connection can be idle before it is removed from the pool. The default value is 60000 (one minute).
maxstatementcached	integer	The maximum number of statements cached by the connection pool. The default value is 50.
createtable	true, false	Specifies whether VoltDB should create the corresponding table in the remote database. By default , VoltDB creates the table(s) to receive the exported data. (That is, the default is true.) If you set this property to false, you must create table(s) with matching names to the VoltDB export sources before starting the export connector.
lowercase	true, false	Specifies whether VoltDB uses lowercase table and column names or not. By default, VoltDB issues SQL statements using uppercase names. However, some databases (such as PostgreSQL) are case sensitive. When this property is set to true, VoltDB uses all lowercase names rather than uppercase. The default is false.
ignoregenerations	true, false	Specifies whether a unique ID for the generation of the database is included as part of the output table name(s). The generation ID changes each time a database restarts or the database schema is updated. The default is false.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skipinternals as true, the output contains only the exported stream data. The default is false.

*Required

15.3.5. The Kafka Export Connector

The Kafka connector receives serialized data from the export sources and writes it to a message queue using the Apache Kafka version 0.10.2 protocols. Apache Kafka is a distributed messaging service that lets you

set up message queues which are written to and read from by "producers" and "consumers", respectively. In the Apache Kafka model, VoltDB export acts as a "producer" capable of writing to any Kafka service using version 0.10.2 or later.

Before using the Kafka connector, we strongly recommend reading the Kafka documentation and becoming familiar with the software, since you will need to set up a Kafka service and appropriate "consumer" clients to make use of VoltDB's Kafka export functionality. The instructions in this section assume a working knowledge of Kafka and the Kafka operational model.

When the Kafka connector receives data from the VoltDB export sources, it establishes a connection to the Kafka messaging service as a Kafka producer. It then writes records to Kafka topics based on the VoltDB stream or table name and certain export connector properties.

The majority of the Kafka export properties are identical in both in name and content to the Kafka producer properties listed in the Kafka documentation. All but one of these properties are optional for the Kafka connector and will use the standard Kafka default value. For example, if you do not specify the `queue.buffering.max.ms` property it defaults to 5000 milliseconds.

The only required property is `bootstrap.servers`, which lists the Kafka servers that the VoltDB export connector should connect to. You must include this property so VoltDB knows where to send the export data. Specify each server by its IP address (or hostname) and port; for example, myserver:7777. If there are multiple servers in the list, separate them with commas.

In addition to the standard Kafka producer properties, there are several custom properties specific to VoltDB. The properties `binaryencoding`, `skipinternals`, and `timezone` affect the format of the data. The `topic.prefix` and `topic.key` properties affect how the data is written to Kafka.

The `topic.prefix` property specifies the text that precedes the stream or table name when constructing the Kafka topic. If you do not specify a prefix, it defaults to "voltdbexport". Alternately, you can map individual sources to topics using the `topic.key` property. In the `topic.key` property you associate a VoltDB export source name with the corresponding Kafka topic as a named pair separated by a period (.). Multiple named pairs are separated by commas (,). For example:

```
Employee.EmpTopic,Company.CoTopic,Enterprise.EntTopic
```

Any mappings in the `topic.key` property override the automated topic name specified by `topic.prefix`.

Note that unless you configure the Kafka brokers with the `auto.create.topics.enable` property set to true, you must create the topics for every export source manually before starting the export process. Enabling auto-creation of topics when setting up the Kafka brokers is recommended.

When configuring the Kafka export connector, it is important to understand the relationship between synchronous versus asynchronous processing and its effect on database latency. If the export data is sent asynchronously, the impact of export on the database is reduced, since the export connector does not wait for the Kafka infrastructure to respond. However, with asynchronous processing, VoltDB is not able to resend the data if the message fails after it is sent.

If export to Kafka is done synchronously, the export connector waits for acknowledgement of each message sent to Kafka before processing the next packet. This allows the connector to resend any packets that fail. The drawback to synchronous processing is that on a heavily loaded database, the latency it introduces means export may not be able to keep up with the influx of export data and and have to write to overflow.

You specify the level of synchronicity and durability of the connection using the Kafka `acks` property. Set `acks` to "0" for asynchronous processing, "1" for synchronous delivery to the Kafka broker, or "all" to ensure durability on the Kafka broker. See the Kafka documentation for more information.

VoltDB guarantees that at least one copy of all export data is sent by the export connector. But when operating in asynchronous mode, the Kafka connector cannot guarantee that the packet is actually received and accepted by the Kafka broker. By operating in synchronous mode, VoltDB can catch errors returned by the Kafka broker and resend any failed packets. However, you pay the penalty of additional latency and possible export overflow.

Finally, the actual export data is sent to Kafka as a comma-separated values (CSV) formatted string. The message includes six columns of metadata (such as the transaction ID and timestamp) followed by the column values of the export stream.

Table 15.5, "Kafka Export Properties" lists the supported properties for the Kafka connector, including the standard Kafka producer properties and the VoltDB unique properties.

Table 15.5. Kafka Export Properties

Property	Allowable Values	Description
bootstrap.servers*	string	A comma-separated list of Kafka brokers (specified as IP-address:port-number). You can use `metadata.broker.list` as a synonym for `bootstrap.servers`.
acks	0, 1, all	Specifies whether export is synchronous (*1* or *all*) or asynchronous (*0*) and to what extent it ensures delivery. The default is *all*, which is recommended to avoid possibly losing messages when a Kafka server becomes unavailable during export. See the Kafka documentation of the producer properties for details.
acks.retry.timeout	integer	Specifies how long, in milliseconds, the connector will wait for acknowledgment from Kafka for each packet. The retry timeout only applies if acknowledgements are enabled. That is, if the `acks` property is set greater than zero. The default timeout is 5,000 milliseconds. When the timeout is reached, the connector will resend the packet of messages.
binaryencoding	hex, base64	Specifies whether VARBINARY data is encoded in hexadecimal or BASE64 format. The default is hexadecimal.
skipinternals	true, false	Specifies whether to include six columns of VoltDB metadata (such as transaction ID and timestamp) in the output. If you specify skipinternals as true, the output contains only the exported stream data. The default is false.
timezone	string	The time zone to use when formatting the timestamp. Specify the time zone as a Java TimeZone identifier. For example, you can specify a continent and region ("America/New_York") or a time zone acronym ("EST"). The default is GMT.
topic.key	string	A set of named pairs each identifying a VoltDB source name and the corresponding Kafka topic name to which the data is written. Separate the names with a period (.) and the name pairs with a comma (,).

Property	Allowable Values	Description
		The specific source/topic mappings declared by topic.key override the automated topic names specified by topic.prefix.
topic.prefix	string	The prefix to use when constructing the topic name. Each row is sent to a topic identified by {prefix}{source-name}. The default prefix is "voltdbexport".
Kafka producer properties	various	You can specify standard Kafka producer properties as export connector properties and their values will be passed to the Kafka interface. However, you cannot modify the property `block.on.buffer.full`.

*Required

15.3.6. The Elasticsearch Export Connector

The Elasticsearch connector receives serialized data from the export source and inserts it into an Elasticsearch server or cluster. Elasticsearch is an open-source full-text search engine built on top of Apache Lucene™. By exporting selected tables and streams from your VoltDB database to Elasticsearch you can perform extensive full-text searches on the data not possible with VoltDB alone.

Before using the Elasticsearch connector, we recommend reading the Elasticsearch documentation and becoming familiar with the software. The instructions in this section assume a working knowledge of Elasticsearch, its configuration and its capabilities.

The only required property when configuring Elasticsearch is the endpoint, which identifies the location of the Elasticsearch server and what index to use when inserting records into the target system. The structure of the Elasticsearch endpoint is the following:

```
<protocol>://<server>:<port>//<index-name>//<type-name>
```

For example, if the target Elasticsearch service is on the server `esearch.lan` using the default port (9200) and the exported records are being inserted into the `employees` index as documents of type `person`, the endpoint declaration would look like this:

```
properties:
- name: endpoint
  value: http://esearch.lan:9200/employees/person
```

You can use placeholders in the endpoint that are replaced at runtime with information from the export data, such as the source name (%t), the partition ID (%p), the export generation (%g), and the date and hour (%d). For example, to use the source name as the index name, the endpoint might look like the following:

```
properties:
- name: endpoint
  value: "http://esearch.lan:9200/%t/person"
```

When you export to Elasticsearch, the export connector creates the necessary index names and types in Elasticsearch (if they do not already exist) and inserts each record as a separate document with the appropriate metadata. Table 15.6, “Elasticsearch Export Properties” lists the supported properties for the Elasticsearch connector.

Table 15.6. Elasticsearch Export Properties

Property	Allowable Values	Description
endpoint*	string	Specifies the root URL of the RESTful interface for the Elasticsearch cluster to which you want to export the data. The endpoint should include the protocol, host name or IP address, port, and path. The path is assumed to include an index name and a type identifier. The export connector will use the Elasticsearch RESTful API to communicate with the server and insert records into the specified locations. You can use placeholders to replace portions of the endpoint with data from the exported records at runtime, including the source name (%t), the partition ID (%p), the date and hour (%d), and the export generation (%g).
batch.mode	true, false	Specifies whether to send multiple rows as a single request or send each export row separately. The default is true.
timezone	string	The time zone to use when formatting timestamps. Specify the time zone as a Java TimeZone identifier. For example, you can specify a continent and region ("America/New_York") or a time zone acronym ("EST"). The default is the local time zone.

*Required

15.4. VoltDB Import Connectors

Just as VoltDB can export data from selected streams and tables to external targets, it supports importing data into selected tables from external sources. Import works in two ways:

- Bulk loading data using one of several standalone utilities VoltDB provides. These data loaders support multiple standard input protocols and can be run from any server, even remotely from the database itself.
- Streaming import as part of the database server process. For data that is imported on an ongoing basis, use of the built-in import functionality ensures that import starts and stops with the database.

The following sections discuss these two approaches to data import.

15.4.1. Bulk Loading Data Using VoltDB Standalone Utilities

Often, when migrating data from one database to another or when pre-loading a set of data into VoltDB as a starting point, you just want to perform the import once and then use the data natively within VoltDB. For these one-time uses, or when you prefer to manage the import process externally, VoltDB provides separate data loader utilities.

Each data loader supports a different source format. You can load data from text files — such as comma-separated value (CSV) files — using the csvloader utility. You can load data from another JDBC-compliant database using the jdbcloader utility. Or you can load data from a streaming message service with the Kafka loader utility, kafkaloader.

All of the data loaders operate in much the same way. For each utility you specify the source for the import and either a table that the data will be loaded into or a stored procedure that will be used to load the data. So, for example, to load records from a CSV file named staff.csv into the table EMPLOYEES, the command might be the following:

```
$ csvloader employees --file=staff.csv
```

If instead you are copying the data from a JDBC-compliant database, the command might look like this:

```
$ jdbcloader employees \
     --jdbcurl=jdbc:postgresql://remotesvr/corphr \
     --jdbctable=employees \
     --jdbcdriver=org.postgresql.Driver
```

Each utility has arguments unique to the data source (such as `--jdbcurl`) that allow you to properly configure and connect to the source. See the description of each utility in Appendix D, *VoltDB CLI Commands* for details.

15.4.2. Streaming Import Using Built-in Import Features

If importing data is an ongoing business process, rather than a one-time event, then it is desirable to make it an integral part of the database system. This can be done by building a custom application to push data into VoltDB using one of its standard APIs, such as the JDBC interface. Or you can take advantage of VoltDB's built-in import infrastructure.

The built-in importers work in much the same way as the data loading utilities, where incoming data is written into one or more database tables using an existing stored procedure. The difference is that the built-in importers start automatically whenever the database starts and stop when the database stops, making import an integral part of the database process.

You set up the built-in importers in the configuration the same way you configure export connections. Within the `import` property, you declare each import stream using separate `configurations` list elements. You use subproperties to specify the `type` and `format` of data being imported and whether the import configuration is enabled or not. Provide information required by the specific importer and/or formatter within the `properties` list. For example:

```
deployment:
  import:
    configurations:
      type: kafka
      format: csv
      enabled: true
      properties:
      - name: brokers
        value: kafkasvr:9092
      - name: topics
        value: employees
      - name: procedure
        value: EMPLOYEE.insert
```

When the database starts, the import infrastructure starts any enabled configurations. If you are importing multiple streams to separate tables through separate procedures, you must include multiple configurations, even if they come from the same source. For example, the following configuration imports data from two Kafka topics from the same Kafka servers into separate VoltDB tables.

```
deployment:
  import:
    configurations:
    - nickname: employees
      type: kafka
      format: csv
      enabled: true
      properties:
      - name: brokers
        value: kafkasvr:9092
      - name: topics
        value: employees
      - name: procedure
        value: EMPLOYEE.insert
    - nickname: managers
      type: kafka
      format: csv
      enabled: true
      properties:
      - name: brokers
        value: kafkasvr:9092
      - name: topics
        value: managers
      - name: procedure
        value: MANAGER.insert
```

Note the addition of the `nickname` property to the configuration. Unlike export where the `target` property provides a unique identifier, there are no unique subproperties in the import configuration. The `nickname` property lets you assign a unique name to each import connector. This way you can modify the settings for individual connectors while the database is running using the **voltadmin set** command. For example, the following command disables the *managers* import connector:

```
$ voltadmin set deployment.import.configurations[managers].enabled=false
```

Without the nickname, you would have to update the entire database configuration to modify just one import connector.

VoltDB currently provides support for a Kafka importer. It also supports two import formats: comma-separated values (csv) and tab-separated values (tsv). Comma-separated values are the default format. So if you are using CSV-formatted input, you can leave out the format attribute, as in the preceding example. The following sections describe the Kafka importer and the CSV/TSV formatter in more detail.

15.4.3. The Kafka Importer

The Kafka importer connects to the specified Kafka messaging service and imports one or more Kafka topics and writes the records into the VoltDB database. The data is decoded according to the specified format — comma-separated values (CSV) by default — and is inserted into the VoltDB database using the specified stored procedure.

The Kafka importer supports Kafka version 0.10 or later. You must specify at least the following properties for each configuration:

- **brokers** — Identifies one or more Kafka brokers. That is, servers hosting the Kafka service and desired topics. Specify a single server or a comma-separated list of brokers.

- **topics** — Identifies the Kafka topics that will be imported. The property value can be a single topic name or a comma-separated list of topics.
- **procedure** — Identifies the stored procedure that is invoked to insert the records into the VoltDB database.

When import starts, the importer first checks to make sure the specified stored procedure exists in the database schema. If not (for example, when you first create a database and before a schema is loaded), the importer issues periodic warnings to the console.

Once the specified stored procedure is declared, the importer looks for the specified Kafka brokers and topics. If the specified brokers cannot be found or the specified topics do not exist on the brokers, the importer reports an error and stops. You will need to restart import once this error condition is corrected. You can restart import using any of the following methods:

- Stop and restart the database
- Pause and resume the database using the **voltadmin pause** and **voltadmin resume** commands
- Update the configuration using the **voltadmin update** command or the web-based Volt Management Center

If the brokers are found and the topics exist, the importer starts fetching data from the Kafka topics and submitting it to the stored procedure to insert into the database. In the simplest case, you can use the default insert procedure for a table to insert records into a single table. For more complex data you can write your own import stored procedure to interpret the data and insert it into the appropriate table(s).

Table 15.7, “Kafka Import Properties” lists the allowable properties for the Kafka importer. You can also specify properties associated with the formatter, as described in Table 15.8, “CSV and TSV Formatter Properties”.

Table 15.7. Kafka Import Properties

Property	Allowable Values	Description
brokers*	string	A comma-separated list of Kafka brokers.
procedure*	string	The stored procedure to invoke to insert the incoming data into the database.
topics*	string	A comma-separated list of Kafka topics.
commit.policy	integer	Because the importer performs two distinct tasks — retrieving records from Kafka and then inserting them into VoltDB — Kafka's automated tracking of the current offset may not match what records are successfully inserted into the database. Therefore, by default, the importer uses a manual commit policy to ensure the Kafka offset matches the completed inserts. Use of the default commit policy is recommended. However, you can, if you choose, use Kafka's automated commit policy by specifying a commit interval, in milliseconds, using this property.
groupid	string	A user-defined name for the group that the client belongs to. Kafka maintains a single pointer for the current position within the stream for all clients in the same group.

Property	Allowable Values	Description
		The default group ID is "voltdb". In the rare case where you have two or more databases importing data from the same Kafka brokers and topics, be sure to set this property to give each database a unique group ID and avoid the databases interfering with each other.
fetch.max.bytes heartbeat.interval.ms max.partition.fetch.bytes max.poll.interval.ms max.poll.records request.timeout.ms session.timeout.ms	string	These Kafka consumer properties are supported as import properties. See the Kafka 0.11 documentation for details.

*Required

15.5. VoltDB Import Formatters

The import infrastructure uses formatters to interpret the incoming data and convert it for insertion into the database. If you use the CSV or TSV formatter, you can control how the data is interpreted by setting additional properties associated with those formatters. For example, the following configuration for the Kafka importer includes the formatter property `blank` specifying that blank entries should generate an error, rather than being interpreted as null or empty values:

```
deployment:
  import:
    configurations:
    - nickname: employees
      type: kafka
      format: csv
      enabled: true
      properties:
      - name: brokers
        value: kafkasvr:9092
      - name: topics
        value: employees
      - name: procedure
        value: EMPLOYEE.insert
      - name: blank
        value: error
```

You include the formatter properties in the import configuration along with the other import connector properties. Table 15.8, “CSV and TSV Formatter Properties” lists the allowable properties for the CSV and TSV import formatters.

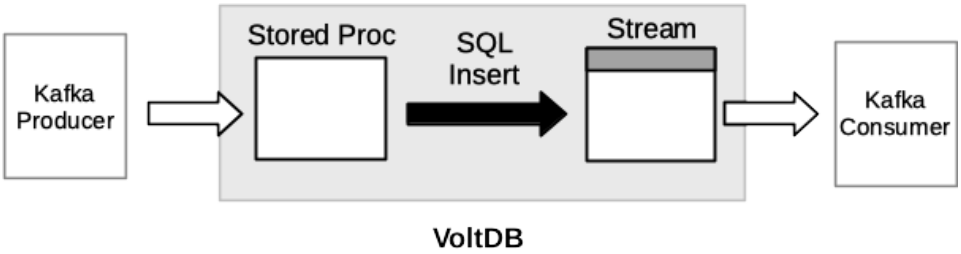
Table 15.8. CSV and TSV Formatter Properties

Property	Allowable Values	Description
blank	empty, error, null	Specifies what to do with missing values in the input. If you specify `empty`, missing entries result in the corresponding "empty" value (that is, zero for INTEGER,

Property	Allowable Values	Description
		a zero-length string for VARCHAR, and so on); if you specify `error`, missing entries generate an error, if you specify `null`, missing entries result in a null value. The default interpretation of missing values is `null`.
nowhitespace	true, false	Specifies whether the input can contain whitespace between data values and separators. If you specify `true`, any input lines containing whitespace will generate an error and not be inserted into the database. The default is `false`.
nullstring	string	Specifies a custom string to be interpreted as a null value. By default, the following entries are interpreted as null: • An empty entry • `NULL` (unquoted, uppercase) • `\N` (quoted or unquoted, either upper or lowercase) If you specify a custom null string, it overrides all default null strings.
trimrawtext	true, false	Specifies whether any white space around unquoted string values is included in the string input or not. If you specify `true`, surrounding white space is dropped; if you specify `false`, surrounding white space between the string value and the separators is included in the input value. The default is `true`.

15.6. VoltDB Topics

Topics use the Apache Kafka protocols when interpreting input from Kafka producers and when generating output for Kafka consumers. The configuration declares the topic and specifies the stored procedure that receives the inbound data. You can use a database table or stream — or even a stream view — as the source for topic output. The CREATE... EXPORT TO TOPIC statement identifies the stream or table that is used to queue outbound data to the specified topic. VoltDB topics operate just like Kafka topics, with the database nodes acting as Kafka brokers. However, unlike Kafka, VoltDB topics also have the ability to analyze, act on, or even modify the data as it passes through.



As the preceding diagram shows, data submitted to the topic from a Kafka producer (either using the Kafka API or using a tool such as Kafka Connect) is passed to the stored procedure, which then interprets and operates on the data before passing it along to the stream or table through standard SQL INSERT semantics. Note that the named procedure must exist before input is accepted. Similarly, the stream or table must be declared using the EXPORT TO TOPIC clause and the topic be defined in the configuration before any

output is queued. So, it is a combination of the database schema *and* configuration that establishes the complete topic workflow.

For example, the following SQL statements declare the necessary stored procedure and stream and the configuration defines a topic *eventLogs* that integrates them:

Schema DDL

```
CREATE STREAM eventlog
  PARTITION ON COLUMN e_type
  EXPORT TO TOPIC eventlogs
  ( e_type INTEGER NOT NULL,
    e_time TIMESTAMP NOT NULL,
    e_msg  VARCHAR(256)
  );

CREATE PROCEDURE
  PARTITION ON TABLE eventlog COLUMN e_type
    FROM CLASS mycompany.myprocs.checkEvent;
```

Configuration

```
deployment:
 topics:
   topic:
   - name: eventlogs
     procedure: checkEvent
```

Concerning Case Sensitivity

The names of Kafka topics are case sensitive. That means that the name of the topic matches exactly how it is specified in the configuration. So in the previous example, the topic name *eventLogs* is all lowercase except for the letter "L". This is how the producers and consumers must specify the topic name. But SQL names — such as table and column names — are case insensitive. As a result, the topic name specified in the EXPORT TO TOPIC clause does not have to match exactly. In other words, the topic "eventLogs" matches any stream or table that specifies the topic name with the same spelling, regardless of case.

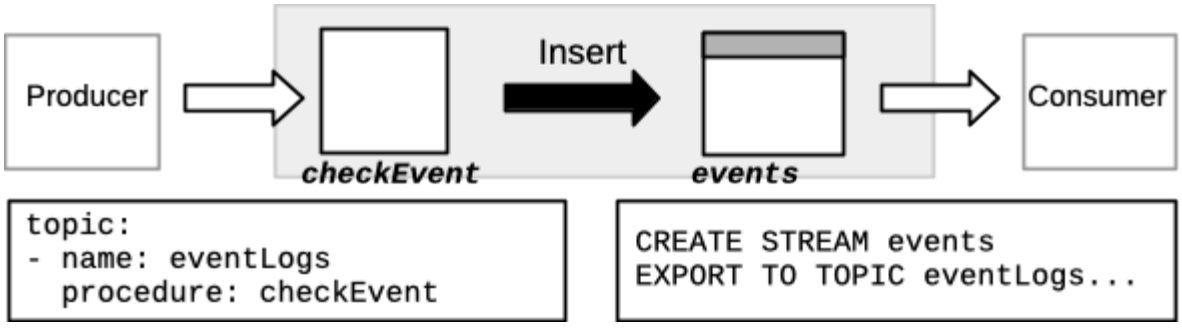
The structure of a topic message — that is, the fields included in the message and the message key — is defined in the schema using the EXPORT TO TOPIC... WITH clause. Other characteristics of how the message is handled, such as the data format, security, and retention policy, are controlled by `property` list in the configuration. The following sections discuss:

- Understanding the different types of topics
- Declaring VoltDB topics
- Configuring and managing topics
- Configuring the topic server
- Calling topics from external consumers and producers
- Using opaque topics

15.6.1. Types of VoltDB Topics

VoltDB supports four different types of topics, depending on how the topic is declared:

- A **fully processed topic** is a pipeline that supports both input and output and passes through a stored procedure. This is defined using both the `procedure` property in the configuration and the EXPORT TO TOPIC clause in the CREATE statement.



- An **input-only topic** only provides for input from Kafka producers. You define an input-only topic by specifying the `procedure` property, without any stream or table including a corresponding EXPORT TO TOPIC clause.

- An **output-only topic** only provides for output to Kafka consumers but can be written to by VoltDB INSERT statements. You define an output-only topic by including the EXPORT TO TOPIC clause, but not specifying a procedure in the topic declaration.

- An **opaque topic** supports input and output but provides for no processing or interpretation. You define an opaque topic setting the `opaque` property to true in the configuration, as described in Section 15.6.6, “Using Opaque Topics”.

15.6.2. Declaring VoltDB Topics

You declare and configure topics by combining SQL stored procedures and streams or tables with topic declarations in the database configuration. The topic itself is defined in the configuration, using the `topics` and `topic` properties. The configuration also lets you identify the stored procedure used for input from producers:

```
deployment:
  topics:
    topic:
    - name: eventLogs
      procedure: eventWatch
```

15.6.2.1. Processing Topic Output

For output, you include the EXPORT TO TOPIC clause when you declare a stream or table. Once the schema object includes the EXPORT TO TOPIC clause and the topic is defined in the configuration, any records written into the stream or table are made available to consumers through the topic port.

You can control what parts of the records are sent to the topic, using the WITH KEY/VALUE clauses. The WITH VALUE clause specifies which columns of the object are included in the body of the topic message and their order. The WITH KEY clause lets you specify one or more columns as a key for the message. Columns can appear in either the message body or the key, in both, or in neither, as needed. In all cases, the lists of columns are enclosed in parentheses and separated by commas.

So, for example, the following stream declaration associates the stream *events* with the topic *eventLogs* and selects two columns for the body of the topic message and one column as the key:

```
CREATE STREAM events
  PARTITION ON COLUMN event_type
  EXPORT TO TOPIC eventLogs
    WITH KEY (event_type) VALUE (when,what)
  ( event_type INTEGER NOT NULL,
    when TIMESTAMP NOT NULL,
    what  VARCHAR(256)
  );
```

15.6.2.2. Processing Topic Input

Since VoltDB does not control what content producers send to the topic, it cannot dictate what columns or datatypes the stored procedure will receive. Instead, VoltDB interprets the content from its format. By default, text data is interpreted as comma-separated values. All other data is interpreted as a single value based on the data itself. On the other hand, if the topic is configured as using either JSON or AVRO formatted data in the configuration, the incoming data from producers will be interpreted in the specified format.

Any errors during the decoding of the input fields is recorded in the log file. If the input can be decoded, the message fields are used, in order, as arguments to the store procedure call.

Only one key field is allowed for input. By default, the key is not passed to the specified stored procedure; only the message fields of the topic are passed as parameters to the stored procedure. If you want to include the key in the list of parameters to the stored procedure, you can set the property `producer.parameters.includeKey` to true and the key will be included as the partitioning parameter for the procedure. For example:

```
deployment:
  topics:
    topic:
    - name: eventLogs
      procedure: eventWatch
      properties:
      - name: producer.parameters.includeKey
        value: true
```

15.6.3. Configuring and Managing Topics

Declaring the topic and its schema object and/or procedure are the only required elements for creating a topic. However, there are several other attributes you can specify either as part of the declaration or as clauses to the stored procedure and stream or table declarations. Those attributes include:

- Permissions — Controlling access to the topic by consumers and producers
- Retention — Managing how long data is retained in the topic queue before being deleted
- Data Format — Choosing a format for the data passed to the external clients

15.6.3.1. Permissions

When security is enabled for the database, the external clients must authenticate using a username and password when they initiate contact with the server. Access to the topic is handled separately for consumers and producers.

For producers, access to the topic is controlled by the security permissions of the associated stored procedure, as defined by the CREATE PROCEDURE... ALLOW clause or the generic permissions of the user account's role. (For example, a role with the ALLPROC or ADMIN permissions can write to any topic.)

For consumers, access to the topic is restricted by the `allow` property of the topic declaration in the configuration. If `allow` is not specified, any authenticated user can read from the topic. If `allow` is included in the declaration, only users with the specified role(s) have access. You specify permissions by providing a comma-separated list of roles that can read from the topic. For example, the following declaration allows users with the *kreader* and *operator* roles to read from the topic eventLogs:

```
deployment:
  topics:
    topic:
    - name: eventLogs
      allow: kreader,operator
```

15.6.3.2. Retention

Unlike export or import, where there is a single destination or source, topics can have multiple consumers and producers. So there is no specific event when the data transfer is complete and can be discarded. Instead, you must set a *retention policy* that defines when data is aged out of the topic queues. You specify the retention policy in terms of either the length of time the data is in queue or the volume of data in the queue.

For example, if you specify a retention policy of five days, after a record has been in the queue for five days, it will be deleted. If, instead, you set a retention policy of five gigabytes, as soon as the volume of

data in the queue exceeds 5GB, data will deleted until the queue size is under the specified limit. In both cases, data aging is a first in, first out process.

You specify the retention policy in the `retention` property of the topic declaration. The retention value is a positive integer and a unit, such as "gb" for gigabytes or "dy" for days. The following is the list of valid retention units:

Time mn — Minutes
hr — Hours
dy — Days
wk — Weeks
mo — Months
yr — Years
Size mb — Megabytes
gb — Gigabytes

If you do not specify a retention value, the default policy is seven days (7 dy).

15.6.3.3. Data Format

VoltDB topics are composed of three elements: a timestamp, a record with one or more fields, and an optional set of keys values. The timestamp is generated automatically when the record is inserted into the stream or table. The format of the record and the key depends on the data itself. Or you can specify a format for the record, for the key, or for both using properties of the topic declaration in the configuration.

For single value records and keys, the data is sent in the native Kafka binary format for that datatype. For multi-value records or keys, VoltDB defaults to sending the content as comma-separated values (CSV) in a text string. Similarly, on input from producers, the topic record is interpreted as a single binary format value or a CSV string, depending on the datatype of the content.

You can control what format is used to send and receive the topic data using either the `format` property of the topic definition, or separate `property` list elements to select the format of individual components. For example, to specify the format for the message and the keys for both input and output, you can use the `format` property:

```
deployment:
  topics:
    topic:
    - name: eventLogs
      format: avro
```

To specify individual formats for input versus output, or message versus keys, you can use separate `property` elements, where the property name is either *consumer* or *producer* followed by *`format`* and, optionally, the component type — all separated by periods. For example, the following declaration specifies Avro for both consumers and producers, and is equivalent to the preceding example using the `format` property:

```
deployment:
  topics:
    topic:
    - name: eventLogs
      properties:
      - name: consumer.format
        value: avro
      - name: producer.format
        value: avro
```

The following are the valid formatting properties:

- `consumer.format`
- `consumer.format.key`
- `consumer.format.value`
- `producer.format.value`

For input, note that you cannot specify the format of the key. This is because only a single key value is supported for producers and it is always assumed to be in native binary or string format.

Depending on what format you choose, you can also control specific aspects of how data is represented in that format. For example, you can specify special characters such as the separator, quote, and escape character in CSV format. Table 15.9, “Topic Formatting Properties” lists all of the supported formatting properties you can use when declaring topics in the configuration.

Table 15.9. Topic Formatting Properties

Property	Values	Description
consumer.format	avro, csv, json	Format of keys and values sent to consumers. Supersedes the format definition in the parent `topic` element. The default is CSV.
consumer.format.value	avro, csv, json	Format of values sent to consumers. Supersedes the format definition in the parent `topic` element and the "consumer.format" property. The default is CSV.
consumer.format.key	avro, csv, json	Format of keys sent to consumers. Supersedes the format definition in the parent `topic` element and the "consumer.format" property. The default is CSV.
producer.format.value	avro, csv, json	Format of values received from producers. Supersedes the format definition in the parent `topic` element. The default is CSV.
config.avro.timestamp	microseconds, milliseconds	Unit of measure for timestamps in AVRO formatted fields. The default is microseconds.
config.avro.geographyPoint	binary, fixed_binary, string	Datatype for GEOGRAPHY_POINT columns in AVRO formatted fields. The default is fixed_binary.
config.avro.geography	binary,string	Datatype for GEOGRAPHY columns in AVRO formatted fields. The default is binary.
config.csv.escape	character	Character used to escape the next character in a quoted string in CSV format. The default is the backslash "\".
config.csv.null	character(s)	Character(s) representing a null value in CSV format. The default is "\N".
config.csv.quote	character	Character used to enclose quoted strings in CSV format. The default is the double quotation character (").
config.csv.separator	character	Character separating values in CSV format. The default is the comma ",".
config.csv.quoteAll	true, false	Whether all string values are quoted or only strings with special characters (such as commas, line breaks, and quotation marks) in CSV format. The default is false.
config.csv.strictQuotes	true, false	Whether all string values are expected to be quoted on input. If true, any characters outside of quotation marks in text fields are ignored. The default is false.

Property	Values	Description
config.csv.ignore-LeadingWhitespace	true, false	Whether leading spaces are included in string values in CSV format. The default is true.
config.json.schema	embedded, none	Whether the JSON representation contains a property named "schema" embedded within it or not. If embedded, the schema property describes the layout of the object. The default is none.
config.json.consumer.attributes	string	Specifies the names of outgoing JSON elements. By default, JSON elements are named after the table columns they represent. This property lets you rename the columns on output.
config.json.producer.attributes	string	Specifies the name and order of the JSON elements that are inserted as parameters to the topic input procedure.
consumer.skip.internals	true, false	Whether the topic includes six columns of internal metadata or not, including information for change data capture. (See Section 15.2.2, "Change Data Capture" for more information.) The default is true.
producer.parameters.includeKey	true, false	Whether the topic key is included as the partitioning parameter to the stored procedure call. The default is false.
opaque.partitioned	true, false	Whether the opaque topic is partitioned. Ignored if not an opaque topic. The default is false
topic.store.encoded	true, false	Whether the topic is stored in the same format as issued by the producer: optimizes transcoding to consumers when producer and consumer formats are identical. The default is false.

When using AVRO format, you must also have access to an AVRO schema registry, which is where VoltDB stores the schema for AVRO-formatted topics. The URL for the registry is specified in the database configuration, as described in the next section.

15.6.4. Configuring the Topic Server

Communication between the VoltDB database and topic clients is handled by a separate server process: the topic server. The topic server process is started whenever VoltDB starts with the `topics` property declared and enabled in the configuration.

By default, the topic server, when running, listens on port 9092. You can specify a different port with the `--topicsport` flag when you start the server with the **voltdb start** command. Other aspects of the topic server operation are configured as `properties` list elements of the `broker` property. The following are the supported subproperties of the `broker` property:

- `cluster.id`
- `group.initial.rebalance.delay.ms`
- `group.load.max.size`
- `group.max.session.timeout.ms`
- `group.max.size`
- `group.min.session.timeout.ms`
- `log.cleaner.dedupe.buffer.size`
- `log.cleaner.delete.retention.ms`
- `log.cleaner.threads`
- `network.thread.count`
- `offsets.retention.check.interval.ms`
- `offsets.retention.minutes`

- quota.throttle.max_ms
- quota.request.bytes_per_second
- quota.request.processing_percent
- quota.response.bytes_per_second
- retention.policy.threads

For example, this declaration configures the broker using a cluster ID of 3 and five network threads:

```
deployment:
  broker:
    properties:
    - name: cluster.id
      value: 3
    - name: network.thread.count
      value: 5
```

Finally, you can additionally tune the performance of the topic server by adjusting the threads that manage the inbound and outbound connections. You can specify a threadpool for the topic server to use for processing client requests using the `threadpool` subproperty of `topics`, then specify a size for the pool in the `threadpools` property:

```
deployment:
  topics:
    threadpool: topics
  threadpools:
  - name: topics
    size: 10
```

15.6.5. Calling Topics from Consumers and Producers

Once the topic has been declared in the database configuration and the appropriate streams, tables, and stored procedures created in the schema, the topic is ready for use by external clients. Since VoltDB topics use the Kafka API protocol, any Kafka consumer or producer with the appropriate permissions can access the topics. For example, you can use the console consumer that comes with Kakfa to read topics from VoltDB:

```
$ bin/kafka-console-consumer.sh --from-beginning \
   --topic eventLogs --bootstrap-server myvoltdb:9092
```

You can even use the console producer. However, whether using an existing producer or creating your own, you must explicitly disable idempotency or else VoltDB will not accept data from your application. You disable idempotency by setting the Kafka property `enable.idempotence` to "false".

To optimize write operations, Kafka also needs to know the VoltDB partitioning scheme. So it is strongly recommended that you define the Kafka `ProducerConfig.PARTITIONER_CLASS_CONFIG` property to point to the VoltDB partitioner for Kafka. By defining the PARTITIONER_CLASS_CONFIG, VoltDB can ensure that the producer sends records to the appropriate cluster node for each partitioning key. For example, a Java-based client application should contain a producer definition similar to the following to disable idempotency and configure the partitioning scheme:

```
Properties props = new Properties();

props.put("bootstrap.servers", "myvoltdb:9092");
props.put(ProducerConfig.PARTITIONER_CLASS_CONFIG, VoltDBKafkaPartitioner.class.getName()
props.put("enable.idempotence","false");
```

```
props.put("client.id","myConsumer");

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");
props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Producer<String, String> producer = new KafkaProducer<>(props);
```

To access the VoltDB partitioner for Kafka, be sure to include the VoltDB client library JAR file in your classpath when compiling and running your producer client.

15.6.6. Using Opaque Topics

Opaque topics are a special type of topic that do not receive any interpretation or modification by the database. If you want to create a topic that is not processed but simply flows through VoltDB from producers to consumers, you declare the topic as "opaque" in the configuration, without either specifying a stored procedure for input or associating a stream or table with the topic for output.

```
deployment:
  topics:
    topic:
    - name: sysmsgs
      opaque: true
```

Opaque topics allow you to use a single set of brokers for all your topics even if you only need to analyze and process certain data feeds. Because there is no interpretation, you cannot specify a stored procedure, stream, table, or format for the topic. However, there are a few properties specific to opaque topics you can use to control how the data are handled.

One important control is whether the opaque topics are partitioned or not. Partitioning the opaque topics improves throughput by distributing processing across the cluster. However, you can only partition opaque topics that have a key. To partition an opaque topic you set the `opaque.partitioned` property to *true*:

```
deployment:
  topics:
    topic:
    - name: sysmsgs
      opaque: true
      - properties:
        - name: opaque.partitioned
          value: true
```

You can specify a retention policy for opaque topics, just like regular topics. In fact, opaque topics have one additional retention option. Since the content is not analyzed in any way, it can be compressed to save space while it is stored. By specifying the retention policy as "compact" with a time limit, the records are stored compressed until the time limit expires. For example, the following configuration compresses the opaque topic data then deletes it after two months:

```
deployment:
  topics:
    topic:
    - name: sysmsgs
      opaque: true
      retention: compact 2 mo
      - properties:
        - name: opaque.partitioned
```

```
value: true
```

Appendix A. Supported SQL DDL Statements

This appendix describes the subset of the SQL Data Definition Language (DDL) that VoltDB supports when defining the schema for a VoltDB database. VoltDB also supports extensions to the standard syntax to allow for the declaration of stored procedures and partitioning information related to tables and procedures.

The following sections are not intended as a complete description of the standard SQL DDL. Instead, they summarize the subset of standard SQL DDL statements that are allowed when defining a VoltDB schema and any exceptions, extensions, or limitations that application developers should be aware of.

The supported standard SQL DDL statements are:

- ALTER TABLE
- CREATE INDEX
- CREATE TABLE
- CREATE VIEW
- DROP INDEX
- DROP TABLE
- DROP VIEW

The supported VoltDB-specific extensions for declaring functions, stored procedures, streams, and partitioning are:

- ALTER STREAM
- ALTER TASK
- ALTER VIEW
- CREATE AGGREGATE FUNCTION
- CREATE FUNCTION
- CREATE PROCEDURE AS
- CREATE PROCEDURE FROM CLASS
- CREATE ROLE
- CREATE STREAM
- CREATE TASK
- DR TABLE
- DROP FUNCTION
- DROP PROCEDURE
- DROP ROLE
- DROP STREAM
- DROP TASK
- PARTITION TABLE

ALTER STREAM

ALTER STREAM — Modifies an existing stream definition.

Syntax

```
ALTER STREAM stream-name DROP [COLUMN] column-name

ALTER STREAM stream-name ADD column-definition [BEFORE column-name]

ALTER STREAM stream-name ALTER column-definition

ALTER STREAM stream-name ALTER [COLUMN] column-name SET {DEFAULT value | [NOT]
        NULL}

column-definition: column-name datatype [DEFAULT value ] [ NOT NULL ]
```

Description

The ALTER STREAM statement modifies an existing stream by adding, dropping, or modifying a column associated with the stream. You cannot drop or modify the column if there are dependencies on that column. For example, if stored procedure queries reference a dropped or modified column, you cannot make the change. In this case, you must drop the stored procedures before making the change to the stream's schema, then recreate the stored procedures afterwards.

If you drop the stream as a whole (using the DROP STREAM statement) and then redefine it using CREATE STREAM, any pending data not already sent to the stream's export target is deleted. ALTER STREAM, on the other hand, does not interrupt pending data. By using ALTER STREAM to modify the schema of the stream, all previously committed data stays in the queue for the target and any inserts after the schema change are added the queue.

Example

The following example modifies an existing stream, *invoice*, to modify the definition of the *customer* column.

```
ALTER STREAM ALTER COLUMN customer SET NOT NULL;
```

ALTER TABLE

ALTER TABLE — Modifies an existing table definition.

Syntax

ALTER TABLE *table-name* DROP CONSTRAINT *constraint-name*

ALTER TABLE *table-name* DROP [COLUMN] *column-name* [CASCADE]

ALTER TABLE *table-name* DROP {PRIMARY KEY | TTL}

ALTER TABLE *table-name* ADD constraint-definition

ALTER TABLE *table-name* ADD column-definition [BEFORE *column-name*]

ALTER TABLE *table-name* ADD ttl-definition

ALTER TABLE *table-name* ALTER column-definition [CASCADE]

ALTER TABLE *table-name* ALTER [COLUMN] *column-name* SET {DEFAULT *value* | [NOT] NULL}

ALTER TABLE *table-name* ALTER export-definition

ALTER TABLE *table-name* ALTER topic-definition

ALTER TABLE *table-name* ALTER ttl-definition

column-definition: [COLUMN] *column-name* datatype [DEFAULT *value*] [NOT NULL] [index-type]

constraint-definition: [CONSTRAINT *constraint-name*] { index-definition }

export-definition: EXPORT TO TARGET *target-name* [ON *action* [,...]]

topic-definition: EXPORT TO TOPIC *topic-name* [ON *action* [,...]]

index-definition: {index-type} (*column-name* [,...])

ttl-definition: USING TTL *value* [*time-unit*] ON COLUMN *column-name* [BATCH_SIZE *number-of-rows*] [MAX_FREQUENCY *value*]

index-type: PRIMARY KEY | UNIQUE | ASSUMEUNIQUE

Description

The ALTER TABLE modifies an existing table definition by adding, removing or modifying a column, constraint, or clause. There are several different forms of the ALTER TABLE statement, depending on what attribute you are altering and how you are changing it. The key point to remember is that you only alter one item at a time. For example, to change two columns or a column and a constraint, you need to issue two ALTER TABLE statements.

There are three ALTER TABLE operations:

- ALTER TABLE ADD
- ALTER TABLE DROP
- ALTER TABLE ALTER

The syntax of each statement depends on whether you are modifying a column, a constraint, or the TTL clause. You can ADD or DROP columns, indexes, and the TTL clause and you can ALTER columns and the TTL clause. However, you cannot ALTER indexes. To alter an existing constraint you must first DROP the constraint and then ADD the new definition.

There are two forms of the ALTER TABLE DROP statement. You can drop a column or constraint by name or you can drop a PRIMARY KEY or a USING TTL clause by identifying the item to drop, since there is only one such item for any given table.

The syntax for the ALTER TABLE ADD statement uses the same syntax to define a new column, constraint, or clause as that used in the CREATE TABLE command. When adding a column you can also specify the BEFORE clause to specify where the new column falls in the order of table columns. If you to not specify BEFORE, the column is added at the end of the list of columns.

When modifying the USING TTL clause, the ALTER TABLE ALTER command specifies the complete replacement definition for the clause, including either or both the BATCH_SIZE or MAX_FREQUENCY clauses.

You cannot alter the MIGRATE TO export target or topic attribute of the table. You also cannot alter any attributes of the table that affect migration. For example, you cannot add, drop, or alter the USING TTL clause if the table is declared with MIGRATE TO. And if the table has both MIGRATE TO and USING TTL, you cannot add, drop, or alter the TTL column. However, you can alter the TTL value, batch size, and frequency.

To add, drop, or alter the MIGRATE action you must drop the table first and redefine it using the CREATE TABLE statement.

When modifying columns, the ALTER TABLE ALTER COLUMN statement can have one of two forms. You can alter the column by providing a complete replacement definition, similar to the ALTER TABLE ADD COLUMN statement, or you can alter a specific attribute using the ALTER TABLE ALTER COLUMN... SET syntax. Use SET DEFAULT to add or modify an existing default. Use SET DEFAULT NULL to remove an existing default. You can also use the SET clause to specify whether the column can be null (SET NULL) or must not contain a null value (SET NOT NULL).

When modifying an export or topic definition using the ALTER TABLE ALTER command, you can only change the ON *action* clause. You cannot add an export or topic definition or change from one target to another.

Handling Dependencies

You can only alter tables if there are no dependencies on the table, column, or index that would be violated by the change. For example, you cannot drop the partitioning column from a partitioned table if there are stored procedures partitioned on that table and column as well. You must first drop the partitioned store procedures before dropping the column. Note that by dropping the partitioning column, you are also automatically changing the table into a replicated table.

The most common dependency is if the table already has data in it. You can add, delete, and (within reasonable bounds) modify the columns of a table with existing data as long as those columns are not named in an index, view, or PARTITION statement. If a column is referenced in a view or index, you can specify CASCADE when you drop the column to automatically drop the referring indexes and views.

When a table has records in it, data associated with dropped columns is deleted. Added columns are interpreted as null or filled in with the specified default value. (You cannot add a column that is defined as NOT NULL, but without a default, if the table has existing data in it.) You can even change the datatype of the column within reason. In other words, you can increase the size of the datatype (for example, from INTEGER to BIGINT) but you cannot decrease the size (say, from INTEGER to TINYINT) since some of the existing data may already violate the size constraint.

You can also add non-unique indexes to tables with existing data. However, you cannot add unique constraints (such as PRIMARY KEY) if data exists.

If a table has no records in it, you can make almost any changes you like to it assuming, again, there are no dependencies. You can add and remove unique constraints, add, remove, and modify columns, even change column datatypes at will.

However, if there are dependencies, such as stored procedure queries that reference a dropped or modified column, you may not be allowed to make the change. If there are such dependencies, it is often easier to do drop the stored procedures before making the changes then recreate the stored procedures afterwards.

Examples

The following example uses ALTER TABLE to drop a unique constraint, add a new column, and then recreate the constraint adding the new column.

```
ALTER TABLE Employee DROP CONSTRAINT UniqueNames;
ALTER TABLE Employee ADD COLUMN MiddleInitial VARCHAR(1);
ALTER TABLE Employee ADD CONSTRAINT UniqueNames
      UNIQUE (FirstName, MiddleInitial, LastName);
```

ALTER TASK

ALTER TASK — Modifies an existing task schedule.

Syntax

```
ALTER TASK task-name [ENABLE | DISABLE]

ALTER TASK task-name ALTER ON ERROR {LOG | IGNORE | STOP}
```

Description

The ALTER TASK statement lets you modify an existing scheduled task. You can enable, disable, or change the error handling for the task.

Examples

The following example changes the error handling for the task *cleanup* to log errors and continue, then enables the task, in case it was previously disabled.

```
ALTER TASK cleanup ALTER ON ERROR LOG;
ALTER TASK cleanup ENABLE;
```

ALTER VIEW

ALTER VIEW — modifies the TTL settings of an existing stream view

Syntax

```
ALTER VIEW view-name ALTER USING TTL value [time-unit] ON COLUMN column-name
      [BATCH_SIZE number-of-rows] [MAX_FREQUENCY value]

time-unit: SECONDS | MINUTES | HOURS | DAYS
```

Description

The ALTER VIEW statement lets you modify the USING TTL clause of a stream view. Note that the ALTER statement can only be used to modify a stream view (not a view on a regular database table) and that you can only change the USING TTL clause.

What's more, although you must specify the entire USING TTL clause including the column specification, you can only change the time value, batch size or maximum frequency. You cannot change the TTL column or add or remove the USING TTL clause entirely. To add a USING TTL clause to a stream view without one, to remove an existing USING TTL clause, or to change the TTL column, you must first drop the view then recreate it (using the DROP VIEW and CREATE VIEW statements).

Examples

The following example creates a stream view with a USING TTL clause, then uses ALTER VIEW to reduce the TTL value and change the batch size to 500.

```
CREATE VIEW logins_per_day
  (userid, session_day, total_count)
  AS SELECT userid, TRUNCATE(DAY, start_time), count(*)
     FROM user_session GROUP BY userid, TRUNCATE(DAY, start_time)
     USING TTL 7 DAY ON COLUMN session_day;
ALTER VIEW logins_per_day ALTER USING TTL 3 DAYS ON COLUMN session_day
  BATCH_SIZE 500;
```

CREATE AGGREGATE FUNCTION

CREATE AGGREGATE FUNCTION — Defines an aggregate SQL function and associates it with a Java class.

Syntax

```
CREATE AGGREGATE FUNCTION function-name FROM CLASS class-path
```

Description

The CREATE AGGREGATE FUNCTION statement declares a user-defined aggregate function and associates it with a Java class. Aggregate functions process multiple values based on a query expression and produce a single result. For example, the built-in AVG aggregate function calculates the average of the values of a specific column or expression based on the query constraints.

The return value of a user-defined aggregate function matches the datatype of the Java method itself. Similarly, the number and datatype of the function's arguments are defined by the arguments of the method.

User-defined aggregate functions allow you to extend the functionality of the SQL language by declaring your own functions that can be used in SQL queries and data manipulation statements. The steps for creating a user-defined aggregate function are:

1. Write, compile, and debug the program code for a class that performs the function's action. The class must include the following methods:
   - `start()` — Initializes the function. Called once for each invocation of the function.
   - `assemble( arg,... )` — Processes the arguments to the function. called once for each record matching the constraints of the query in which the function appears.
   - `combine( class-instance )` — For partitioned queries, combines the results of one partition into the results of another.
   - `end()` — Finalizes the function and returns the function result. Called once at the completion of the function invocation.
2. Package the class in a JAR file, just as you would a stored procedure. (Classes for functions and stored procedures can be packaged in the same JAR file.)
3. Load the JAR file into the database using the LOAD CLASSES statement.
4. Declare and name the user-defined function using the CREATE AGGREGATE FUNCTION statement.

The Java methods that implement the user-defined function must follow the same rules for determinism as user-defined stored procedures, as outlined in Section 5.1.2.1, “Avoid Introducing Non-deterministic Values from External Functions”. See the chapter on "Creating Custom SQL Functions" in the *VoltDB Guide to Performance and Customization* for details on designing the Java class and methods necessary for a user-defined aggregate function.

To declare a scalar rather than an aggregate function, see the description of the CREATE FUNCTION statement.

Examples

The following example defines an aggregate function called longest_word from the start(), assemble(), combine(), and end() methods in the class LongestWord:

```
CREATE AGGREGATE FUNCTION longest_word FROM CLASS myapp.functions.LongestWord;
```

CREATE FUNCTION

CREATE FUNCTION — Defines a SQL scalar function and associates it with a Java method.

Syntax

```
CREATE FUNCTION function-name FROM METHOD class-path.method-name
```

Description

The CREATE FUNCTION statement declares a user-defined function and associates it with a Java method. The return value of the function matches the datatype of the Java method itself. Similarly, the number and datatype of the function's arguments are defined by the arguments of the method.

User-defined functions allow you to extend the functionality of the SQL language by declaring your own functions that can be used in SQL queries and data manipulation statements. The steps for creating a user-defined function are:

1. Write, compile, and debug the program code for the method that will perform the function's action.
2. Package the class and method in a JAR file, just as you would a stored procedure. (Classes for functions and stored procedures can be packaged in the same JAR file.)
3. Load the JAR file into the database using the LOAD CLASSES statement.
4. Declare and name the user-defined function using the CREATE FUNCTION statement.

For example, let's say you want to create function that decodes an HTML-encoded string. The beginning of the Java method might look like this, declaring a method of type String and accepting two arguments: the string to encode and an integer value for the maximum length.

```
package myapp.datatypes;

  public class Html {

    public String decode( String html, int maxlength )
                throws VoltAbortException {
```

After compiling and packaging this class into a JAR file, you can load the class and declare it as a SQL function:

```
sqlcmd
1> LOAD CLASSES myfunctions.jar;
2> CREATE FUNCTION html_decode FROM METHOD myapp.datatypes.Html.decode;
```

Note that the function name and method name do not have to be identical. Also, the function name is not case sensitive. However, the Java class and method names *are* case sensitive. Finally, the Java methods for user-defined functions must follow the same rules for determinism as user-defined stored procedures, as outlined in Section 5.1.2.1, “Avoid Introducing Non-deterministic Values from External Functions”.

Examples

The following example defines a function called emoticon from a Java method findEmojiCode:

```
CREATE FUNCTION emoticon FROM METHOD utils.Charcode.findEmojiCode;
```

CREATE INDEX

CREATE INDEX — Creates an index for faster access to a table.

Syntax

```
CREATE [UNIQUE|ASSUMEUNIQUE] INDEX index-name
        ON {table-name | view-name} ( index-column [,...])
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
```

Description

Creating an index on a table or view makes read access to the data faster when using the columns of the index as a key. Note that VoltDB creates an index automatically when you specify a constraint, such as a primary key, in the CREATE TABLE statement.

When you specify that the index is UNIQUE, VoltDB constrains the table to at most one row for each set of index column values. If an INSERT or UPDATE statement attempts to create a row where all the index column values match an existing indexed row, the statement fails.

Because the uniqueness constraint is enforced separately within each partition, only indexes on replicated tables or containing the partitioning column of partitioned tables can ensure global uniqueness for partitioned tables and therefore support the UNIQUE keyword.

If you wish to create an index on a partitioned table that acts like a unique index but does not include the partitioning column, use the keyword ASSUMEUNIQUE instead of UNIQUE. Assumed unique indexes are treated like unique indexes (VoltDB verifies they are unique within the current partition). However, it is your responsibility to ensure these indexes are actually globally unique. Otherwise, it is possible an index will generate a constraint violation during an operation that modifies the partitioning of the database (such as adding nodes on the fly or restoring a snapshot to a different cluster configuration).

The indexed items (*index-column*) are either columns of the specified table or expressions, including functions, based on the table. For example, the following statements index a table based on the calculated area and its distance from a set location:

```
CREATE INDEX areaofplot ON plot (width * height);
CREATE INDEX distancefrom49 ON plot ( ABS(latitude - 49) );
```

You can create a *partial index* by including a WHERE clause in the index definition. The WHERE clause limits the number of rows that get indexed. This is useful if certain columns in the index are not evenly distributed. For example, if you are not interested in records where a column is null, you can use a WHERE clause to exclude those records and optimize the size and performance of the index.

The partial index is utilized by the database when a query's WHERE clause contains the same condition as the partial index definition. A special case is if the index condition is `{column} IS NOT NULL`. In this situation, the index may be applied even in the query does not contain that exact condition, as long as the query contains a WHERE condition that *implies* the column is not null, such as `{column} > 0`.

VoltDB uses tree indexes. They provide the best general performance for a wide range of operations, including exact value matches and queries involving a range of values, such as `SELECT ... WHERE Score > 1 AND Score < 10`.

Examples

The following example creates two indexes on a single table. The first is, by default, a non-unique index based on the departure time The second is a unique index based on the columns for the airline and flight number.

```
CREATE INDEX flightTimeIdx ON FLIGHT ( departtime );
CREATE UNIQUE INDEX FlightKeyIdx ON FLIGHT ( airline, flightID );
```

You can also use functions in the index definition. For example, the following is an index based on the element *movie* within a JSON-encoded VARCHAR column named *favorites* and the member's ID.

```
CREATE INDEX FavoriteMovie ON MEMBER (
       FIELD( favorites, 'movie' ), memberID
);
```

The following example demonstrates the use of a partial index, by including a WHERE clause, to exclude records with a null column.

```
CREATE INDEX completed_tasks
    ON tasks (task_id, startdate, enddate)
    WHERE enddate IS NOT NULL;
```

CREATE PROCEDURE AS

CREATE PROCEDURE AS — Defines a stored procedure composed of one or more SQL statements.

Syntax

```
CREATE PROCEDURE procedure-name
        [PARTITION ON TABLE table-name COLUMN column-name [PARAMETER position]]
        [ALLOW role-name [,...]]
        AS {sql-statement; | multi-statement-procedure}

CREATE DIRECTED PROCEDURE procedure-name
        [ALLOW role-name [,...]]
        AS {sql-statement; | multi-statement-procedure}

multi-statement-procedure:
        BEGIN
          sql-statement; [,...]
        END;
```

Description

You must declare stored procedures as part of the schema to make them accessible at runtime. The CREATE PROCEDURE AS statement lets you create a procedure from one or more SQL statements directly within the DDL statement. The SQL statements can contain question marks (?) as placeholders that are filled in at runtime with the arguments to the procedure call.

There are two ways to define a procedure as part of the CREATE PROCEDURE AS statement:

- A single statement procedure where the CREATE PROCEDURE AS statement is followed by one SQL statement terminated by a semi-colon.
- A multi-statement procedure where the CREATE PROCEDURE AS statement is followed by multiple SQL statements enclosed in a BEGIN-END clause.

For a single statement, the stored procedure returns the results of the query as a VoltTable. For multi-statement procedures, the results are returned as an array of VoltTable structures, one for each statement.

For all CREATE PROCEDURE AS statements, the procedure name must follow the naming conventions for Java class names. For example, the name is case-sensitive and cannot contain any white space.

You can create three types of stored procedures:

- *Multi-Partition Procedures* — By default, the CREATE PROCEDURE statement declares a multi-partition procedure. A multi-partition procedure runs as a single transaction and has access to data from the entire database. However, it also means that the procedure will access all of the partitions at once, blocking the transaction queues until the procedure is done.
- *Single-Partition Procedures* — If you include the PARTITION ON clause, the procedure is partitioned and runs on only one partition of the database. The partition it runs on is determined by the value of one of the parameters you pass to the procedure at runtime, as described below.
- *Directed Procedures* — if you include the DIRECTED clause, the procedure is a directed procedure and will run separate transactions on each of the partitions. However, the individual transactions are not

coordinated. Directed procedures must be invoked as a scheduled task or using the `callAllPartitionProcedure` method. See Section 7.5, "Directed Procedures: Distributing Transactions to Every Partition" and the description of the CREATE TASK statement for more information on directed procedures.

When creating single-partitioned procedures, you specify the partitioning in the PARTITION ON clause. Partitioning a stored procedure means that the procedure executes within a unique partition of the database. The partition in which the procedure executes is chosen at runtime based on the table and column specified by *table-name* and *column-name*. By default, VoltDB uses the first parameter to the stored procedure as the partitioning value. However, you can use the PARAMETER clause to specify a different parameter. The *position* value specifies the parameter position, counting from zero. (In other words, position 0 is the first parameter, position 1 is the second, and so on.) The specified table must be a partitioned table or stream.

If security is enabled at runtime, only those roles named in the ALLOW clause (or with the ALLPROC or ADMIN permissions) have permission to invoke the procedure. If security is not enabled at runtime, the ALLOW clause is ignored and all users have access to the stored procedure.

Examples

The following example defines a stored procedure, *CountUsersByCountry*, as a single SQL query with a placeholder for matching the *country* column:

```
CREATE PROCEDURE CountUsersByCountry AS
    SELECT COUNT(*) FROM Users WHERE country=?;
```

The next example restricts access to the stored procedure to only users with the *operator* role. It also partitions the stored procedure on the *userID* column of the *Accounts* table. Note that the PARAMETER clause is used since the userID is the second parameter to the procedure:

```
CREATE PROCEDURE ChangeUserPassword
    PARTITION ON TABLE Accounts COLUMN userID PARAMETER 1
    ALLOW operator
    AS UPDATE Accounts SET HashedPassword=? WHERE userID=?;
```

The last example uses a BEGIN-END clause to include four SQL statements in the procedure. In this case, the procedure performs two INSERT INTO SELECT statements, a DELETE statement and then selects the total count of records after the operation. The stored procedure returns four VoltTables, one for each statement, with the last one containing the final record count since SELECT is the last statement in the procedure.

```
CREATE PROCEDURE MoveOrders
  AS BEGIN
    INSERT INTO enroute SELECT * FROM Orders
        WHERE ship_date < NOW() AND delivery_date > NOW();
    INSERT INTO history SELECT * FROM enroute
        WHERE delivery_date < NOW();
    DELETE FROM enroute
        WHERE delivery_date < NOW();
    SELECT COUNT(*) FROM enroute;
  END;
```

CREATE PROCEDURE FROM CLASS

CREATE PROCEDURE FROM CLASS — Defines a stored procedure associated with a Java class.

Syntax

```
CREATE PROCEDURE
        [PARTITION ON TABLE table-name COLUMN column-name [PARAMETER position]]
        [ALLOW role-name [,...]]
        FROM CLASS class-name

CREATE DIRECTED PROCEDURE
        [ALLOW role-name [,...]]
        FROM CLASS class-name

CREATE COMPOUND PROCEDURE
        [ALLOW role-name [,...]]
        FROM CLASS class-name
```

Description

You must declare stored procedures to make them accessible to client applications and the sqlcmd utility. CREATE PROCEDURE FROM CLASS lets you declare stored procedures that are written as Java classes. The *class-name* is the name of the Java class.

Before you declare the stored procedure, you must create, compile, and load the associated Java class. It is usually easiest to do this by compiling all of your Java stored procedures and packaging the resulting class files into a single JAR file that can be loaded once. For example:

```
$ javac -d ./obj src/procedures/*.java
$ jar  cvf  myprocs.jar –C obj .
$ sqlcmd
1> LOAD CLASSES myprocs.jar;
2> CREATE PROCEDURE FROM CLASS procedures.AddCustomer;
```

You can create four types of stored procedures:

- *Multi-Partition Procedures* — By default, the CREATE PROCEDURE statement declares a multi-partition procedure. A multi-partition procedure runs as a single transaction and has access to data from the entire database. However, it also means that the procedure will access all of the partitions at once, blocking the transaction queues until the procedure is done.

- *Single-Partition Procedures* — If you include the PARTITION ON clause, the procedure is partitioned and runs on only one partition of the database. The partition it runs on is determined by the value of one of the parameters you pass to the procedure at runtime, as described below.

- *Directed Procedures* — If you include the DIRECTED clause, the procedure is a directed procedure and will run separate transactions on each of the partitions. However, the individual transactions are not coordinated. Directed procedures must be invoked as a scheduled task or using the `callAllPartitionProcedure` method. See Section 7.5, “Directed Procedures: Distributing Transactions to Every Partition” and the description of the CREATE TASK statement for more information on directed procedures.

- *Compound Procedures* — If you include the COMPOUND clause, and the specified class extends the `VoltCompoundProcedure` class, the procedure is defined as a compound procedure. Compound procedures are *not* transactional, but can call multiple regular stored procedures, perform analysis and error handling, and return custom results. See the chapter on Compound Procedures in the VoltDB Performance and Customization Guide for more information.

When creating single-partitioned procedures, you specify the partitioning in the PARTITION ON clause. Partitioning a stored procedure means that the procedure executes within a unique partition of the database. The partition in which the procedure executes is chosen at runtime based on the table and column specified by *table-name* and *column-name*. By default, VoltDB uses the first parameter to the stored procedure as the partitioning value. However, you can use the PARAMETER clause to specify a different parameter. The *position* value specifies the parameter position, counting from zero. (In other words, position 0 is the first parameter, position 1 is the second, and so on.)

The specified table must be a partitioned table and cannot be an export stream or replicated table.

If security is enabled at runtime, only those roles named in the ALLOW clause (or with the ALLPROC or ADMIN permissions) have permission to invoke the procedure. If security is not enabled at runtime, the ALLOW clause is ignored and all users have access to the stored procedure.

Directed Procedures

A *directed procedure* is a special type of stored procedure, declared using the DIRECTED clause. Multi-partition procedures run as a single transaction with access to all partitions. Single-partitioned procedures run as a single transaction on one partition, determined by the table and column on which the procedure is partitioned. A directed procedure queues multiple transactions at once, one on each partition without regard to any specific table or partitioning column.

By running separate transactions on each partition, directed procedures do not block the partition queues the way multi-partition procedures do. However, there is no guarantee that the individual transactions are executed at the same time. Consequently, directed procedures are good for work activities that need access to all of the partitions but do not need to be coordinated as a single transaction. For example, cleanup tasks that delete orphaned records or update non-critical columns across the database.

Directed procedures are transactional and must obey the same rules as other stored procedures. That is, they are Java classes that extend voltProcedure, use the voltQueueSQL method to queue SQL statements, and they must be deterministic. (See Chapter 5, *Designing Stored Procedures to Access the Database* for more information about determinism and the structure of stored procedures.)

However, because of the different execution model of the directed procedure, you cannot call them with the `callProcedure` method the way you call partitioned or multi-partitioned procedures. Instead, directed procedures are invoked using the `callAllPartitionProcedure` method or as a scheduled task with the ON PARTITIONS clause. See the description of the CREATE TASK statement for more information on invoking directed procedures as an automated task.

Compound Procedures

A *compound procedure* is another special type of stored procedure, declared using the COMPOUND clause. Unlike all other procedures, compound procedures themselves are *not* transactional. That is, the procedure does not succeed or rollback as a unit. Instead, compound procedures let you call multiple regular stored procedures that *are* transactional, analyze their results, handle errors, and perform other program logic.

Because they can invoke multiple transactions, compound procedures are particularly useful for processing input from topics, where there is no client application to process the results of SQL queries. Compound

procedures let you embed the processing of results and conditional programming into the procedure itself. The drawback is that, if anything unexpected happens, it is up to the procedure to handle any exceptions.

In general, compound procedures are intended for making multiple calls to single-partitioned procedures. However, compound procedures can call both partitioned and multi-partitioned procedures. And when it does call partitioned procedures, it can specify different partitioning values. So the compound procedure itself cannot be partitioned. It also runs in a separate queue from the transactional procedures it calls. See the chapter on Using Compound Procedures in the *VoltDB Performance and Customization Guide* for more information.

Example

The following example declares a stored procedure matching the Java class MakeReservation. Note that the class name includes its location within the current class path (in this case, as a child of *flight* and *procedures*). However, the name itself, *MakeReservation*, must be unique within the schema because at runtime stored procedures are invoked by name only.

```
CREATE PROCEDURE FROM CLASS flight.procedures.MakeReservation;
```

CREATE ROLE

CREATE ROLE — Defines a role and the permissions associated with that role.

Syntax

```
CREATE ROLE role-name [WITH permission [,...]]
```

Description

The CREATE ROLE statement defines a named role that can be used to assign access rights to specific procedures and functions. When security is enabled in the database configuration, the permissions assigned in the CREATE ROLE and CREATE PROCEDURE statements specify which users can access which functions.

Use the CREATE PROCEDURE statement to assign permissions to named roles for accessing specific stored procedures. The CREATE ROLE statement lets you assign certain generic permissions. The following table describes the permissions that can be assigned the WITH clause.

Permission	Description	Inherits
DEFAULTPROCREAD	Access to read-only default procedures (*TABLE*.select)	
DEFAULTPROC	Access to all default procedures (*TABLE*.select, *TABLE*.insert, *TABLE*.delete, *TABLE*.update, and *TABLE*.upsert)	DEFAULTPROCREAD
SQLREAD	Access to read-only ad hoc SQL queries (SELECT)	DEFAULTPROCREAD
SQL	Access to all ad hoc SQL queries and default procedures	SQLREAD, DEFAULTPROC
ALLPROC	Access to all user-defined stored procedures	
ADMIN	Full access to all system procedures, all user-defined procedures, as well as default procedures, ad hoc SQL, and DDL statements.	ALLPROC, DEFAULTPROC, SQL
Note: For backwards compatibility, the special permissions ADHOC and SYSPROC are still recognized. They are interpreted as synonyms for SQL and ADMIN, respectively.		

The generic permissions are denied by default. So you must explicitly enable them for those roles that need them. For example, if users assigned to the "interactive" role need to run ad hoc queries, you must explicitly assign that permission in the CREATE ROLE statement:

```
CREATE ROLE interactive WITH sql;
```

Also note that the permissions are additive. So if a user is assigned to one role that allows access to defaultproc but not allproc, but that user also is assigned to another role that allows allproc, the user has both permissions.

Example

The following example defines three roles — *admin*, *developer*, and *batch* — each with a different set of permissions:

```
CREATE ROLE admin WITH admin;
CREATE ROLE developer WITH sql, allproc;
CREATE ROLE batch WITH defaultproc;
```

CREATE STREAM

CREATE STREAM — Creates an output stream in the database.

Syntax

```
CREATE STREAM stream-name
        [PARTITION ON COLUMN column-name]
        [export-definition | topic-definition]
        ( column-definition [,...] );

export-definition: EXPORT TO TARGET export-target-name

topic-definition: EXPORT TO TOPIC topic-name
        [ WITH [KEY (column-name [,...])] [VALUE (column-name [,...])]]

column-definition: column-name datatype [DEFAULT value ] [ NOT NULL ]
```

Description

The CREATE STREAM statement defines a stream and its associated columns in the database. A stream can be thought of as a virtual table. It has the same structure as a table, consisting of a list of columns and supporting all the same datatypes (Table A.1, “Supported SQL Datatypes”) as tables. The columns have the same rules in terms of naming and size. You can also use the INSERT statement to insert data into the stream once it is defined.

The three differences between streams and tables are:

- No data is stored in the database for a stream, it is only used as a passthrough.
- Because no data is stored, you cannot SELECT, UPDATE, or DELETE the stream contents.
- No indexes or constraints (such as primary keys) are allowed on a stream.

Data inserted into the stream is not stored in the database. The stream is an ephemeral container used only for analysis and/or passing data through VoltDB to other systems via the export function.

Combining streams with views lets you perform summary analysis on data passing through VoltDB without having to store all of the underlying data. For example, you might want to know how many times users access a website and their most recent visit. But you do not need to store a record for each visit. In this case, you can create a stream, *visits*, to capture the event and a view, *visit_by_user*, to capture the cumulative data:

```
CREATE STREAM visits PARTITION ON COLUMN user_id (
    user_id BIGINT NOT NULL,
    ip_address VARCHAR(128),
    login TIMESTAMP
);
CREATE VIEW visit_by_user
    ( user_id, total_visits, last_visit )
    AS SELECT user_id, COUNT(*), MAX(login)
       FROM visits GROUP BY user_id;
```

When creating a view on a stream, the stream must be partitioned and the partition column must appear in the view. Another special feature of views on streams is that, because there is no underlying data stored for the view, VoltDB lets you modify the views content manually by issuing UPDATE and DELETE statements on the view. (This ability to manipulate the view is only available for views on streams. You cannot UPDATE or DELETE a view on a table; you must modify the data in the underlying table instead.)

For example, if you only care about a daily rollup of visits, you can use DELETE with the stream name to clear the data at midnight every night:

```
DELETE FROM visit_by_user;
```

Or if you need to adjust the cumulative analysis to, say, "reset" the entry for a specific user, you can use UPDATE:

```
UPDATE visit_by_user
    SET total_visits = 0, last_visit = NULL
    WHERE user_id = ?;
```

Export Streams

Streams can be used to export data out of VoltDB into other systems, such as Kafka, CSV files, and so on. To export data into another system, you start by declaring one or more streams defining the data that will be sent to the external system. In the CREATE STREAM statement you specify the named target for the export:

```
CREATE STREAM visits
  EXPORT TO TARGET archive
  PARTITION ON COLUMN user_id (
    user_id BIGINT NOT NULL,
    ip_address VARCHAR(128),
    login TIMESTAMP
);
```

As soon as you declare the EXPORT TO TARGET clause for a stream, any data inserted into the stream is queued for export. If the export target is not defined in the database configuration, then the data waits in the queue. Once the export target is configured, the export connector begins sending the queued data to the configured destination. See Chapter 15, *Streaming Data: Import, Export, and Migration* for more information on configuring export targets.

Topic Streams

Alternately, you can output a stream to a VoltDB topic. Topics stream data to and from external systems, similar to import and export, with two distinct differences. First, topics share data written into the stream with multiple external consumers. Second, rather than *pushing* data to a single target the way export does, topics allow multiple consumers to *pull* the data when they need it or when they are ready for it.

To identify a stream as an output source for a topic, you include the EXPORT TO TOPIC clause in the CREATE STREAM statement, naming the topic to use:

```
CREATE STREAM visits PARTITION ON COLUMN user_id
  EXPORT TO TOPIC visitors
  PARTITION ON COLUMN user_id (
    user_id BIGINT NOT NULL,
    ip_address VARCHAR(128),
    login TIMESTAMP
```

```
);
```

The topic itself is configured in the database configuration file. If the topic is not configured before the stream is declared, no data written to the stream is added to the queue until the topic is added to the configuration. Similarly, if the topic is removed from the configuration, the queue for the topic and its contents are deleted.

There are two optional clauses associated with EXPORT TO TOPIC, KEY and VALUE, which are preceded with the WITH keyword. KEY identifies one or more columns to use as a key for the topic. So, for example, if the column *user_id* is defined as the key and you execute INSERT INTO visits (123, "1.2.3.4", NOW()), the value 123 is used as the key for the topic message. VALUE identifies which columns (and in which order) to include in the body of the topic message. In the following example, *user_id* is used as the key and *user_id* and *login* are included in the body of the message (leaving out *ip_address*):

```
CREATE STREAM visits PARTITION ON COLUMN user_id
  EXPORT TO TOPIC visitors
    WITH KEY (user_id) VALUE (user_id, login)
  PARTITION ON COLUMN user_id (
    user_id BIGINT NOT NULL,
    ip_address VARCHAR(128),
    login TIMESTAMP
);
```

If you do not specify a key, there is no key for the topic. If you do not specify values, all columns from the stream are included in the order specified in the CREATE STREAM statement. See the section on Section 15.6, “VoltDB Topics” for more information on defining and using topics.

Multi-Purpose Streams

Finally, you can combine analysis with export by creating a stream with an export target and also creating a view on that stream. So in our earlier example, if we want to warehouse data about each visit but use VoltDB to perform the real-time summary analysis, we would add an export definition, along with the partitioning clause, to the CREATE STREAM statement for the *visits* stream:

```
CREATE STREAM visits
  PARTITION ON COLUMN user_id
  EXPORT TO TARGET warehouse (
    user_id BIGINT NOT NULL,
    ip_address VARCHAR(128),
    login TIMESTAMP
);
```

Example

The following example defines a stream and a view on that stream. Note the use of the PARTITION ON clause to ensure the stream is partitioned, since it is being used in a view.

```
CREATE STREAM flightdata
  PARTITION ON COLUMN airport (
    flight_id BIGINT NOT NULL,
    airport VARCHAR(3) NOT NULL,
    passengers INTEGER,
    eta TIMESTAMP
);
```

```
CREATE VIEW all_flights
  (airport, flight_count, passenger_count)
  AS SELECT airport, count(*),sum(passengers)
     FROM flightdata GROUP BY airport;
```

CREATE TABLE

CREATE TABLE — Creates a table in the database.

Syntax

```
CREATE TABLE table-name

[ export-definition | topic-definition | migrate-target-definition | migrate-topic-definition ]

( column-definition [,...]
        [, constraint-definition [,...]]

) [ttl-definition] ;

export-definition: EXPORT TO TARGET target-name [ON action [,...]]

topic-definition: EXPORT TO TOPIC topic-name [ON action [,...]]
        [ WITH [KEY (column-name [,...])] [VALUE (column-name [,...])]]

migrate-target-definition: MIGRATE TO TARGET target-name

migrate-topic-definition: MIGRATE TO TOPIC topic-name
        [ WITH [KEY (column-name [,...])] [VALUE (column-name [,...])]]

column-definition: column-name datatype [DEFAULT value ] [ NOT NULL ] [index-type]

constraint-definition: [CONSTRAINT constraint-name] { index-definition }

index-definition: {index-type} (column-name [,...])

index-type: PRIMARY KEY | UNIQUE | ASSUMEUNIQUE

ttl-definition: USING TTL value [time-unit] ON COLUMN column-name
        [BATCH_SIZE number-of-rows] [MAX_FREQUENCY value]

time-unit: SECONDS | MINUTES | HOURS | DAYS
```

Description

The CREATE TABLE statement creates a table and its associated columns in the database. The supported datatypes are described in Table A.1, “Supported SQL Datatypes”.

Table A.1. Supported SQL Datatypes

SQL Datatype	Equivalent Java Datatype	Description
TINYINT	byte	1-byte signed integer, -127 to 127[a]
SMALLINT	short	2-byte signed integer, -32,767 to 32,767
INTEGER	int	4-byte signed integer, -2,147,483,647 to 2,147,483,647
BIGINT	long	8-byte signed integer, -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807

SQL Datatype	Equivalent Java Datatype	Description
FLOAT	double	8-byte numeric, $-(2-2^{-52})\cdot 2^{1023}$ to $(2-2^{-52})\cdot 2^{1023}$ (Note that values less than or equal to -1.7E+308 are interpreted as null.)
DECIMAL	BigDecimal	16-byte fixed scale of 12 and precision of 38, -99999999999999999999999999.999999999999 to 99999999999999999999999999.999999999999
GEOGRAPHY or GEOGRAPHY()		A geospatial region. The storage requirement for geospatial data varies depending on the geometry. The default maximum size in memory is 32768. However, you can specify a different value by specifying the maximum size (in bytes) in the declaration. For example: GEOGRAPHY(80000). See the section on entering geospatial data in the *VoltDB Guide to Performance and Customization* for details.
GEOGRAPHY_POINT		A geospatial location identified by its latitude and longitude. Requires 16 bytes of storage.
VARCHAR()	String	Variable length text string, with a maximum length specified in either characters (the default) or bytes. To specify the length in bytes, use the BYTES keyword after the length value. For example: VARCHAR(28 BYTES). Maximum allowable length is 1MB.
VARBINARY()	byte array	Variable length binary string (sometimes referred to as a "blob") with a maximum length specified in bytes. Maximum allowable length is 1MB.
TIMESTAMP	long, VoltDB TimestampType	Time in microseconds. Although any long integer value can be inserted into a TIMESTAMP column, for time related functions such as EXTRACT and DATEDIFF the value must fall between the first microsecond of the year 1583 and the last microsecond of the year 9999.
DATE	java.sql.Date, java.util.Date, VoltDB DateType	Date expressed in year, month, and day between 1583-01-01 and 9999-12-31 inclusive.
BOOLEAN	VoltDB BooleanType	A boolean value equating to true, false, or null.

[a]For integer and floating-point datatypes, VoltDB reserves the largest possible negative value to denote a null value. For example -128 is interpreted as null for TINYINT, -32768 for SMALLINT, and so on.

The following limitations are important to note when using the CREATE TABLE statement in VoltDB:

- CHECK and FOREIGN KEY constraints are not supported.
- VoltDB does not support AUTO_INCREMENT, the automatic incrementing of column values.
- A table can have up to 1024 columns. Each column has a maximum size of 1 megabyte and the total declared size of all of the columns in a table cannot exceed 2 megabytes. For VARCHAR columns where the length is specified in characters, the declared size is calculated as 4 bytes per character to allow for the longest potential UTF-8 string.

- If you intend to use a column to partition a table, that column cannot contain null values. You must specify NOT NULL in the definition of the column or VoltDB issues an error when compiling the schema.

- To specify an index — either for an individual column or as a table constraint — that is globally unique across the database, use the standard SQL keywords UNIQUE and PRIMARY KEY. However, for partitioned tables, VoltDB can only ensure uniqueness if the index includes the partitioning column. Otherwise, these keywords are not allowed.

  It can be a performance advantage to define indexes or constraints on non-partitioning columns that you, as the developer, know are going to contain unique values. Although VoltDB cannot ensure uniqueness across the entire database, it does allow you to define indexes that are assumed to be unique by using the ASSUMEUNIQUE keyword.

  When you define an index on a partitioned table as ASSUMEUNIQUE, VoltDB verifies uniqueness within the current partition when creating an index entry. However, it is your responsibility as developer or administrator to ensure that the values are actually globally unique. If the database is repartitioned due to adding new nodes or restoring a snapshot to a different cluster configuration, non-unique ASSUME-UNIQUE index entries may collide. When this occurs it results in a constraint violation error and the database will not be able to complete its current action.

  Therefore, ASSUMEUNIQUE should be used with caution. Also, it is not necessary and should not be used with replicated tables or indexes that contain the partitioning column, which can be defined as UNIQUE.

- EXPORT TO TARGET and EXPORT TO TOPIC allow you to connect a table to either an export target or a topic, so that by default data written into the table is also sent to the topic or the export connector for delivery to the specified target. By default, only insert operations (INSERT and UPSERT when it inserts a new row) initiate export records. However, you can use the ON clause to specify which actions you want to trigger export. For example, the following table declaration generates export records whenever rows are created or modified.

  ```
  CREATE TABLE RESERVATION
     EXPORT TO TARGET airlines ON INSERT, UPDATE_NEW
       (reserv_id INT NOT NULL,
        flight_id INT NOT NULL,
        . . . );
  ```

  The following table defines the actions that you can specify in the ON clause.

Keyword	Description
INSERT	Contents of new record from INSERT, or UPSERT that creates new record
DELETE	Contents of a record that is deleted
UP-DATE_OLD	Contents of a record before it is updated
UP-DATE_NEW	Contents of a record after it is updated
UPDATE	Two records are exported, the contents before and after a record is updated (shorthand equivalent for specifying both UPDATE_OLD and UPDATE_NEW)

- EXPORT TO TOPIC (and MIGRATE TO TOPIC) also allow you to specify which column values are used as the topic key and which as the body of the topic message using the WITH KEY/VALUE clause. See the description of topic streams in the description of CREATE STREAM for more information.

- The length of VARCHAR columns can be specified in either characters (the default) or bytes. To specify the length in bytes, include the BYTES keyword after the length value; for example VARCHAR(16 BYTES).

  Specifying the VARCHAR length in characters is recommended because UTF-8 characters can require a variable number of bytes to store. By specifying the length in characters you can be sure the column has sufficient space to store any string of the specified length. Specifying the length in bytes is only recommended when all values contain only single byte (ASCII) characters or when conserving space is required and the strings are less than 64 bytes in length.

- The VARBINARY datatype provides variable storage for arbitrary strings of binary data and operates similarly to VARCHAR(n BYTES) strings. You assign byte arrays to a VARBINARY column when passing in variables, or you can use a hexidecimal string for assigning literal values in the SQL statement.

- The VoltDB TIMESTAMP datatype is a long integer representing the number of microseconds since the epoch. Two important points to note about this timestamp:

  - The VoltDB TIMESTAMP is not the same as the Java Timestamp datatype or traditional Linux time measurements, which are measured in milliseconds rather than microseconds. Appropriate conversion is needed when casting values between a VoltDB TIMESTAMP and other timestamp datatypes.

  - The VoltDB TIMESTAMP is interpreted as a Greenwich Meantime (GMT) value. Depending on how time values are created, their value may or may not account for the local machine's default time zone. Mixing timestamps from different time zones (for example, in WHERE clause comparisons) can result in unexpected behavior.

- For TIMESTAMP columns, you can define a default value using the NOW or CURRENT_TIMESTAMP keywords in place of a specific value. For DATE columns you can use the NOW or CURRENT_DATE keywords. For example:

  ```
  CREATE TABLE Event (
      Event_Id INTEGER UNIQUE NOT NULL,
      Event_Timestamp TIMESTAMP DEFAULT NOW,
      Event_Description VARCHAR(128)
  );
  ```

  The default value is evaluated at runtime as an approximation, in milliseconds, of when the transaction begins execution.

Automatic Aging and Data Migration

When you define a database table you can also define a "time to live" (TTL) when records in the table expire and are automatically deleted. The USING TTL clause specifies a lifetime for each record, based on the difference between the specified TTL value, the value of the specified column, and the current time (in GMT microseconds). In the simplest case, you can define a time to live based on a TIMESTAMP column defined as DEFAULT NOW, so the record expires the specified amount of time after it is inserted. For example, the records in the following table will be deleted five minutes after they are inserted into the database (assuming the default value is used for the `created` column):

```
CREATE TABLE current_alerts (
   id BIGINT NOT NULL,
   message VARCHAR(128),
   created TIMESTAMP DEFAULT NOW NOT NULL,
) USING TTL 5 MINUTES ON COLUMN created;
```

You specify the time to live value as an integer number of seconds, minutes, hours, or days. (The default, if you do not specify a time unit, is seconds.) The TTL column must be declared as a TIMESTAMP and NOT NULL.

TTL records are evaluated and deleted by a parallel process within the database. As a result, records are deleted shortly *after* the specified time to live arrives, rather than at the exact time specified. But the deletion of records is handled as a proper database transaction, guaranteeing consistency with any user-invoked transactions. One consequence of automating the expiration of database records, is that the evaluation and deletion of records produces additional transactions that may impact database performance.

When you define an expiration time for database records, you can also specify an export target or a topic using MIGRATE TO TARGET or MIGRATE TO TOPIC. If you specify both USING TTL and MIGRATE TO TARGET, before the data is deleted by the TTL process, the data is migrated — through the specified export connector or topic — to the target location. The combination of TTL and data migration creates an automated archiving process, where aged data is moved to another repository or stream while VoltDB continues to operate on current data. VoltDB does not delete the records until *after* the target system acknowledges their receipt; that is, the export connector successfully sends the data to the remote target or the data is inserted into the specified output topic. In this way you are assured that the data is always present in at least one of the participating systems.

For example, the following table definition establishes an automatic archiving policy that removes sessions with no activity for an hour, migrating old records to a historical repository:

```
CREATE TABLE sessions
    MIGRATE TO TARGET oldsessions
 (
   login TIMESTAMP DEFAULT NOW,
   last_update TIMESTAMP NOT NULL,
   user_id BIGINT NOT NULL
 ) USING TTL 1 HOURS ON COLUMN last_update;
```

It is also possible to migrate data manually. If you add the MIGRATE TO TARGET or MIGRATE TO TOPIC clause by itself, without USING TTL, no data is automatically migrated. However, you can explicitly initiate migration by invoking the MIGRATE SQL statement with the WHERE clause to specify which rows are migrated. Use of MIGRATE TO TARGET or MIGRATE TO TOPIC without USING TTL is useful when the application logic to select what data to migrate requires multiple or non-numeric variables. For example, if the schedule for archiving a record varies based on which user created it:

```
CREATE TABLE messages
  MIGRATE TO TARGET oldmessages
 (
   posted TIMESTAMP DEFAULT NOW,
   message_text VARCHAR(128),
   user_id BIGINT NOT NULL,
   user_type VARCHAR(5) NOT NULL
 );
```

In this case, no data is migrated until you explicitly initiate migration with the MIGRATE statement:

```
MIGRATE FROM messages
  WHERE
    ( (posted < DATEADD(DAY,-3,NOW()) AND user_type='USER')
      OR  (posted < DATEADD(DAY,-14,NOW()) AND user_type='ADMIN')
    ) AND NOT MIGRATING;
```

You can also migrate data manually, even if the table declaration includes the USING TTL clause. In this case you can use MIGRATE to preemptively migrate data before the TTL column expires. For example, using the *sessions* table defined above, you might want to migrate all sessions for a user when their account is deleted:

```
MIGRATE FROM sessions WHERE user_id=? AND NOT MIGRATING;
```

Note that use of the MIGRATING function is not required to filter on rows that are not already migrating, because the MIGRATE statement will not initiate export if rows are already migrating. However, explicitly including AND NOT MIGRATING in your MIGRATE statement can improve performance.

The MIGRATING function is also useful so you can avoid accidentally modifying records that are already marked for deletion, especially since any changes to migrating records will cancel the delete operation but not the export. For example, if you want to update the last_update column of a user's records but only if they are not already being migrated, your UPDATE statement should include NOT MIGRATING:

```
UPDATE sessions SET last_update=NOW() WHERE user_id=? AND NOT MIGRATING;
```

Time to live and data migration are powerful concepts. However, there are some important details to consider when using these features:

- There must be a usable index on the TTL column for the table. VoltDB uses that index to optimize the evaluation of the TTL values. If not, the USING TTL clause is accepted, but *no automated deletion will* occur at runtime until a usable index is defined.
- The CREATE TABLE... USING TTL statement is not rejected if the index is missing. This way you can define the index in a subsequent DDL statement. However, a warning message is issued if the USING TTL clause has no supporting index available. A similar warning is issued if you delete the last usable index.
- When the table definition includes both USING TTL and MIGRATE TO TARGET or MIGRATE TO TOPIC, there must be an index including the TTL column for the USING TTL clause and a separate index including only the TTL column and a WHERE NOT MIGRATING clause. This index is required to effectively find and schedule the migration of expired records. For example, the sessions table in the previous example would require the following index. If the index is not present, records for the table will neither be deleted nor migrated and a warning will be logged on the server:

  ```
  CREATE INDEX sessions_migrate_index ON sessions
     (last_update) WHERE NOT MIGRATING;
  ```

- TTL clauses are most effective when used on partitioned tables. Defining TTL for a replicated table, especially a large replicated table, can have a significant impact on database performance because the TTL delete actions must be processed as multi-partition transactions.
- You can specify the frequency and maximum size of the TTL processing cycle.
  - The BATCH_SIZE argument specifies the maximum number of records that will be deleted during each processing cycle. Specify the batch size as a positive integer. The default is 1000 rows.
  - The MAX_FREQUENCY argument specifies how often the TTL clause is evaluated. You specify the frequency in terms of the maximum number of times it is processed per second. For example a MAX_FREQUENCY of 10 means that the table's TTL value is processed at most 10 times per second. Specify the frequency as a positive integer. The default frequency is once per second (1).

  Under extreme loads or sudden bursts of inserts, it is possible for TTL processing to fall behind. Or if the records are extremely large, attempting to delete too many records at one time can cause the TTL

process to exceed the temporary table limit. The BATCH_SIZE and MAX_FREQUENCY clauses let you customize the TTL processing per table to meet the specific requirements of your application. The TTL selector for the `@Statistics` system procedure can help you evaluate TTL performance against your application workload to determine what settings you need.

- Evaluation of the time to live is made against the current value of the TTL column, not its initial value. So if a subsequent transaction alters the column value (either increasing or decreasing it) that modification will impact the subsequent lifetime of the record.
- When using database replication (DR), it is possible for the TTL transaction to exceed the 50MB limit on the DR binary log. If this happens, a warning is issued and TTL processing is suspended.
- When using MIGRATION TO TARGET or MIGRATE TO TOPIC, there is an interval after the TTL value is triggered and before the record is successfully migrated and deleted from the VoltDB database. During this interval, the record is available for read access from SELECT queries. You can also update or delete the record; but modifying the record will cancel the pending delete. So if, for example, you update the record to extend the TTL column, the record will remain in the database until the new TTL column value is reached. However the update does *not* cancel the export of the original data to the specified target or topic that had already been triggered. So two records will eventually be migrated.
- In most cases, you can ignore whether a record is currently being migrated and scheduled for delete or not. For example, if you delete a record that is currently being migrated, you cancel the pending delete but you delete the record anyway, so the results end up the same. However, if you do want to distinguish between currently active and currently migrating records, you can use the MIGRATING function, that identifies records that are currently "in flight". For example, to select records for a specific user ID and only those records that are not being migrated, you can use the following query:

  ```
  SELECT user_id, login FROM sessions WHERE user_id = ? AND NOT MIGRATING;
  ```

Example

The following example defines a table with five columns. The first column, *Company*, is not allowed to be null, which is important since it is used as the partitioning column in the following PARTITION TABLE statement. That column is also contained in the PRIMARY KEY constraint. Again, it is important to include the partitioning column in any fully unique indexes for partitioned tables.

```
CREATE TABLE Inventory (
    Company VARCHAR(32) NOT NULL,
    ProductID BIGINT NOT NULL,
    Price DECIMAL,
    Category VARCHAR(32),
    Description VARCHAR(256),
    PRIMARY KEY (Company, ProductID)
);
PARTITION TABLE Inventory ON COLUMN Company;
```

CREATE TASK

CREATE TASK — Schedules a procedure to run periodically.

Syntax

```
CREATE TASK task-name
        ON SCHEDULE {CRON cron-definition | DELAY time-interval | EVERY time-interval |
        FROM CLASS class-path}
        PROCEDURE { procedure-name | FROM CLASS class-path } [WITH (argument [,...])]
        [ON ERROR {LOG | IGNORE | STOP} ]
        [RUN ON {DATABASE | HOSTS | PARTITIONS} ]
        [AS USER user-name]
        [ENABLE | DISABLE]

CREATE TASK task-name
        FROM CLASS class-path [WITH (argument[,...])]
        [ON ERROR {LOG | IGNORE | STOP} ]
        [RUN ON {DATABASE | HOSTS | PARTITIONS } ]
        [AS USER user-name]
        [ENABLE | DISABLE]

time-interval: integer {MILLISECONDS | SECONDS | MINUTES | HOURS | DAYS}
```

Description

The CREATE TASK statement schedules a stored procedure to run iteratively on a set schedule. In its simplest form, the CREATE TASK statement schedules a specified stored procedure to be run at a regular interval. The PROCEDURE clause specifies the stored procedure and any arguments it requires. The ON SCHEDULE clause specifies when the procedure will be run. You can schedule a procedure to run on three types of schedule:

- CRON — Specifies a cron-style schedule to run the procedure as set times per day or week.
- DELAY — Specifies a time interval between each run of the stored procedure, where the time interval starts at the end of each run.
- EVERY — Specifies a time interval between the start of each run of the stored procedure.

The difference between DELAY and EVERY is how the interval is measured. For example, if you specify EVERY 5 SECONDS, the stored procedure runs every 5 seconds, no matter how long it takes to execute (assuming it does not take more than 5 seconds). If, on the other hand, you specify DELAY 5 SECONDS, each run starts 5 seconds *after* the previous run completes. In other words, EVERY results in invocations at a regular interval no matter how long they take, while DELAY results in a regular interval between when one run ends and the next begins.

For DELAY and EVERY you specify the interval as a positive integer and a time unit, where the supported time units are milliseconds, seconds, minutes, hours, and days. For EVERY, if the previous run takes longer than the interval to run, the schedule is reset at the end of the previous run. So, for example, if the schedule specifies EVERY 2 SECONDS but the procedure takes 2.5 seconds to run, the next scheduled interval will already be past when the previous run ends. In this case, the next invocation of the task is reset to 2 seconds after the previous run ends.

The CRON option requires a standard cron schedule, which consists of six values separated by spaces. Cron schedules set specific times of day, week, or month, rather than an interval. The six values of the cron string represent seconds, minutes, hours, day of the month, month, and day of the week. Asterisks indicate all possible values. For example, the cron specification `ON SCHEDULE CRON 0 0 * * * *` schedules the task on the hour, every hour of every day. More information about scheduling tasks with cron can be found on the web.

You can also specify details about how the procedure is run:

- ON ERROR specifies how errors are handled. The default is ON ERROR STOP.
  - ON ERROR LOG — The error is logged but the procedure continues to be scheduled and run.
  - ON ERROR IGNORE — The procedure continues to be scheduled and run and the error is ignored and *not* logged.
  - ON ERROR STOP — The error is logged and the scheduling process stops. No further invocations of the procedure will occur until the task is explicitly re-enabled (by using ALTER TASK to disable and then enable the task) or the database restarts.
- RUN ON specifies where the procedure executes. The default is RUN ON DATABASE.
  - RUN ON DATABASE — For multi-partitioned procedures, each invocation of the procedure is run as a single transaction coordinated across all partitions.
  - RUN ON PARTITIONS — For directed procedures, the procedure is scheduled and run independently on all partitions in the database. Directed procedures are useful for performing distributed tasks that are transactional on each partition but do not need to be coordinated and therefore are less disruptive to the ongoing database workload.
- AS USER specifies the user account under which the procedure is run. When security is enabled, you must specify a valid username and that user must have sufficient privileges to run the procedure.

Finally, you can use the ENABLE and DISABLE keywords to specify whether the task is enabled or not. (The task is enabled by default.) If the task is disabled, the procedure is not invoked. If the task is enabled, the procedure is invoked according to the schedule until the database shuts down or the task is disabled by an ALTER TASK statement or an error while ON ERROR STOP is active.

Creating Custom Tasks

If the standard schedules do not meet your needs — you want to change the interval between runs, modify the arguments to the procedure , or the procedure itself — you can define a custom task using Java classes that implement one of three special interfaces:

- When you only want to dynamically control the schedule of the procedure but keep the procedure and its parameters the same, you can use the ON SCHEDULE FROM CLASS clause specifying a Java class that implements the `IntervalGenerator` interface.
- When you want to use a regular schedule but dynamically change the procedure and/or its parameters, you can use the PROCEDURE FROM CLASS clause specifying a Java class that implements the `ActionGenerator` interface.
- When you want to dynamically control both the schedule and the procedure being invoked, you can use the second form of the CREATE TASK syntax which replaces both the ON SCHEDULE and PROCEDURE clauses with a single FROM CLASS clause specifying a Java class that implements the `ActionScheduler` interface.

Before declaring a custom task, you must load the specified Java class, the same way you load Java classes before declaring a user-defined stored procedure, by packaging it in a JAR file and using the LOAD CLASSES directive in **sqlcmd**. It is also important to note that the classes used for custom tasks are *not* stored procedures and do not run in the normal transactional path for VoltDB transactions. The custom task classes run in a separate thread to identify the characteristics of the next task invocation *before* the specified stored procedure is run. For all three task interfaces, the task management infrastructure provides the results from the previous run as input to the callback method, which can then use that information to determine how to modify the next instantiation of the task's procedure, parameters, or run interval.

Many of the CREATE TASK statement's clauses — ON ERROR, AS USER, and ENABLE|DISABLE — operate exactly the same for both custom tasks and the simple case of scheduling a single stored procedure. The two exceptions are the WITH and RUN ON clauses.

For custom tasks that alter the procedure and procedure parameters, the arguments in the WITH clause are passed to the custom task's `initialize()` method rather than to the stored procedure that it runs. The custom task can then decide what to do with those arguments. For example, it may use them as initial, maximum, and minimum values for adjusting arguments to the stored procedure.

The RUN ON clause for a custom task has one additional option beyond just DATABASE and PARTITIONS. Custom tasks can also be RUN ON HOSTS, which means one instance of the task is run on each server in the cluster.

Examples

The following example declares a procedure to reset the DailyStats view, and a task scheduled as a cron event at midnight every night to run the procedure.

```
CREATE PROCEDURE ResetDailyStats AS
   DELETE FROM DailyStats;
CREATE TASK nightly
   ON SCHEDULE CRON 0 0 0 * * *
   PROCEDURE ResetDailyStats
   RUN ON DATABASE;
```

The next example creates a custom task that dynamically changes the interval between invocations of the stored procedure. The example first loads the JAR file containing a custom task class that implements the IntervalGenerator interface and then declares the task using PROCEDURE FROM CLASS clause.

```
sqlcmd
1> LOAD CLASSES mytasks.jar;
2> CREATE TASK DailyNoHolidays
   ON SCHEDULE FROM CLASS mytasks.NoHolidays
   PROCEDURE ResetDailyStats
   RUN ON DATABASE;
```

CREATE VIEW

CREATE VIEW — Creates a view into one or more tables, optimizing access to a summary of their contents.

Syntax

```
CREATE VIEW view-name ( view-column-name [,...] )
        AS SELECT { column-name | selection-expression } [AS alias] [,...]
        FROM table-reference [join-clause...]
        [where-clause] [group-clause]

CREATE VIEW view-name
        [migrate-target-definition | migrate-topic-definition]
        ( view-column-name [,...] )
        AS SELECT { column-name | selection-expression } [AS alias] [,...]
        FROM stream-reference
        [where-clause] [group-clause] [ttl-definition]

table-reference: { table-name [AS alias] }

stream-reference: { view-name [AS alias] }

join-clause: , table-reference [INNER] JOIN [{table-reference}] [join-condition]

join-condition: ON conditional-expression USING (column-reference [,...])

where-clause WHERE [NOT] boolean-expression
        [ {AND | OR} [NOT] boolean-expression]...

group-clause GROUP BY { column-name | selection-expression } [,...]

migrate-target-definition MIGRATE TO TARGET {target-name}

migrate-topic-definition MIGRATE TO TOPIC {topic-name}
        [WITH [KEY (column-name[,...])] [VALUE (column-name[,...])]]

ttl-definition: USING TTL value [time-unit] ON COLUMN column-name
        [BATCH_SIZE number-of-rows] [MAX_FREQUENCY value]

time-unit: SECONDS | MINUTES | HOURS | DAYS
```

Description

The CREATE VIEW statement creates a view of a table, a stream, or joined tables with selected columns and aggregates. VoltDB implements views as materialized views. In other words, the view is stored as a special table in the database and is updated each time the corresponding database contents are modified. This means there is a small, incremental performance impact for any inserts or updates to the tables, but selects on the view will execute efficiently.

The following limitations are important to note when using the CREATE VIEW statement with VoltDB:

- If the SELECT statement contains a GROUP BY clause, all of the columns and expressions listed in the GROUP BY must be listed in the same order at the start of the SELECT statement. Aggregate functions, including COUNT(*), are allowed following the GROUP BY columns.
- Views are allowed on individual tables or streams, or joins of multiple tables. Joining streams is not supported.
- Joins must be inner joins and cannot be self-joins. All other limitations for joins as described in the SELECT statement also apply to joins in views.
- Views that join multiple tables must include a COUNT(*) field listed after all GROUP BY columns.
- To avoid performance problems when inserting data into a view that joins multiple tables, it is strongly recommended you define indexes on the table columns involved in the join.

Managing Stream Views

You can create views on both tables and streams. Views on tables are *materialized*, which means the content of the view is tied to the current contents of the table. You cannot modify the records of the view except by changing the contents of the table.

Views on streams, however, are historical. As data is inserted into the stream, the view contents are incremented. However, streams are ephemeral and have no storage associated with them, so there is no way update or delete the view records from the stream itself. Instead, you are allowed to use the DELETE and UPDATE statements to manage the contents of stream views.

For example, you might want to track user sessions, keeping a log of daily logins and time online per user. You do not need to keep a record of every session, so you can use a stream to capture each event and a view on the stream to track the daily totals.

```
CREATE STREAM user_session
  PARTITION ON COLUMN userid (
    userid INTEGER NOT NULL,
    start_time TIMESTAMP,
    session_length BIGINT
);
CREATE VIEW session_view
  (userid, session_day, total_count, total_time)
  AS SELECT userid, TRUNCATE(DAY, start_time),
            count(*), SUM(session_length)
     FROM user_session GROUP BY userid, TRUNCATE(DAY, start_time);
```

But you do not need to keep these records forever, so you might choose to delete views for any days more than a week old. You could do this with the following DELETE statement, deleting records from the stream view:

```
DELETE FROM session_view WHERE session_day < DATEADD(DAY, -7, NOW());
```

Similarly, if you knew there was an invalid session logged earlier, you might update the view to remove the incorrect data from the view using an UPDATE statement:

```
UPDATE session_view
  SET total_count = total_count-1,
      total_time = total_time-72
  WHERE userid = 1234
        AND session_day = TRUNCATE(DAY,NOW());
```

Automating Stream View Management

As mentioned in the preceding section, you can "clean up" stream views by using the DELETE statement. However, doing this manually is usually impractical. So, VoltDB lets you use the same time-to-live (TTL) and migration capabilities available to tables for stream views.

When you create the stream view you can define how long to keep the view data by adding the USING TTL clause to specify when records should be deleted based on a timestamp column in the view. If the timestamp is older than the specified value, the record will be deleted. For example, the following view definition automates the delete operation shown earlier by specifying that records should be deleted after the value of the *session_day* column is older than seven days.

```
CREATE VIEW session_view
  (userid, session_day, total_count, total_time)
  AS SELECT userid, TRUNCATE(DAY, start_time),
            count(*), SUM(session_length)
     FROM user_session GROUP BY userid, TRUNCATE(DAY, start_time)
  USING TTL 7 DAYS ON COLUMN session_day;
```

If you want to delete the records from the stream view but not lose them, you can use the MIGRATE clause to specify where to send the records, then use the MIGRATE rather than DELETE statement. Stream view records can be migrated to either export connectors or to topics, depending on whether you specify MIGRATE TO TARGET or MIGRATE TO TOPIC. The key difference between the two is that MIGRATE TO TARGET ensures that the record reaches the external export target system and receipt is acknowledged before it is deleted. Whereas MIGRATE TO TOPIC deletes the record as soon as it has been inserted into the VoltDB topic queue and made available to external consumers.

For example, the following CREATE VIEW statement specifies that migrated records are sent to the topic *OldSessions*. When migrating to a topic, you can also specify which columns compose the topic key and which columns to include in the message itself.

```
CREATE VIEW session_view
  MIGRATE TO TOPIC OldSessions
    WITH KEY(userid) VALUES(session_day,total_count,total_time)
  (userid, session_day, total_count, total_time)
  AS SELECT userid, TRUNCATE(DAY, start_time),
            count(*), SUM(session_length)
     FROM user_session GROUP BY userid, TRUNCATE(DAY, start_time);
```

Finally, you can combine USING TTL and MIGRATE TO to automate the removal and transfer of old records. Whenever the TTL value is triggered, the records are sent to the specified target or topic and then deleted from the view.

```
CREATE VIEW session_view
  MIGRATE TO TOPIC OldSessions
    WITH KEY(userid) VALUES(session_day,total_count,total_time)
  (userid, session_day, total_count, total_time)
  AS SELECT userid, TRUNCATE(DAY, start_time),
            count(*), SUM(session_length)
     FROM user_session GROUP BY userid, TRUNCATE(DAY, start_time)
  USING TTL 7 DAYS ON COLUMN session_day;
```

It is important when evaluating view contents for TTL to be aware of the possibilities of additional stream inserts arriving "late". That is, the timestamp value being used for the TTL evaluation may not be synchronous to the system clock. So if you attempt to delete view records immediately after they expire (in

the preceding example, when the session start time is only one day old), late arriving records will create a new record for that older time window. It is always a good idea to allow a certain leeway before deleting or migrating records to make sure any potentially late arrivals are included before the record is archived.

Note that you *cannot* define TTL or migration on table views. However, a similar effect can be achieved by defining the TTL and migration constraints on the table itself.

Examples

The following example defines a view that counts the number of records for a specific product item grouped by its location (that is, the warehouse the item is in).

```
CREATE VIEW inventory_count_by_warehouse (
     productID,
     warehouse,
     total_inventory
) AS SELECT
     productID,
     warehouse,
     COUNT(*)
FROM inventory GROUP BY productID, warehouse;
```

The next example uses a WHERE clause but no GROUP BY to provide a count and minimum and maximum aggregates of all records that meet a certain criteria.

```
CREATE VIEW small_towns ( number, minimum, maximum )
    AS SELECT count(*), min(population), max(population)
    FROM TOWNS WHERE population < 10000;
```

The final example demonstrates joining two tables in a view. This definition provides a similar view to the first example, except it uses the productID column to join two tables, Product and Inventory:

```
CREATE VIEW inventory_count_by_warehouse (
     productName,
     warehouse,
     total_inventory
) AS SELECT
     product.productName,
     inventory.warehouse,
     COUNT(*)
FROM product JOIN inventory
     ON product.productID = inventory.productID
     GROUP BY product.productName, inventory.warehouse;
```

DR TABLE

DR TABLE — Identifies a table as a participant in database replication (DR)

Syntax

```
DR TABLE table-name [DISABLE]
```

Description

The DR TABLE statement identifies a table as a participant in database replication (DR). If DR is not enabled, the DR TABLE statement has no effect on the operation of the table or the database as a whole. However, once DR is enabled, the results of any updates to the contents of DR tables are sent to the associated XDCR clusters.

The DR TABLE ... DISABLE statement reverses the effect of a previous DR TABLE statement, removing the specified table from participation in DR. The databases must be paused when you issue the DR TABLE... DISABLE command and you must apply it to all participating clusters before resuming operation. The DR TABLE command *cannot* be used with dynamic schema change.

See Chapter 11, *Active(N) Database Replication* for more information about how database replication works and Section 11.3.1, “Safely Updating the Schema While the Clusters are Paused” for using DR TABLE on a running database.

Examples

The following example identifies the tables *Employee* and *Department* as participants in database replication.

```
DR TABLE Employee;
DR TABLE Department;
```

DROP FUNCTION

DROP FUNCTION — Removes the definition of a SQL function.

Syntax

```
DROP FUNCTION function-name [IF EXISTS]
```

Description

The DROP FUNCTION statement deletes the definition of the specified user-defined function. Note that, for functions declared using CREATE FUNCTION and a class file, the statement does *not* delete the class that implements the function, it only deletes the definition. To remove the Java class that contains the associated function method, you must first drop the function definition then use the sqlcmd **remove classes** directive to remove the class.

The IF EXISTS clause allows the statement to succeed even if the specified function name does not exist. If the function does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Examples

The following example removes the definitions of the HTML_ENCODE and HTML_DECODE functions, then uses **remove classes** to remove the class containing their corresponding methods.

```
$ sqlcmd
1> DROP FUNCTION html_encode;
1> DROP FUNCTION html_decode;
2> remove classes "*.HtmlFunctions";
```

DROP INDEX

DROP INDEX — Removes an index.

Syntax

```
DROP INDEX index-name [IF EXISTS]
```

Description

The DROP INDEX statement deletes the specified index, and any data associated with it, from the database. The IF EXISTS clause allows the statement to succeed even if the specified index does not exist. If the index does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

You must use the name of the index as specified in the original DDL when dropping the index. You cannot drop an index if it was not explicitly named in the CREATE INDEX command. This is why you should always name indexes and other constraints wherever possible.

Examples

The following example removes the index named employee_idx_by_lastname:

```
DROP INDEX Employee_idx_by_lastname;
```

DROP PROCEDURE

DROP PROCEDURE — Removes the definition of a stored procedure.

Syntax

```
DROP PROCEDURE procedure-name [IF EXISTS]
```

Description

The DROP PROCEDURE statement deletes the definition of the named stored procedure. Note that, for procedures declared using CREATE PROCEDURE FROM and a class file, the statement does *not* delete the class that implements the procedure, it only deletes the definition and any partitioning information associated with the procedure. To remove the associated stored procedure class, you must first drop the procedure definition then use the sqlcmd **remove classes** directive to remove the class.

The IF EXISTS clause allows the statement to succeed even if the specified procedure name does not exist. If the stored procedure does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Examples

The following example removes the definition of the FindCanceledReservations stored procedure, then uses **remove classes** to remove the corresponding class.

```
$ sqlcmd
1> DROP PROCEDURE FindCanceledReservations;
2> remove classes "*.FindCanceledReservations";
```

DROP ROLE

DROP ROLE — Removes a role.

Syntax

```
DROP ROLE role-name [IF EXISTS]
```

Description

The DROP ROLE statement deletes the specified role. The IF EXISTS clause allows the statement to succeed even if the specified role does not exist. If the role does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Examples

The following example removes the role named debug:

```
DROP ROLE debug;
```

DROP STREAM

DROP STREAM — Removes a stream and, optionally, any views associated with it.

Syntax

```
DROP STREAM stream-name [IF EXISTS] [CASCADE]
```

Description

The DROP STREAM statement deletes the specified stream from the database. The IF EXISTS clause allows the statement to succeed even if the specified stream does not exist. If the stream does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

If you use the CASCADE clause, VoltDB automatically drops any referencing views as well as the stream itself.

If the stream is associated with an export target (that is, the stream was created with the EXPORT TO TARGET clause), dropping the stream also deletes any pending records that were inserted into the stream but have not been committed to the export target yet. If you want to change the stream definition without losing any pending export data, use the ALTER STREAM statement. If you want to remove the stream but ensure all export data is flushed before it is dropped, you can either use the **voltadmin pause --wait** command (to flush all queues) or the @Statistics system procedure with the EXPORT selector to check that the specified target has no pending records.

Example

The following example uses DROP STREAM with the IF EXISTS clause to remove the MeterReadings stream definition.

```
DROP STREAM MeterReadings IF EXISTS;
```

DROP TABLE

DROP TABLE — Removes a table and any data associated with it.

Syntax

```
DROP TABLE table-name [IF EXISTS] [CASCADE]
```

Description

The DROP TABLE statement deletes the specified table, and any data associated with it, from the database. The IF EXISTS clause allows the statement to succeed even if the specified tables does not exist. If the table does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Before dropping a table, you must first remove any stored procedures that reference the table. For example, if the table EMPLOYEE is partitioned and the stored procedure AddEmployee is partitioned on the EMPLOYEE table, you must drop the procedure first before dropping the table:

```
PARTITION TABLE Employee ON COLUMN EmpID;
CREATE PROCEDURE
   PARTITION ON TABLE Employee COLUMN EmpID
   FROM CLASS myapp.procedures.AddEmployee;

 [. . . ]

DROP PROCEDURE AddEmployee;
DROP TABLE Employee;
```

Attempting to drop the table before dropping the procedure will result in an error. The same will normally happen if there are any views or indexes that reference the table. However, if you use the CASCADE clause VoltDB will automatically drop any referencing indexes and views as well as the table itself.

Examples

The following example uses DROP TABLE with the IF EXISTS clause to remove any existing MailAddress table definition and data before adding a new definition.

```
DROP TABLE UserSignin IF EXISTS;
CREATE TABLE UserSignin (
   userID BIGINT NOT NULL,
   lastlogin TIMESTAMP DEFAULT NOW
);
```

DROP TASK

DROP TASK — Removes a task and cancels any future execution.

Syntax

```
DROP TASK task-name [IF EXISTS]
```

Description

The DROP TASK statement deletes the specified task and cancels any future execution. The IF EXISTS clause allows the statement to succeed even if the specified task does not exist. If the task does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Examples

The following example removes the task named *cleanup*:

```
DROP TASK cleanup;
```

DROP VIEW

DROP VIEW — Removes a view and any data associated with it.

Syntax

```
DROP VIEW view-name [IF EXISTS]
```

Description

The DROP VIEW statement deletes the specified view, and any data associated with it, from the database. The IF EXISTS clause allows the statement to succeed even if the specified view does not exist. If the view does not exist and you *do not* include the IF EXISTS clause, the statement will return an error.

Dropping a view has the same constraints as dropping a table, in that you cannot drop a view that is referenced by existing stored procedure queries. Before dropping the view, you must drop any stored procedures that reference it.

Examples

The following example removes the view named Votes_by_state:

```
DROP VIEW votes_by_state;
```

PARTITION TABLE

PARTITION TABLE — Specifies that a table is partitioned and which is the partitioning column.

Syntax

```
PARTITION TABLE table-name ON COLUMN column-name
```

Description

Partitioning a table specifies that different records are stored in different unique partitions, based on the value of the specified column. The table *table-name* and column *column-name* must be valid, declared elements in the current DDL file or VoltDB generates an error when compiling the schema.

For a table to be partitioned, the partitioning column must be declared as NOT NULL. If you do not declare a partitioning column of a table in the DDL, the table is assumed to be a replicated table.

Example

The following example partitions the table *Employee* on the column *EmployeeID*.

```
PARTITION TABLE Employee on COLUMN EmployeeID;
```

Appendix B. Supported SQL Statements

This appendix describes the SQL syntax that VoltDB supports in stored procedures and ad hoc queries.

This is not intended as a complete description of the SQL language and how it operates. Instead, it summarizes the subset of standard SQL statements that are allowed in VoltDB and any exceptions or limitations that application developers should be aware of.

The supported SQL statements are:

- DELETE
- INSERT
- MIGRATE
- SELECT
- TRUNCATE TABLE
- UPDATE
- UPSERT

DELETE

DELETE — Deletes one or more records from the database.

Syntax

```
DELETE FROM table-name
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
        [ORDER BY {column-name [ ASC | DESC ]}[,...] [LIMIT integer] [OFFSET integer]]
```

Description

The DELETE statement deletes rows from the specified table that meet the constraints of the WHERE clause. The following limitations are important to note when using the DELETE statement in VoltDB:

- The DELETE statement can operate on only one table at a time. It does not support joins. However, it does support subqueries in the WHERE expression.
- The WHERE expression supports the boolean operators: equals (=), not equals (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), IS NULL, AND, OR, and NOT. Note, however, although OR is supported syntactically, VoltDB does not optimize these operations and use of OR may impact the performance of your queries.
- You can use subqueries in the WHERE clause of the DELETE statement, with the following provisions:
  - See the description of subqueries in the SELECT statement for general rules concerning the construction of subqueries.
  - In a multi-partition procedure, subqueries of the DELETE statement can only reference replicated tables.
  - In single-partitioned procedures, the subquery can reference both partitioned and replicated tables.
  - For ad hoc DELETE statements, the same rules apply except the SQL statement itself determines whether VoltDB executes it as a single-partitoned or multi-partitioned procedure. Statements that delete rows from a partitioned table based on a specific value of the partitioning column are executed as single-partitioned procedures. All other statements are multi-partitioned.
- The ORDER BY clause lets you order the selection results and then select a subset of the ordered records to delete. For example, you could delete only the five oldest records, chronologically, sorting by timestamp:

  ```
  DELETE FROM events ORDER BY event_time, event_id ASC LIMIT 5;
  ```

  Similarly, you could choose to keep only the five most recent:

  ```
  DELETE FROM events ORDER BY event_time, event_id DESC OFFSET 5;
  ```

- You cannot use ORDER BY to delete rows from a partitioned table in a multi-partitioned query. In other words, for partitioned tables DELETE... ORDER BY must be executed as part of a single-partitioned stored procedure or as an ad hoc query with a WHERE clause that uniquely identifies the partitioning column value.

Examples

The following example removes rows from the EMPLOYEE table where the EMPLOYEE_ID column is equal to 145303.

```
DELETE FROM employee WHERE employee_id = 145303;
```

The following example removes rows from the BID table where the BIDDERID is 12345 and the BIDPRICE is less than 100.00.

```
DELETE FROM bid WHERE bidderid=12345 AND bidprice<100.0;
```

INSERT

INSERT — Creates new rows in the database, using the specified values for the columns.

Syntax

INSERT INTO *table-name* [(*column-name* [,...])] VALUES (*value-expression* [,...])

INSERT INTO *table-name* [(*column-name* [,...])] SELECT *select-expression*

Description

The INSERT statement creates one or more new rows in the database. There are two forms of the INSERT statement, INSERT INTO... VALUES and INSERT INTO... SELECT. The INSERT INTO... VALUES statement lets you enter specific values for a adding a single row to the database. The INSERT INTO... SELECT statement lets you insert multiple rows into the database, depending upon the number of rows returned by the select expression.

The INSERT INTO... SELECT statement is often used for copying rows from one table to another. For example, say you want to export all of the records associated with a particular column value. The following INSERT statement copies all of the records from the table ORDERS with a warehouseID of 25 into the table EXPORT_ORDERS:

```
INSERT INTO Export_Orders SELECT * FROM Orders WHERE CustomerID=25;
```

However, the select expression can be more complex, including joining multiple tables. The following limitations currently apply to the INSERT INTO... SELECT statement:

- INSERT INTO... SELECT can join partitioned tables only if they are joined on equality of the partitioning columns. Also, the resulting INSERT must apply to a partitioned table and be inserted using the same partition column value, whether the query is executed in a single-partitioned or multi-partitioned stored procedure.
- INSERT INTO... SELECT does not support UNION statements.

In addition to the preceding limitations, there are certain instances where the select expression is too complex to be processed. Cases of invalid select expressions in INSERT INTO... SELECT include:

- A LIMIT or TOP clause applied to a partitioned table in a multi-partitioned query
- A GROUP BY of a partitioned table where the partitioning column is not in the GROUP BY clause

If you specify the column names following the table name, the values will be assigned to the columns in the order specified. If you do not specify the column names, values will be assigned to columns based on the order specified in the schema definition. However, if you specify a subset of the columns, you must specify values for any columns that are explicitly defined in the schema as NOT NULL and do not have a default value assigned.

You can use subqueries within the VALUES clause of the INSERT statement, with the following provisions:

- See the description of subqueries in the SELECT statement for general rules concerning the construction of subqueries.

- In a multi-partition procedure, subqueries of the INSERT statement can only reference replicated tables.
- In single-partitioned procedures, the subquery can reference both partitioned and replicated tables.
- For ad hoc INSERT statements, the same rules apply except the SQL statement itself determines whether VoltDB executes it as a single-partitoned or multi-partitioned procedure. Statements that insert rows into a partitioned table based on a specific value of the partitioning column are executed as single-partitioned procedures. All other statements are multi-partitioned.

Examples

The following example inserts values into the columns (firstname, mi, lastname, and emp_id) of an EMPLOYEE table:

```
INSERT INTO employee VALUES ('Jane', 'Q', 'Public', 145303);
```

The next example performs the same operation with the same results, except this INSERT statement explicitly identifies the column names and changes the order:

```
INSERT INTO employee (emp_id, lastname, firstname, mi)
       VALUES (145303, 'Public', 'Jane', 'Q');
```

The last example assigns values for the employee ID and the first and last names, but not the middle initial. This query will only succeed if the MI column is nullable or has a default value defined in the database schema.

```
INSERT INTO employee (emp_id, lastname, firstname)
       VALUES (145304, 'Doe', 'John');
```

MIGRATE

MIGRATE — queues table rows for migration to an export target.

Syntax

```
MIGRATE FROM table-name
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
```

Description

The MIGRATE statement selects rows from the specified table for migration to an export target and marks the rows for deletion. When rows are migrated, they are first exported to the export target defined in the table definition (in the MIGRATE TO TARGET clause). Once the export target acknowledges receipt of the data, the rows are deleted from the VoltDB table.

For example, assume the *reservations* table contains information about airline reservations. Once the flight is over, you want to archive the reservation records. But you do not want them to be deleted until you are sure they reach the archive. To achieve this you can declare the table using the MIGRATE TO TARGET clause:

```
CREATE TABLE Reservation
  MIGRATE TO TARGET oldreserve
  ( Reserve_ID INT NOT NULL,
    Flight_ID INT NOT NULL,
    Customer_ID INT);
```

Then, when the flight is completed, you can migrate all associated reservations to the external system associated with the *oldreserve* target, ensuring they are not deleted from the VoltDB database until they reach the target.

```
MIGRATE FROM Reservation WHERE Reserve_ID= ?;
```

The MIGRATE statement applies to any tables declared with a MIGRATE TO TARGET clause. You can use MIGRATE to manually migrate rows from tables that do not have an automated "time to live" (USING TTL) value defined or you can use it to preemptively migrate rows in a table declared with USING TTL.

Example

The following example migrates user accounts if the account type is "trial" and the user hasn't logged in for two weeks.

```
MIGRATE FROM accounts
  WHERE acct_type="TRIAL" AND last_login < DATEADD(DAY,-14,NOW());
```

SELECT

SELECT — Fetches the specified rows and columns from the database.

Syntax

```
[common-table-expression] Select-statement [{set-operator} Select-statement ] ...

Select-statement:
        SELECT [ TOP integer-value ]
        { * | [ ALL | DISTINCT ] { column-name | selection-expression } [AS alias] [,...] }
        FROM { table-reference } [ join-clause ]...
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
        [clause...]

table-reference:
        { table-name [AS alias] | view-name [AS alias] | sub-query AS alias }

sub-query:
        (Select-statement)

join-clause:
        , table-reference
        [INNER | {LEFT | RIGHT | FULL } [OUTER]] JOIN [{table-reference}] [join-condition]

join-condition:
        ON conditional-expression
        USING (column-reference [,...])

clause:
        ORDER BY { column-name | alias } [ ASC | DESC ] [,...]
        GROUP BY { column-name | alias } [,...]
        HAVING boolean-expression
        LIMIT integer-value [OFFSET row-count]

set-operator:
        UNION [ALL]
        INTERSECT [ALL]
        EXCEPT

common-table-expression:
        WITH common-table-name [(column-name [,...])] AS ( Select-statement )
        WITH RECURSIVE common-table-name [(column-name [,...])] AS (
            Select-statement UNION ALL Select-statement
        )
```

Description

The SELECT statement retrieves the specified rows and columns from the database, filtered and sorted by any clauses that are included in the statement. In its simplest form, the SELECT statement retrieves the values associated with individual columns. However, the selection expression can be a function such as COUNT and SUM.

The following features and limitations are important to note when using the SELECT statement with VoltDB:

- See Appendix C, *SQL Functions* for a full list of the SQL functions that VoltDB supports.
- VoltDB supports the following operators in expressions: addition (+), subtraction (-), multiplication (*), division (*) and string concatenation (||).
- `TOP` *n* is a synonym for `LIMIT` *n*.
- The WHERE expression supports the boolean operators: equals (=), not equals (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), LIKE, STARTS WITH, IS NULL, IS DISTINCT, IS NOT DISTINCT, AND, OR, and NOT. Note, however, although OR is supported syntactically, VoltDB does not optimize these operations and use of OR may impact the performance of your queries.
- The boolean expression LIKE provides text pattern matching in a VARCHAR column. The syntax of the LIKE expression is `{string-expression} LIKE '{pattern}'` where the pattern can contain text and wildcards, including the underscore (_) for matching a single character and the percent sign (%) for matching zero or more characters. The string comparison is case sensitive.

  Where an index exists on the column being scanned and the pattern starts with a text prefix (rather than starting with a wildcard), VoltDB will attempt to use the index to maximize performance, For example, a query limiting the results to rows from the EMPLOYEE table where the primary index, the JOB_CODE column, begins with the characters "Temp" looks like this:

  ```
  SELECT * from EMPLOYEE where JOB_CODE like 'Temp%';
  ```

- The STARTS WITH clause is useful in stored procedures because it uses indexed scans where the LIKE clause cannot. The expression `STARTS WITH '{string-expression}'` is syntactically identical to `LIKE '{string-expression}%'` in that it matches any string value that starts with *string-expression*. The difference is that in a stored procedure, use of the STARTS WITH clause with a placeholder (that is, "START WITH ?") utilizes available indexes, whereas LIKE ? requires a sequential scan, since the compiler cannot tell if the replacement text ends in a percent sign or not and must plan for any possible string value. For example, if KEYWORD is the primary key for the ENTRY table, then VoltDB can use the primary key index to optimize the following stored procedure:

  ```
  CREATE PROCEDURE SimpleSearch AS
     SELECT keyword FROM entry WHERE keyword STARTS WITH ?;
  ```

- The boolean expression IN determines if a given value is found within a list of alternatives. For example, in the following code fragment the IN expression looks to see if a record is part of Hispaniola by evaluating whether the column COUNTRY is equal to either "Dominican Republic" or "Haiti":

  ```
  WHERE Country IN ('Dominican Republic', 'Haiti')
  ```

  Note that the list of alternatives must be enclosed in parentheses. The result of an IN expression is equivalent to a sequence of equality conditions separated by OR. So the preceding code fragment produces the same boolean result as:

  ```
  WHERE Country='Dominican Republic' OR Country='Haiti'
  ```

  The advantages are that the IN syntax provides more compact and readable code and can provide improved performance by using an index on the initial expression where available.

- The boolean expression BETWEEN determines if a value falls within a given range. The evaluation is inclusive of the end points. In this way BETWEEN is a convenient alias for two boolean expressions determining if a value is greater than or equal to (>=) the starting value and less than or equal to (<=) the end value. For example, the following two WHERE clauses are equivalent:

```
WHERE salary BETWEEN ? AND ?
WHERE salary >= ? AND salary <= ?
```

- The boolean expressions IS DISTINCT FROM and IS NOT DISTINCT FROM are similar to the equals ("=") and not equals ("<>") operators respectively, except when evaluating null operands. If either or both operands are null, the equals and not equals operators return a boolean null value, or *false*. IS DISTINCT FROM and IS NOT DISTINCT FROM consider null a valid operand. So if only one operand is null IS DISTINCT FROM returns *true* and IS NOT DISTINCT FROM returns *false*. If both operands are null IS DISTINCT FROM returns *false* and IS NOT DISTINCT FROM returns *true*.

- When using placeholders in SQL statements involving the IN list expression, you can either do replacement of individual values within the list or replace the list as a whole. For example, consider the following statements:

  ```
  SELECT * from EMPLOYEE where STATUS IN (?, ?,?);
  SELECT * from EMPLOYEE where STATUS IN ?;
  ```

  In the first statement, there are three parameters that replace individual values in the IN list, allowing you to specify exactly three selection values. In the second statement the placeholder replaces the entire list, including the parentheses. In this case the parameter to the procedure call must be an array and allows you to change not only the values of the alternatives but the number of criteria considered.

  The following Java code fragment demonstrates how these two queries can be used in a stored procedure, resulting in equivalent SQL statements being executed:

  ```
  String arg1 = "Salary";
  String arg2 = "Hourly";
  String arg3 = "Parttime";
  voltQueueSQL( query1, arg1, arg2, arg3);

  String listargs[] = new String[3];
  listargs[0] = arg1;
  listargs[1] = arg2;
  listargs[2] = arg3;
  voltQueueSQL( query2, (Object) listargs);
  ```

  Note that when passing arrays as parameters in Java, it is a good practice to explicitly cast them as an object to avoid the array being implicitly expanded into individual call parameters.

- VoltDB supports the use of CASE-WHEN-THEN-ELSE-END for conditional operations. For example, the following SELECT expression uses a CASE statement to return different values based on the contents of the price column:

  ```
  SELECT Prod_name,
      CASE WHEN price > 100.00
            THEN 'Expensive'
            ELSE 'Cheap'
      END
  FROM products ORDER BY Prod_name;
  ```

  For more complex conditional operations with multiple alternatives, use of the DECODE() function is recommended.

- VoltDB supports both inner and outer joins.

- The SELECT statement supports subqueries as a table reference in the FROM clause. Subqueries must be enclosed in parentheses and must be assigned a table alias.
- You can only join two or more partitioned tables if those tables are partitioned on the same value and joined on equality of the partitioning column. Joining two partitioned tables on non-partitioned columns or on a range of values is not supported. However, there are no limitations on joining to replicated tables.
- Extremely large result sets (greater than 100 megabytes in size) are not supported. If you execute a SELECT statement that generates a result set of more than 100 megabytes, VoltDB will return an error.

Window Functions

Window functions, which can appear in the selection list, allow you to perform more selective calculations on the statement results than you can do with plain aggregation functions such as COUNT() or SUM(). Window functions execute the specified operation on a subset of the total selection results, controlled by the PARTITION BY and ORDER BY clauses. The overall syntax for a window function is as follows:

function-name([*expression*])
OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY { *expression* [,...]}])

Where:

- The PARTITION BY[1] clause defines how the selection results are grouped.
- The ORDER BY clause defines the order in which the rows are evaluated within each group.

An example may help explain the behavior of the two clauses. Say you have a database table that lists the population of individual cities and includes columns for country and state. You can use the window function `COUNT(city) OVER (PARTITION BY state)` to include a count of all of the cities within each state as part of each city record. You can also control the order the records are evaluated using the ORDER BY clause. Note, however, when you use the ORDER BY clause the window function results are calculated sequentially. So rather than show the count of all cities in the state each time, the window function will return the count of cities incrementally up to the current record in the group. So rather than use COUNT() you can use RANK() to more accurately indicate the values being returned. For example, `RANK() OVER (PARTITION BY state, ORDER BY city_population)` lists the cities for each state with a rank value showing their ranking in order of their population.

Please be aware of the following limitations when using the window functions:

- There can be only one window function per SELECT statement.
- You cannot use a window function and GROUP BY in the same SELECT statement.
- The argument(s) to the ORDER BY clause can be either integer or TIMESTAMP expressions only.

The following list describes the operation and constraints for each window function separately.

RANK() OVER ([PARTITION BY {*expression* [,...]}] ORDER BY {*expression* [,...]})

The RANK() window function generates a BIGINT value (starting at 1) representing the ranking of the current result within the group defined by the PARTITION BY expression(s) or of the entire result

[1]Use of the keyword PARTITION is for compatibility with SQL syntax from other databases and is unrelated to the columns used to partition single-partitioned tables. You can use the RANK() functions with either partitioned or replicated tables and the ranking column does *not* need to be the same as the partitioning column for VoltDB partitioned tables.

set if PARTITION BY is not specified. No function argument is allowed and the ORDER BY clause is required.

For example, if you rank a column (say, *city_population*) and use the *country* column as the partitioning column for the ranking, the cities of each country will be ranked separately. If you use both *state* and *country* as partitioning columns, then the cities for each state in each country will be ranked separately.

DENSE_RANK() OVER ([PARTITION BY {*expression* [,...]}] ORDER BY {*expression* [,...]})

The DENSE_RANK() window function generates a BIGINT value (starting at 1) representing the ranking of the current result, in the same way the RANK() window function does. The difference between RANK() and DENSE_RANK() is how they handle ranking when there is more than one row with the same ORDER BY value.

If more than one row has the same ORDER BY value, those rows receive the same rank value in both cases. However, with the RANK() function, the next rank value is incremented by the number of preceding rows. For example, if the ORDER BY values of four rows are 100, 98, 98, and 73 the respective rank values using RANK() will be 1, 2, 2, and 4. Whereas, with the DENSE_RANK() function, the next rank value is always only incremented by one. So, if the ORDER BY values are 100, 98, 98, and 73, the respective rank values using DENSE_RANK() will be 1, 2, 2, and 3.

As with the RANK() window function, no function argument is allowed for the DENSE_RANK() function and the ORDER BY clause is required.

ROW_NUMBER() OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY {*expression* [,...]}])

The ROW_NUMBER() window function generates a BIGINT value representing the ordinal order of the current result within the group defined by the PARTITION BY expression(s) or of the entire result set if PARTITION BY is not specified. No function argument is allowed.

For example, if you order a column (say, *animal*) and use the *class* column as the partitioning column, the animals in each class will be ordered separately. So "angelfish" might receive row number 1 in the type "finned fish" while "aardvark" is row number 1 in the type "mammal". But if you do not specify PARTITION BY, "angelfish" would be numbered after "aardvark".

COUNT(*{expression}*) OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY {*expression* [,...]}])

The COUNT() window function generates a sub-count of the number of rows within the current result set, where the PARTITION BY clause defines how the rows are grouped. The function argument is required.

LAG(*{expression}* [, *{offset}* [, *{default-value}*]]) OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY {*expression* [,...]}])

The LAG() window function takes the result set generated by the ORDER BY and PARTITION BY clauses and returns the column value of a preceding row, as specified by the *offset*. The *offset* can be any integer value between 0 and 16384, with the default being 1. If the offset results in a value outside the bounds of the window, the result is NULL, or a default value specified in the third parameter to the function.

MAX(*{expression}*) OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY {*expression* [,...]}])

The MAX() window function reports the maximum value of a column within the current result set, where the PARTITION BY clause defines how the rows are grouped. If the ORDER BY clause is

specified, the maximum value is calculated incrementally over the rows in the order specified. The function argument is required.

MIN(*{expression}*) OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY {*expression* [,...]}])

The MIN() window function reports the minimum value of a column within the current result set, where the PARTITION BY clause defines how the rows are grouped. If the ORDER BY clause is specified, the minimum value is calculated incrementally over the rows in the order specified. The function argument is required.

SUM(*{expression}*) OVER ([PARTITION BY {*expression* [,...]}] [ORDER BY {*expression* [,...]}])

The SUM() window function generates a sub-total of the specified column within the current result set, where the PARTITION BY clause defines how the rows are grouped. The function argument is required.

Subqueries

The SELECT statement can include subqueries. Subqueries are separate SELECT statements, enclosed in parentheses, where the results of the subquery are used as values, expressions, or arguments within the surrounding SELECT statement.

Subqueries, like any SELECT statement, are extremely flexible and can return a wide array of information. A subquery might return:

- A single row with a single column — this is sometimes known as a *scalar subquery* and represents a single value
- A single row with multiple columns — this is also known as a *row value expression*
- Multiple rows with one or more columns

In general, VoltDB supports subqueries in the FROM clause, in the selection expression, and in boolean expressions in the WHERE clause or in CASE-WHEN-THEN-ELSE-END operations. However, different types of subqueries are allowed in different situations, depending on the type of data returned.

- In the FROM clause, the SELECT statement supports all types of subquery as a table reference. The subquery must be enclosed in parentheses and must be assigned a table alias.
- In the selection expression, scalar subqueries can be used in place of a single column reference.
- In the WHERE clause and CASE operations, both scalar and non-scalar subqueries can be used as part of boolean expressions. Scalar subqueries can be used in place of any single-valued expression. Non-scalar subqueries can be used in the following situations:
  - **Row value comparisons** — Boolean expressions that compare one row value expression to another can use subqueries that resolve to one row with multiple columns. For example:

    ```
    select * from t1
       where (a,c) > (select a, c from t2 where b=t1.b);
    ```

  - **IN and EXISTS** — Subqueries that return multiple rows can be used as an argument to the IN or EXISTS predicate to determine if a value (or set of values) exists within the rows returned by the subquery. For example:

    ```
    select * from t1
    ```

```
   where a in (select a from t2);
select * from t1
   where (a,c) in (select a, c from t2 where b=t1.b);
select * from t1 where c > 3 and
   exists (select a, b from t2 where a=t1.a);
```

- **ANY and ALL** — Multi-row subqueries can also be used as the target of an ANY or ALL comparison, using either a scalar or row expression comparison. For example:

```
select * from t1
   where a > ALL (select a from t2);
select * from t1
   where (a,c) = ANY (select a, c from t2 where b=t1.b);
```

Note that VoltDB does not support subqueries in the HAVING, ORDER BY, or GROUP BY clauses. Subqueries are also not supported for any of the data manipulation language (DML) statements: DELETE, INSERT, UPDATE, and UPSERT.

For the initial release of subqueries in selection and boolean expressions, only replicated tables can be used in the subquery. Both replicated and partitioned tables can be used in subqueries in place of table references in the FROM clause.

Finally, the VoltDB planner does not optimize subqueries as thoroughly as simple SELECT queries. As a result, the intended index might not be applied to a subquery, resulting in table scans that can dramatically impact performance against large tables. It is important to analyze the execution plan for your subqueries to ensure the indexes are being utilized effectively. (See the chapter on "Understanding Execution Plans" in the *Guide to Performance and Customization* for instructions on evaluating execution plans.) If the correct index is not being applied, the recommendation is to replace the subquery with the corresponding basic SELECT statements to maintain application performance.

Set Operations

VoltDB also supports the set operations UNION, INTERSECT, and EXCEPT. These keywords let you perform set operations on two or more SELECT statements. UNION includes the combined results sets from the two SELECT statements, INTERSECT includes only those rows that appear in both SELECT statement result sets, and EXCEPT includes only those rows that appear in one result set but not the other.

Normally, UNION and INTERSECT provide a set including unique rows. That is, if a row appears in both SELECT results, it only appears once in the combined result set. However, if you include the ALL modifier, all matching rows are included. For example, UNION ALL will result in single entries for the rows that appear in only one of the SELECT results, but two copies of any rows that appear in both.

The UNION, INTERSECT, and EXCEPT operations obey the same rules that apply to joins:

- You cannot perform set operations on SELECT statements that reference the same table.
- All tables in the SELECT statements must either be replicated tables or partitioned tables partitioned on the same column value, using equality of the partitioning column in the WHERE clause.

Common Table Expressions

Common table expressions let you declare a named subquery that can be used in the main query the same way regular tables and columns are used. Common expressions are useful for simplifying queries that use an expression multiple times or for separating out two distinct aspects of a larger query. You declare a common table expression by placing the WITH clause before the main SELECT query. The WITH clause:

- Defines the name of the common table expression
- Optionally, renames the resulting columns
- Declares the expression itself using standard SELECT statement syntax

VoltDB supports two forms of common table expressions:

- Basic common expressions, with a name, optional column names, and the expression itself
- Recursive expressions, using the WITH RECURSIVE keywords and merging two expressions with a UNION ALL set operation

You can use the results of the common table expression in the subsequent SELECT statement the same way you would reference regular tables in the database. For example, the following common table expression determines how many members live in each city, then uses that information to return a list of members who live in a city with fewer than the specified number of members:

```
WITH city_count (city,membercount) AS (
  SELECT cityname, count(*) FROM members
    GROUP BY cityname
)
SELECT m.fullname, m.cityname FROM members AS m
  JOIN city_count AS cc ON m.city = cc.city
  WHERE membercount < ?
  ORDER BY m.cityname,m.fullname;
```

Recursive common expressions are like regular table expressions, except they are self-referencing, so you can iterate over the results in a recursive fashion. Recursive common expressions are particularly useful for evaluating tree or graph structures that cannot be natively represented in flat database records or queries.

You declare a recursive expression with the WITH RECURSIVE keywords followed by:

- The table name and, optionally, alias names for the columns
- A base query that defines the starting condition
- A UNION ALL set operator
- A second, recursive query that iterates over the common table expression results

For example, assume you wanted to know all the employees in a specific branch of the company's organizational structure. However, organizational charts are hierarchical. Each employee record may only record that employee's direct manager. Recursive common expressions let you start at the top of a branch of the organizational "tree" and iteratively look for any employee reporting to that manager, then employees reporting to that person, and so on. The common table expression might look like this.

```
WITH RECURSIVE org (id) AS (
   SELECT mgr_id AS mgr FROM department
     WHERE dept_name=?
   UNION ALL
     SELECT  emp_id FROM employee, org
         WHERE employee.mgr_id = org.id
)
SELECT e.emp_id, e.emp_name, e.emp_address
   FROM employee AS e, org
```

```
WHERE e.emp_id = org.id;
```

Warning

As with any recursive programming, you are responsible for ensuring the common table expression does not result in an infinite loop. VoltDB cannot determine at compile time whether the expression is sufficiently bounded. The preceding example succeeds because the application ensures all employee/manager relationships are hierarchical — no manager reports to a employee lower in the tree. If evaluation of a common table expression results in a loop, VoltDB will eventually exceed some limit (such as the query timeout or maximum temporary table space) and fail the transaction. In certain cases, an infinite loop could use up so much memory it exceeds the resource limit and pauses the database.

Common table expressions in VoltDB have the following limitations:

- There can be only one common table expression per query.
- In multi-partition transactions, the common expression can reference replicated tables only.
- In single-partition transactions, the common expression can reference both replicated and partitioned table, with the caveat that as in any partitioned transaction partitioned tables have access to only that data in the current partition.
- For basic (non-recursive) common table expressions, the common expression cannot be self-referencing. That is, the SELECT statement within the WITH clause can reference actual database table and view names only, it cannot reference the common expression name itself.

Examples

The following example retrieves all of the columns from the EMPLOYEE table where the last name is "Smith":

```
SELECT * FROM employee WHERE lastname = 'Smith';
```

The following example retrieves selected columns for two tables at once, joined by the employee_id using an implicit inner join and sorted by last name:

```
SELECT lastname, firstname, salary
    FROM employee AS e, compensation AS c
    WHERE e.employee_id = c.employee_id
    ORDER BY lastname DESC;
```

TRUNCATE TABLE

TRUNCATE TABLE — Deletes all records from the specified table.

Syntax

```
TRUNCATE TABLE table-name
```

Description

The TRUNCATE TABLE statement deletes all of the records from the specified table. TRUNCATE TABLE is the same as the statement `DELETE FROM {table-name}` with no selection clause. These statements contain optimizations to increase performance and reduce memory usage over an equivalent DELETE statement containing a WHERE selection clause.

The goal of the TRUNCATE TABLE statement is to remove all records from a table. Since this is not possible in a partitioned stored procedure, VoltDB does not allow TRUNCATE TABLE statements within partitioned stored procedures. You can perform TRUNCATE TABLE statements in ad hoc or multi-partition procedures only.

Examples

The following example removes all data from the CURRENT_STANDINGS table:

```
TRUNCATE TABLE Current_standings;
```

UPDATE

UPDATE — Updates the values within the specified columns and rows of the database.

Syntax

```
UPDATE table-name SET column-name = value-expression [, ...]
        [WHERE [NOT] boolean-expression [ {AND | OR} [NOT] boolean-expression]...]
```

Description

The UPDATE statement changes the values of columns within the specified records. The following limitations are important to note when using the UPDATE statement with VoltDB:

- VoltDB supports the following arithmetic operators in expressions: addition (+), subtraction (-), multiplication (*), and division (*).
- The WHERE expression supports the boolean operators: equals (=), not equals (!= or <>), greater than (>), less than (<), greater than or equal to (>=), less than or equal to (<=), IS NULL, AND, OR, and NOT. Note, however, although OR is supported syntactically, VoltDB does not optimize these operations and use of OR may impact the performance of your queries.
- You can use subqueries in place of value expressions within the SET and WHERE clauses of the UPDATE statement, with the following provisions:
  - See the description of subqueries in the SELECT statement for general rules concerning the construction of subqueries.
  - In a multi-partition procedure, subqueries of the UPDATE statement can only reference replicated tables.
  - In single-partitioned procedures, the subquery can reference both partitioned and replicated tables.
  - For ad hoc UPDATE statements, the same rules apply except the SQL statement itself determines whether VoltDB executes it as a single-partitoned or multi-partitioned procedure. Statements that modify a partitioned table based on a specific value of the partitioning column are executed as single-partitioned procedures. All other statements are multi-partitioned.

Examples

The following example changes the ADDRESS column of the EMPLOYEE record with an employee ID of 145303:

```
UPDATE employee
    SET address = '49 Lavender Sweep'
    WHERE employee_id = 145303;
```

The following example increases the starting price by 25% for all ITEM records with a category ID of 7:

```
UPDATE item SET startprice = startprice * 1.25 WHERE categoryid = 7;
```

UPSERT

UPSERT — Either inserts new rows or updates existing rows depending on the primary key value.

Syntax

UPSERT INTO *table-name* [(*column-name* [,...])] VALUES (*value-expression* [,...])

UPSERT INTO *table-name* [(*column-name* [,...])] SELECT *select-expression*

Description

The UPSERT statement has the same syntax as the INSERT statement and will perform the same function, assuming a record with a matching primary key does *not* already exist in the database. If such a record does exist, UPSERT updates the existing record with the new column values. Note that the UPSERT statement can only be executed on tables that have a primary key.

UPSERT has the same two forms as the INSERT statement: UPSERT INTO... VALUES and UPSERT INTO... SELECT. The UPSERT statement also has similar constraints and limitations as the INSERT statement with regards to joining partitioned tables and overly complex SELECT clauses. (See the description of the INSERT statement for details.)

However, UPSERT INTO... SELECT has an additional limitation: the SELECT statement must produce deterministically ordered results. That is, the query must not only produce the same rows, they must be in the same order to ensure the subsequent inserts and updates produce identical results.

You can use subqueries within the VALUES clause of the UPSERT statement, with the following provisions:

- See the description of subqueries in the SELECT statement for general rules concerning the construction of subqueries.
- In a multi-partition procedure, subqueries of the UPSERT statement can only reference replicated tables.
- In single-partitioned procedures, the subquery can reference both partitioned and replicated tables.
- For ad hoc UPSERT statements, the same rules apply except the SQL statement itself determines whether VoltDB executes it as a single-partitoned or multi-partitioned procedure. Statements that modify a partitioned table based on a specific value of the partitioning column are executed as single-partitioned procedures. All other statements are multi-partitioned.

Examples

The following examples use two tables, Employee and Manager, both of which define the column emp_id as a primary key. In the first example, the UPSERT statement either creates a new row with the specified values or updates an existing row with the primary key 145303.

```
UPSERT INTO employee (emp_id, lastname, firstname, title, department)
       VALUES (145303, 'Public', 'Jane', 'Manager', 'HR');
```

The next example copies records from the Employee table to the Manager table, if the employee's title is "Manager". Again, new records will be created or existing records updated depending on whether the employee already has a record in the Manager table.

```
UPSERT  INTO Manager (emp_id, lastname, firstname, title, department)
       SELECT * from Employee WHERE title='Manager' ORDER BY emp_id;
```

Appendix C. SQL Functions

Functions let you aggregate column values and perform other calculations and transformations on data within your SQL queries. This appendix lists the functions alphabetically, describing for each their syntax and purpose. The functions can also be grouped by the type of data they produce or operate on, as listed below.

Bitwise Functions

- BIT_SHIFT_LEFT()
- BIT_SHIFT_RIGHT()
- BITAND()
- BITNOT()
- BITOR()
- BITXOR()

Column Aggregation Functions

- APPROX_COUNT_DISTINCT()
- AVG()
- COUNT()
- MAX()
- MIN()
- SUM()

Date and Time Functions

- CURRENT_DATE()
- CURRENT_TIMESTAMP()
- DATEADD()
- DATEDIFF()
- DAY(), DAYOFMONTH()
- DAYOFWEEK()
- DAYOFYEAR()
- EXTRACT()
- FORMAT_TIMESTAMP()
- FROM_UNIXTIME()
- HOUR()
- IS_VALID_TIMESTAMP()
- MAX_VALID_TIMESTAMP()
- MIN_VALID_TIMESTAMP()
- MINUTE()
- MONTH()
- NOW()
- QUARTER()
- SECOND()
- SINCE_EPOCH()
- TIME_WINDOW()
- TO_TIMESTAMP()
- TRUNCATE()
- WEEK(), WEEKOFYEAR()
- WEEKDAY()

- YEAR()

Geospatial Functions

- AREA()
- ASTEXT()
- CENTROID()
- CONTAINS()
- DISTANCE()
- DWITHIN()
- ISINVALIDREASON()
- ISVALID()
- LATITUDE()
- LONGITUDE()
- MAKEVALIDPOLYGON()
- NUMINTERIORRINGS()
- NUMPOINTS()
- POINTFROMTEXT()
- POLYGONFROMTEXT()
- VALIDPOLYGONFROMTEXT()

JSON Functions

- ARRAY_ELEMENT()
- ARRAY_LENGTH()
- FIELD()
- SET_FIELD()

Internet Functions

- INET6_ATON()
- INET6_NTOA()
- INET_ATON()
- INET_NTOA()

Logic and Conversion Functions

- CAST()
- COALESCE()
- DECODE()
- FORMAT()

Math Functions

- ABS()
- CEILING()
- EXP()
- FLOOR()
- LN(), LOG()
- LOG10()
- MOD()
- POWER()
- ROUND()
- SIGN()

- SQRT()

String Functions

- BIN()
- CHAR()
- CHAR_LENGTH()
- CONCAT()
- FORMAT_CURRENCY()
- HEX()
- LEFT()
- LOWER()
- OCTET_LENGTH()
- OVERLAY()
- POSITION()
- REGEXP_POSITION()
- REPEAT()
- REPLACE()
- RIGHT()
- SPACE()
- STR()
- SUBSTRING()
- TRIM()
- UPPER()

Trigonometric Functions

- COS()
- COT()
- CSC()
- DEGREES()
- PI()
- RADIANS()
- SEC()
- SIN()
- TAN()

Miscellaneous Functions

- MIGRATING()

ABS()

ABS() — Returns the absolute value of a numeric expression.

Syntax

ABS(*numeric-expression*)

Description

The ABS() function returns the absolute value of the specified numeric expression.

Example

The following example sorts the results of a SELECT expression by its proximity to a target value (specified by a placeholder), using the ABS() function to normalize values both above and below the intended target.

```
SELECT price, product_name FROM product_list
    ORDER BY ABS(price - ?) ASC;
```

APPROX_COUNT_DISTINCT()

APPROX_COUNT_DISTINCT() — Returns an approximate count of the number of distinct values for the specified column expression.

Syntax

```
APPROX_COUNT_DISTINCT( column-expression )
```

Description

The APPROX_COUNT_DISTINCT() function returns an approximation of the number of distinct values for the specified column expression. APPROX_COUNT_DISTINCT(*column-expression*) is an alternative to the SQL expression "`COUNT(DISTINCT column-expression)`".

The reason for using APPROX_COUNT_DISTINCT() is because it can be significantly faster and use less temporary memory than performing a precise COUNT DISTINCT operation. This is particularly true when calculating a distinct count of a partitioned table across all of the partitions. The approximation usually falls within ±1% of the actual count.

You can use the APPROX_COUNT_DISTINCT() function on column expressions of decimal, date, timestamp, or any size integer datatype. You cannot use the function on floating point (FLOAT) or variable length (VARCHAR and VARBINARY) columns.

Example

The following example returns an approximation of the number of distinct products available in each store.

```
SELECT store, APPROX_COUNT_DISTINCT(product_id) FROM catalog
    GROUP BY store ORDER BY store;
```

AREA()

AREA() — Returns the area of a polygon in square meters.

Syntax

AREA(*polygon*)

Description

The AREA() function returns the area of a GEOGRAPHY value in square meters. The area is the total area of the outer ring minus the area of any inner rings within the polygon. The area is returned as a FLOAT value.

Example

The following example calculates the sum of the areas of multiple polygons representing fields on a farm.

```
SELECT farmer, SUM(AREA(field)) FROM farm
    WHERE farmer = 'Old MacDonald' GROUP BY farmer;
```

ARRAY_ELEMENT()

ARRAY_ELEMENT() — Returns the element at the specified location in a JSON array.

Syntax

ARRAY_ELEMENT(*JSON-array*, *element-position*)

Description

The ARRAY_ELEMENT() function extracts a single element from a JSON array. The array position is zero-based. In other words, the first element in the array is in position "0". The function returns the element as a string. For example, the following function invocation returns the string "two":

```
ARRAY_ELEMENT('["zero","one","two","three"]',2)
```

Note that the array element is always returned as a string. So in the following example, the function returns "2" as a string rather than an integer:

```
ARRAY_ELEMENT('[0,1,2,3]',2)
```

Finally, the element may itself be a valid JSON-encoded object. For example, the following function returns the string "[0,1,2,3]":

```
ARRAY_ELEMENT('[[0,1,2,3],["zero","one","two","three"]]',0)
```

The ARRAY_ELEMENT() function can be combined with other functions, such as FIELD(), to traverse more complex JSON structures. The function returns a NULL value if any of the following conditions are true:

- The position argument is less than zero
- The position argument is greater than or equal to the length of the array
- The JSON string does not represent an array (that is, the string is a valid JSON scalar value or object)

The function returns an error if the first argument is not a valid JSON string.

Example

The following example uses the ARRAY_ELEMENT() function along with FIELD() to extract specific array elements from one field in a JSON-encoded VARCHAR column:

```
SELECT language,
       ARRAY_ELEMENT(FIELD(words,'colors'),1) AS color,
       ARRAY_ELEMENT(FIELD(words,'numbers'),2) AS number
       FROM world_languages WHERE language = 'French';
```

Assuming the column *words* has the following structure, the query returns the strings "French', "vert", and "trois".

```
{"colors":["rouge","vert","bleu"],
 "numbers":["un","deux","trois"]}
```

ARRAY_LENGTH()

ARRAY_LENGTH() — Returns the number of elements in a JSON array.

Syntax

```
ARRAY_LENGTH( JSON-array )
```

Description

The ARRAY_LENGTH() returns the length of a JSON array; that is, the number of elements the array contains. The length is returned as an integer.

The ARRAY_LENGTH() function can be combined with other functions, such as FIELD(), to traverse more complex JSON structures.

The function returns NULL if the argument is a valid JSON string but does not represent an array. The function returns an error if the argument is not a valid JSON string.

Example

The following example uses the ARRAY_LENGTH(), ARRAY_ELEMENT(), and FIELD() functions to return the last element of an array in a larger JSON string. The functions perform the following actions:

- Innermost, the FIELD() function extracts the JSON field "alerts", which is assumed to be an array, from the column *messages*.
- ARRAY_LENGTH() determines the number of elements in the array.
- ARRAY_ELEMENT() returns the last element based on the value of ARRAY_LENGTH() minus one (because the array positions are zero-based).

```
SELECT ARRAY_ELEMENT(FIELD(messages,'alerts'),
       ARRAY_LENGTH(FIELD(messages,'alerts'))-1) AS last_alert,
       station FROM reportlog
       WHERE station=?;
```

ASTEXT()

ASTEXT() — Returns the Well Known Text (WKT) representation of a GEOGRAPHY or GEOGRAPHY_POINT value.

Syntax

```
ASTEXT( polygon | point )
```

Description

The ASTEXT() function returns a text string containing a Well Known Text (WKT) representation of a GEOGRAPHY or GEOGRAPHY_POINT value. ASTEXT(*value*) produces the same results as calling CAST(*value* AS VARCHAR).

Note that ASTEXT() does not return the identical text string that was originally input using POINTFROMTEXT() or POLYGONFROMTEXT(). When geospatial data is converted from WKT to its internal representation, the string representations of longitude and latitude are converted to double floating point values. Rounding and differing levels of precision may result in small differences in the stored values. The use of spaces and capitalization may also vary between the original input strings and the computed output of the ASTEXT() function.

Examples

The following SELECT statement uses the ASTEXT() function to return the WKT representation of a GEOGRAPHY_POINT value in the column *location*.

```
SELECT name, ASTEXT(location) FROM city
   WHERE state = 'NY' ORDER BY name;
```

AVG()

AVG() — Returns the average of a range of numeric column values.

Syntax

AVG(*column-expression*)

Description

The AVG() function returns the average of a range of numeric column values. The values being averaged depend on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example returns the average price for each product category.

```
SELECT AVG(price), category FROM product_list
    GROUP BY category ORDER BY category;
```

BIN()

BIN() — Returns the binary representation of a BIGINT value as a string.

Syntax

```
BIN( value )
```

Description

The BIN() function returns the binary representation of a BIGINT value as a string. The function will return the shortest valid string representation, truncating any preceding zeros (except in the case of the value zero, which is returned as the string "0").

Example

The following example use the BIN and BITAND functions to return the binary representations of two BIGINT values and their binary intersection.

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits values(55,99);
3> select bin(a) as int1, bin(b) as int2,
4>     bin(bitand(a,b)) as intersection from bits;
INT1     INT2      INTERSECTION
-------- --------- -------------
110111   1100011   100011
```

BIT_SHIFT_LEFT()

BIT_SHIFT_LEFT() — Shifts the bits of a BIGINT value to the left a specified number of places.

Syntax

```
BIT_SHIFT_LEFT( value, offset )
```

Description

The BIT_SHIFT_LEFT() function shifts the bit values of a BIGINT value to the left the number of places specified by *offset*. The offset must be a positive integer value. The unspecified bits to the right are padded with zeros. So, for example, if the offset is 5, the left-most 5 bits are dropped, the remaining bits are shifted 5 places to the left, and the right-most 5 bits are set to zero. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example shifts the bits in a BIGINT value three places to the left and displays the hexadecimal representation of both the initial value and the resulting value.

```
$ sqlcmd
1> create table bits (a bigint);
2> insert into bits values (112);
3> select hex(a), hex(bit_shift_left(a,3)) from bits;
C1       C2
-------- ---------
70       380
```

BIT_SHIFT_RIGHT()

BIT_SHIFT_RIGHT() — Shifts the bits of a BIGINT value to the right a specified number of places.

Syntax

```
BIT_SHIFT_RIGHT( value, offset )
```

Description

The BIT_SHIFT_RIGHT() function shifts the bit values of a BIGINT value to the right the number of places specified by *offset*. The offset must be a positive integer value. The unspecified bits to the left are padded with zeros. So, for example, if the offset is 5, the right-most 5 bits are dropped, the remaining bits are shifted 5 places to the right, and the left-most 5 bits are set to zero. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example shifts the bits in a BIGINT value three places to the right and displays the hexadecimal representation of both the initial value and the resulting value.

```
$ sqlcmd
1> create table bits (a bigint);
2> insert into bits values (112);
3> select hex(a), hex(bit_shift_right(a,3)) from bits;
C1       C2
-------- -------
70       E
```

BITAND()

BITAND() — Returns the mask of bits set in both of two BIGINT values

Syntax

```
BITAND( value, value )
```

Description

The BITAND() function returns the mask of bits set in both of two BIGINT integers. In other words, it performs a bitwise AND operation on the two arguments. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes values into two BIGINT columns of the table *bits* and then returns the bitwise AND of the columns:

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits (a,b) values (7,13);
3> select bitand(a,b) from bits;
C1
---
 5
```

BITNOT()

BITNOT() — Returns the mask reversing every bit of a BIGINT value.

Syntax

```
BITNOT( value )
```

Description

The BITNOT() function returns the mask reversing every bit in a BIGINT value. In other words, it performs a bitwise NOT operation, returning the complement of the argument. The result is returned as a new BIGINT value — the argument to the function is not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes a value into a BIGINT column of the table *bits* and then returns the bitwise NOT of the column:

```
$ sqlcmd
1> create table bits (a bigint);
2> insert into bits (a) values (1234567890);
3> select bitnot(a) from bits;
C1
------------
 -1234567891
```

BITOR()

BITOR() — Returns the mask of bits set in either of two BIGINT values

Syntax

BITOR(*value*, *value*)

Description

The BITOR) function returns the mask of bits set in either of two BIGINT integers. In other words, it performs a bitwise OR operation on the two arguments. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes values into two BIGINT columns of the table *bits* and then returns the bitwise OR of the columns:

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits (a,b) values (7,13);
3> select bitor(a,b) from bits;
C1
---
 15
```

BITXOR()

BITXOR() — Returns the mask of bits set in one but not both of two BIGINT values

Syntax

```
BITXOR( value, value )
```

Description

The BITXOR() function returns the mask of bits set in one but not both of two BIGINT integers. In other words, it performs a bitwise XOR operation on the two arguments. The result is returned as a new BIGINT value — the arguments to the function are not modified.

The left-most bit of an integer number is the sign bit, but has no special meaning for bitwise operations. However, The left-most bit set to 1 followed by all zeros is reserved as the NULL value. If you use a NULL value as an argument, you will receive a NULL response. But in all other circumstances (using non-NULL BIGINT arguments), the bitwise functions should never return a NULL result. Consequently any bitwise operation that would result in only the left-most bit being set, will generate an error at runtime.

Examples

The following example writes values into two BIGINT columns of the table *bits* and then returns the bitwise XOR of the columns:

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits (a,b) values (7,13);
3> select bitxor(a,b) from bits;
C1
---
 10
```

CAST()

CAST() — Explicitly converts an expression to the specified datatype.

Syntax

```
CAST( expression AS datatype )
```

Description

The CAST() function converts an expression to a specified datatype. Cases where casting is beneficial include when converting between numeric types (such as integer and float) or when converting a numeric value to a string.

All numeric datatypes can be used as the source and numeric or string datatypes can be the target. When converting from decimal values to integers, values are truncated. You can also cast from a TIMESTAMP to a VARCHAR or from a VARCHAR to a TIMESTAMP, assuming the text string is formatted as *YYYY-MM-DD* or *YYYY-MM-DD HH:MM:SS.nnnnnnn*. Similarly, you can cast DATE to and from VARCHAR or TIMESTAMP. Where the runtime value cannot be converted (for example, the value exceeds the maximum allowable value of the target datatype) an error is thrown.

You cannot use VARBINARY as either the target or the source datatype. To convert between numeric and TIMESTAMP values, use the TO_TIMESTAMP(), FROM_UNIXTIME(), and EXTRACT() functions.

The result of the CAST() function of a null value is the corresponding null in the target datatype.

Example

The following example uses the CAST() function to ensure the result of an expression is also a floating point number and does not truncate the decimal portion.

```
SELECT contestant, CAST( (votes * 100) as FLOAT) / ? as percentage
    FROM contest ORDER BY votes, contestant;
```

CEILING()

CEILING() — Returns the smallest integer value greater than or equal to a numeric expression.

Syntax

CEILING(*numeric-expression*)

Description

The CEILING() function returns the next integer greater than or equal to the specified numeric expression. In other words, the CEILING() function "rounds up" numeric values. For example:

```
CEILING(3.1415) = 4
CEILING(2.0) = 2
CEILING(-5.32) = -5
```

Example

The following example uses the CEILING function to calculate the shipping costs for a product based on its weight in the next whole number of pounds.

```
SELECT shipping.cost_per_lb * CEILING(product.weight),
       product.prod_id FROM product
       JOIN shipping ON product.prod_id=shipping.prod_id
       ORDER BY product.prod_id;
```

CENTROID()

CENTROID() — Returns the central point of a polygon.

Syntax

CENTROID(*polygon*)

Description

The CENTROID() returns the central point of a GEOGRAPHY polygon. The centroid is the point where any line passing through the centroid divides the polygon into two segments of equal area. The return value of the CENTROID() function is a GEOGRAPHY_POINT value.

Note that the centroid may fall outside of the polygon itself. For example, if the polygon is a ring (that is, a circle with an inner circle removed) or a horseshoe shape.

Example

The following example uses the CENTROID() and LATITUDE() functions to return a list of countries where the majority of the land mass falls above the equator.

```
SELECT name, capital FROM country
   WHERE LATITUDE(CENTROID(outline)) > 0
   ORDER BY name, capital;
```

CHAR()

CHAR() — Returns a string with a single UTF-8 character associated with the specified character code.

Syntax

CHAR(*integer*)

Description

The CHAR() function returns a string containing a single UTF-8 character that matches the specified UNICODE character code. One use of the CHAR() function is to insert non-printing and other hard to enter characters into string expressions.

Example

The following example uses CHAR() to add a copyright symbol into a VARCHAR field.

```
UPDATE book SET copyright_notice= CHAR(169) || CAST(? AS VARCHAR)
       WHERE isbn=?;
```

CHAR_LENGTH()

CHAR_LENGTH() — Returns the number of characters in a string.

Syntax

```
CHAR_LENGTH( string-expression )
```

Description

The CHAR_LENGTH() function returns the number of text characters in a string.

Note that the number of characters and the amount of physical space required to store those characters can differ. To measure the length of the string, in bytes, use the OCTET_LENGTH() function.

Example

The following example returns the string in the column LastName as well as the number of characters and length in bytes of that string.

```
SELECT LastName, CHAR_LENGTH(LastName), OCTET_LENGTH(LastName)
    FROM Customers ORDER BY LastName, FirstName;
```

COALESCE()

COALESCE() — Returns the first non-null argument, or null.

Syntax

```
COALESCE( expression [, ... ] )
```

Description

The COALESCE() function takes multiple arguments and returns the value of the first argument that is not null, or — if all arguments are null — the function returns null.

Examples

The following example uses COALESCE to perform two functions:

- Replace possibly null column values with placeholder text
- Return one of several column values

In the second usage, the SELECT statement returns the value of the column State, Province, or Territory depending on the first that contains a non-null value. Or the function returns a null value if none of the columns are non-null.

```
SELECT lastname, firstname,
   COALESCE(address,'[address unkown]'),
   COALESCE(state, province, territory),
   country FROM users ORDER BY lastname;
```

CONCAT()

CONCAT() — Concatenates two or more strings and returns the result.

Syntax

CONCAT(*string-expression* { , ... })

Description

The CONCAT() function concatenates two or more strings and returns the resulting string. The string concatenation operator || performs the same function as CONCAT().

Example

The following example concatenates the contents of two columns as part of a SELECT expression.

```
SELECT price, CONCAT(category,part_name) AS full_part_name
   FROM product_list ORDER BY price;
```

The next example does something similar but uses the || operator as a shorthand to concatenate three strings, two columns and a string constant, as part of a SELECT expression.

```
SELECT lastname || ', ' || firstname AS full_name
   FROM customers ORDER BY lastname, firstname;
```

CONTAINS()

CONTAINS() — Returns true or false depending if a point falls within the specified polygon.

Syntax

```
CONTAINS( polygon, point )
```

Description

The CONTAINS() function determines if a given point falls within the specified GEOGRAPHY polygon. If so, the function returns a boolean value of true. If not, it returns false.

Example

The following example uses the CONTAINS function to see if a specific user is with the boundaries of a city or not by evaluating if the user.location GEOGRAPHY_POINT column value falls within the polygon defined by the city.boundary GEOGRAPHY column.

```
SELECT user.name, user.id, city.name FROM user, city
    WHERE user.id = ? AND CONTAINS(city.boundary,user.location);
```

COS()

COS() — Returns the cosine of an angle specified in radians.

Syntax

COS({*numeric-expression*})

Description

The COS() function returns the cosine of a specified angle as a FLOAT value. The angle must be specified in radians as a numeric expression.

Example

The following example returns the sine, cosine, and tangent of angles from 0 to 90 degrees (where the angle is specified in radians).

```
SELECT SIN(radians), COS(radians), TAN(radians)
     FROM triangles WHERE radians >= 0 AND radians <= PI()/2;
```

COT()

COT() — Returns the cotangent of an angle specified in radians.

Syntax

```
COT( {numeric-expression} )
```

Description

The COT() function returns the cotangent of a specified angle as a FLOAT value. The angle must be specified in radians as a numeric expression.

Examples

The following example returns the secant, cosecant, and cotangent of angles from 0 to 90 degrees (where the angle is specified in radians).

```
SELECT SEC(radians), CSC(radians), COT(radians)
     FROM triangles WHERE radians >= 0 AND radians <= PI()/2;
```

COUNT()

COUNT() — Returns the number of rows selected containing the specified column.

Syntax

COUNT(*column-expression*)

Description

The COUNT() function returns the number of rows selected for the specified column. Since the actual value of the column is not used to calculate the count, you can use the asterisk (*) as a wildcard for any column. For example the query `SELECT COUNT(*) FROM widgets` returns the number of rows in the table `widgets`, without needing to know what columns the table contains.

The one case where the column name is significant is if you use the DISTINCT clause to constrain the selection expression. For example, `SELECT COUNT(DISTINCT last_name) FROM customer` returns the count of unique last names in the customer table.

Examples

The following example returns the number of rows where the product name starts with the captial letter A.

```
SELECT COUNT(*) FROM product_list
    WHERE product_name LIKE 'A%';
```

The next example returns the total number of unique product categories in the product list.

```
SELECT COUNT(DISTINCT category) FROM product_list;
```

CSC()

CSC() — Returns the cosecant of an angle specified in radians.

Syntax

CSC({*numeric-expression*})

Description

The CSC() function returns the cosecant of a specified angle as a FLOAT value. The angle must be specified in radians as a numeric expression.

Examples

The following example returns the secant, cosecant, and cotangent of angles from 0 to 90 degrees (where the angle is specified in radians).

```
SELECT SEC(radians), CSC(radians), COT(radians)
     FROM triangles WHERE radians >= 0 AND radians <= PI()/2;
```

CURRENT_DATE()

CURRENT_DATE() — Returns the current date as a date value.

Syntax

```
CURRENT_DATE()

CURRENT_DATE
```

Description

The CURRENT_DATE() function returns the current date as a VoltDB DATE. The value of the date is determined when the query or stored procedure is invoked. Since there are no arguments to the function, the parentheses following the function name are optional.

Several important aspects of how the CURRENT_DATE() function operates are:

- The value returned is guaranteed to be identical for all partitions that execute the query.
- During command logging, the returned value is stored as part of the log, so when the command log is replayed, the same value is used during the replay of the query.
- Similarly, for database replication (DR) the value returned is passed and reused by the replica database when replaying the query.
- You can specify CURRENT_DATE() as a default value in the CREATE TABLE statement when defining the schema of a VoltDB database.
- The CURRENT_DATE() function *cannot* be used in the CREATE INDEX or CREATE VIEW statements.

Example

The following example uses CURRENT_DATE() and DATEADD() in the WHERE clause to select customers who are 21 years old or older:

```
SELECT * FROM customers WHERE DATEADD(YEAR,21,customers.birthday) <= CURRENT_DATE;
```

CURRENT_TIMESTAMP()

CURRENT_TIMESTAMP() — Returns the current time as a timestamp value.

Syntax

```
CURRENT_TIMESTAMP()

CURRENT_TIMESTAMP
```

Description

The CURRENT_TIMESTAMP() function returns the current time as a VoltDB timestamp. The value of the timestamp is determined when the query or stored procedure is invoked. Since there are no arguments to the function, the parentheses following the function name are optional.

Several important aspects of how the CURRENT_TIMESTAMP() function operates are:

- The value returned is guaranteed to be identical for all partitions that execute the query.
- The value returned is measured in milliseconds then padded to create a timestamp value in microseconds.
- During command logging, the returned value is stored as part of the log, so when the command log is replayed, the same value is used during the replay of the query.
- Similarly, for database replication (DR) the value returned is passed and reused by the replica database when replaying the query.
- You can specify CURRENT_TIMESTAMP() as a default value in the CREATE TABLE statement when defining the schema of a VoltDB database.
- The CURRENT_TIMESTAMP() function *cannot* be used in the CREATE INDEX or CREATE VIEW statements.

The NOW() and CURRENT_TIMESTAMP() functions are synonyms and perform an identical function.

Example

The following example uses CURRENT_TIMESTAMP() in the WHERE clause to delete alert events that occurred in the past:

```
DELETE FROM Alert_event WHERE event_timestamp < CURRENT_TIMESTAMP;
```

DATEADD()

DATEADD() — Returns a new timestamp or date value by adding a specified time interval to an existing value.

Syntax

```
DATEADD( time-unit, interval, input-value )
```

Description

The DATEADD() function creates a new TIMESTAMP or DATE value by adding (or subtracting for negative values) the specified time interval from the input value. The first argument specifies the time unit of the interval. The valid time unit keywords are:

- MICROSECOND (or MICROS)
- MILLISECOND (or MILLIS)
- SECOND
- MINUTE
- HOUR
- DAY
- MONTH
- QUARTER
- YEAR

The second argument is an integer value specifying the interval to add to the third argument. A positive interval moves the time ahead. A negative interval moves the time value backwards.

The third argument specifies the initial input value to which the interval is applied. The datatype of the third argument (TIMESTAMP or DATE) determines the datatype of the return value.

The DATEADD function takes into account leap years and the variable number of days in a month. Therefore, if the year of either the specified input or the resulting output is a leap year, the day is adjusted to its correct value. For example, DATEADD(YEAR, 1, '2008-02-29') returns '2009-02-28'. Similarly, if the original input value is the last day of a month, then the resulting output will be adjusted as necessary. For example, DATEADD(MONTH, 1, '2008-03-31') returns '2008-04-30'.

Example

The following example uses the DATEADD() function to find all records where the TIMESTAMP column, incident, occurs within one day before a specified timestamp (entered as a POSIX time value).

```
SELECT incident, description FROM securityLog
   WHERE DATEADD(DAY, 1, incident) > FROM_UNIXTIME(?)
     AND incident < FROM_UNIXTIME(?)
     ORDER BY incident, description;
```

DATEDIFF()

DATEDIFF() — Returns the difference between two timestamps or dates as an integer.

Syntax

```
DATEDIFF( time-unit, timestamp-or-date, timestamp-or-date )
```

Description

The DATEDIFF() function returns an integer value representing the difference between two timestamps or dates for the specified time unit. Specifically, the second value minus the first value, measured in the specified unit. Note that both values must be of the *same* datatype — either TIMESTAMP or DATE — and the values are converted to the specified time unit *before* the subtraction occurs. So, for example, the difference between the timestamps Dec-31-1999 11:59 and Jan-01-2000 01:01 measured in days would be 1 day, but in hours it would be 2 hours, and in minutes 62 minutes.

The valid time unit keywords are:

- MICROSECOND (or MICROS)
- MILLISECOND (or MILLIS)
- SECOND
- MINUTE
- HOUR
- DAY
- MONTH
- QUARTER
- YEAR

Example

The following example uses the DATEDIFF() function to return the number of whole minutes between the start and end of an event.

```
INSERT INTO billing
    (account_id, when, length)
    SELECT user, start_time,
       DATEDIFF(MINUTE,start_time, end_time)
       FROM event;
```

DAY(), DAYOFMONTH()

DAY(), DAYOFMONTH() — Returns the day of the month as an integer value.

Syntax

DAY(*timestamp-or-date*)

DAYOFMONTH(*timestamp-or-date*)

Description

The DAY() function returns an integer value between 1 and 31 representing the date or timestamp's day of the month. The DAY() and DAYOFMONTH() functions are synonyms. These functions produce the same result as using the DAY or DAY_OF_MONTH keywords with the EXTRACT() function.

Examples

The following example uses the DAY(), MONTH(), and YEAR() functions to return a timestamp column as a formatted date string.

```
SELECT  CAST( MONTH(starttime) AS VARCHAR) || '/' ||
        CAST( DAY(starttime)   AS VARCHAR) || '/' ||
        CAST( YEAR(starttime)  AS VARCHAR), title, description
        FROM event ORDER BY starttime;
```

DAYOFWEEK()

DAYOFWEEK() — Returns the day of the week as an integer between 1 and 7.

Syntax

```
DAYOFWEEK( timestamp-or-date )
```

Description

The DAYOFWEEK() function returns an integer value between 1 and 7 representing the day of the week in a date or timestamp value. For the DAYOFTHEWEEK() function, the week starts (1) on Sunday and ends (7) on Saturday.

This function produces the same result as using the DAY_OF_WEEK keyword with the EXTRACT() function.

Examples

The following example uses DAYOFWEEK() and the DECODE() function to return a string value representing the day of the week for the specified TIMESTAMP value.

```
SELECT eventtime,
   DECODE(DAYOFWEEK(eventtime),
       1, 'Sunday',
       2, 'Monday',
       3, 'Tuesday',
       4, 'Wednesday',
       5, 'Thursday',
       6, 'Friday',
       7, 'Saturday') AS eventday
   FROM event ORDER BY eventtime;
```

DAYOFYEAR()

DAYOFYEAR() — Returns the day of the year as an integer between 1 and 366.

Syntax

```
DAYOFYEAR( timestamp-or-date )
```

Description

The DAYOFYEAR() function returns an integer value between 1 and 366 representing the day of the year of a date or timestamp value. This function produces the same result as using the DAY_OF_YEAR keyword with the EXTRACT() function.

Examples

The following example uses the DAYOFYEAR() function to determine the number of days until an event occurs.

```
SELECT DECODE(YEAR(NOW), YEAR(starttime),
     CAST(DAYOFYEAR(starttime) - DAYOFYEAR(NOW) AS VARCHAR)
           || ' days remaining',
     CAST(YEAR(starttime) - YEAR(NOW) AS VARCHAR)
           || ' years remaining'),
  eventname FROM event;
```

DECODE()

DECODE() — Evaluates an expression against one or more alternatives and returns the matching response.

Syntax

DECODE(*expression*, { *comparison-value*, *result* } [,...] [,*default-result*])

Description

The DECODE() function compares an expression against one or more possible comparison values. If the *expression* matches the *comparison-value*, the associated *result* is returned. If the expression does not match any of the comparison values, the *default-result* is returned. If the expression does not match any comparison value and no default result is specified, the function returns NULL.

The DECODE() function operates the same way an IF-THEN-ELSE, or CASE statement does in other languages.

Example

The following example uses the DECODE() function to interpret a coded data column and replace it with the appropriate meaning for each code.

```
SELECT title, industry, DECODE(salary_range,
             'A', 'under $25,000',
             'B', '$25,000 - $34,999',
             'C', '$35,000 - $49,999',
             'D', '$50,000 - $74,999',
             'E', '$75,000 - $99,000',
             'F', '$100,000 and over',
             'unspecified') from survey_results
       order by industry, title;
```

The next example tests a value against three columns and returns the name of the column when a match is found, or a message indicating no match if none is found.

```
SELECT product_name, DECODE(?,product_name,'PRODUCT NAME',
                            part_name, 'PART NAME',
                            category, 'CATEGORY',
                            'NO MATCH FOUND')
     FROM product_list ORDER BY product_name;
```

DEGREES()

DEGREES() — Converts an angle in radians to degrees

Syntax

DEGREES(*angle-in-radians*)

Description

The DEGREES() function converts a floating-point value representing an angle measured in radians to the equivalent angle measured in degrees.

Example

The following SELECT statement converts a column value stored in radians to degrees before returning it to the user.

```
SELECT test_number, distance, DEGREES(angle) as angle_in_degrees
   FROM tests ORDER BY test_number;
```

DISTANCE()

DISTANCE() — Returns the distance between two points or a point and a polygon.

Syntax

DISTANCE(*point-or-polygon*, *point-or-polygon*)

Description

The DISTANCE() function returns the distance, measured in meters, between two points or a point and a polygon. The arguments to the function can be either two GEOGRAPHY_POINT values or a GEOGRAPHY_POINT and GEOGRAPHY value.

The DISTANCE() function accepts multiple datatypes for its two arguments, but there are constraints on which combination of datatypes are allowed. For example, the two arguments cannot both be of type GEOGRAPHY. Consequently, the VoltDB planner must know the datatype of the arguments when the statement is compiled. So using generic, untyped placeholders for these arguments is not valid. This means you cannot use syntax such as DISTANCE(?,?) in a stored procedure. However, you can use placeholders as long as they are cast to specific types. For example:

```
DISTANCE(POINTFROMTEXT(?),POLYGONFROMTEXT(?))
```

Examples

The following example finds the closest city to a specified user, using the GEOGRAPHY_POINT column user.location and the GEOGRAPHY column city.boundary.

```
SELECT TOP 1 user.name, city.name,
    DISTANCE(user.location, city.boundary)
       FROM user, city WHERE user.id = ?
       ORDER BY DISTANCE(user.location, city.boundary) ASC;
```

The next example finds the distance in kilometers from a truck to stores, listed in order with closest first, using the two GEOGRAPHY_POINT columns truck.loc and store.loc.

```
SELECT store.address,
    DISTANCE(store.loc,truck.loc) / 1000 AS distance
       FROM store, truck WHERE truck.id = ?
       ORDER BY DISTANCE(store.loc,truck.loc)/1000 ASC;
```

DWITHIN()

DWITHIN() — Returns true or false depending whether two geospatial entities are within a specified distance of each other.

Syntax

DWITHIN(*polygon-or-point*, *polygon-or-point*, *distance*)

Description

The DWITHIN() function determines if two geospatial values are within the specified distance of each other. The values can be two points (GEOGRAPHY_POINT) or a point and a polygon (GEOGRAPHY). The maximum distance is specified as a numeric value measured in meters. If the distance between the two geospatial values is less than or equal to the specified distance, the function returns true. If not, it returns false.

The DWITHIN() function accepts multiple datatypes for its first two arguments, but there are constraints on which combination of datatypes are allowed. For example, the two arguments cannot both be of type GEOGRAPHY. Consequently, the VoltDB planner must know the datatype of the arguments when the statement is compiled. So using generic, untyped placeholders for these arguments is not valid. This means you cannot use syntax such as DWITHIN(?,?,?) in a stored procedure. However, you can use placeholders as long as they are cast to specific types. For example:

```
DWITHIN(POINTFROMTEXT(?),POLYGONFROMTEXT(?), ?)
```

Examples

The following example finds all the cities within five kilometers of a given user, by evaluating the distance between the GEOGRAPHY_POINT column user.loc and the GEOGRAPHY column city.boundary.

```
SELECT user.name, city.name, DISTANCE(user.loc, city.boundary)
   FROM user, city WHERE user.id=?
   AND DWITHIN(user.loc, city.boundary, 5000)
   ORDER BY DISTANCE(user.loc, city.boundary) ASC;
```

The next is a more generalized example, where the query returns all delivery trucks within a specified distance of a store, where both the distance and the store ID are parameterized and can be input at runtime.

```
SELECT store.address, truck.license_number,
   DISTANCE(store.loc, truck.loc)/1000 AS distance_in_km
      FROM store, truck
      WHERE DWITHIN(store.loc, truck.loc, ?) and store.id=?
      ORDER BY DISTANCE(store.loc,truck.loc)/1000 ASC;
```

EXP()

EXP() — Returns the exponential of the specified numeric expression.

Syntax

```
EXP( numeric-expression )
```

Description

The EXP() function returns the exponential of the specified numeric expression. In other words, EXP(x) is the equivalent of the mathematical expression e^x.

Example

The following example uses the EXP function to calculate the potential population of certain species of animal projecting out ten years.

```
SELECT species, population AS current,
       (population/2.0) * EXP(10*(gestation/365.0)*litter) AS future
       FROM animals
       WHERE species = 'rabbit'
       ORDER BY population;
```

EXTRACT()

EXTRACT() — Returns the value of a selected portion of a timestamp or date.

Syntax

EXTRACT(*selection-keyword* FROM *timestamp-or-date*)

EXTRACT(*selection-keyword, timestamp-or-date*)

Description

The EXTRACT() function returns the value of the selected portion of a timestamp or date. Both timestamp and date expressions are allowable input for date selections. However, only timestamp values are valid for selecting the HOUR, MINUTE, or SECOND portions of a time value. Table C.1, “Selectable Values for the EXTRACT Function” lists the supported keywords, the datatype of the value returned by the function, and a description of its contents.

Table C.1. Selectable Values for the EXTRACT Function

Keyword	Datatype	Description
YEAR	INTEGER	The year as a numeric value.
QUARTER	TINYINT	The quarter of the year as a single numeric value between 1 and 4.
MONTH	TINYINT	The month of the year as a numeric value between 1 and 12.
DAY	TINYINT	The day of the month as a numeric value between 1 and 31.
DAY_OF_MONTH	TINYINT	The day of the month as a numeric value between 1 and 31 (same as DAY).
DAY_OF_WEEK	TINYINT	The day of the week as a numeric value between 1 and 7, starting with Sunday.
DAY_OF_YEAR	SMALLINT	The day of the year as a numeric value between 1 and 366.
WEEK	TINYINT	The week of the year as a numeric value between 1 and 52.
WEEK_OF_YEAR	TINYINT	The week of the year as a numeric value between 1 and 52 (same as WEEK).
WEEKDAY	TINYINT	The day of the week as a numeric value between 0 and 6, starting with Monday.
HOUR	TINYINT	The hour of the day as a numeric value between 0 and 23.
MINUTE	TINYINT	The minute of the hour as a numeric value between 0 and 59.
SECOND	DECIMAL	The whole and fractional part of the number of seconds within the minute as a floating point value between 0 and 60.

Example

The following example lists all the contacts by name and birthday, listing the birthday as three separate fields for month, day, and year.

```
SELECT Last_name, first_name, EXTRACT(MONTH FROM dateofbirth),
```

```
EXTRACT(DAY FROM dateofbirth), EXTRACT(YEAR FROM dateofbirth)
FROM contact_list
ORDER BY last_name, first_name;
```

FIELD()

FIELD() — Extracts a field value from a JSON-encoded string column.

Syntax

FIELD(*column*, *field-name-path*)

Description

The FIELD() function extracts a field value from a JSON-encoded string. For example, assume the VARCHAR column Profile contains the following JSON string:

```
{"first":"Charles","last":"Dickens","birth":1812,
 "description":{"genre":"fiction",
                "period":"Victorian",
                "output":"prolific",
                "children":["Charles","Mary","Kate","Walter","Francis",
                            "Alfred","Sydney","Henry","Dora","Edward"]
               }
}
```

It is possible to extract individual field values using the FIELD() function, as in the following SELECT statement:

```
SELECT FIELD(profile,'first') AS firstname,
       FIELD(profile,'last') AS lastname FROM Authors;
```

It is also possible to find records based on individual JSON fields by using the FIELD() function in the WHERE clause. For example, the following query retrieves all records from the Authors table where the JSON field *birth* is 1812. Note that the FIELD() function always returns a string, even if the JSON type is numeric. The comparison must match the string datatype, so the constant `'1812'` is in quotation marks:

```
SELECT * FROM Authors WHERE FIELD(profile,'birth') = '1812';
```

The second argument to the FIELD() function can be a simple field name, as in the previous examples. In which case the function returns a first-level field matching the specified name. Alternately, you can specify a path representing a hierarchy of names separated by periods. For example, you can specify the genre element of the description field by specifying "description.genre" as the second argument, like so

```
SELECT * FROM Authors WHERE
  FIELD(profile,'description.genre') = 'fiction';
```

You can also use array notation — with square brackets and an integer value — to identify array elements by their position. So, for example, the function can return "Kate", the third child, by using the path specifier "description.children[2]", where "[2]" identifies the third array element because JSON arrays are zero-based.

Two important points to note concerning input to the FIELD() function:

- If the requested field name does not exist, the function returns a null value.
- The first argument to the FIELD() function *must* be a valid JSON-encoded string. However, the content is not evaluated until the function is invoked at runtime. Therefore, it is the responsibility of the database

application to ensure the validity of the content. If the FIELD() function encounters invalid content, the query will fail.

Example

The following example uses the FIELD() function to both return specific JSON fields within a VARCHAR column and filter the results based on the value of a third JSON field:

```
SELECT product_name, sku,
       FIELD(specification,'color') AS color,
       FIELD(specification,'weight') AS weight FROM Inventory
   WHERE FIELD(specification, 'category') = 'housewares'
   ORDER BY product_name, sku;
```

FLOOR()

FLOOR() — Returns the largest integer value less than or equal to a numeric expression.

Syntax

FLOOR(*numeric-expression*)

Description

The FLOOR() function returns the largest integer less then or equal to the specified numeric expression. In other words, the FLOOR() function truncates fractional numeric values. For example:

```
FLOOR(3.1415) = 3
FLOOR(2.0) = 2
FLOOR(-5.32) = -6
```

Example

The following example uses the FLOOR function to calculate the whole number of stocks owned by a specific shareholder.

```
SELECT customer, company,
       FLOOR(num_of_stocks) AS stocks_available_for_sale
       FROM shareholders WHERE customer_id = ?
       ORDER BY company;
```

FORMAT()

FORMAT() — Returns a formatted text string with values inserted as defined by placeholders in the template.

Syntax

```
FORMAT( format-string [, expression...] )
```

Description

The FORMAT() function returns a formatted text string based on the template in the first argument, where placeholders in the template are replaced with values from the subsequent arguments. This function is equivalent to the *format* function found in Java or the *sprintf* function in C.

Placeholders in the format string consist of a percent sign (%) followed by a single character, such as %s for a string value or %d for a numeric value. These placeholders can be further modified by additional syntactic expressions. See the Boost documentation for a complete description of the format string syntax.

Note that, because the intended datatype of the inserted value is not known at compile time, there is no datatype conversion performed on the arguments to the FORMAT() function. Therefore it is important you use CAST() or other techniques to convert datatypes as necessary. For example, you must explicitly cast DECIMAL datatypes to FLOAT when providing values for a floating point placeholder such as %f .

Example

The following example uses the FORMAT() function to return the query results as a meaningful sentence.

```
SELECT FORMAT('%d people voted during the contest.',COUNT(phone_number))
  AS summary
    FROM votes;
```

FORMAT_CURRENCY()

FORMAT_CURRENCY() — Converts a DECIMAL to a text string as a monetary value.

Syntax

```
FORMAT_CURRENCY( decimal-value, rounding-position )
```

Description

The FORMAT_CURRENCY() function converts a DECIMAL value to its string representation, rounding to the specified position. The resulting string is formatted with commas separating every three digits of the whole portion of the number (indicating thousands, millions, and so on) and a decimal point before the fractional portion, as needed.

The *rounding-position* argument must be an integer between 12 and -25 and indicates the place to which the numeric value should be rounded. Positive values indicate a decimal place; for example 2 means round to 2 decimal places. Negative values indicate rounding to a whole number position; for example, -2 indicates the number should be rounded to the nearest hundred. A zero indicates that the value should be rounded to the nearest whole number.

Rounding is performed using "banker's rounding", in that any fractional half is rounded to the nearest even number. So, for example, if the rounding-position is 2, the value 22.225 is rounded to 22.22, but the value 33.335 is rounded to 33.34. The following list demonstrates some sample results.

FORMAT_CURRENCY(.123456789, 4) = 0.1235
FORMAT_CURRENCY(123456789.123, 2) = 123,456,789.12
FORMAT_CURRENCY(123456789.123, 0) = 123,456,789
FORMAT_CURRENCY(123456789.123, -2) = 123,456,800
FORMAT_CURRENCY(123456789.123, -6) = 123,000,000
FORMAT_CURRENCY(123456789.123, 6) = 123,456,789.123000

Example

The following example uses the FORMAT_CURRENCY() function to return a DECIMAL column as a string representation of its monetary value, rounding to two decimal places and appending the appropriate currency symbol from a VARCHAR column.

```
SELECT country,
       currency_symbol || format_currency(budget,2) AS annual_budget
    FROM world_economy ORDER BY country;
```

FORMAT_TIMESTAMP()

FORMAT_TIMESTAMP() — Takes a timestamp as input and returns a formatted string in the specified timezone.

Syntax

```
FORMAT_TIMESTAMP( timestamp-value, timezone-or-offset )
```

Description

The FORMAT_TIMESTAMP() returns the timestamp input value as a formatted string in the specified timezone. VoltDB stores timestamps as a time value in Greenwich Mean Time (GMT). The FORMAT_TIMESTAMP() function lets you return that value as a date and time string in a different timezone. You can specify the timezone as a either an offset of GMT or as a region as described by the Internet Assigned Numbers Authority (IANA) time zone database (tz). You can find a list of the IANA time zones on the Wikipedia tz page.

Time zone names are case sensitive. Time offsets are specified as a time value preceded by a plus or minus sign. The offset time value can be specified in hours (one or two digits); hours and minutes (four digits); or hours, minutes, and seconds (six digits). You can optionally use colons to separate the time units. For example, all of the following offsets specify the same amount of time — a positive offset of five hours:

```
+5
+05
+0500
+05:00
+050000
+05:00:00
```

Examples

The following example uses the FORMAT_TIMESTAMP() function to return a timestamp column as an Eastern United States timezone date and time:

```
SELECT FORMAT_TIMESTAMP(e_time,'America/New_York'), e_log FROM event;
```

The next example uses an offset to go back 15 minutes:

```
SELECT FORMAT_TIMESTAMP(alarm.expires,'-00:15') AS warning FROM alarm;
```

The last example uses the FORMAT_TIMESTAMP() function to return timestamp values in the customer's chosen timezone.

```
CREATE PROCEDURE get_reservation AS
  SELECT r.id,
   FORMAT_TIMESTAMP(r.departure,c.timezone) AS departure,
   FORMAT_TIMESTAMP(r.arrival,c.timezone) AS arrival
   FROM reservation AS r, customer AS c
   WHERE r.id = ?;
```

FROM_UNIXTIME()

FROM_UNIXTIME() — Converts a UNIX time value to a VoltDB timestamp.

Syntax

```
FROM_UNIXTIME( integer-expression )
```

Description

The FROM_UNIXTIME() function converts an integer expression to a VoltDB timestamp, interpreting the integer value as a POSIX time value; that is the number of seconds since the epoch (00:00.00 on January 1, 1970 Consolidated Universal Time). This function is a synonym for TO_TIMESTAMP(second, *integer-expression*).

Example

The following example inserts a record using FROM_UNIXTIME to convert the first argument, a POSIX time value, into a VoltDB timestamp:

```
INSERT INTO event (e_when, e_what, e_where)
       VALUES (FROM_UNIXTIME(?),?,?);
```

HEX()

HEX() — Returns the hexadecimal representation of a BIGINT value as a string.

Syntax

HEX(*value*)

Description

The HEX() function returns the hexadecimal representation of a BIGINT value as a string. The function will return the shortest valid string representation, truncating any preceding zeros (except in the case of the value zero, which is returned as the string "0").

Examples

The following example use the HEX and BITAND functions to return the hexadecimal representations of two BIGINT values and their binary intersection.

```
$ sqlcmd
1> create table bits (a bigint, b bigint);
2> insert into bits values(555,999);
3> select hex(a) as int1, hex(b) as int2,
4>     hex(bitand(a,b)) as intersection from bits;
INT1     INT2      INTERSECTION
-------- --------- -------------
22B      3E7       223
```

HOUR()

HOUR() — Returns the hour of the day as an integer value.

Syntax

```
HOUR( timestamp-value )
```

Description

The HOUR() function returns an integer value between 0 and 23 representing the hour of the day in a timestamp value. This function produces the same result as using the HOUR keyword with the EXTRACT() function.

Examples

The following example uses the HOUR(), MINUTE(), and SECOND() functions to return the time portion of a TIMESTAMP value in a formatted string.

```
SELECT eventname,
   CAST(HOUR(starttime) AS VARCHAR) || ' hours, ' ||
   CAST(MINUTE(starttime) AS VARCHAR) || ' minutes, and ' ||
   CAST(SECOND(starttime) AS VARCHAR) || ' seconds.'
   AS timestring FROM event;
```

INET6_ATON()

INET6_ATON() — Converts an IPv6 internet address from a string to a VARBINARY(16) value

Syntax

```
INET6_ATON( {string} )
```

Description

The INET6_ATON() function converts a VARCHAR value representing an IPv6 internet address in hexidecimal notation to a 16-byte VARBINARY value in network byte order. The VARCHAR value must consist of up to eight hexidecimal values separated by colons, such as "2600:141b:4:290::2add", or a null value. Note that in IPv6 addresses, two colons together ("::") can and should be used in place of two or more consecutive zero values in the sequence.

You can use the INET6_NTOA() function to reverse the conversion or you can use the INET_ATON and INET_NTOA functions to perform similar conversions on IPv4 addresses.

Example

The following example converts a string representation of an IPv6 internet address to a VARBINARY(16) value before storing it in the Address table

```
INSERT INTO Address (v6ip, owner, date) VALUES (INET6_ATON(?),?,?);
```

INET6_NTOA()

INET6_NTOA() — Converts an IPv6 internet address from a VARBINARY(16) value to a string

Syntax

```
INET_NTOA( {binary-value} )
```

Description

The INET6_NTOA() function converts a 16-byte VARBINARY value representing an IPv6 internet address to its corresponding string representation as a VARCHAR value. Or, if the argument is null, the function returns a null VARCHAR as the result.

You can use the INET6_ATON() function to perform the reverse operation, from a VARCHAR IPv6 address to a VARBINARY(16) value, or you can use the INET_ATON and INET_NTOA functions to perform similar operations on IPv4 addresses.

Examples

The following example converts a VARBINARY(16) representation of an IPv6 internet address into its string representation for output.

```
SELECT INET6_NTOA(v6ip), owner FROM Address
   WHERE owner=? ORDER BY v6ip;
```

INET_ATON()

INET_ATON() — Converts an IPv4 internet address from a string to a numeric value

Syntax

```
INET_ATON( {string} )
```

Description

The INET_ATON() function converts a VARCHAR value representing an IPv4 internet address in dot notation to a single BIGINT value. The VARCHAR value must consist of four integer values separated by dots, such as "104.112.152.119", or a null value. The string representations of the integer values must be between 0 and 256 and cannot contain any spaces or leading zeros. For example, string values of "0" and "12" are valid but "012" is not.

You can use the INET_NTOA() function to reverse the conversion or you can use the INET6_ATON and INET6_NTOA functions to perform similar conversions on IPv6 addresses.

Example

The following example converts a string representation of an internet address to a BIGINT value before storing it in the Address table

```
INSERT INTO Address (ip, owner, date) VALUES (INET_ATON(?),?,?);
```

INET_NTOA()

INET_NTOA() — Converts an IPv4 internet address from a numeric value to a string

Syntax

```
INET_NTOA( {numeric-value} )
```

Description

The INET_NTOA() function converts a BIGINT value representing an IPv4 internet address to its corresponding dot representation as a VARCHAR value. Or, if the argument is null, the function returns a null VARCHAR as the result.

You can use the INET_ATON() function to perform the reverse operation, from a VARCHAR IPv4 address in dot notation to a BIGINT value, or you can use the INET6_ATON and INET6_NTOA functions to perform similar operations on IPv6 addresses.

Examples

The following example converts a BIGINT representation of an internet address into its string representation for output.

```
SELECT INET_NTOA(ip), owner FROM Address
   WHERE owner=? ORDER BY ip;
```

ISINVALIDREASON()

ISINVALIDREASON() — Explains why a GEOGRAPHY polygon is invalid

Syntax

```
ISINVALIDREASON( polygon )
```

Description

The ISINVALIDREASON() function returns a text string explaining if the specified GEOGRAPHY value is valid or not and, if not, why not. The argument to the ISINVALIDREASON() function must be a GEOGRAPHY value describing a polygon. This function is especially useful when validating geospatial data.

Example

The following example uses the ISVALID() and ISINVALIDREASON() functions to report on any invalid polygons in the *border* column of the *country* table.

```
SELECT country_name, ISINVALIDREASON(border)
  FROM Country WHERE NOT ISVALID(border);
```

ISVALID()

ISVALID() — Determines if the specified GEOGRAPHY value is a valid polygon.

Syntax

```
ISVALID( polygon )
```

Description

The ISVALID() function returns true or false depending on whether the specified GEOGRAPHY value is a valid polygon or not. Polygons must follow rules defined by the Open Geospatial Consortium (OGC) standard for Well Known Text (WKT). Specifically:

- A GEOGRAPHY polygon consists of one or more *rings*, where a ring is a closed boundary described by a sequence of vertices and the lines, or *edges*, between those vertices.
- The first ring must be the outer ring and the vertices must be listed in counter clockwise order.
- All subsequent rings represent "holes" in the outer ring. The inner rings must be wholly contained within the outer ring and their vertices must be listed in clockwise order.
- Rings cannot intersect or have adjacent edges.
- The edges of an individual ring cannot cross (for example, a figure "8" is invalid).
- For each ring, the first vertex is listed twice: as both the first and last vertex.

If the specified GEOGRAPHY value is a valid polygon, the function returns true. If not, it returns false.

To maximize performance, VoltDB does not validate the GEOGRAPHY values when they are inserted. However, if you are not sure the WKT strings are valid, you can use ISVALID() to validate the resulting GEOGRAPHY values before inserting them or after they are inserted into the database.

Examples

The first example shows an UPDATE statement that uses the ISVALID() function to remove the contents of a GEOGRAPHY column (by setting it to NULL), if the current contents are invalid.

```
UPDATE REGION SET border = NULL WHERE NOT ISVALID(border);
```

The next example shows part of a stored procedure that uses ISVALID() to conditionally set the value of a column, *mustbevalid*, that is defined as NOT NULL. By setting the column valid to NULL, the procedure ensures that the INSERT statement fails and the stored procedure rolls back if the WKT border column is invalid.

```
public class ValidateBorders extends VoltProcedure {

  public final SQLStmt insertrec = new SQLStmt(
    "INSERT INTO REGION (name, border, mustbevalid)" +
    " SELECT name, border," +
    " CASE WHEN ISVALID(border) THEN 1 ELSE NULL END" +
    " FROM anothertable WHERE name = ? LIMIT 1;"
```

```
);

public VoltTable[] run( String name)
  throws VoltAbortException
    { voltQueueSQL( insertrec, name); return voltExecuteSQL(); }
}
```

IS_VALID_TIMESTAMP()

IS_VALID_TIMESTAMP() — Identifies whether a given value is a valid timestamp.

Syntax

```
IS_VALID_TIMESTAMP( value )
```

Description

The IS_VALID_TIMESTAMP() function returns either true or false depending on whether the specified value is a valid timestamp or not. The minimum valid timestamp equates to the beginning of the year 1583. That is, the first microsecond of that year. The maximum valid timestamp equates to the last microsecond of the year 9999.

Because TIMESTAMP values are stored and can be entered as an eight byte integer, it is possible to enter a numeric value that is not actually a valid timestamp. The functions MIN_VALID_TIMESTAMP() and MAX_VALID_TIMESTAMP() give you access to the valid minimum and maximum values. The function IS_VALID_TIMESTAMP() compares a TIMESTAMP value and returns true or false depending on whether the value falls within the valid range or not.

Example

The following example uses the TIMESTAMP functions to return an informational string for any event records that contain an invalid timestamp value.

```
SELECT 'TIMESTAMP must be between ' ||
        CAST(MIN_VALID_TIMESTAMP() as VARCHAR) ||
        ' and ' ||
        CAST(MAX_VALID_TIMESTAMP() as VARCHAR),
        log_time,
        log_event
   FROM events WHERE NOT IS_VALID_TIMESTAMP(log_time);
```

LATITUDE()

LATITUDE() — Returns the latitude of a GEOGRAPHY_POINT value.

Syntax

```
LATITUDE( point )
```

Description

The LATITUDE() function returns the latitude, as a floating point value, from a GEOGRAPHY_POINT expression.

Example

The following example returns all ships that are located in the northern hemisphere by examining the latitude of their current location.

```
SELECT ship.number, ship.country FROM ship
   WHERE LATITUDE(ship.location) > 0;
```

LEFT()

LEFT() — Returns a substring from the beginning of a string.

Syntax

```
LEFT( string-expression, numeric-expression )
```

Description

The LEFT() function returns the first *n* characters from a string expression, where *n* is the second argument to the function.

Example

The following example uses the LEFT function to return an abbreviation (the first three characters) of the product category as part of the SELECT expression.

```
SELECT LEFT(category,3), product_name, price FROM product_list
    ORDER BY category, product_name;
```

LN(), LOG()

LN(), LOG() — Returns the natural logarithm of a numeric value.

Syntax

LN(*numeric-value*)

LOG(*numeric-value*)

Description

The LN() function returns the natural logarithm of the specified input value. The log is returned as a floating point (FLOAT) value. LN() and LOG() are synonyms and perform the same function.

Example

The following example uses the LN() function to calculate the rate of population growth from census data.

```
SELECT  city, current_population,
    ( ( LN(current_population) - LN(base_population) )
       / (current_year - base_year)
    ) * 100.0 AS percent_growth
    FROM  census ORDER BY city;
```

LOG10()

LOG10() — Returns the base-10 logarithm of a numeric value.

Syntax

LOG10(*numeric-value*)

Description

The LOG10() function returns the base-10, or decimal, logarithm of the specified input value. The log is returned as a floating point (FLOAT) value.

Example

The following example uses the LOG10() function to calculate the magnitude of difference between two values.

```
SELECT  LOG10(YR2.profit/YR1.profit) AS Magnitude_of_growth
    FROM account AS YR1, account AS YR2
    WHERE YR1.fiscalyear=? AND YR2.fiscalyear=?;
```

LONGITUDE()

LONGITUDE() — Returns the longitude of a GEOGRAPHY_POINT value.

Syntax

```
LONGITUDE( point )
```

Description

The LONGITUDE() function returns the longitude, as a floating point value, from a GEOGRAPHY_POINT expression.

Example

The following example returns all ships that are located in the western hemisphere by examining the longitude of their current location.

```
SELECT ship.number, ship.country FROM ship
   WHERE LONGITUDE(ship.location) < 0
   AND  LONGITUDE(ship.location) > -180;
```

LOWER()

LOWER() — Returns a string converted to all lowercase characters.

Syntax

LOWER(*string-expression*)

Description

The LOWER() function returns a copy of the input string converted to all lowercase characters.

Example

The following example uses the LOWER function to perform a case-insensitive search of a VARCHAR field.

```
SELECT product_name, product_id  FROM product_list
    WHERE LOWER(product_name) LIKE 'acme%'
    ORDER BY product_name, product_id;
```

MAKEVALIDPOLYGON()

MAKEVALIDPOLYGON() — Attempts to return a valid GEOGRAPHY value from a GEOGRAPHY polygon

Syntax

```
MAKEVALIDPOLYGON( polygon )
```

Description

A common problem when generating polygons from Well Known Text (WKT) is listing the rings within the polygon in the correct orientation. The vertices of the outer ring must be listed counter-clockwise, while the vertices of the inner rings must be listed in a clockwise direction.

If you use the POLYGONFROMTEXT() function to create GEOGRAPHY values from WKT strings, the rings can be individually correct but, if they are not oriented properly, the resulting polygon will not match your intended geographic region. As a consequence, using the polygon in VoltDB geospatial functions, such as CONTAINS() and DISTANCE(), will produce unexpected answers. You can use ISVALID() to test if the polygon is valid, but ISVALID() simply tests correctness, it does not fix simple errors, such as ring orientation.

MAKEVALIDPOLYGON() both tests the polygon and corrects any errors in ring orientation. The argument to the MAKEVALIDPOLYGON() function is a GEOGRAPHY object representing a polygon. The output is another GEOGRAPHY object, identical to the input if the input is valid, or with the orientation of the rings corrected if they are listed in the wrong direction. If there are any other issues with the polygon that cannot be corrected (such as an incomplete ring or crossed lines), the function throws an error.

Example

The following example uses the MAKEVALIDPOLYGON() function to correct any potential orientation issues with the *location* column in the *country* table.

```
UPDATE country SET boundaries = MAKEVALIDPOLYGON(boundaries);
```

MAX()

MAX() — Returns the maximum value from a range of column values.

Syntax

```
MAX( column-expression )
```

Description

The MAX() function returns the highest value from a range of column values. The range of values depends on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example returns the highest price in the product list.

```
SELECT MAX(price) FROM product_list;
```

The next example returns the highest price for each product category.

```
SELECT category, MAX(price) FROM product_list
   GROUP BY category
   ORDER BY category;
```

MAX_VALID_TIMESTAMP()

MAX_VALID_TIMESTAMP() — Returns the maximum valid timestamp.

Syntax

```
MAX_VALID_TIMESTAMP()

MAX_VALID_TIMESTAMP
```

Description

The MAX_VALID_TIMESTAMP() function returns the maximum valid value for the VoltDB TIMESTAMP datatype. The minimum valid timestamp equates to the beginning of the year 1583. That is, the first microsecond of that year. The maximum valid timestamp equates to the last microsecond of the year 9999.

Because TIMESTAMP values are stored and can be entered as an eight byte integer, it is possible to enter a numeric value that is not actually a valid timestamp. The functions MIN_VALID_TIMESTAMP() and MAX_VALID_TIMESTAMP() give you access to the valid minimum and maximum values. The function IS_VALID_TIMESTAMP() compares a TIMESTAMP value and returns true or false depending on whether the value falls within the valid range or not.

Since there are no arguments to the function, the parentheses following the function name are optional.

Example

The following example uses the TIMESTAMP functions to return an informational string for any event records that contain an invalid timestamp value.

```
SELECT 'TIMESTAMP must be between ' ||
        CAST(MIN_VALID_TIMESTAMP() as VARCHAR) ||
        ' and ' ||
        CAST(MAX_VALID_TIMESTAMP() as VARCHAR),
        log_time,
        log_event
   FROM events WHERE NOT IS_VALID_TIMESTAMP(log_time);
```

MIGRATING()

MIGRATING() — Identifies table rows currently migrating to an export target.

Syntax

```
MIGRATING()

MIGRATING
```

Description

The MIGRATING function identifies rows of a table that are currently being migrated to an export target.

If a table declaration includes the MIGRATE TO TARGET clause, when the migration is triggered (either by the USING TTL clause or an explicit MIGRATE statement), the row's contents are queued for export to the specified export target. Until the export is completed and acknowledged, the row remains in the database but marked for deletion. Once the export is acknowledged, the row is deleted. The MIGRATING function lets you identify rows that have expired but not completed the export action.

The MIGRATING function can only be used in the WHERE clause under the following conditions:

- The selection expression selects from only one table.
- The table in question is declared with MIGRATE TO TARGET.

Examples

The following example selects records for a particular customer where the records are currently being migrated to an export target.

```
SELECT * FROM Requests WHERE customer=? AND MIGRATING;
```

The next example performs the opposite operation — selecting only those records that are *not* currently being migrated.

```
SELECT * FROM Requests WHERE customer=? AND NOT MIGRATING;
```

MIN()

MIN() — Returns the minimum value from a range of column values.

Syntax

```
MIN( column-expression )
```

Description

The MIN() function returns the lowest value from a range of column values. The range of values depends on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example returns the lowest price in the product list.

```
SELECT MIN(price) FROM product_list;
```

The next example returns the lowest price for each product category.

```
SELECT category, MIN(price) FROM product_list
   GROUP BY category
   ORDER BY category;
```

MIN_VALID_TIMESTAMP()

MIN_VALID_TIMESTAMP() — Returns the minimum valid timestamp.

Syntax

```
MIN_VALID_TIMESTAMP()

MIN_VALID_TIMESTAMP
```

Description

The MIN_VALID_TIMESTAMP() function returns the minimum valid value for the VoltDB TIMESTAMP datatype. The minimum valid timestamp equates to the beginning of the year 1583. That is, the first microsecond of that year. The maximum valid timestamp equates to the last microsecond of the year 9999.

Because TIMESTAMP values are stored and can be entered as an eight byte integer, it is possible to enter a numeric value that is not actually a valid timestamp. The functions MIN_VALID_TIMESTAMP() and MAX_VALID_TIMESTAMP() give you access to the valid minimum and maximum values. The function IS_VALID_TIMESTAMP() compares a TIMESTAMP value and returns true or false depending on whether the value falls within the valid range or not.

Since there are no arguments to the function, the parentheses following the function name are optional.

Example

The following example uses the TIMESTAMP functions to return an informational string for any event records that contain an invalid timestamp value.

```
SELECT 'TIMESTAMP must be between ' ||
        CAST(MIN_VALID_TIMESTAMP() as VARCHAR) ||
        ' and ' ||
        CAST(MAX_VALID_TIMESTAMP() as VARCHAR),
        log_time,
        log_event
   FROM events WHERE NOT IS_VALID_TIMESTAMP(log_time);
```

MINUTE()

MINUTE() — Returns the minute of the hour as an integer value.

Syntax

```
MINUTE( timestamp-value )
```

Description

The MINUTE() function returns an integer value between 0 and 59 representing the minute of the hour in a timestamp value. This function produces the same result as using the MINUTE keyword with the EXTRACT() function.

Examples

The following example uses the HOUR(), MINUTE(), and SECOND() functions to return the time portion of a TIMESTAMP value in a formatted string.

```
SELECT eventname,
   CAST(HOUR(starttime) AS VARCHAR) || ' hours, ' ||
   CAST(MINUTE(starttime) AS VARCHAR) || ' minutes, and ' ||
   CAST(SECOND(starttime) AS VARCHAR) || ' seconds.'
   AS timestring FROM event;
```

MOD()

MOD() — Returns the result of a modulo operation.

Syntax

MOD(*dividend*, *divisor*)

Description

The MOD() function performs a modulo operation. That is, it divides one value, the dividend, by another value, the divisor, and returns the remainder of the division operation as a new integer value. The sign of the result, whether positive or negative, matches the sign of the first argument, the dividend.

Both the dividend and the divisor must either both be integer values or both be DECIMAL values and the divisor must *not* be zero. Use of mixed input datatypes or a divisor of zero will result in a runtime error. When using DECIMAL input values, the result is the integer portion of the remainder. In other words, the decimal result is truncated and returned as an integer using the following formula:

MOD(a,b) = a - INT(a/b) * b

Example

The following example uses the HOUR() and MOD() functions to return the hour of a timestamp in 12 hour format

```
SELECT event,
       MOD(HOUR(eventtime)+11, 12)+1,
       CASE WHEN HOUR(eventtime)/12 < 1
          THEN 'AM'
          ELSE 'PM'
       END
   FROM schedule ORDER BY 3, 2;
```

MONTH()

MONTH() — Returns the month of the year as an integer value.

Syntax

```
MONTH( timestamp-or-date )
```

Description

The MONTH() function returns an integer value between 1 and 12 representing the date or timestamp's month of the year. The MONTH() function produces the same result as using the MONTH keyword with the EXTRACT() function.

Examples

The following example uses the DAY(), MONTH(), and YEAR() functions to return a timestamp column as a formatted date string.

```
SELECT  CAST( MONTH(starttime) AS VARCHAR) || '/' ||
        CAST( DAY(starttime)   AS VARCHAR) || '/' ||
        CAST( YEAR(starttime)  AS VARCHAR), title, description
        FROM event ORDER BY starttime;
```

NOW()

NOW() — Returns the current time as a timestamp value.

Syntax

```
NOW()

NOW
```

Description

The NOW() function returns the current time as a VoltDB timestamp. The value of the timestamp is determined when the query or stored procedure is invoked. Since there are no arguments to the function, the parentheses following the function name are optional.

Several important aspects of how the NOW() function operates are:

- The value returned is guaranteed to be identical for all partitions that execute the query.
- The value returned is measured in milliseconds then padded to create a timestamp value in microseconds.
- During command logging, the returned value is stored as part of the log, so when the command log is replayed, the same value is used during the replay of the query.
- Similarly, for database replication (DR) the value returned is passed and reused by the replica database when replaying the query.
- You can specify NOW() as a default value in the CREATE TABLE statement when defining the schema of a VoltDB database.
- The NOW() function *cannot* be used in the CREATE INDEX or CREATE VIEW statements.

The NOW() and CURRENT_TIMESTAMP() functions are synonyms and perform an identical function.

Example

The following example uses NOW(0 in the WHERE clause to delete alert events that occurred in the past:

```
DELETE FROM Alert_event WHERE event_timestamp < NOW;
```

NUMINTERIORRINGS()

NUMINTERIORRINGS() — Returns the number of interior rings within a polygon GEOGRAPHY value.

Syntax

```
NUMINTERIORRINGS( polygon )
```

Description

The NUMINTERIORRINGS() function returns the number of interior rings within a polygon GEOGRAPHY value. Polygon GEOGRAPHY values can contain multiple polygons: one and only one outer polygon and one or more optional inner polygons that define "holes" in the outer polygon. The NUMINTERIORRINGS() function counts the number of inner polygons and returns the result as an integer value.

Example

The following example lists the countries of the world based on the number of interior polygons within the outline GEOGRAPHY column.

```
SELECT NUMINTERIORRINGS(outline), name, capital FROM country
   ORDER BY NUMINTERIORRINGS(outline);
```

NUMPOINTS()

NUMPOINTS() — Returns the number of points within a polygon GEOGRAPHY value.

Syntax

```
NUMPOINTS( polygon )
```

Description

The NUMPOINTS() function returns the total number of points that comprise a polygon GEOGRAPHY value. The number of points includes the points from both the outer polygon and any inner polygons. It also includes all of the points defining the polygon. Which means the starting point for each polygon is counted twice — once as the starting point and once and the ending point — because this is required in the WKT representation of a polygon.

Example

The following example lists the countries of the world based on the number of points in their outlines.

```
SELECT NUMPOINTS(outline), name, capital FROM country
   ORDER BY NUMPOINTS(outline);
```

OCTET_LENGTH()

OCTET_LENGTH() — Returns the number of bytes in a string.

Syntax

```
OCTET_LENGTH( string-expression )
```

Description

The OCTET_LENGTH() function returns the number of bytes of data in a string.

Note that the number of bytes required to store a string and the actual characters that make up the string can differ. To count the number of characters in the string use the CHAR_LENGTH() function.

Example

The following example returns the string in the column LastName as well as the number of characters and length in bytes of that string.

```
SELECT LastName, CHAR_LENGTH(LastName), OCTET_LENGTH(LastName)
    FROM Customers ORDER BY LastName, FirstName;
```

OVERLAY()

OVERLAY() — Returns a string overwriting a portion of the original string with the specified replacement.

Syntax

```
OVERLAY( string PLACING replacement-string FROM position [FOR length] )
```

Description

The OVERLAY() function overwrites a portion of the original string with the replacement string and returns the result. The replacement starts at the specified position in the original string and either replaces the characters one-for-one for the length of the replacement string or, if a FOR *length* is specified, replaces the specified number of characters.

For example, if the original string is 12 characters in length, the replacement string is 3 characters in length and starts at position 4, and the FOR clause is left off, the resulting string consists of the first 3 characters of the original string, the replacement string, and the last 6 characters of the original string:

OVERLAY('abcdefghijkl' PLACING 'XYZ' FROM 4) = 'abcXYZghijkl'

If the FOR clause is included specifying that the replacement string replaces 6 characters, the result is the first 3 characters of the original string, the replacement string, and the last 3 characters of the original string:

OVERLAY('abcdefghijkl' PLACING 'XYZ' FROM 4 FOR 6) = 'abcXYZjkl'

If the combination of the starting position and the replacement length exceed the length of the original string, the resulting output is extended as necessary to include all of the replacement string:

OVERLAY('abcdef' PLACING 'XYZ' FROM 5) = 'abcdXYZ'

If the starting position is greater than the length of the original string, the replacement string is appended to the original string:

OVERLAY('abcdef' PLACING 'XYZ' FROM 20) = 'abcdefXYZ'

Similarly, if the combination of the starting position and the FOR length is greater than the length of the original string, the replacement string simply overwrites the remainder of the original string:

OVERLAY('abcdef' PLACING 'XYZ' FROM 2 FOR 20) = 'aXYZ'

The starting position and length must be specified as non-negative integers. The starting position must be greater than zero and the length can be zero or greater.

Example

The following example uses the OVERLAY function to redact part of a name.

```
SELECT OVERLAY( fullname PLACING '****' FROM 2
                FOR CHAR_LENGTH(fullname)-2
       ) FROM users ORDER BY fullname;
```

PI()

PI() — Returns the value of the mathematical constant pi (π) as a FLOAT value.

Syntax

```
PI()

PI
```

Description

The PI() function returns the value of the mathematical constant pi (π) as a double floating point (FLOAT) value. Since there are no arguments to the function, the parentheses following the function name are optional.

Example

The following example uses the PI() function to return the surface area of a sphere.

```
SELECT radius, 4*PI*POWER(radius, 2) FROM Sphere ORDER BY radius;
```

POINTFROMTEXT()

POINTFROMTEXT() — Returns a GEOGRAPHY_POINT value from the corresponding WKT

Syntax

```
POINTFROMTEXT( string )
```

Description

The POINTFROMTEXT() function generates a GEOGRAPHY_POINT value from a string containing a well known text (WKT) representation of a geographic point. The WKT string must be in the form 'POINT(*longitude latitude*)' where longitude and latitude are floating point values.

if the argument is not a valid WKT representation of a point, the function generates an error.

Example

The following example uses the POINTFROMTEXT() function to update a record containing a GEOGRAPHY_POINT column using two floating point input values (representing longitude and latitude).

```
UPDATE user SET location =
   POINTFROMTEXT(
     CONCAT('POINT(',CAST(? AS VARCHAR),' ',CAST(? AS VARCHAR),')')
   )
   WHERE id = ?;
```

POLYGONFROMTEXT()

POLYGONFROMTEXT() — Returns a GEOGRAPHY value from the corresponding WKT

Syntax

```
POLYGONFROMTEXT( text )
```

Description

The POLYGONFROMTEXT() function generates a GEOGRAPHY value from a string containing a well known text (WKT) representation of a geographic polygon. The WKT string must be a valid representation of a polygon with only one outer polygon and, optionally, one or more inner polygons.

if the argument is not a valid WKT representation of a polygon, the function generates an error.

Example

The following example uses the POLYGONFROMTEXT() function to insert a record containing a GEOGRAPHY column using a text input value containing the WKT representation of a geographic polygon.

```
INSERT INTO city (name, state, boundary) VALUES(?, ?, POLYGONFROMTEXT(?));
```

POSITION()

POSITION() — Returns the starting position of a substring in another string.

Syntax

```
POSITION( substring-expression IN string-expression )
```

Description

The POSITION() function returns the starting position of a substring in another string. The position, if a match is found, is an integer number between one and the length of the string being searched. If no match is found, the function returns zero.

Example

The following example selects all books where the title contains the word "poodle" and returns the book's title and the position of the substring "poodle" in the title.

```
SELECT Title, POSITION('poodle' IN Title) FROM Books
    WHERE Title LIKE '%poodle%'  ORDER BY Title;
```

POWER()

POWER() — Returns the value of the first argument raised to the power of the second argument.

Syntax

POWER(*numeric-expression*, *numeric-expression*)

Description

The POWER() function takes two numeric expressions and returns the value of the first raised to the power of the second. In other words, POWER(x,y) is the equivalent of the mathematical expression x^y.

Example

The following example uses the POWER function to return the surface area and volume of a cube.

```
SELECT length, 6 * POWER(length,2) AS surface,
       POWER(length,3) AS volume FROM Cube
      ORDER BY length;
```

QUARTER()

QUARTER() — Returns the quarter of the year as an integer value

Syntax

QUARTER(*timestamp-or-date*)

Description

The QUARTER() function returns an integer value between 1 and 4 representing the quarter of the year in a DATE or TIMESTAMP value. The QUARTER() function produces the same result as using the QUARTER keyword with the EXTRACT() function.

Examples

The following example uses the QUARTER() and YEAR() functions to group and sort records containing a timestamp.

```
SELECT year(starttime), quarter(starttime),
    count(*) as eventsperquarter
    FROM event
    GROUP BY year(starttime), quarter(starttime)
    ORDER BY year(starttime), quarter(starttime);
```

RADIANS()

RADIANS() — Converts an angle in degrees to radians

Syntax

```
RADIANS( angle-in-degrees )
```

Description

The RADIANS() function converts a floating-point value representing an angle measured in degrees to the equivalent angle measured in radians.

Example

The following INSERT statement converts input entered in degrees to radians before inserting the record into the database.

```
INSERT INTO tests (test_number, distance, angle)
   VALUES (?, ?, RADIANS(?) );
```

REGEXP_POSITION()

REGEXP_POSITION() — Returns the starting position of a regular expression within a text string.

Syntax

```
REGEXP_POSITION( string, pattern [, flag] )
```

Description

The REGEXP_POSITION() function returns the starting position of the first instance of the specified regular expression within a text string. The position value starts at one (1) for the first position in the string and the functions returns zero (0) if the regular expression is not found.

The first argument to the function is the VARCHAR character string to be searched. The second argument is the regular expression pattern to look for. The third argument is an optional flag that specifies whether the search is case sensitive or not. The flag must be single character VARCHAR with one of the following values:

Flag	Description
c	Case-sensitive matching (default)
i	Case-insensitive matching

There are several different formats for regular expressions. The REGEXP_POSITION() uses the revised Perl compatible regular expression (PCRE2) syntax, which is described in detail on the PCRE website.

Examples

The following example uses the REGEXP_POSITION() to filter all records where the column description matches a specific pattern. The examples uses the optional *flag* argument to make the pattern match text regardless of case.

```
SELECT incident, description FROM securityLog
    WHERE REGEXP_POSITION(description,
          'host:\s*10\.186\.[0-9]+\.[0-9]+',
          'i') > 0
    ORDER BY incident;
```

REPEAT()

REPEAT() — Returns a string composed of a substring repeated the specified number of times.

Syntax

```
REPEAT( string-expression, numeric-expression )
```

Description

The REPEAT() function returns a string composed of the substring *string-expression* repeated *n* times where *n* is defined by the second argument to the function.

Example

The following example uses the REPEAT and the CHAR_LENGTH functions to replace a column's actual contents with a mask composed of the letter "X" the same length as the original column value.

```
SELECT username, REPEAT('X', CHAR_LENGTH(password)) as Password
    FROM accounts ORDER BY username;
```

REPLACE()

REPLACE() — Returns a string replacing the specified substring of the original string with new text.

Syntax

```
REPLACE( string, substring, replacement-string )
```

Description

The REPLACE() function returns a copy of the first argument, replacing all instances of the substring identified by the second argument with the third argument. If the substring is not found, no changes are made and a copy of the original string is returned.

Example

The following example uses the REPLACE function to update the Address column, replacing the string "Ceylon" with "Sri Lanka".

```
UPDATE Customers SET address=REPLACE( address,'Ceylon', 'Sri Lanka')
        WHERE address LIKE '%Ceylon%';
```

RIGHT()

RIGHT() — Returns a substring from the end of a string.

Syntax

RIGHT(*string-expression*, *numeric-expression*)

Description

The RIGHT() function returns the last *n* characters from a string expression, where *n* is the second argument to the function.

Example

The following example uses the LEFT() and RIGHT() functions to return an abbreviated summary of the Description column, ensuring the result fits within 20 characters.

```
SELECT product_name,
       LEFT(description,10) || '...' || RIGHT(description,7)
       FROM product_list ORDER BY product_name;
```

ROUND()

ROUND() — Returns a numeric value rounded to the specified decimal place

Syntax

```
ROUND( numeric-value, rounding-position )
```

Description

The ROUND() functions returns the input value rounded to the specific decimal place. The result is returned as a DECIMAL value.

The *numeric-value* argument must be a FLOAT or DECIMAL value. The *rounding-position* argument must be an integer between 12 and -25 and indicates the place to which the numeric value should be rounded. Positive values indicate a decimal place; for example 2 means round to 2 decimal places. Negative values indicate rounding to a whole number position; for example, -2 indicates the number should be rounded to the nearest hundred. A zero indicates that the value should be rounded to the nearest whole number.

Rounding is performed using "banker's rounding", in that any fractional half is rounded to the nearest even number. So, for example, if the rounding-position is 2, the value 22.225 is rounded to 22.22, but the value 33.335 is rounded to 33.34. The following list demonstrates some sample results.

ROUND (.123456789, 4) = 0.123500000000
ROUND(123456789.123, 2) = 123456789.120000000000
ROUND(123456789.123, 0) = 123456789.000000000000
ROUND(123456789.123, -2) = 123456800.000000000000
ROUND(123456789.123, -6) = 123000000.000000000000
ROUND(123456789.123, 6) = 123456789.123000000000

Examples

The following example uses the ROUND() function to return a DECIMAL value, rounding the value of the *budget* column to two decimal places.

```
SELECT country, ROUND(budget,2) AS annual_budget
    FROM world_economy ORDER BY country;
```

SEC()

SEC() — Returns the secant of an angle specified in radians.

Syntax

SEC({*numeric-expression*})

Description

The SEC() function returns the secant of a specified angle as a FLOAT value. The angle must be specified in radians as a numeric expression.

Examples

The following example returns the secant, cosecant, and cotangent of angles from 0 to 90 degrees (where the angle is specified in radians).

```
SELECT SEC(radians), CSC(radians), COT(radians)
     FROM triangles WHERE radians >= 0 AND radians <= PI()/2;
```

SECOND()

SECOND() — Returns the seconds of the minute as a floating point value.

Syntax

```
SECOND( timestamp-value )
```

Description

The SECOND() function returns an floating point value between 0 and 60 representing the whole and fractional part of the number of seconds in the minute of a timestamp value. This function produces the same result as using the SECOND keyword with the EXTRACT() function.

Examples

The following example uses the HOUR(), MINUTE(), and SECOND() functions to return the time portion of a TIMESTAMP value in a formatted string.

```
SELECT eventname,
   CAST(HOUR(starttime) AS VARCHAR) || ' hours, ' ||
   CAST(MINUTE(starttime) AS VARCHAR) || ' minutes, and ' ||
   CAST(SECOND(starttime) AS VARCHAR) || ' seconds.'
   AS timestring FROM event;
```

SET_FIELD()

SET_FIELD() — Returns a copy of a JSON-encoded string, replacing the specified field value.

Syntax

SET_FIELD(*column*, *field-name-path, string-value*)

Description

The SET_FIELD() function finds the specified field within a JSON-encoded string and returns a copy of the string with the new value replacing that field's previous value. Note that the SET_FIELD() function returns an altered copy of the JSON-encoded string — it does not change any column values in place. So to change existing database columns, you must use SET_FIELD() with an UPDATE statement.

For example, assume the Product table contains a VARCHAR column Productinfo which for one row contains the following JSON string:

```
{"product":"Acme widget",
 "availability":"plenty",
 "info": { "description": "A fancy widget.",
           "sku":"ABCXYZ",
           "part_number":1234},
 "warehouse":[{"location":"Dallas","units":25},
              {"location":"Chicago","units":14},
              {"location":"Troy","units":67}]
}
```

It is possible to change the value of the `availability` field using the SET_FIELD function, like so:

```
UPDATE Product SET Productinfo =
    SET_FIELD(Productinfo,'availability','"limited"')
    WHERE FIELD(Productinfo,'product') = 'Acme widget';
```

The second argument to the SET_FIELD() function can be a simple field name, as in the previous example, In which case the function replaces the value of the top field matching the specified name. Alternately, you can specify a path representing a hierarchy of names separated by periods. For example, you can replace the SKU number by specifying "info.sku" as the second argument, or you can replace the number of units in the second warehouse by specifying the field path "warehouse[1].units". For example, the following UPDATE statement does both by nesting SET_FIELD commands:

```
UPDATE Product SET Productinfo =
    SET_FIELD(
       SET_FIELD(Productinfo,'info.sku','"DEFGHI"'),
          'warehouse[1].units', '128')
    WHERE FIELD(Productinfo,'product') = 'Acme widget';
```

Note that the third argument is the string value that will be inserted into the JSON-encoded string. To insert a numeric value, you enclose the value in single quotation marks, as in the preceding example where '128' is used as the replacement value for the `warehouse[1].units` field. To insert a string value, you must include the string quotation marks within the replacement string itself. For example, the preceding code uses the SQL string constant '"DEFGHI"' to specify the replacement value for the text field `info.sku`.

Finally, the replacement string value can be any valid JSON value, including another JSON-encoded object or array. It does not have to be a scalar string or numeric value.

Example

The following example uses the SET_FIELD() function to add a new array element to the warehouse field.

```
UPDATE Product SET Productinfo =
    SET_FIELD(Productinfo,'warehouse',
        '[{"location":"Dallas","units":25},
          {"location":"Chicago","units":14},
          {"location":"Troy","units":67},
          {"location":"Phoenix","units":23}]')
    WHERE FIELD(Productinfo,'product') = 'Acme widget';
```

SIGN()

SIGN() — Indicates whether a numeric value is positive, negative, or zero.

Syntax

SIGN(*numeric-expression*)

Description

The SIGN() function evaluates a numeric expression and returns +1 if it is positive, -1 if it is negative, and 0 or NULL if it is zero or null, respectively. For example:

```
SIGN(3.1415) = 1
SIGN(5 - 12) = -1
SIGN(0) = 0
```

Example

The following example uses the SIGN() and DECODE() functions to select a graphical representation of the difference between two fields. In this case, up and down arrows based on the difference between the purchase price and current price of stocks.

```
SELECT name, current_price,
   DECODE(SIGN(Current_price - Purchase_price),
      +1, CHAR(11014), -- Up arrow
      -1, CHAR(11015), -- Down arrow
       0, CHAR(8212))  -- M-dash
   FROM Stocks
     ORDER BY name ASC;
```

SIN()

SIN() — Returns the sine of an angle specified in radians.

Syntax

SIN({*numeric-expression*})

Description

The SIN() function returns the sine of a specified angle as a FLOAT value. The angle must be specified in radians as a numeric expression.

Example

The following example returns the sine, cosine, and tangent of angles from 0 to 90 degrees (where the angle is specified in radians).

```
SELECT SIN(radians), COS(radians), TAN(radians)
     FROM triangles WHERE radians >= 0 AND radians <= PI()/2;
```

SINCE_EPOCH()

SINCE_EPOCH() — Converts a VoltDB timestamp to an integer number of time units since the POSIX epoch.

Syntax

```
SINCE_EPOCH( time-unit, timestamp-expression )
```

Description

The SINCE_EPOCH() function converts a VoltDB timestamp into an 64-bit integer value (BIGINT) representing the equivalent number since the POSIX epoch in a specified time unit. POSIX time is usually represented as the number of seconds since the epoch; that is, since 00:00.00 on January 1, 1970 Consolidated Universal Time (UTC). So the function SINCE_EPOCH(SECONDS, timestamp) returns the POSIX time equivalent for the value of timestamp. However, you can also request the number of milliseconds or microseconds since the epoch. The valid keywords for specifying the time units are:

- SECOND — Seconds since the epoch
- MILLISECOND, MILLIS — Milliseconds since the epoch
- MICROSECOND, MICROS — Microseconds since the epoch

You cannot perform arithmetic on timestamps directly. So SINCE_EPOCH() is useful for performing calculations on timestamp values in SQL expressions. For example, the following SQL statement looks for events that are less than a minute in length, based on the timestamp columns STARTTIME and ENDTIME:

```
SELECT * FROM Event
  WHERE ( SINCE_EPOCH(Second, endtime)
        - SINCE_EPOCH(Second, starttime) ) < 60;
```

The TO_TIMESTAMP() function performs the inverse of SINCE_EPOCH(), by converting an integer value to a VoltDB timestamp based on the specified time units.

Example

The following example returns a timestamp column as the equivalent POSIX time value.

```
SELECT event_id, event_name,
    SINCE_EPOCH(Second, starttime) as posix_time FROM Event
    ORDER BY event_id;
```

The next example uses SINCE_EPOCH() to return the length of an event, in microseconds, by calculating the difference between two timestamp columns.

```
SELECT event_id, event_type,
    SINCE_EPOCH(Microsecond, endtime)
    -SINCE_EPOCH(Microsecond, starttime) AS delta
    FROM Event ORDER BY event_id;
```

SPACE()

SPACE() — Returns a string of spaces of the specified length.

Syntax

SPACE(*numeric-expression*)

Description

The SPACE() function returns a string composed of *n* spaces where the string length *n* is specified by the function's argument. SPACE(n) is a synonym for REPEAT(' ', n).

Example

The following example uses the SPACE and CHAR_LENGTH functions to ensure the result is a fixed length, padded with blank spaces.

```
SELECT product_name || SPACE(80 - CHAR_LENGTH(product_name))
   FROM product_list ORDER BY product_name;
```

SQRT()

SQRT() — Returns the square root of a numeric expression.

Syntax

SQRT(*numeric-expression*)

Description

The SQRT() function returns the square root of the specified numeric expression.

Example

The following example uses the SQRT and POWER functions to return the distance of a graph point from the origin.

```
SELECT location, x, y,
       SQRT(POWER(x,2) + POWER(y,2)) AS distance
       FROM points ORDER BY location;
```

STR()

STR() — Returns the string representation of a numeric value.

Syntax

```
STR( numeric-value [string-length [decimal-precision]] )
```

Description

The STR() function returns a string representation of the numeric input. The first argument can be either a FLOAT or DECIMAL value.

The optional second argument specifies the maximum size of the output string and must be an integer between 0 and 38. If the maximum string length is less than the number of characters required to represent the numeric value, the resulting string is filled with asterisk (*) characters. The default length is 10 characters.

The optional third argument specifies the number of decimal places included in the output, which is specified as an integer between 0 and 12. If the numeric value requires more decimal places than specified, the value is rounded using "banker's rounding". (See the description of the FORMAT_CURRENCY() function for a description of banker's rounding.) If the decimal precision is not specified, the value is rounded and only the integer portion is returned.

Example

The following example uses STR() to return a percentage, rounded to two decimal places and including the percent sign.

```
SELECT STR( 100.0 * c.population / total_pop ) || '%'
  FROM countries AS c,
    (SELECT SUM(population) AS total_pop FROM countries) as w
  WHERE c.name=?;
```

SUBSTRING()

SUBSTRING() — Returns the specified portion of a string expression.

Syntax

SUBSTRING(*string-expression* FROM *position* [FOR *length*])

SUBSTRING(*string-expression*, *position* [, *length*])

Description

The SUBSTRING() function returns a specified portion of the string expression, where *position* specifies the starting position of the substring (starting at position 1) and *length* specifies the maximum length of the substring. The length of the returned substring is the lower of the remaining characters in the string expression or the value specified by *length*.

For example, if the string expression is "ABCDEF" and position is specified as 3, the substring starts with the character "C". If length is also specified as 3, the return value is "CDE". If, however, the length is specified as 5, only the remaining four characters "CDEF" are returned.

If *length* is not specified, the remainder of the string, starting from the specified by *position*, is returned. For example, SUBSTRING("ABCDEF",3) and SUBSTRING("ABCDEF"3,4) return the same value.

Example

The following example uses the SUBSTRING function to return the month of the year, which is a VARCHAR column, as a three letter abbreviation.

```
SELECT event, SUBSTRING(month,1,3), day, year FROM calendar
    ORDER BY event ASC;
```

SUM()

SUM() — Returns the sum of a range of numeric column values.

Syntax

SUM(*column-expression*)

Description

The SUM() function returns the sum of a range of numeric column values. The values being added together depend on the constraints defined by the WHERE and GROUP BY clauses.

Example

The following example uses the SUM() function to determine how much inventory exists for each product type in the catalog.

```
SELECT category, SUM(quantity) AS inventory FROM product_list
    GROUP BY category ORDER BY category;
```

TAN()

TAN() — Returns the tangent of an angle specified in radians.

Syntax

```
TAN( {numeric-expression} )
```

Description

The TAN() function returns the tangent of a specified angle as a FLOAT value. The angle must be specified in radians as a numeric expression.

Example

The following example returns the sine, cosine, and tangent of angles from 0 to 90 degrees (where the angle is specified in radians).

```
SELECT SIN(radians), COS(radians), TAN(radians)
     FROM triangles WHERE radians >= 0 AND radians <= PI()/2;
```

TIME_WINDOW()

TIME_WINDOW() — Returns the time "window" into which the specified timestamp value falls.

Syntax

```
TIME_WINDOW( time-unit, window-length, timestamp [, START|END ] )
```

Description

The TIME_WINDOW() function defines time "slices" or "windows" of the requested size and returns a value identifying which window the specified timestamp argument falls in. This can help you group events or records by time. The size of the window is defined by the first two arguments: the unit of measurement (second, minute, hour, and so on) and the length, which must be an integer. The function returns a TIMESTAMP value identifying the beginning or end of the window, depending on the optional fourth argument. (The default is START, the beginning of the window.)

The valid time unit keywords are:

- MILLISECOND (or MILLIS)
- SECOND
- MINUTE
- HOUR
- DAY
- WEEK
- MONTH
- QUARTER
- YEAR

The TIME_WINDOW function is particularly useful when aggregating data into views based on time. For example, the following view definition uses the TIME_WINDOW() function to count the number of logins grouped into half hour (30 minute) segments:

```
CREATE VIEW active_users (users, window)
   AS SELECT count(*), TIME_WINDOW(MINUTE, 30, login)
      FROM user_login
      GROUP BY TIME_WINDOW(MINUTE, 30, login);
```

The time windows themselves are all relative to the standard UNIX start time, midnight January 1, 1970, with the exception of weeks. Weeks are assumed to begin on Monday so any time windows measured in weeks are relative to midnight Monday, December 29, 1969.

Example

The following example uses the TIME_WINDOW() function to group temperature measurements into 2 hour windows, recording the minimum, maximum, and average temperature for each window and location.

```
CREATE VIEW temp_view (location, window, minimum, maximum, average)
  AS SELECT location, TIME_WINDOW(HOUR, 2, measure_timestamp),
            MIN(temperature), MAX(temperature), AVG(temperature)
     FROM temp_record
     GROUP BY location, TIME_WINDOW(HOUR, 2, measure_timestamp)
```

TO_TIMESTAMP()

TO_TIMESTAMP() — Converts an integer value to a VoltDB timestamp based on the time unit specified.

Syntax

TO_TIMESTAMP(*time-unit*, *integer-expression*)

Description

The TO_TIMESTAMP() function converts an integer expression to a VoltDB timestamp, interpreting the integer value as the number of specified time units since the POSIX epoch. POSIX time is usually represented as the number of seconds since the epoch; that is, since 00:00.00 on January 1, 1970 Consolidated Universal Time (UTC). So the function TO_TIMESTAMP(Second, timeinsecs) returns the VoltDB TIMESTAMP equivalent of timeinsecs as a POSIX time value. However, you can also request the integer value be interpreted as milliseconds or microseconds since the epoch. The valid keywords for specifying the time units are:

- SECOND — Seconds since the epoch
- MILLISECOND. MILLIS — Milliseconds since the epoch
- MICROSECOND, MICROS — Microseconds since the epoch

You cannot perform arithmetic on timestamps directly. So TO_TIMESTAMP() is useful for converting the results of arithmetic expressions to VoltDB TIMESTAMP values. For example, the following SQL statement uses TO_TIMESTAMP to convert a POSIX time value before inserting it into a VoltDB TIMESTAMP column:

```
INSERT INTO Event
   (event_id,event_name,event_type, starttime)
   VALUES(?,?,?,TO_TIMESTAMP(Second, ?));
```

The SINCE_EPOCH() function performs the inverse of TO_TIMESTAMP(), by converting a VoltDB TIMESTAMP to an integer value based on the specified time units.

Example

The following example updates a TIMESTAMP column, adding one hour (in seconds) to the current value using SINCE_EPOCH() and TO_TIMESTAMP() to perform the conversion and arithmetic:

```
UPDATE Contest
   SET deadline=TO_TIMESTAMP(Second, SINCE_EPOCH(Second,deadline) + 3600)
   WHERE expired=1;
```

TRIM()

TRIM() — Returns a string with leading and/or training spaces removed.

Syntax

TRIM([[LEADING | TRAILING | BOTH] [*character*] FROM] *string-expression*)

Description

The TRIM() function returns a string with leading and/or trailing spaces removed. By default, the TRIM function removes spaces from both the beginning and end of the string. If you specify the LEADING or TRAILING clause, spaces are removed from either the beginning or end of the string only.

You can also specify an alternate character to remove. By default only spaces (UTF-8 character code 32) are removed. If you specify a different character, only that character will be removed. For example, the following INSERT statement uses the TRIM function to remove any TAB characters from the beginning of the string input for the ADDRESS column:

```
INSERT INTO Customers (first, last, address)
      VALUES(?, ?,
         TRIM( LEADING CHAR(9) FROM CAST(? AS VARCHAR) )
      );
```

Example

The following example uses TRIM() to remove extraneous leading and trailing spaces from the output for three VARCHAR columns:

```
SELECT TRIM(first), TRIM(last), TRIM(address) FROM Customer
      ORDER BY last, first;
```

TRUNCATE()

TRUNCATE() — Truncates a VoltDB date or timestamp to the specified time unit.

Syntax

```
TRUNCATE( time-unit, input-value )
```

Description

The TRUNCATE() function truncates a date or timestamp value to the specified time unit. The datatype of the input value (DATE or TIMESTAMP) determines the datatype of the return value. For example, if the TIMESTAMP column Apollo has the value July 20, 1969 4:17:40 P.M, then using the function TRUNCATE(hour,apollo) would return the value July 20, 1969 4:00:00 P.M. as a TIMESTAMP. Allowable time units for truncation include the following:

- YEAR
- QUARTER
- MONTH
- DAY
- HOUR
- MINUTE
- SECOND
- MILLISECOND, MILLIS

Example

The following example uses the TRUNCATE function to find records where the timestamp column, incident, falls within a specific day, entered as a POSIX time value.

```
SELECT incident, description FROM securitylog
       WHERE TRUNCATE(DAY, incident) = TRUNCATE(DAY,FROM_UNIXTIME(?))
       ORDER BY incident, description;
```

UPPER()

UPPER() — Returns a string converted to all uppercase characters.

Syntax

```
UPPER( string-expression )
```

Description

The UPPER() function returns a copy of the input string converted to all uppercase characters.

Example

The following example uses the UPPER function to return results alphabetically regardless of case.

```
SELECT UPPER(product_name), product_id  FROM product_list
    ORDER BY UPPER(product_name);
```

VALIDPOLYGONFROMTEXT()

VALIDPOLYGONFROMTEXT() — Returns a validated GEOGRAPHY value from the corresponding WKT

Syntax

```
VALIDPOLYGONFROMTEXT( text )
```

Description

The VALIDPOLYGONFROMTEXT() function generates a valid GEOGRAPHY value from a string containing a well known text (WKT) representation of a geographic polygon. If the GEOGRAPHY value resulting from the WKT string cannot be made into a valid representation of a polygon, the function returns an error. The error message includes an explanation of why the WKT is not valid. If the polygon is valid except the rings are drawn in the wrong direction (that is, the outer ring is clockwise or the inner rings are counterclockwise), the VALIDPOLYGONFROMTEXT() function will correct the rings and generate a valid polygon.

The difference between the POLYGONFROMTEXT() function and the VALIDPOLYGONFROMTEXT() function is that the VALIDPOLYGONFROMTEXT() verifies that the resulting polygon meets all of the requirements for use by VoltDB. If a valid polygon cannot be generated, the VALIDPOLYGONFROMTEXT() function returns an error. The POLYGONFROMTEXT() function, on the other hand, simply constructs a GEOGRAPHY value without validating all of the requirements of a VoltDB polygon and may need separate validation (using the ISVALID() or MAKEVALIDPOLYGON() function) before it can be used effectively with other geospatial functions. See the description of the ISVALID() function for a description of the requirements for a valid polygon.

Example

The following example uses the VALIDPOLYGONFROMTEXT() function to insert a record containing a GEOGRAPHY column using a text input value containing the WKT representation of a geographic polygon. Note that if

```
INSERT INTO city (name, state, boundary)
    VALUES(?, ?, VALIDPOLYGONFROMTEXT(?));
```

WEEK(), WEEKOFYEAR()

WEEK(), WEEKOFYEAR() — Returns the week of the year as an integer value.

Syntax

```
WEEK( timestamp-or-date )

WEEKOFYEAR( timestamp-or-date )
```

Description

The WEEK() and WEEKOFYEAR() functions are synonyms and return an integer value between 1 and 52 representing the date or timestamp's week of the year. These functions produce the same result as using the WEEK_OF_YEAR keyword with the EXTRACT() fucntion.

Examples

The following example uses the WEEK() function to group and sort records containing a timestamp.

```
SELECT week(starttime), count(*) as eventsperweek
  FROM event GROUP BY week(starttime) ORDER BY week(starttime);
```

WEEKDAY()

WEEKDAY() — Returns the day of the week as an integer between 0 and 6.

Syntax

```
WEEKDAY( timestamp-or-date )
```

Description

The WEEKDAY() function returns an integer value between 0 and 6 representing the day of the week in a date or timestamp value. For the WEEKDAY() function, the week starts (0) on Monday and ends (6) on Sunday.

This function is provided for compatibility with MySQL and produces the same result as using the WEEKDAY keyword with the EXTRACT() function.

Examples

The following example uses WEEKDAY() and the DECODE() function to return a string value representing the day of the week for the specified TIMESTAMP value.

```
SELECT eventtime,
   DECODE(WEEKDAY(eventtime),
       0, 'Monday',
       1, 'Tuesday',
       2, 'Wednesday',
       3, 'Thursday',
       4, 'Friday',
       5, 'Saturday',
       6, 'Sunday') AS eventday
   FROM event ORDER BY eventtime;
```

YEAR()

YEAR() — Returns the year as an integer value.

Syntax

```
YEAR( timestamp-or-date )
```

Description

The YEAR() function returns an integer value representing the year of a DATE or TIMESTAMP value. The YEAR() function produces the same result as using the YEAR keyword with the EXTRACT() function.

Examples

The following example uses the DAY(), MONTH(), and YEAR() functions to return a timestamp column as a formatted date string.

```
SELECT  CAST( MONTH(starttime) AS VARCHAR) || '/' ||
        CAST( DAY(starttime)   AS VARCHAR) || '/' ||
        CAST( YEAR(starttime)  AS VARCHAR), title, description
        FROM event ORDER BY starttime;
```

Appendix D. VoltDB CLI Commands

VoltDB provides shell or CLI (command line interpreter) commands to perform common functions for developing, starting, and managing VoltDB applications and databases. This appendix describes those shell commands in detail.

The commands are listed in alphabetical order.

- csvloader
- jdbcloader
- kafkaloader
- sqlcmd
- voltadmin
- voltdb

Using CLI Commands with TLS/SSL

When TLS (Transport Layer Security) encryption is enabled for the cluster external ports (that is, the client and admin ports, not just the httpd port), you must explicitly tell VoltDB CLI commands that interact with a running cluster to use TLS. The simplest way to do this, if you are using a certificate from an external certificate authority, is to include the **--ssl** flag on the command line. For example:

```
$ sqlcmd --ssl
```

Alternately, if you are using a locally generated certificate, you must specify an Java properties file that points to the trust store as an argument to the flag, like so:

```
$ sqlcmd --ssl=mytruststore.conf
```

The properties file verifies that the database server is passing credentials that you trust. That is, credentials that match the trust store you reference. The format of file is a Java properties files declaring two properties, one per line, that identify the trust store and trust store password:

trustStore={path-to-trust-store}
trustStorePassword={trust-store password}

See Section 12.7, “Encrypting VoltDB Communication Using TLS/SSL” for more information on configuring TLS encryption on the external ports.

csvloader

csvloader — Imports the contents of a CSV file and inserts it into a VoltDB table.

Syntax

csvloader *table-name* [arguments]

csvloader -p *procedure-name* [arguments]

Description

The csvloader command reads comma-separated values and inserts each valid line of data into the specified table in a VoltDB database. The most common way to use csvloader is to specify the database table to be loaded and a CSV file containing the data, like so:

```
$ csvloader employees -f acme_employees.csv
```

Alternately, you can use standard input as the source of the data:

```
$ csvloader employees < acme_employees.csv
```

In addition to inserting all valid content into the specified database table, csvloader creates three output files:

- **Error log** — The error log provides details concerning any errors that occur while processing the input file. This includes errors in the format of the input as well as errors that occur attempting the insert into VoltDB. For example, if two rows contain the same value for a column that is declared as unique, the error log indicates that the second insert fails due to a constraint violation.
- **Failed input** — A separate file contains the contents of each line that failed to load. This file is useful because it allows you to correct any formatting issues and retry just the failed content, rather than having to restart and reload the entire table.
- **Summary report** — Once all input lines are processed, csvloader generates a summary report listing how many lines were read, how many were successfully loaded and how long the operation took.

All three files are created, by default, in the current working directory using "csvloader" and the table name as prefixes. For example, using csvloader to insert contestants into the sample voter database creates the following files:

csvloader_contestants_insert_log.log
csvloader_contestants_invalidrows.csv
csvloader_contestants_insert_report.log

It is possible to use csvloader to load text files other than CSV files, using the `--separator`, `--quotechar`, and `--escape` flags. Note that csvloader uses Python to process the command line arguments. So to enter certain non-alphanumeric characters, you must use the appropriate escaping mechanism for Python command lines. For example, to use a tab-delimited file as input, you need to use the `--separator` flag, escaping the tab character like so:

```
$ csvloader --separator=$'\t'  \
```

```
-f employees.tab  employees
```

It is also important to note that, unlike VoltDB native clients, when interpreting string values for TIMESTAMP columns, csvloader interprets the values according to the local time zone. That is, the time zone set by the local system. To have string values interpreted as Greenwich Mean Time, you can either use the qualifier `--timezone="GMT"` or set the system variable TZ to "GMT" prior to invoking the csvloader. For example, the following commands are equivalent:

```
$ csvloader employees --timezone="GMT" -f employees.csv
$ export TZ=GMT; csvloader employees -f employees.csv
```

Arguments

--batch=*{integer}*

Specifies the number of rows to submit in a batch. If you do not specify an insert procedure, rows of input are sent in batches to maximize overall throughput. You can specify how many rows are sent in each batch using the `--batch` flag. The default batch size is 200. If you use the `--procedure` flag, no batching occurs and each row is sent separately.

--blank={error | null | empty }

Specifies what to do with missing values in the input. By default, if a line contains a missing value, it is interpreted as a null value in the appropriate datatype. If you do not want missing values to be interpreted as nulls, you can use the --blank argument to specify other behaviors. Specifying `--blank error` results in an error if a line contains any missing values and the line is not inserted. Specifying `--blank empty` returns the corresponding "empty" value in the appropriate datatype. An empty value is interpreted as the following:

- Zero for all numeric columns
- Null for date and timestamp columns
- An empty or zero-length string for VARCHAR and VARBINARY columns

-c, --charset=*{character-set}*

Specifies the character set of the input file. The default character set is UTF-8.

--columnsizelimit=*{integer}*

Specifies the maximum size of quoted column input, in bytes. Mismatched quotation marks in the input can cause csvloader to read all subsequent input — including line breaks — as part of the column. To avoid excessive memory use in this situation, the flag sets a limit on the maximum number of bytes that will be accepted as input for a column that is enclosed in quotation marks and spans multiple lines. The default is 16777216 (that is, 16MB).

--credentials=*{properties-file}*

Specifies a file that lists the username and password of the account to use when connecting to a database with security enabled. This is useful when writing shell scripts because it avoids having to hardcode the password as plain text in the script. The credentials file is interpreted as a Java properties file defining the properties `username` and `password`. For example:

```
username: johndoe
password: 4tUn8
```

Because it is a Java properties file, you must escape certain special characters in the username or password, including the colon or equals sign.

--escape=*{character}*

Specifies the escape character that must precede a separator or quotation character that is supposed to be interpreted as a literal character in the CSV input. The default escape character is the backslash (\). This character escapes the next character both inside and outside of quoted strings *except* when using the default quote character, the double quotation ("). When using the default quote character, the escape character only operates outside quoted strings; inside quoted strings, all characters are interpreted as literal characters except two double quotation marks in a row, which are interpreted as a single double quote.

-f, --file=*{file-specification}*

Specifies the location of a CSV file to read as input. If you do not specify an input file, csvloader reads input from standard input.

--header

Specifies that the first line of the CSV file is a header row, containing the names of the columns. The column names must match columns in the VoltDB table. However, by using --header, the columns can appear in a different order in the CSV file from the order in the database schema. Note that you must specify all of the table column names in the header. The arguments --header and --procedure are mutually exclusive.

--kerberos=*{service-name}*

Specifies the use of kerberos authentication when connecting to the database server(s). The service name identifies the Kerberos client service module, as defined by the JAAS login configuration file.

--limitrows=*{integer}*

Specifies the maximum number of rows to be read from the input stream. This argument (along with --skip) lets you load a subset of a larger CSV file.

-m, --maxerrors=*{integer}*

Specifies the target number of errors before csvloader stops processing input. Once csvloader encounters the specified number of errors while trying to insert rows, it will stop reading input and end the process. Note that, since csvloader performs inserts asynchronously, it often attempts more inserts before the target number of exceptions are returned from the database. So it is possible more errors could be returned after the target is met. This argument lets you conditionally stop a large loading process if more than an acceptable number of errors occur.

--noquotechar

Disables the interpretation of quotation characters in the CSV input. All input other than the separator character and line break will be treated as literal input characters.

--nowhitespace

Specifies that the CSV input must not contain any whitespace between data values and separators. By default, csvloader ignores extra space between values, quotation marks, and the value separators. If you use this argument, any input lines containing whitespace will generate an error and not be inserted into the database.

--password=*{text}*

Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--port=*{port-number}*

Specifies the network port to use when connecting to the database. If you do not specify a port, csvloader uses the default client port 21212.

-p, --procedure=*{procedure-name}*

Specifies a stored procedure to use for loading each record from the data file. The named procedure must exist in the database schema and must accept the fields of the data record as input parameters. By default, csvloader uses a custom procedure to batch multiple rows into a single insert operation. If you explicitly name a procedure, batching does not occur.

--quotechar=*{character}*

Specifies the quotation character that is used to enclose values. By default, the quotation character is the double quotation mark (").

-r, --reportdir=*{directory}*

Specifies the directory where csvloader writes the three output files. By default, csvloader writes output files to the current working directory. This argument lets you redirect output to an alternative location.

--s, --servers=*{server-id}[,...]*

Specifies the network address of one or more nodes of a database cluster. When specifying an IPv6 address, enclose the address in square brackets. By default, csvloader attempts to insert the CSV data into a database on the local system (localhost). To load data into a remote database, use the --servers argument to specify the database nodes the loader should connect to.

--separator=*{character}*

Specifies the character used to separate individual values in the input. By default, the separator character is the comma (,).

--skip=*{integer}*

Specifies the number of lines from the input stream to skip before inserting rows into the database. This argument (along with --limitrows) lets you load a subset of a larger CSV file.

--ssl=*{ssl-config-file}*

Specifies the use of TLS encryption when communicating with the server. Only necessary if the cluster is configured to use TLS encryption for the external ports. See the section called “Using CLI Commands with TLS/SSL” for more information.

--stopondisconnect

Specifies that if connections to all of the VoltDB servers are broken, the loader will stop. Normally, if the connection to the database is lost, csvloader periodically attempts to reconnect until the servers

come back online and it can complete the loading process. However, you can use this argument to have the loader process stop if the VoltDB cluster becomes unavailable.

--strictquotes

Specifies that all values in the CSV input must be enclosed in quotation marks. If you use this argument, any input lines containing unquoted values will generate an error and not be inserted into the database.

--update

Specifies that existing records with a matching primary key are updated, rather than being rejected. By default, csvloader attempts to create new records. The --update flag lets you load updates to existing records — and create new records where the primary key does not already exist. To use --update, the table *must* have a primary key.

--user=*{text}*

Specifies the username to use when connecting to the database. You must specify a username and password if security is enabled for the database.

-z, --timezone=*{text}*

Specifies the timezone to use when interpreting input for a TIMESTAMP column You specify the timezone as a Java TimeZone identifier. For example, you can specify a continent and region ("America/New_York") or a time zone acronym ("EST"). The --timezone qualifier is only valid when specifying a table name as the target; not when specifying a stored procedure.

Examples

The following example loads the data from a CSV file, `languages.csv`, into the helloworld table from the Hello World example database and redirects the output files to the ./logs subfolder.

```
$ csvloader helloworld -f languages.csv -r ./logs
```

The following example performs the same function, providing the input interactively.

```
$ csvloader helloworld -r ./logs
"Hello", "World", "English"
"Bonjour", "Monde", "French"
"Hola", "Mundo", "Spanish"
"Hej", "Verden", "Danish"
"Ciao", "Mondo", "Italian"
CTRL-D
```

jdbcloader

jdbcloader — Extracts a table from another database via JDBC and inserts it into a VoltDB table.

Syntax

jdbcloader *table-name* [arguments]

jdbcloader -p *procedure-name* [arguments]

Description

The jdbcloader command uses the JDBC interface to fetch all records from the specified table in a remote database and then insert those records into a matching table in VoltDB. The most common way to use jdbcloader is to copy matching tables from another database to VoltDB. In this case, you specify the name of the table, plus any JDBC-specific arguments that are needed. Usually, the required arguments are the JDBC connection URL, the source table, the username, password, and local JDBC driver. For example:

```
$ jdbcloader employees \
     --jdbcurl=jdbc:postgresql://remotesvr/corphr \
     --jdbctable=employees \
     --jdbcuser=charlesdickens \
     --jdbcpassword=bleakhouse \
     --jdbcdriver=org.postgresql.Driver
```

In addition to inserting all valid content into the specified database table, jdbcloader creates three output files:

- **Error log** — The error log provides details concerning any errors that occur while processing the input file. This includes errors that occur attempting the insert into VoltDB. For example, if two rows contain the same value for a column that is declared as unique, the error log indicates that the second insert fails due to a constraint violation.
- **Failed input** — A separate file contains the contents of each record that failed to load. The records are stored in CSV (comma-separated value) format. This file is useful because it allows you to correct any formatting issues and retry just the failed content using the csvloader.
- **Summary report** — Once all input records are processed, jdbcloader generates a summary report listing how many records were read, how many were successfully loaded and how long the operation took.

All three files are created, by default, in the current working directory using "jdbcloader" and the table name as prefixes. For example, using jdbcloader to insert contestants into the sample voter database creates the following files:

jdbcloader_contestants_insert_log.log
jdbcloader_contestants_insert_invalidrows.csv
jdbcloader_contestants_insert_report.log

It is possible to use jdbcloader to perform other input operations. For example, if the source table does not have the same structure as the target table, you can use a custom stored procedure to perform the necessary translation from one to the other by specifying the procedure name on the command line with the --procedure flag:

```
$ jdbcloader --procedure translateEmpRecords \
```

```
 --jdbcurl=jdbc:postgresql://remotesvr/corphr \
  --jdbctable=employees \
  --jdbcuser=charlesdickens \
  --jdbcpassword=bleakhouse \
  --jdbcdriver=org.postgresql.Driver
```

Arguments

--batch=*{integer}*

Specifies the number of rows to submit in a batch to the target VoltDB database. If you do not specify an insert procedure, rows of input are sent in batches to maximize overall throughput. You can specify how many rows are sent in each batch using the `--batch` flag. The default batch size is 200. If you use the `--procedure` flag, no batching occurs and each row is sent separately.

--credentials=*{properties-file}*

Specifies a file that lists the username and password of the account to use when connecting to a database with security enabled. This is useful when writing shell scripts because it avoids having to hardcode the password as plain text in the script. The credentials file is interpreted as a Java properties file defining the properties `username` and `password`. For example:

```
username: johndoe
password: 4tUn8
```

Because it is a Java properties file, you must escape certain special characters in the username or password, including the colon or equals sign.

--fetchsize=*{integer}*

Specifies the number of records to fetch in each JDBC call to the source database. The default fetch size is 100 records,

--jdbcdriver=*{class-name}*

Specifies the class name of the JDBC driver to invoke. The driver must exist locally and be accessible either from the CLASSPATH environment variable or in the `lib/extension` directory where VoltDB is installed.

--jdbcpassword=*{text}*

Specifies the password to use when connecting to the source database via JDBC. You must specify a username and password if security is enabled on the source database.

--jdbctable=*{table-name}*

Specifies the name of source table on the remote database. By default, jdbcloader assumes the source table has the same name as the target VoltDB table.

--jdbcurl=*{connection-URL}*

Specifies the JDBC connection URL for the source database. This argument is required.

--jdbcuser=*{text}*

Specifies the username to use when connecting to the source database via JDBC. You must specify a username and password if security is enabled on the source database.

--limitrows=*{integer}*

Specifies the maximum number of rows to be read from the input stream. This argument lets you load a subset of a remote database table.

-m, --maxerrors=*{integer}*

Specifies the target number of errors before jdbcloader stops processing input. Once jdbcloader encounters the specified number of errors while trying to insert rows, it will stop reading input and end the process. Note that, since jdbcloader performs inserts asynchronously, it often attempts more inserts before the target number of exceptions are returned from the database. So it is possible more errors could be returned after the target is met. This argument lets you conditionally stop a large loading process if more than an acceptable number of errors occur.

--password=*{text}*

Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--port=*{port-number}*

Specifies the network port to use when connecting to the VoltDB database. If you do not specify a port, jdbcloader uses the default client port 21212.

-p, --procedure=*{procedure-name}*

Specifies a stored procedure to use for loading each record from the input source. The named procedure must exist in the VoltDB database schema and must accept the fields of the data record as input parameters. By default, jdbcloader uses a custom procedure to batch multiple rows into a single insert operation. If you explicitly name a procedure, batching does not occur.

-r, --reportdir=*{directory}*

Specifies the directory where jdbcloader writes the three output files. By default, jdbcloader writes output files to the current working directory. This argument lets you redirect output to an alternative location.

--s, --servers=*{server-id}[,...]*

Specifies the network address of one or more nodes of a VoltDB cluster. When specifying an IPv6 address, enclose the address in square brackets. By default, jdbcloader attempts to insert the data into a VoltDB database on the local system (localhost). To load data into a remote database, use the --servers argument to specify the VoltDB database nodes the loader should connect to.

--ssl=*{ssl-config-file}*

Specifies the use of TLS encryption when communicating with the server. Only necessary if the cluster is configured to use TLS encryption for the external ports. See the section called “Using CLI Commands with TLS/SSL” for more information.

--stopondisconnect

Specifies that if connections to all of the VoltDB servers are broken, the loader will stop. Normally, if the connection to the database is lost, jdbcloader periodically attempts to reconnect until the servers

come back online and it can complete the loading process. However, you can use this argument to have the loader process stop if the VoltDB cluster becomes unavailable.

--user=*{text}*

Specifies the username to use when connecting to the VoltDB database. You must specify a username and password if security is enabled on the target database.

Example

The following example loads records from the Products table of the Warehouse database on server hq.mycompany.com and writes the records into the Products table of the VoltDB database on servers svrA, svrB, and svrC, using the MySQL JDBC driver to access to source database. Note that the --jdbctable flag is not needed since the source and target tables have the same name.

```
$ jdbcloader Products --servers="svrA,svrB,svrC" \
    --jdbcurl="jdbc:mysql://hq.mycompany.com/warehouse" \
    --jdbcdriver="com.mysql.jdbc.Driver" \
    --jdbcuser="ceo" \
    --jdbcpassword="headhoncho"
```

kafkaloader

kafkaloader — Imports data from a Kafka message queue into the specified database table.

Syntax

```
kafkaloader table-name [arguments]
```

Description

The kafkaloader utility loads data from a Kafka message queue and inserts each message as a separate record into the specified database table. Apache Kafka is a distributed messaging service that lets you set up message queues which are written to and read from by "producers" and "consumers", respectively. In the Apache Kafka model, the kafkaloader acts as a "consumer".

When you start the kafkaloader, you must specify at least three arguments:

- The database table
- The Kafka server to read messages from, specified using the --brokers flag
- The Kafka "topic" where the messages are stored, specified using the --topic flag

For example:

```
$ kafkaloader --brokers=quesvr:2181 --topic=voltdb_customer customer
```

Note that Kafka does not impose any specific format on the messages it manages. The format of the messages are application specific. In the case of kafkaloader, VoltDB assumes the messages are encoded as standard comma-separated value (CSV) strings, with the values representing the columns of the table in the order listed in the schema definition. Each Kafka message contains a single row to be inserted into the database table.

It is also important to note that, unlike the csvloader which reads a static file, the kafkaloader is reading from a queue where messages can be written at any time, on an ongoing basis. Therefore, the kafkaloader process does not stop when it reads the last message on the queue; instead it continues to monitor the queue and process any new messages it receives. The kafkaloader process will continue to read from the queue until one of the following events occur:

- The maximum number of errors (specified by `--maxerrors`) is reached.
- The user explicitly stops the process.
- If `--stopondisconnect` is specified and connection to all of the VoltDB servers is broken (that is, kafkaloader can no longer access the VoltDB database).

The kafkaloader will *not* terminate if it loses its connection to the Kafka zookeeper. Therefore, it is important to monitor the Kafka service and restart the kafkaloader if and when the Kafka service is interrupted. Similarly, the kafkaloader will not stop if it loses connection to the VoltDB database, unless you include the `--stopondisconnect` argument on the command line.

Arguments

Note

The arguments `--servers` and `--port` are deprecated in favor of the new, more flexible argument `--host`. Also, the argument `--zookeeper` is deprecated in favor of the new argument `--brokers`. The deprecated arguments continue to work but may be removed in a future major release.

--batch=*{integer}*

Specifies the number of rows to submit in a batch. By default, rows of input are sent in batches to maximize overall throughput. You can specify how many rows are sent in each batch using the `--batch` flag. The default batch size is 200.

Note that --batch and --flush work together. Whichever limit is reached first triggers an insert to the database.

-b, -brokers=*{kafka-broker[:port]}[,...]*

Specifies one or more Kafka brokers to connect to. Specify multiple brokers as a comma-separated list. The Kafka service must be running Kafka 0.10.2 or later (including 1.0.0).

-c, --config=*{file}*

Specifies a Kafka configuration file that lets you set Kafka consumer properties, such as *group.id*. The file should contain the names of the properties you want to set, one per line, followed by an equals sign and the desired value. For example:

```
group.id=mydb
client.id=myappname
```

--commitpolicy=*{interval}*

Because the loader performs two distinct tasks — retrieving records from Kafka and then inserting them into VoltDB — Kafka's automated tracking of the current offset may not match what records are successfully inserted into the database. Therefore, by default, the importer uses a manual commit policy to ensure the Kafka offset matches the completed inserts.

Use of the default commit policy is recommended. However, you can, if you choose, use Kafka's automated commit policy by specifying a commit interval, in milliseconds, using this property.

--credentials=*{properties-file}*

Specifies a file that lists the username and password of the account to use when connecting to a database with security enabled. This is useful when writing shell scripts because it avoids having to hardcode the password as plain text in the script. The credentials file is interpreted as a Java properties file defining the properties `username` and `password`. For example:

```
username: johndoe
password: 4tUn8
```

Because it is a Java properties file, you must escape certain special characters in the username or password, including the colon or equals sign.

-f, --flush=*{integer}*

Specifies the maximum number of seconds before pending data is written to the database. The default flush period is 10 seconds.

If data is inserted into the kafka queue intermittently, there could be a long delay between when data is read from the queue and when enough records have been read to meet the `--batch` limit. The flush value avoids unnecessary delays in this situation by periodically writing all pending data. If the flush limit is reached, all pending records are written to the database, even if the `--batch` limit has not been satisfied.

--formatter=*{file}*

Specifies a configuration file identifying properties for a custom formatter. The file must set the property `formatter` to the class for the custom implementation of the Formatter interface. (Note, this is different than the attribute set when declaring a formatter for a built-in import connector. For the kaflaloader utility you specify the Formatter class, not the Formatter Factory.) You can also declare additional custom properties used by the formatter itself. For example:

```
formatter=myformatter.MyFormatter
column_width=12
```

Before running kafkaloader with a custom formatter, you must define two environment variables: ZK_LIB pointing to the location of the Apache Zookeeper libraries and FORMATTER_LIB pointing to the location of your custom formatter JAR file. See the chapter on "Custom Importers,Exporters, and Formatters" in the VoltDB Guide to Performance and Customization for more information about using custom formatters.

-H, --host=*{server[:port]}[,...]*

Specifies one or more nodes of the database cluster where the records are to be inserted. You can specify servers as a network address or hostname, plus an optional port number. When specifying an IPv6 address, enclose the address (exclusive of the optional colon and port number) in square brackets. By default, kafkaloader attempts to connect to the default client port on the local system (localhost). To load data into a remote database, use the --host argument to specify one or more VoltDB servers the loader should connect to. Once kafkaloader connects to at least one cluster node, it will automatically connect to the other servers in the cluster.

-m, --maxerrors=*{integer}*

Specifies the target number of input errors before kafkaloader stops processing input. Once kafkaloader encounters the specified number of errors while trying to insert rows, it will stop reading input and end the process.

The default maximum error count is 100. Since kafka import can be an persistent process, you can avoid having input errors cancel ongoing import by setting the maximum error count to zero, which means that the loader will continue to run no matter how many input errors are generated.

--maxpollinterval=*{integer}*

Specifies the maximum time (in milliseconds) allowed between polls of the Kafka brokers before Kafka assumes the kafkaloader client has failed and drops it from the client group. The default poll interval is 300 seconds (5 minutes).

--maxpollrecords=*{integer}*

Specifies the maximum number of records fetched in each batch from the kafka brokers. The default maximum is 500 records.

--maxrequesttimeout=*{integer}*

Specifies the maximum length of time (in milliseconds) VoltDB waits for a response from the Kafka brokers before retrying the request or timing out the session. The default time out is 305 seconds (just over 5 minutes).

--maxsessiontimeout=*{integer}*

Specifies the maximum interval between heart beats from the consumer (kafkaloader) and the Kafka brokers before Kafka drops the kafkaloader from the client group identified by group.id. The default time out is 20 seconds.

-n, --consumercount=*{integer}*

Specifies the number of concurrent Kafka consumers kafakloader uses to pull data from the brokers. The default is one consumer.

--password=*{text}*

Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

-p, --procedure=*{procedure-name}*

Specifies a stored procedure to use for loading each record from the data file. The named procedure must exist in the database schema and must accept the fields of the data record as input parameters. By default, kafkaloader uses a custom procedure to batch multiple rows into a single insert operation. If you explicitly name a procedure, batching does not occur.

--ssl=*{ssl-config-file}*

Specifies the use of TLS encryption when communicating with the server. Only necessary if the cluster is configured to use TLS encryption for the external ports. See the section called “Using CLI Commands with TLS/SSL” for more information.

--stopondisconnect

Specifies that if connections to all of the VoltDB servers are broken, the kafkaloader process will stop. The kafkaloader connects to servers automatically as the topology of the cluster changes. Normally, if *all* connections are broken, kafkaloader will periodically attempt to reconnect until the servers come back online. However, you can use this argument to have the loader process stop when the VoltDB cluster becomes unavailable.

-t, --topic=*{kafka-topic}*

Specifies the Kafka topic to read from the Kafka queue.

--update

Specifies that existing records with a matching primary key are updated, rather than being rejected. By default, kafkaloader attempts to create new records. The --update flag lets you load updates to existing records — and create new records where the primary key does not already exist. To use --update, the table *must* have a primary key.

--user=*{text}*

Specifies the username to use when connecting to the database. You must specify a username and password if security is enabled for the database.

Examples

The following example starts the kafkaloader to read messages from the voltdb_customer topic on the Kafka broker quebkr:9092, inserting the resulting records into the CUSTOMER table in the VoltDB cluster that includes the servers dbsvr1, dbsvr2, and dbsvr3. The process will continue, regardless of errors, until the user explicitly ends the process.

```
$ kafkaloader --maxerrors=0 customer \
   --brokers=quebkr:2181 --topic=voltdb_customer \
   --host=dbsvr1,dbsvr2,dbsvr3
```

sqlcmd

sqlcmd — Starts an interactive command prompt for issuing SQL queries to a running VoltDB database

Syntax

```
sqlcmd [args...]
```

Description

The sqlcmd command lets you query a VoltDB database interactively. You can execute SQL statements, invoke stored procedures, or use directives to examine the structure of the database. When sqlcmd starts it provides its own command line prompt until you exit the session. When you start the session, you can optionally specify one or more database servers to access. By default, sqlcmd accesses the database on the local system via localhost.

At the sqlcmd prompt, you have several options:

- **SQL queries** — You can enter ad hoc SQL queries that are run against the database and the results displayed. You must terminate the query with a semi-colon and carriage return.
- **Procedure calls** — You can have sqlcmd execute a stored procedure. You identify a procedure call with the **exec** directive, followed by the procedure class name, the procedure parameters, and a closing semi-colon. For example, the following sqlcmd directive executes the @SystemCatalog system procedure requesting information about the stored procedures.

  ```
  $ sqlcmd
  1> exec @SystemCatalog procedures;
  ```

  Note that string values can be entered as plain text or enclosed in single quotation marks. Also, the **exec** directive must be terminated by a semi-colon.
- **Echo directives** — The **echo** and **echoerror** directives let you add comments or informational messages to the sqlcmd output. Any text following the directive up to and including the line break or carriage return is repeated verbatim:
  - ECHO *[text]* — Writes the specified text, as is, to standard output (stdout).
  - ECHOERROR *[text]* — Writes the specified text, as is, to standard error (stderr).
- **Show, Describe, and Explain directives** — The **show**, **describe**, and **explain** directives let you examine the structure of the schema and user-defined stored procedures. Valid directives are:
  - SHOW CLASSES — Lists the user-defined classes in the database. Classes are grouped into procedures classes (those that can be invoked as a stored procedure) and non-procedure classes (shared classes that cannot themselves be called as stored procedures but can be invoked from within stored procedures).
  - SHOW PROCEDURES — Lists the user-defined, default, and system procedures for the current database, including the type and number of arguments for each.
  - SHOW TABLES — Lists the tables in the schema.
  - DESCRIBE *{table-name}* — Lists the columns of a table, stream, or view.

    - EXPLAIN *{sql-query}* — Displays the execution plan for the specified SQL statement.
    - EXPLAINPROC *{procedure-name}* — Displays the execution plans for the specified stored procedure.
    - EXPLAINVIEW *{view-name}* — Displays the execution plans for the components of the specified view.
- **Class management directives** — The **load classes** and **remove classes** directives let you add and remove Java classes from the database:
    - LOAD CLASSES —Loads any classes or resource files in the specified JAR file. If a class or resource already exists in the database, it is replaced by the new definition from the JAR file.
    - REMOVE CLASSES — Removes any classes that match the specified class name string. The class specification can include wildcards.
- **Command recall** — You can recall previous commands using the up and down arrow keys. Or you can recall a specific command by line number (the command prompt shows the line number) using the **recall** command. For example:

  ```
  $ sqlcmd
  1> select * from votes;
  2> show procedures;
  3> recall 1
  select * from votes;
  ```

  Once recalled, you can edit the command before reissuing it using typical editing keys, such as the left and right arrow keys and backspace and delete.
- **Script files** — You can run multiple queries or stored procedures in a single command using the **file** directive. The **file** directive takes one or more text files as an argument and executes all of the SQL queries and **exec** directives in the file(s) as if they were entered interactively. (Do not use control directives such as **recall** and **exit** in script files.) Separate multiple script files with spaces. Enclose file names that contain spaces with single quotation marks. For example, the first command in the following example processes all of the SQL queries and procedure invocations in the file `myscript.sql`. The second command processes the SQL queries from two files:

  ```
  $ sqlcmd
  1> file myscript.sql;
  2> file yourscript.sql 'their script.sql';
  ```

  If the file(s) contain only data definition language (DDL) statements, you can also have the files processed as a single batch by including the `-batch` argument:

  ```
  $ sqlcmd
  1> file -batch myscript.sql;
  ```

  If a file or set of statements includes both DDL and DML statements, you can still batch process a group of DDL statements by enclosing the statements in a `file -inlinebatch` directive and the specified end marker. For example, in the following code the three CREATE PROCEDURE statements are processed as a batch:

  ```
  load classes myprocs.jar;
  file -inlinebatch END_OF_BATCH
  CREATE PROCEDURE FROM CLASS procs.AddEmployee;
  ```

```
CREATE PROCEDURE FROM CLASS procs.ChangeDept;
CREATE PROCEDURE FROM CLASS procs.PromoteEmployee;
END_OF_BATCH
```

Batch processing the DDL statements has two effects:

- Batch processing can significantly improve performance since all of the schema changes are processed and distributed to the cluster nodes at one time, rather than individually for each statement.
- The batch operates as a transaction, succeeding or failing as a unit. If any statement fails, all of the schema changes are rolled back.

- **Exit** — When you are done with your interactive session, enter the **exit** directive to end the session and return to the shell prompt.

To run a sqlcmd command without starting the interactive prompt, you can pipe the command through standard input to the sqlcmd command. For example:

```
$ echo "select * from contestants;" | sqlcmd
```

In general, the sqlcmd commands are not case sensitive and must be terminated by a semi-colon. However, the semi-colon is optional for the **exit**, **file**, and **recall** directives. Also, **list** and **quit** are supported as synonyms for the **show** and **exit** directives, respectively.

Textual Data and Character Sets

All text data in Volt is processed and stored as UTF-8 encoded strings. However, it is possible to enter and display data in other character sets if desired.

For interactive sessions, the **sqlcmd** utility automatically translates between the localized character set for the user's interactive session and UTF-8 used by Volt internally. For example, when executing an INSERT statement into a VARCHAR column, the text being inserted is converted from the localized character set into UTF-8 before being inserted. Similarly, when displaying results from a SELECT statement, text fields are converted from UTF-8 into the user's session character set before being sent to the terminal. The web-based Volt Management Console (VMC) provides a similar service automatically converting to and from the user's session character set to UTF-8 on input and output.

When processing data from a file, **sqlcmd** provides the **--charset** qualifier that lets you specify the character set used both for processing files for input (either with the **--file** qualifier or the **FILE** directive) and writing files for output (with the **--output** qualifier). You can also use the **--charset** qualifier with the **csvloader** utility to specify the character set of the CSV input file. Finally, the file export connector supports a `charset` property for specifying what character set to use when exporting data to a file. (See Table 15.1, “File Export Properties” for more information on the file export connector properties.)

Which character sets are supported depends on which Java virtual machine (JVM) release you are using on your servers (for export) or client machines (for **sqlcmd** and **csvloader**). For established character sets, such as Shift_JIS or ISO-8859-1, all supported JVM releases provide support. For newer character sets, you may need a more recent release of the JVM. For example, the recent Simplified Chinese character set GB18030-2022 requires a JVM released in 2023 or later. For OpenJDK this includes the following releases:

- Java 8 — release 8u32-b05
- Java 11 — release 11.0.20+8
- Java 17 — release 17.0.8+7

Arguments

--batch

When used with the `--file` qualifier, processes the data definition language (DDL) statements in the file as a single batch (rather than one at a time) and returns to the shell prompt. Batch processing of DDL statements is orders of magnitude faster than processing each statement separately. However, the file must contain DDL statements only, no data manipulation language (DML) queries such as SELECT or INSERT. If the file contains both DDL and DML, use the `file --inlinebatch` directive in the file to delineate batches of DDL statements then process the file with the `--file` qualifier only.

--charset=*{character-set-name}*

Specifies the character encoding to use when interpreting file input (using the `--file` qualifier or FILE directive) and when writing file output (using the `--output-file` qualifier). If no character set is specified, **sqlcmd** uses UTF-8 for file input and output. Note that the `--charset` qualifier does not affect text entered at the **sqlcmd** prompt, which is always interpreted using the session's default character set, as is also true of output displayed on the terminal.

--credentials=*{properties-file}*

Specifies a file that lists the username and password of the account to use when connecting to a database with security enabled. This is useful when writing shell scripts because it avoids having to hardcode the password as plain text in the script. The credentials file is interpreted as a Java properties file defining the properties `username` and `password`. For example:

```
username: johndoe
password: 4tUn8
```

Because it is a Java properties file, you must escape certain special characters in the username or password, including the colon or equals sign.

--file=*{file-spec}*

Executes the SQL statements in the specified file and then returns to the shell prompt.

--help

Displays the sqlcmd help text then returns to the shell prompt.

--kerberos=*{service-name}*

Specifies the use of kerberos authentication when connecting to the database server(s). The service name identifies the Kerberos client service module, as defined by the JAAS login configuration file.

--output-file=*{file-spec}*

Write the output from sqlcmd commands and statements to the specified file. Note that only the out of the commands are written to the file, errors and informational messages (such as how many rows were affected) are still written to the terminal. By default all output is written to the terminal.

--output-format={csv | fixed | tab}

Specifies the format of the output of query results. Output can be formatted as comma-separated values (csv), fixed monospaced text (fixed), or tab-separated text fields (tab). By default, the output is in fixed monospaced text.

--output-skip-metadata

Specifies that the column headings and other metadata associated with query results are not displayed. By default, the output includes such metadata. However, you can use this argument, along with the `--output-format` argument, to write just the data itself to an output file.

--port=*{port-num}*

Specifies the port number to use when connecting to the database servers. All servers must be using the same port number. By default, sqlcmd connects to the standard client port (21212).

--query=*{text}*

Submits the specified text as an SQL query to the database, displays the results and returns to the shell prompt.

--query-timeout=*{time-limit}*

Specifies a time limit for read-only queries. Any read-only queries that exceed the time limit are canceled and control returned to the user. Specify the time out as an integer number of milliseconds. The default timeout is set in the cluster configuration (or set to 10 seconds by default). Only users with ADMIN privileges can set a sqlcmd timeout longer than the cluster-wide setting.

--servers=*{server-id}[,...]*

Specifies the network address of one or more nodes in the database cluster. When specifying an IPv6 address, enclose the address in square brackets. By default, sqlcmd attempts to connect to a database on localhost.

--ssl[=*ssl-config-file]*

Specifies the use of TLS encryption when communicating with the server. Only necessary if the cluster is configured to use TLS encryption for the external ports. See the section called "Using CLI Commands with TLS/SSL" for more information.

--stop-on-error=*{true|false}*

When processing file input (with the `--file` qualifier or the `file` directive) stops any further execution when an error occurs, if set to true. If false, the statement causing the error is ignored and processing continues to the next statement in the file. The default is true.

--user=*{user-id}*

Specifies the username to use for authenticating to the database. The username is required if the database has security enabled.

--password=*{text}*

Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument but not the --password argument, VoltDB prompts for the password.

Example

The following example demonstrates an sqlcmd session, accessing the voter sample database running on node zeus.

```
$ sqlcmd --servers=zeus
SQL Command :: zeus:21212
1> select * from contestants;
 1 Edwina Burnam
 2 Tabatha Gehling
 3 Kelly Clauss
 4 Jessie Alloway
 5 Alana Bregman
 6 Jessie Eichman

(6 row(s) affected)
2> select sum(num_votes) as total, contestant_number from
v_votes_by_contestant_number_State group by contestant_number
order by total desc;
TOTAL   CONTESTANT_NUMBER
------- ------------------
 757240                  1
 630429                  6
 442962                  5
 390353                  4
 384743                  2
 375260                  3


(6 row(s) affected)
3> exit
$
```

voltadmin

voltadmin — Performs administrative functions on a VoltDB database.

Syntax

```
voltadmin defrag [--full | --maxcount={block-count}] [--tables={table-name[,..]}]

voltadmin dr drop

voltadmin dr protocol [--update]

voltadmin dr reset [--all | --cluster={cluster-id} [--force]]

voltadmin export release --source={source-table} --target={export-target}

voltadmin get [configuration-property [...]]

voltadmin help [command]

voltadmin inspect

voltadmin jstack [server-id]

voltadmin license {license-file}

voltadmin log4j {log4j-configuration-file}

voltadmin log4j {component}={level}

voltadmin note {text}

voltadmin pause [--wait [--timeout={seconds}]]

voltadmin resize [--ignore=disabled_export] [--delay={minutes}] [--restart | --status | --test | --update]

voltadmin restore [--skiptables={table-name[,..]}] [--tables={table-name[,..]}]

voltadmin resume

voltadmin save [{directory} {unique-id}] [--format={csv|native}] [--blocking}]
        [--skiptables={table-name[,..]}] [--tables={table-name[,..]}]

voltadmin set [--file=configuration-file] [configuration-property=value [...]]

voltadmin show [license|snapshots]

voltadmin shutdown [--cancel|--force|--save] [--timeout={seconds}]

voltadmin ssl reload

voltadmin status [--continuous] [--dr] [--json]

voltadmin stop {server-id} [--force]

voltadmin update {configuration-file}
```

```
global qualifiers:
        --credentials={properties-file}
        --help
        --host={server-id}
        --kerberos
        --password={text}
        --ssl={ssl-config-file}
        --user={user-id}
```

Description

The voltadmin command allows you to perform administrative tasks on a VoltDB database. You specify the database server to access and, optionally, authentication credentials using arguments to the voltadmin command. Individual administrative commands may have they own unique arguments as well.

Arguments

The following global arguments are available for all voltadmin commands.

--credentials=*{properties-file}*

Specifies a file that lists the username and password of the account to use when connecting to a database with security enabled. This is useful when writing shell scripts because it avoids having to hardcode the password as plain text in the script. The credentials file is interpreted as a Java properties file defining the properties `username` and `password`. For example:

```
username: johndoe
password: 4tUn8
```

Because it is a Java properties file, you must escape certain special characters in the username or password, including the colon or equals sign.

-h, --help

Displays information about how to use a command. The --help flag and the help command perform the same function.

-H, --host=*{server-id}[:port]*

Specifies which database server to connect to. You can specify the server as a network address or hostname. When specifying an IPv6 address, enclose the address (exclusive of the optional colon and port number) in square brackets. By default, voltadmin attempts to connect to a database on localhost. You can optionally specify the port number. If you do not specify a port, voltadmin uses the default admin port.

--kerberos

Specifies the use of Kerberos authentication when connecting to the database. You must login to your Kerberos account using kinit before issuing the voltadmin with this argument.

-p, --password=*{text}*

Specifies the password to use when connecting to the database. You must specify a username and password if security is enabled for the database. If you specify a username with the --user argument

but not the --password argument, VoltDB prompts for the password. This is useful when writing shell scripts because it avoids having to hardcode passwords as plain text in the script.

--ssl=*{ssl-config-file}*

Specifies the use of TLS encryption when communicating with the server. Only necessary if the cluster is configured to use TLS encryption for the external ports. See the section called “Using CLI Commands with TLS/SSL” for more information.

-u, --user=*{user-id}*

Specifies the username to use for authenticating to the database. The username is required if the database has security enabled.

-v, -verbose

Displays additional information about the specific commands being executed.

Commands

The following are the administrative functions that you can invoke using voltadmin.

defrag

Performs a defragmentation operation on the memory used to store database records. By default, one iteration of memory compaction is performed using the standard settings for memory management. However, you can use optional arguments to the command to control how much compaction is done and which tables are affected. See the chapter on "Understanding Memory Usage" in the *Volt Guide to Performance and Customization* for more information about how memory compaction is done and the minimum requirements and controls available for managing the defragmentation process.

-f, --full

Compacts all available spaces, assuming the partition's table data meets the minimum requirements for defragmentation.

--maxcount=*{block-count}*

Specifies the maximum number of blocks' worth of tuples that will be moved for each table and partition. For example, if the table schema allows 500 tuples per block and you specify a count of 2, at most 1,000 tuples will be moved per table and partition during the compaction event.

--tables=*{table-name} [,...]*

Specifies which tables to defragment.

The --full and --maxcount options are mutually exclusive. If you specify both, --maxcount is ignored.

help [*command*]

Displays information about the usage of individual commands or, if you do not specify a command, summarizes usage information for all commands. The **help** command and **--help** qualifier are synonymous.

dr drop

Removes the current cluster from an XDCR environment. Performing a drop breaks existing DR connections, deletes pending binary logs and stops the queuing of DR data on the current cluster. It

also tells all other clusters in the XDCR relationship to drop their connection to the current cluster and remove any associated binary logs for that cluster.

This command will wait until all other clusters respond before returning to the shell prompt. If one (or more) of the clusters are unreachable, the command will periodically report which clusters it is waiting for. Be aware that if you CTRL-C out of the command before it returns to the shell prompt, one or more of the remote clusters will not have received the appropriate message and will not have cleared their logs for the targeted cluster. In that case, you need to clear that cluster's queues manually after it comes back online using the **dr reset --cluster** command.

The **dr drop** command lets you effectively remove a single cluster — the cluster on which the the command is executed — from a multi-cluster XDCR environment in a single command.

dr protocol [--update]

Reports the current DR protocol that is in use and that is supported by each of the clusters with a DR connection to the current cluster. This information is useful when upgrading XDCR clusters and determining if they are capable of using dynamic schema change or not.

If all of the clusters support a higher protocol version than the current cluster is using, you can use the --update argument to have the cluster move to the higher protocol version.

dr reset

Resets the database replication (DR) connection(s) for the database. Performing a reset breaks existing DR connections, deletes pending binary logs and stops the queuing of DR data on the current cluster.

After a reset, DR must start over from scratch; it cannot be restarted where it left off. Similarly, if there are two clusters in an XDCR environment, you can use **dr reset** for one cluster to drop the connection to the other cluster.

If you are using multiple XDCR clusters, the **dr drop** command is the recommended way to remove a running cluster from the environment. Otherwise, if you use the **dr reset** command you must choose between removing the connections to all other clusters or just one cluster using the following options. You must issue the appropriate command on all applicable clusters:

--all

Resets DR connections and queues to all other clusters on the current cluster. Choose this option if you want the current cluster to survive and then restart DR from scratch on the other remaining clusters.

--cluster=*{remote-cluster-ID}*

Drops the connection to just one cluster. Specify the ID of the remote cluster you wish to drop from the XDCR environment as an argument to the --cluster option. For example, if one cluster has stopped and you want to remove it from the XDCR environment, you can reset the connections to that cluster by issuing the **dr reset --cluster=*{id}*** command on all the remaining clusters. You must also specify --force when you specify --cluster.

--force

Verifies that you want to drop one cluster from a multi-cluster XDCR environment. There is a risk, when a cluster fails, that it has not sent the same binary logs to all other clusters. In this situation, if you drop the one cluster from the XDCR environment, the remaining clusters can diverge. Which is why you must confirm that you really want to drop just one cluster.

Stopping the remote cluster with an orderly shutdown (**voltadmin shutdown**) ensures that all binary logs are delivered. So it is then safe to do a **dr reset** with --cluster and --force. Otherwise, the recommended approach is to choose one cluster as the source, stop all DR connections from that cluster, then restart DR from scratch on the remaining clusters. However, you can, if you choose, use --force to drop the one cluster if you are sure no divergence has occurred.

The --all and --cluster options are mutually exclusive.

export release --source=*{source-table}* --target=*{export-target}*

Resets any blocked export queues, resuming export at the next available export record. You must specify both of the **export release** qualifiers:

-s, --source=*{source-table}*

Specifies the source stream of the table whose queue you want to reset.

-t, --target=*{export-target}*

Specifies the export target you want to reset.

get *[configuration-property [...]]*

Displays information about the configuration of the database. If you specify a specific property (such as `deployment.cluster.kfactor`), it displays the value of that property. If you specify the name of a parent in the property hierarchy (such as `deployment.export`), it displays all of the configuration values under that branch. If you do not specify a property or specify the top-most parent, `deployment`, all of the configuration properties are displayed.

inspect

Displays information about the software, license, and cluster operating environment. Primarily used when communicating with customer support.

jstack *[{server-id}[:port]]*

Saves the current state of all Java threads on one or more of the cluster nodes. If you specify a server on the command line, Jstacks are taken for that node only. If you do not specify a server, Jstacks are saved on all nodes of the cluster. The Jstack files are saved in the `thread_dumps` subfolder under the database root directory. This command is primarily for use when working with VoltDB support to debug application or database issues.

license *{license-file}*

Updates the software license for the database. After validating the license matches the current configuration, the license is saved to the database root directory for each node in the cluster.

log4j *{configuration-file}*

Updates the logging configuration for the cluster. You specify the new configuration as a Log4j version 2 properties file.

log4j *{component}={level}*

Changes the logging level of a Volt component. Enter the argument, *component=level*, as a single text string with no spaces.

note *{text}*

Writes the specified text message to the VoltDB log file. When security is enabled, the user must have admin permissions to write to the log file.

pause [--wait [--timeout=*{seconds}*]]

Pauses the database, stopping any additional activity on the client port. Normally, pause returns immediately. However, you can use the **--wait** flag to have the command wait until all pending transactions are processed and all database replication (DR) and export queues are flushed. Use of **--wait** is recommended if you are shutting down the database and do not intend to restart with recover, since **--wait** ensures all associated DR or export data is delivered prior to shutdown.

Since it is possible that lost connections to external systems or other abnormal conditions can cause queues to hang, the **pause --wait** command waits for up to two minutes if transactions are pending but not being cleared. After two minutes of inactivity, the command times out and stops waiting, leaving the database in a paused state. You can change the timeout period by using the **--timeout** (or **-t**) flag and specifying a different timeout period in seconds.

If the **pause --wait** command times out, review any error messages to determine the cause of the delay. Once you correct the problem, you can either reissue the **pause --wait** command or check the @Statistics system procedure results to make sure all pending transactions and queues are clear.

resize

Reduces the size of a cluster by removing one or more nodes. The number of nodes removed depends on the K-safety factor. See Section 9.3.2, “Removing Nodes with Elastic Scaling” for more information on resizing clusters. Without any action qualifiers, the **voltadmin resize** command tests to ensure a resize is possible, reports which nodes will be removed, and starts the resize process There are two options that affect how the resizing is performed:

--delay=*{minutes}*

Specifies the number of minutes to wait before shutting down each node. Setting a delay of at least 1 minute is recommended when there are topics active in the database. The default value is -1, or no delay.

--ignore=disabled_export

Ignores any pending data for export targets that are disabled when performing the resize process. Normally, resize waits for all export queues to drain before starting the resize process, even if the target is currently disabled in the database configuration.

There are four action qualifiers that alter what resize operation is performed. These qualifiers are mutually exclusive:

--restart

Restarts the resize process after an unexpected failure.

--status

Reports on the current status of the resize operation.

--test

Tests to see if the cluster has enough nodes to perform a resize operation while retaining its K-safety factor. If so, it reports which nodes will be removed during resizing.

--update

Updates the options, such as --ignore and --delay, during a resize operation.

resume

Resumes normal database operation after a pause.

save [*{directory} {unique-ID}*]

Creates a snapshot containing the current database contents. Snapshot files are saved to each server in the cluster, depending on the format used. (Native format snapshots save snapshot files on every server. CSV format snapshots are deduplicated, so only one copy of the data is saved. On K-safe clusters, this means only certain servers may be used to save the resulting files.)

If you use **save** without any arguments, the snapshot is saved into the database's snapshots directory where it can automatically be restored the next time the database starts. If you specify an alternate directory and ID, the snapshot files are saved to the specified path using the unique ID as a file prefix.

When saving into the default snapshots directory, VoltDB automatically performs a full snapshot in native mode. The following are additional arguments you can specify when saving to a specific location and unique ID. (Only the --blocking argument is allowed when saving to the default snapshots directory.)

--format={ csv | native }

Specifies the format of the snapshot files. The allowable formats are CSV (comma-separated value) and native, with native being the default. Native format snapshots can be used for restoring the database. CSV files can be used by other utilities (such as spreadsheets or the VoltDB CSV loader) but cannot be restored using the **voltadmin restore** command.

--blocking

Specifies that the snapshot will block all other transactions until the snapshot is complete. The advantage of blocking snapshots is that once the command completes you know the snapshot is finished. The disadvantage is that the snapshot blocks ongoing use of the database.

By default, voltadmin performs non-blocking snapshots so as not to interfere with ongoing database operation. However, note that the non-blocking **save** command only starts the snapshot. You must use **show snapshots** to determine when the snapshot process is finished if you want to know when it is safe, for example, to shutdown the database.

--skiptables=*{table-name} [,...]*

Specifies one or more tables to leave out of the snapshot. Separate multiple table names with commas.

--tables=*{table-name} [,...]*

Specifies what table(s) to include in the snapshot. Only the specified tables will be included. Separate multiple table names with commas.

restore *{directory} {unique-ID}*

Restores the data from a snapshot to the database. The data is read from a snapshot using the same unique ID and directory path that were used when the snapshot was created. If no tables exist in the

database (that is, no schema has been defined) the restore command will also restore the original schema, including stored procedure classes, before restoring the data.

The following arguments let you selectively include or exclude data from certain tables during the restore operation.

--skiptables=*{table-name} [,...]*

Data for the specified tables is *not* restored. All other tables are restored. Separate multiple table names with commas.

--tables=*{table-name} [,...]*

Only data for the specified tables is restored. Data for all other tables is ignored. Separate multiple table names with commas.

Note that if the database is empty (that is, has no existing schema), the full schema from the snapshot is always loaded even if you choose not to load the data for certain tables. Also, you can specify either --skiptables or --tables but not both on the same command.

set [--file=*configuration-file*] [*configuration-property={value} [...]*]

Updates or adds the configuration option with the specified value. You can include multiple configuration options separated by spaces. The properties are specified in dot notation. You can also set multiple properties using --file to specify a YAML file of configuration properties.

The **set** command is *cumulative*. Values for existing properties are replaced and new properties are added to the existing configuration. For example, specifying a value for deployment.systemsettings.query.timeout sets the value for query timeout but does not alter other system settings. Similarly, if you specify a new export configuration it is added to the existing configurations. See the **voltadmin update** command for instructions on *replacing* the existing configuration.

Not all configuration settings can be updated while the database is running. See Section E.2, “YAML Properties for Configuring Volt Databases on Bare Metal” for information on which options can and cannot be updated on the fly.

show license

Displays information about the cluster's current license.

show snapshots

Displays information about up to ten previous snapshots. This command is useful for determining the success or failure of snapshots started with the **save** command.

ssl reload

Allows the update or replacement of the TLS/SSL certificate keystore and truststore files. Place new files in the same location (using the identical file name and directory) specified in the `deployment.ssl.keystore.path` and `deployment.ssl.truststore.path` configuration properties. Then use the **ssl reload** command to update the server. This command also reloads any certificate revocation lists (CRLs) in the `deployment.ssl.crls.path` directory.

status

Displays information on the state of the cluster, such as the number of nodes and uptime. You can use the following options to customize the content and presentation of the status information:

--dr

Adds information about the current status of data replication to the display.

-j, --json

Outputs the information in JSON format.

--continuous

Specifies that the information be continuously updated until you interrupt the command (with CTRL-C, for example).

update *{configuration-file}*

Updates the configuration on a running database with the values in the configuration file. The specified configuration replaces the current configuration. That is, values specified in the configuration file replace the current values. Configuration options not specified in the configuration file are reset to their default values, even if they were set explicitly in the previous configuration. Alternately, you can use the **voltadmin set** command to change or add individual properties.

There are limitations on what changes can be made to the configuration of a running database cluster. See Section E.2, "YAML Properties for Configuring Volt Databases on Bare Metal" for information on the allowable changes. The update either succeeds or fails as a whole. If any of the requested changes are for properties that cannot be set on a running database, no changes are made to the configuration.

stop [--force] *{server-id}*

Stops an individual node in the cluster. The **voltadmin stop** command can only be used on a K-safe cluster and will not intentionally shutdown the database. That is, the command will only stop a node if there are enough nodes left for the cluster to remain viable.

By default, the **stop** command waits for all partition and export leadership on the specified node to be redistributed to other nodes in the cluster in an orderly fashion before stopping the node. You can use the **--force** argument to stop the node immediately. However, if you force the node to stop, the remainder of the cluster must negotiate leadership after the node stops, which can have several negative effects. The advantages of using the default, orderly stop command are:

- In-flight transactions queued to the stopped node are completed and returned to the client. A forced stop interrupts these transactions resulting in lost connection and other errors being returned to the clients.
- Stopping the node has reduced impact on the ongoing transactions and workload for the database. A forced stop disrupts ongoing transactions while the cluster negotiates the migration of partition leadership.
- Export queues are transitioned correctly, avoiding gaps and potentially lost export data that can result if nodes are interrupted and restarted in quick succession.

shutdown [--force | --save | --cancel] [--timeout=*{seconds}*]

Shuts down the database process on all nodes of the cluster. By default, voltadmin shutdown performs an orderly shutdown, pausing the database, completing all pending transactions and writing any queued export, import, or DR data to disk before shutting down the database. You can also use one of the following arguments to modify the behavior of shutdown:

--force

Stops the database immediately. If you do not need to save any in-process work, you can use the `--force` argument to stop the database immediately.

--save

Specifies that not only will all data be made durable, all pending DR and export data will be sent to the corresponding external systems and a final snapshot will be taken before the cluster is shutdown. The resulting snapshot will be used, in place of command logs, the next time the database is started with the **voltdb start** command. Using the final snapshot on startup permits changes not normally allowed by command logs, such as upgrading the VoltDB software.

--cancel

Cancels a pending shutdown. The **shutdown --save** command can be blocked if the targets for pending DR or export are currently unavailable. If this happens, you can do a CTRL-C to interrupt the **shutdown --save** command, but that does not cancel the shutdown itself and your database is not operational. The **shutdown --cancel** command cancels the shutdown operation and returns the database to an operational state.

Since it is possible that lost connections to external systems or other abnormal conditions can cause queues to hang, the **shutdown** command (without the **--force** flag) waits for up to two minutes if transactions are pending but not being cleared. After two minutes of inactivity, the command times out, leaving the database in a paused state but not shutdown. You can change the timeout period by using the **--timeout** (or **-t**) flag and specifying a different timeout period in seconds.

If the **shutdown** command times out, review any error messages to determine the cause of the delay. You can:

- Do a **shutdown --cancel** to cancel the shutdown, correct the problem, then reissue the **shutdown** command
- Do a **shutdown --cancel** to cancel the shutdown and resume normal database operations
- Do a **shutdown --force** to initiate an immediate shutdown

Note that if you do a **shutdown --force** after a **shutdown --save** command, the system will *not* have created a final snapshot.

Examples

The following example performs an orderly shutdown.

```
$ voltadmin shutdown
```

The next example uses **pause** and **save** to create a snapshot of the database contents as a backup before shutting down.

```
$ voltadmin pause --wait
$ voltadmin save --blocking ./ mydb
$ voltadmin shutdown
```

The last example uses the **shutdown --save** command to create a snapshot of the database contents, similar to the previous example. However, in this case, the snapshot that is created will be used automatically to restore the database on the next **start** command.

```
$ voltadmin shutdown --save
```

voltdb

voltdb — Performs management tasks on the current server, such as starting and recovering the database.

Syntax

```
voltdb collect [args]

voltdb get classes [args]

voltdb get deployment [args]

voltdb get schema [args]

voltdb mask [args] source-configuration-file [new-configuration-file]

voltdb init [args]

voltdb start [args]
```

Description

The voltdb command performs local management functions on the current system, including:

- Initializing the database root directory and setting configuration options
- Starting the database process
- Collecting log files into a single compressed file
- Retrieving the classes, deployment, or schema from a database root directory
- Hiding passwords in the configuration file

The action that is performed depends on which start action you specify to the voltdb command:

- **collect** — the collect option collects system and process logs related to the VoltDB database process on the current system and compresses them into a single file. This command is helpful when reporting problems to VoltDB support.
- **get** — the get option retrieves the current configuration, procedure classes, or schema from the database root directory. The requested item is then written to a file. This command can be used whether the database is running or not. You can use options to specify either or both the parent of the root directory (`--dir`) or the name and location of the output file (`--output`). Note that the get option can only be used on databases created using init and start.
- **mask** — the mask option disguises the passwords associated with user accounts in the security section of the configuration file. The output of the voltdb mask command is either a new configuration file with hashed passwords or, if you do not specify an output file, the original input file is modified in place.
- **init** — the init option initializes the root directory VoltDB uses for storing the configuration, logs, and other disk-based information (such as snapshots and command logs) for the database process. You only

need to initialize the root directory once. After that, VoltDB manages the content and selecting the appropriate start actions to maintain the database state. If you choose to re-initialize an existing root directory, you can use the --force argument to delete any previous data.[1]

- **start** — the starts option starts the database process after the root directory has been initialized. The actual action that VoltDB takes depends on the current state of the database cluster:
  - If this is the first time the database has started, it creates a new database.
  - If the database has run before and is configured to use command logs or there is at least one snapshot in the snapshots directory, the database is restarted and previous data recovered.
  - If the cluster is already running and a server is missing (assuming the use of K-safety) the current node will rejoin the running cluster.
  - If the cluster is already running with all servers present, the current node will be added to expand the size of the cluster — as long as you use the --add argument on the start command.

The **voltdb start** command uses Java to instantiate the process. It is possible to customize the Java environment, if necessary, by passing command line arguments to Java through the following environment variables:

- **LOG4J_VOLTSERVER_PATH** — Specifies an alternate Log4J configuration file.
- **VOLTDB_GC_OPTS** — Lets you specify which Java garbage collector to use and other GC-related options. Specify the options using standard Java -XX format. For example:

  ```
  export VOLTDB_GC_OPTS="-XX+useG1GC -XX+UseStringDeduplication"
  ```

- **VOLTDB_HEAPMAX** — Specifies the maximum heap size for the Java process. Specify the value as an integer number of megabytes. By default, the maximum heap size is set to 2048.
- **VOLTDB_OPTS** — Specifies all other Java command line arguments. You must include both the command line flag and argument. For example, this environment variable can be used to specify system properties using the -D flag:

  ```
  export VOLTDB_OPTS="-DmyApp.DebugFlag=true"
  ```

Log Collection (voltdb collect) Arguments

The following arguments apply specifically to the **collect** action.

-D --dir=*{directory}*

Specifies the parent location for the database root directory from which to collect information. The default, if you do not specify a directory, is the current working directory.

--days=*{integer}*

Specifies the number of days of log files to collect. For example, using `--days=1` will collect data from the last 24 hours. By default, VoltDB collects 14 days (2 weeks) worth of logs.

[1]The **init --force** command deletes command logs and overflow subfolders within the database root directory. However, to avoid accidentally deleting backups, the snapshots subfolder is renamed rather than deleted. This way, it is possible to restore a snapshot in case of an unintended re-initialization. On the other hand, this means you should periodically check your database root directories and purge any archived snapshots folders (named `snapshots.`*nn*) that are no longer needed.

--dry-run

Lists the actions that will be taken, including the files that will be collected, but does not actually perform the collection or upload.

--no-prompt

Specifies that the process will not prompt for input, such as whether to delete the output file after uploading is complete. This argument is useful when starting the collect action from within a script.

--output=*{file}*

Specifies the name and location of the resulting output file. The default output file name starts with "voltdb_collect_" and includes the current server IP or hostname, with a file extension of ".zip" saved to the current working directory.

--skip-heap-dump

Specifies that the heap dump not be included in the collection. The heap dump is usually significantly larger than the other log files and can be excluded to save space.

Get Resource (voltdb get) Arguments

The following arguments apply specifically to the **get classes**, **get deployment**, and **get schema** actions.

-D --dir=*{directory}*

Specifies the parent location for the database root directory. The default, if you do not specify a directory, is the current working directory.

-f, --force

Allows the command to overwrite an existing file. By default, the get actions will *not* overwrite existing files.

-o --output=*{file-path}*

Specifies the name and, optionally, location for the resulting output file. The default location is the current working directory. The default file depends on the resource being requested:

- `procedures.jar` for **get classes**
- `deployment.yaml` for **get deployment**
- `schema.sql` for **get schema**

In addition, the following arguments are specific to get deployment and are mutually exclusive:

--xml

Specifies that the output be in XML format.

--yaml

Specifies that the output be in YAML format.

If neither the `--xml` or `--yaml` argument is specified, the default output format is YAML. However, if the format is not specified but the output file is and the specified file has ".xml" as its file type, the output is generated as XML.

Initialization (voltdb init) Arguments

The following arguments apply to the **voltdb init** command.

-C, --config=*{configuration-file}*[,...]

Specifies the location of one or more database configuration files. The configuration files are YAML files that define the logical structure of the database , including which options to enable when the database starts. See Appendix E, *YAML Configuration Properties* for a complete description of the syntax of the configuration file. Use of YAML properties is recommended. However, you can specify a single XML file as an alternative method for specifying the configuration.

If you do not specify a configuration file, default is a configuration that includes command logging (where available), no K-safety, and eight sites per host.

-D --dir=*{directory}*

Specifies the parent location for the database root directory. The root directory is named `voltdb-root` and is created if it does not already exist in the specified location. If a `voltdbroot` directory *does* already exist, you must use the --force argument to override any existing data. The default, if you do not specify a directory, is the current working directory.

-f, --force

Initializes the database root directory, even if files (such as command logs or snapshots) already exist in the specified directory. Initializing the root directory after previously running a database could overwrite and therefore erase old command logs. Therefore, VoltDB will not, by default, initialize the database if such files exist. If you do not need the files from the previous session, you can use the --force argument to overwrite these files.

-j, --classes=*{JAR-file} [, ...]*

Specifies the location of one or more JAR files containing classes used to declare user-defined stored procedures. The JAR files (and any schema definitions included with the --schema argument) are loaded automatically when the database starts. Separate multiple file names with commas. You can also use asterisk (*) as a wildcard character in the file specification. If durability is enabled (through command logs or a shutdown snapshot) the classes specified on the init command are loaded only the first time the database starts and the command logs are used for subsequent starts. If no durability is provided, the initialized classes are loaded on every start.

-l, --license=*{license-file}*

Specifies the location of the license file, which is required. If no license is specified, Volt looks for a file named `license.xml` in the current working directory, the `/voltdb` subfolder where the VoltDB software is installed, or the current user's home directory.

-r, --retain=*{integer}*

Specifies the maximum number of snapshot directories to save when performing a `voltdb init --force`. When initializing a root directory with `--force`, VoltDB deletes all previous files in the directory except the *snapshot* subfolder, which is renamed *snapshots.1*, *snapshots.2*, and so on. By default, VoltDB saves only two older snapshot folders. The `--retain` argument lets you specify a different maximum number of folders to save.

-s, --schema=*{schema-file} [, ...]*

Specifies the location of one or more files containing database definition language (DDL) statements. The DDL statements (and any classes included with the --classes argument) are loaded automatically

when the database starts. Separate multiple file names with commas. You can also use asterisk (*) as a wildcard character in the file specification. If durability is enabled (through command logs or a shutdown snapshot) the schema specified on the init command is loaded only the first time the database starts and the command logs are used for subsequent starts. If no durability is provided, the initialized schema is loaded on every start.

Database Startup (voltdb start) Arguments

The following arguments apply to the **voltdb start** command.

-D --dir=*{directory}*

Specifies the parent location for the database root directory. This is the same directory specified on the **voltdb init** command. (You must initialize the root directory before you can start the database.) The default, if you do not specify a directory, is the current working directory.

-H, --host=*{host-id} [,...]*

Specifies the network address of one or more nodes in the database cluster. VoltDB selects one of these nodes to coordinate the start of the database or the adding or rejoining of servers. When starting a database, all nodes must specify the same list of host addresses. Note that once the database starts and the cluster is complete, the role of the host node is complete and all nodes become peers.

When rejoining or adding a server to a running cluster, you can specify any node(s) still in the cluster. The host for an add or rejoin operation does *not* have to be the same node specified when the database started.

The default if you do not specify a host when creating or recovering the database is `localhost`. In other words, a single node cluster running on the current system. You must specify a host on the command line when adding or rejoining a node or when starting a cluster.

If the host node is using an internal port other than the default (3021), you must specify the port as part of the host string, in the format host:port.

When used in conjunction with the --missing flag, the first host in the list must be one of the current hosts, *not* one of the missing nodes.

-c, --count=*{number-of-nodes}*

Specifies the number of nodes in the database cluster.

--add

When joining a running cluster, specifies that the new node can be "added", elastically expanding the size of the cluster. The --add flag only takes affect when a node is joining a complete, running cluster. If the cluster is starting or if a node is missing from a K-safe cluster, the current node will join the cluster as normal. But if the cluster is already running and has its full complement of members, you must specify --add if you want to increase the size of the cluster.

-B, --background

Starts the server process in the background (as a daemon process).

-g, --placement-group=*{group-name}*

Specifies the location of the server. When the K-safety value is greater than zero, VoltDB uses this argument to assist in rack-aware partitioning. The cluster will attempt to place multiple copies of each

partition on different nodes to keep them physically as far apart as possible. The physical location is specified by the *group-name*, which is an alphanumeric name. The names might represent physical servers, racks, switches, or anything meaningful to the user to avoid multiple copies failing at the same time.

To be effective, placement groups must adhere to the following rules:

- There must be more than one placement group specified for the cluster.
- The number of nodes must be a multiple of the number of placement groups.
- The number of placement groups must be a multiple of K+1.

Otherwise, there are no guarantees the partitions will be evenly distributed.

--ignore=thp

For Linux systems, allows the database to start even if the server is configured to use Transparent Huge Pages (THP). THP is a known problem for memory-intense applications like VoltDB. So under normal conditions VoltDB will not start if the use of THP is enabled. This flag allows you to ignore that restriction for test purposes. *Do not use this flag on production systems.*

--missing=*{number-of-nodes}*

Allows a K-safe cluster to start without the full complement of nodes. This argument specifies how many nodes are missing from the cluster at startup. For example, if the arguments are `--count=5` and `--missing=2`, then the database will start once three nodes join the cluster, assuming those nodes can support at least one copy of each partition. Note that use of the `--missing` option means that the cluster is *not fully K-safe* until the specified number of missing nodes rejoin the cluster after the database starts. Also, the `--hosts` flag should list currently available hosts, not the missing nodes.

--pause

For the create and recover operations only, starts the database in admin mode. Admin mode stops applications from performing write operations to the database through the client interface. This is useful when performing administrative functions such as restoring a snapshot before allowing client access. Once all administrative operations are complete, you can use the **voltadmin resume** command to resume normal operation for the database. If any nodes in the cluster start with the `--pause` switch, the entire cluster starts paused.

--safemode

When using command logs to recover an existing database that cannot recover under normal circumstances, the `--safemode` argument recovers the database to the last valid transaction. This argument should only be used when troubleshooting a failed recovery. See the description of safe mode recovery in the *VoltDB Administrator's Guide* for details.

Network Configuration Arguments

In addition to the arguments listed above for the **voltdb start** command, there are additional arguments that specify the network configuration for server ports and interfaces when starting a VoltDB database. In most cases, the default values can and should be accepted for these settings. The exceptions are the external and internal interfaces that should be specified whenever there are multiple network interfaces on a single machine.

You can choose to set network ports and interfaces either using individual arguments for each port or as a YAML file identified by the `--network` qualifier. When specifying network settings in YAML, you identify the type of port followed by individual properties for the port number and the network interface, either of which can be defaulted. For example, the following YAML network configuration file shows all of the possible settings. Of course, normally you only need to specify those settings that are *different* from the default.

```
network:
    externalinterface: 192.168.0.100
    publicinterface: 192.168.0.200
    admin:
        address: 192.168.0.100
        port: 21211
    client:
        address: 192.168.0.100
        port: 21212
    drpublic:
        address: 192.168.0.100
        port: 5555
    internal:
        address: 192.168.0.100
        port: 3021
    metrics:
        address: 192.168.0.100
        port: 11781
    replication:
        address: 192.168.0.100
        port: 5555
    topics:
        address: 192.168.0.100
        port: 9092
    topicspublic:
        address: 192.168.0.100
        port: 9092
    zookeeper:
        address: 127.0.0.1
        port: 7181
```

When specifying network ports using individual command line arguments, you can optionally specify a unique network interface by preceding the port number with the interface's IP address (or hostname) followed by a colon. Specifying the port and/or network interface for an individual port setting overrides the default interface for that port, the interface set by `--externalinterface` or `--internalinterface`, and any properties defined in a YAML file specified using the `--network` qualifier.

The network configuration arguments to the **voltdb start** command are listed below. See the appendix on server configuration options in the *VoltDB Administrator's Guide* for more information about network configuration options.

--network=*{YAML-file}*

Specifies a YAML file that defines the interfaces and port numbers of one or more network ports and interfaces.

--externalinterface=*{ip-address}*

Specifies the default network interface to use for external ports, such as the admin and client ports.

--internalinterface =*{ip-address}*

Specifies the default network interface to use for internal communication, such as the internal port.

--publicinterface=*{ip-address}*

Specifies the public network interface. This argument is useful for hosted systems where the internal and external interfaces may not be generally reachable from the Internet. In which case, specifying the public interface helps the Volt Management Center provide publicly accessible links for the cluster nodes.

--drpublic=*{ip-address}[:port-number]*

Specifies the publicly advertised network interface and, optionally, port number for database replication (DR) communication. This is the address that is sent from the producer cluster to consumers. This argument is useful for hosted systems where the internal interfaces are not reachable from outside the hosted environment and the producer cluster must return an externally mapped port as the public DR interface to remote consumers.

--admin=*[ip-address:]{port-number}*

Specifies the admin port.

--client=*[ip-address:]{port-number}*

Specifies the client port.

--http=*[ip-address:]{port-number}*

Specifies the http port. The --http flag both sets the port number (and optionally the interface) and enables the http port, overriding the http setting, if any, in the configuration file.

--internal=*[ip-address:]{port-number}*

Specifies the internal port used to communicate between cluster nodes.

--metrics=*[ip-address:]{port-number}*

Specifies the metrics port used for distributing Prometheus-compliant metrics data.

--replication=*[ip-address:]{port-number}*

Specifies the replication port used for database replication. The --replication flag overrides the replication port setting in the configuration.

--topicsport=*[ip-address:]{port-number}*

Specifies the port used for receiving and sending topics data.

--topicspublic=*{ip-address}[:port-number]*

Specifies the network address advertised as the public topics port. For cases where the server's interfaces are not accessible to external systems, the --topicspublic flag identifies a publicly accessible interface and, optionally, an alternative port number.

--zookeeper=*[ip-address:]{port-number}*

Specifies the zookeeper port. By default, the zookeeper port is bound to the server's internal interface (127.0.0.1).

Examples

The first example shows the commands for initializing and starting a three-node database cluster using three custom configuration files, `common.yaml`, `boston.yaml`, and `users.yaml` — and the node zeus as the host. This example demonstrates how multiple XDCR clusters could share a common set of configuration options, then have unique settings for, say, the XDCR settings (including the cluster-specific ID) and the user accounts for the cluster.

```
$ voltdb init --dir=~/mydb --config=configuration.yaml,boston.yaml,users.yaml
$ voltdb start --dir=~/mydb --count=3 --host=zeus
```

The second example takes advantage of the defaults for the host and configuration arguments to initialize and start a single-node database in the current directory.

```
$ voltdb init
$ voltdb start
```

The next example shows the use of the --force argument to re-initialize the directory used in the first example, to delete old data and set new configuration options from a different configuration file.

```
$ voltdb init --dir=~/mydb --config=newconfig.yaml --force
```

Appendix E. YAML Configuration Properties

Configuration properties define the physical layout of a VoltDB database cluster at runtime — including the number of sites per hosts and K-safety value, among other things — as well as what features are in use and how they will operate. This appendix describes the syntax for specifying YAML properties as well as a table of all the supported configuration properties and their meaning.

> **Note**
>
> For a list and description of the equivalent configuration properties when running Volt in Kubernetes, see the appendix of Helm Properties in the *Volt Kubernetes Administrator's Guide*.

E.1. Understanding YAML Syntax

YAML is a simple syntax for describing properties and their values in text files. The advantage of YAML as a syntax is its simplicity and flexibility. YAML files are easy to read and easy to edit with a plain text editor. The properties are defined as a hierarchy, so related properties are grouped into sets of subproperties of a parent element.

There are actually two ways to specify YAML properties:

- In a YAML file using indentation to describe the hierarchy of properties
- On the command line, using dot notation to identify a property and its value on a single line

The following sections provide an overview of both styles of YAML syntax and their uses.

E.1.1. YAML Files

When specifying a property in a YAML file, you represent the property hierarchy as a set of indented keywords on separate lines, each keyword terminated by a colon. The actual value for the final node of the hierarchy follows the colon. For example, the following YAML file defines two properties, both of which are children of the `deployment` and `cluster` elements:

```
deployment:
   cluster:
      sitesperhost: 12
      kfactor: 1
```

Certain elements can contain a list rather than individual subproperties. In this case, the child list elements are preceded by a hyphen. The child elements themselves can be either individual values or further subproperties. For example, the `user` property contains a list of users, each specifying a user name, password and comma-separated list of roles.

```
deployment:
  users:
  - name: Washington
    password: cherrytree
    roles: citizen,president
  - name: Jefferson
    password: monticello
    roles: citizen,administrator
```

One advantage of YAML is that you can modularize the configuration, putting different options into separate files. For example, you might choose to save the security settings separately from the configuration of the individual database features (export, command logging, etc.) that are in use. You can then combine the configuration options on the command line when you initialize the database root directory:

```
$ voltdb init -f -D ~/mydbroot --config=features.yaml,security.yaml
```

For more on the YAML syntax, visit https//yaml.org/.

E.1.2. Dot Notation

YAML is easy to read and easy to edit in text files. However, the multi-line format is not well suited to command line options. As an alternative, you can use dot notation for specifying YAML properties on shell commands, such as **voltadmin get** and **set**.

Dot notation separates the elements of a YAML property with periods. For example, the following command fetches the current value of the `sitesperhost` property, which is a child of `deployment` and `cluster`:

```
$ voltadmin get deployment.cluster.sitesperhost
---
deployment:
  cluster:
    sitesperhost: 12
```

You can retrieve a specific value by specifying the dot notation for the scalar property, as in the preceding example. Or you can display a group of properties by specifying their parent element, like so:

```
$ voltadmin get deployment.cluster
---
deployment:
  cluster:
    sitesperhost: 12
    kfactor: 1
```

You can also specify a new value for a scalar property by using the dot notation for the property followed by an equals sign and the new value, as in the following **voltadmin set** command:

```
$ voltadmin set deployment.systemsettings.query.timeout=5000
INFO: The configuration update succeeded.
```

Of course, there are only some properties that can be changed on the fly (that is, while the database is running). The table of configuration properties in Table E.1, “Complete List of Volt Configuration Properties” includes a checkmark next to those properties that can be changed dynamically.

E.1.2.1. Identifying Individual Elements of a YAML List

Some configuration properties are lists. So normally the only way to see the values for an element of a property would be to show all of the elements of the property. For example, the **voltadmin get deployment.users** command shows you all of the user accounts defined for the database:

```
$ voltadmin get deployment.users
---
deployment:
  users:
  - name: "Washington"
```

```
    roles: "citizen,president"
    password: "cherrytree"
    plaintext: true
    expires: null
  - name: "Jefferson"
    roles: "citizen,administrator"
    password: "monticello"
    plaintext: true
    expires: null
```

However, as a shortcut Volt lets you identify the specific element of the list you want to operate on by specifying the unique identifier for that list in square brackets after the property name. For example, the following command displays the properties associated with the Jefferson account only:

```
$ voltadmin get deployment.users[Jefferson]
---
deployment:
  users:
  - name: "Jefferson"
    roles: "citizen,administrator"
    password: "monticello"
    plaintext: true
    expires: null
```

You can also use the unique name to assign values to other subproperties. For example, the following example changes the list of roles for the Jefferson account:

```
$ voltadmin set deployment.users[Jefferson].roles="citizen,president,administrator
INFO: The configuration update succeeded.
```

In most cases, the unique identifier is the `name` property of the list element (such as for the list of `user` accounts). For export configurations, the unique identifier is the `target` property. For those cases where a unique identifier does not already exist or cannot be used, the YAML schema includes a `nickname` property so you can assign a unique identifier to list elements. The current properties that use nicknames are `deployment.import.configuration` and `deployment.ldap.group`. For example, The following import configuration has assigned a nickname so the properties can be modified individually.

```
$ voltadmin get deployment.import
---
deployment:
  import:
    configuration:
    - nickname: "employees"
      enabled: true
      type: "KAFKA"
      format: "csv"
      version: "10"
      module: null
      priority: 4
      property:
      - name: "broker"
        value: "kafkasvr:9092"
      - name: "topics"
        value: "employees"
      - name: "procedure"
```

```
        value: "EMPLOYEE.insert"
      format-property: []
$ voltadmin set deployment.import.configuration[employees].priority=6
INFO: The configuration update succeeded.
```

Without the nicknames, you would need to save the current configuration to a file (with the **voltadmin get deployment --output=*{filename}*** command), edit the file as desired, then reapply the changes using the **voltadmin update** command.

Note

If a property that allows a nickname does not have one explicitly assigned, Volt will generate one automatically. These auto-generated nicknames are visible when you retrieve the configuration with **voltadmin get**. However, these implicit nicknames are not persistent; they may change any time the import configuration changes. For that reason, it is best to use only user-assigned nicknames when modifying individual import configurations.

E.2. YAML Properties for Configuring Volt Databases on Bare Metal

Table E.1, “Complete List of Volt Configuration Properties” lists all of the configuration properties supported by Volt Active Data on bare metal systems. (For information on configuring Volt Active Data on Kubernetes, see the *Volt Kubernetes Administrator's Guide*.) The table lists the properties in alphabetical order and includes both a description of the property and a checkmark next to those properties that can be updated "on the fly" (that is, while the database is running.) Property names followed by a pair of square brackets indicate properties that are a list of subproperties.

Table E.1. Complete List of Volt Configuration Properties

Property	Live Update	Description
deployment.avro.namespace		Avro namespace (since V11)
deployment.avro.prefix		Avro configuration prefix
deployment.avro.properties		Avro configuration properties
deployment.avro.registry		Avro registry URL
deployment.cluster.kfactor		K-factor to use for database durability and data safety replication
deployment.cluster.sitesperhost		SitesPerHost for VoltDB Cluster
deployment.commandlog.enabled		Command logging for database durability (recommended)
deployment.commandlog.frequency.time	✓	How often the command log is written, by time (milliseconds)
deployment.commandlog.frequency.transactions	✓	How often the command log is written, by transaction commands
deployment.commandlog.logsize	✓	Command logging allocated disk space (MB)
deployment.commandlog.synchronous		Transactions do not complete until logged to disk
deployment.compoundproc.callsperstage		Maximum number of procedure invocations per stage of each compound procedure

Property	Live Update	Description
deployment.compoundproc.queuelimit		Maximum number of compound procedures queued to the compound procedure executor
deployment.compoundproc.threads		Number of threads used by the compound procedure executor
deployment.dr.conflictretention	✓	Automatic pruning of XDCR conflict logs; integer followed by one of m/h/d, for minutes/hours/days(since V11)
deployment.dr.connection.connectiontimeout	✓	Connection timeout for connecting to producer cluster in milliseconds.
deployment.dr.connection.enabled	✓	Specifies whether this DR connection is enabled
deployment.dr.connection.preferredSource	✓	Cluster ID of preferred DR source
deployment.dr.connection.receivetimeout	✓	Timeout receiving response from producer cluster in milliseconds
deployment.dr.connection.source	✓	If role is replica or xdcr: list of host names or IP addresses of remote nodes
deployment.dr.connection.ssl	✓	-
deployment.dr.connection.ssl-host-check		Enable validation of Subject Alternative Name in received certificates
deployment.dr.consumerlimit.maxbuffers	✓	Enable DR consumer flow control; either maxsize or maxbuffers must be specified
deployment.dr.consumerlimit.maxsize	✓	Enable DR consumer flow control; either maxsize or maxbuffers must be specified. maxsize is an integer optionally followed by 'g' (gigabytes) or 'm' (megabytes)
deployment.dr.exportconflicts	✓	Optional setting. If false, XDCR conflict logs are not generated into export stream and therefore are not exported. Conflicts are still handled and shown in statistics.
deployment.dr.flushInterval	✓	-
deployment.dr.id		Unique cluster id, 0-127
deployment.dr.role		Role for this cluster, either xdcr or none
deployment.dr.schemachange.enabled	✓	Enable DR consumer to continue while compatible schema changes are being made (since V12)
deployment.dr.schemachange.truncate	✓	Enable values to be truncated if a VARCHAR column is wider on another cluster while schema changes are being made
deployment.export.configurations[].enabled	✓	Is this export enabled?
deployment.export.configurations[].exportconnectorclass	✓	Class name for custom exporters
deployment.export.configurations[].properties	✓	List of type-specific properties
deployment.export.configurations[].reconnect	✓	Reconnect interval, integer followed by 's' or 'm' (V12 or later)
deployment.export.configurations[].target	✓	Name of export target

Property	Live Update	Description
deployment.export.configurations[].threadpool	✓	Name of thread pool to use for export processing
deployment.export.configurations[].type	✓	Export type (file, custom, etc.)
deployment.export.defaultpoolsize	✓	Size of the default export thread pool (if not set, sites per host value used); only used for Volt V13+
deployment.heartbeat.timeout	✓	Internal VoltDB cluster verification of presence of other nodes (seconds)
deployment.import.configurations[].enabled	✓	Is this import enabled?
deployment.import.configurations[].format	✓	Format of import data
deployment.import.configurations[].formatProperties	✓	List of formatter-specific properties (only used for Volt V13+)
deployment.import.configurations[].module	✓	Name of jar file for custom importer
deployment.import.configurations[].nickname	✓	Arbitrary but unique name for import configuration (Volt V14+)
deployment.import.configurations[].priority	✓	Priority of transactions for this import configuration
deployment.import.configurations[].properties	✓	List of type-specific properties
deployment.import.configurations[].type	✓	Import type e.g. kafka
deployment.import.configurations[].version	✓	The kafka version (only used for Volt V13+)
deployment.metrics.enabled		Enables cloud-native metrics system on each VoltDB pod. It is an alternative to using Prometheus Agent.
deployment.metrics.interval		How often the metrics system prepares a new metrics slice to be sourced by an external system like prometheus. Supported unit indicators are s (seconds), m (minutes), h (hours), d (days).
deployment.metrics.maxbuffersize		Limits memory that the metric system can use for internal metric buffering (system will always retain at least one metrics slice in the buffer). In megabytes.
deployment.partitiondetection.enabled		Controls detection of network partitioning. Setting this false is deprecated in V14, and doing so will be ignored in V15+.
deployment.paths		-
deployment.security.enabled	✓	Controls whether user-based authentication and authorization are used
deployment.security.ldap.group[].name	✓	Distinguished name of LDAP group
deployment.security.ldap.group[].nickname	✓	Arbitrary but unique name for this group entry (Volt V14+)
deployment.security.ldap.group[].role	✓	Comma-list of Volt roles equivalent to this LDAP group
deployment.security.ldap.groupclass	✓	Name of the LDAP schema's objectClass defining a group of users
deployment.security.ldap.groupmemberuid	✓	Name of the LDAP schema's objectClass defining a group of users

Property	Live Update	Description
deployment.security.ldap.password	✓	Password corresponding to LDAP server username, required
deployment.security.ldap.rootdn	✓	Distinguished Name of the root of the LDAP schema that defines users and groups, required
deployment.security.ldap.server	✓	URL for LDAP server, required; as 'ldap://server:port', or 'ldaps://server:port', port optional
deployment.security.ldap.ssl.insecure	✓	-
deployment.security.ldap.ssl.truststore.auxFile		-
deployment.security.ldap.ssl.truststore.file		Truststore file used to validate LDAPS server certificate (Java KeyStore format)
deployment.security.ldap.ssl.truststore.indirectPassword		-
deployment.security.ldap.ssl.truststore.password		Password for LDAP truststore file
deployment.security.ldap.timeout	✓	Timeout, in seconds, for requests to the LDAP server
deployment.security.ldap.user	✓	Username used by VoltDB for read-only access on LDAP server, required
deployment.security.ldap.userclass	✓	Name of the LDAP schema's objectClass containing user information
deployment.security.ldap.useruid	✓	Name of the LDAP attribute in the userObjectClass that should contain the username provided by the VoltDB client
deployment.security.provider	✓	Sets authentication provider as hash (local) or ldap (using a customer-specified LDAP/LDAP server)
deployment.snapshot.enabled	✓	Enable/disable periodic automatic snapshots
deployment.snapshot.frequency	✓	Frequency of automatic snapshots (in s,m,h)
deployment.snapshot.prefix	✓	Unique prefix for snapshot files
deployment.snapshot.retain	✓	Number of snapshots to retain
deployment.snmp.authkey	✓	SNMPv3 authentication key if protocol is not NoAuth
deployment.snmp.authprotocol	✓	SNMPv3 authentication protocol. One of: SHA, MD5, NoAuth
deployment.snmp.community	✓	Name of SNMP community
deployment.snmp.enabled	✓	Enables or disables use of SNMP
deployment.snmp.privacykey	✓	SNMPv3 privacy key if protocol is not NoPriv
deployment.snmp.privacyprotocol	✓	SNMPv3 privacy protocol. One of: AES, DES, 3DES, AES192, AES256, NoPriv
deployment.snmp.target	✓	Host name or IP address, and optional port (default 162), for SNMP server
deployment.snmp.username	✓	Username for SNMPv3 authentication; else SNMPv2c is used
deployment.ssl.clientauthrequired		Enables mTLS on external port; client applications must be configured with a certificate

Property	Live Update	Description
deployment.ssl.crl.path		-
deployment.ssl.dr		Extends TLS/SSL security to the DR port (5555).
deployment.ssl.drclientauthrequired		Enables mTLS on DR port; DR consumer will use local cluster keystore to identify itself to remote
deployment.ssl.enabled		Enable or disable TLS/SSL on the cluster. Other properties control activation of TLS/SSL for specific ports and features.
deployment.ssl.external		Extends TLS/SSL security to all external ports (default admin 21211, client 21212).
deployment.ssl.internal		Extends TLS/SSL security to the internal port (default 3021).
deployment.ssl.keystore.auxFile		-
deployment.ssl.keystore.file		Keystore file for cluster, in Java Key Store format; ignored if sslSecret is used
deployment.ssl.keystore.indirectPassword		-
deployment.ssl.keystore.password		Password for VoltDB keystore file
deployment.ssl.truststore.auxFile		-
deployment.ssl.truststore.file		Truststore file, in Java Key Store format; ignored if sslSecret is used
deployment.ssl.truststore.indirectPassword		-
deployment.ssl.truststore.password		Password for VoltDB truststore file
deployment.systemsettings.clockskew.interval	✓	Interval of the scheduled clock skew collection (minutes). 0 is allowed and it disables collection. Interval cannot be less than 0 and if set below such value it will be reset to default.
deployment.systemsettings.compaction.interval	✓	Interval to indicate how often memory compaction should run (seconds)
deployment.systemsettings.compaction.maxcount	✓	Set a target block count compaction should try and achieve if there is memory fragmentation
deployment.systemsettings.elastic.duration	✓	Target value for the length of time each rebalance transaction will take (milliseconds)
deployment.systemsettings.elastic.throughput	✓	Target value for rate of data processing by rebalance transactions (MB)
deployment.systemsettings.flushinterval.dr.interval	✓	Interval for flushing DR data (milliseconds)
deployment.systemsettings.flushinterval.export.interval	✓	Interval for flushing export data (milliseconds)
deployment.systemsettings.flushinterval.minimum	✓	Interval between checking for need to flush (milliseconds)
deployment.systemsettings.priorities.batchsize		Modifies priority scheduling algorithm to execute multiple requests before rescheduling
deployment.systemsettings.priorities.dr.priority		Priority for DR requests (1-8, 1 is highest priority)

Property	Live Update	Description
deployment.systemsettings.priorities.enabled		Enables priority scheduling of requests by VoltDB cluster (true/false)
deployment.systemsettings.priorities.maxwait		Modifies priority scheduling by setting a limit on time waiting while higher priority requests execute (millisecs)
deployment.systemsettings.priorities.snapshot.priority		Priority for snapshot requests (1-8, 1 is highest priority)
deployment.systemsettings.procedure.copyparameters	✓	If set, mutable array parameters should be copied before processing
deployment.systemsettings.procedure.loginfo	✓	Threshold for long-running task detection (milliseconds)
deployment.systemsettings.query.timeout	✓	Timeout on SQL queries (milliseconds)
deployment.systemsettings.resourcemonitor.disklimit	✓	-
deployment.systemsettings.resourcemonitor.frequency	✓	Interval between Resource Monitor resource checks (seconds)
deployment.systemsettings.resourcemonitor.memorylimit.alert	✓	Alert level for memory use (in GB or as percentage)
deployment.systemsettings.resourcemonitor.memorylimit.compact	✓	Enable memory compaction
deployment.systemsettings.resourcemonitor.memorylimit.size	✓	Limit on memory use (in GB or as percentage)
deployment.systemsettings.snapshot.autotune.enabled	✓	Autotuning for snapshot work (for beta test only, in 14.1.0)
deployment.systemsettings.snapshot.priority	✓	Delay factor for snapshot work; larger values cause longer waits between bursts of execution. Ignored if systemsettings.priorities.enabled is true.
deployment.systemsettings.temptables.maxsize		Limit the size of temporary database tables (MB)
deployment.task.host.size		-
deployment.task.maxfrequency		Task scheduling parameter
deployment.task.mininterval		Task scheduling parameter
deployment.task.partition.size		-
deployment.threadpools[].name	✓	-
deployment.threadpools[].size	✓	-
deployment.topics.broker.properties	✓	Kafka topics: broker configuration properties
deployment.topics.enabled	✓	Kafka topics: enabled or not (supported since V11)
deployment.topics.threadpool	✓	Kafka topics: threadpool to use
deployment.topics.topic[].allow	✓	List of roles allowed to access the topic
deployment.topics.topic[].format	✓	Format of topic message
deployment.topics.topic[].name	✓	Topic name
deployment.topics.topic[].opaque	✓	Is this an opaque topic?

Property	Live Update	Description
deployment.topics.topic[].priority	✓	Priority for topics requests (if priority scheduling is enabled)
deployment.topics.topic[].procedure	✓	Procedure to invoke upon getting message
deployment.topics.topic[].properties	✓	Topic configuration properties
deployment.topics.topic[].retention	✓	Topic retention policy
deployment.users[].expires	✓	Expiration date for this user account
deployment.users[].name	✓	A VoltDB user name
deployment.users[].password	✓	User password
deployment.users[].plaintext	✓	If false, password was pre-hashed
deployment.users[].roles	✓	Comma-list of roles assigned to user

Appendix F. VoltDB Datatype Compatibility

VoltDB supports thirteen datatypes. When invoking stored procedures from different programming languages or queuing SQL statements within a Java stored procedure, you must use an appropriate language-specific value and datatype for arguments corresponding to placeholders in the query. This appendix provides the mapping of language-specific datatypes to the corresponding VoltDB datatype.

In several cases, there are multiple possible language-specific datatypes that can be used. The following tables highlight the best possible matches in bold.

F.1. Java and VoltDB Datatype Compatibility

Table F.1, "Java and VoltDB Datatype Compatibility" shows the compatible Java datatypes for each VoltDB datatype when:

- Calling simple stored procedures defined using the CREATE PROCEDURE AS statement
- Calling default stored procedures created for each table in the schema

In reverse, the table also shows which SQL datatype is used for the arguments and return values of user-defined functions written in Java. The highlighted Java datatype listed in the second column results in the corresponding SQL datatype being accepted or returned at runtime.

Note that when calling user-defined stored procedures written in Java, you can use additional datatypes, including arrays and the VoltTable object, as arguments to the stored procedure, as long as the actual query invocations within the stored procedure use the following datatypes. Within the stored procedure, when queuing SQL statements using the voltdbQueueSql method, implicit type casting is not guaranteed so using the highlighted Java type is recommended.

VoltDB accepts both primitive numeric types (byte, short, int, and so on) and their reference type equivalents (Byte, Short, Integer, etc.). The reference types can be useful, especially when passing null values, where you can send a Java null. In most cases when using the primitive types, you must pass the largest possible negative value for the type in place of null.

Table F.1. Java and VoltDB Datatype Compatibility

SQL Datatype	Compatible Java Datatypes	Notes
TINYINT	**byte**/Byte short/Short int/Integer long/Long BigDecimal String	Larger datatypes (short, int, long, and BigDecimal) are valid input types. However, VoltDB throws a runtime error if the value exceeds the allowable range of a TINYINT. String input must be a properly formatted text representation of an integer value in the correct range.
SMALLINT	byte/Byte **short**/Short int/Integer long/Long BigDecimal String	Larger datatypes (int, long, and BigDecimal) are valid input types. However, VoltDB throws a runtime error if the value exceeds the allowable range of a SMALLINT.

SQL Datatype	Compatible Java Datatypes	Notes
		String input must be a properly formatted text representation of an integer value in the correct range.
INTEGER	byte/Byte short/Short **int**/Integer long/Long BigDecimal String	Larger datatypes (long and BigDecimal) are valid input type. However, VoltDB throws a runtime error if the value exceeds the allowable range of an INTEGER. String input must be a properly formatted text representation of an integer value in the correct range.
BIGINT	byte/Byte short/Short int/Integer **long**/Long BigDecimal String	String input must be a properly formatted text representation of an integer value in the correct range.
FLOAT	**double**/Double byte/Byte short/Short int/Integer long/Long BigDecimal String	Because of the difference in how numbers are represented in the two types, there can be a loss of precision when using BigDecimal as input to a FLOAT value. String input must be a properly formatted text representation of a floating point value.
DECIMAL	**BigDecimal** double/Double byte/Byte short/Short int/Integer long/Long String	String input must be a properly formatted text representation of a decimal number.
GEOGRAPHY	(none)	Geospatial input should be converted from Well Known Text (WKT) to a VoltDB native format either using the GeographyValue.fromWKT() method or by passing a String and using the POLYGONFROMTEXT function within the SQL statement.
GEOGRAPHY_POINT	(none)	Geospatial input should be converted from Well Known Text (WKT) to a VoltDB native format either using the GeographyPointValue.fromWKT() method or by passing a String and using the POINTFROMTEXT function within the SQL statement.
VARCHAR()	**String** byte[] byte/Byte short/Short int/Integer long/Long BigDecimal	VoltDB stores VARCHAR text data as UTF-8 encoded strings. String and byte array input from Java programs is interpreted as UTF-8. However, for interactive input through the **sqlcmd** utility, VoltDB uses the character set encoding for the current user session to interpret and map the data to UTF-8 for processing. Similarly, for file input

SQL Datatype	Compatible Java Datatypes	Notes
	VoltDB TimestampType VoltDB DateType	(such as CSV files) you can use the `--charset` qualifier to the **sqlcmd** or **csvloader** command to specify which character set to use when interpreting input. Numeric and timestamp values are converted to their string representation. For example, the double value 13.25 is interpreted as "13.25" when converted to a VARCHAR.
VARBINARY()	String **byte[]** ByteBuffer	String input is interpreted as a hex-encoded binary value. For ByteBuffers, the starting position and limit properties are ignored and the entire buffer is interpreted as a byte array starting in position zero.
TIMESTAMP	**VoltDB TimestampType** int/Integer long/Long String	For String variables, the text must be formatted as either `YYYY-MM-DD hh.mm.ss.nnnnnn` or just the date portion `YYYY-MM-DD`.
DATE	**VoltDB DateType** java.sql.Date java.util.Date String	For String variables, the text must be formatted as either `YYYY-MM-DD` or `YYYY-MM-DD hh.mm.ss.nnnnnn`, where the extraneous hours, minutes, and seconds are truncated.
BOOLEAN	**VoltDB BooleanType**	Because the SQL Boolean type supports both true/false and null, it is not compatible with the Java native boolean type, which does not support null. However, BooleanType provides constants that let you check for all three possible values.

Appendix G. System Procedures

VoltDB provides system procedures that perform system-wide administrative functions. You can invoke system procedures interactively using the sqlcmd utility, or you can invoke them programmatically like other stored procedures, using the VoltDB client methods like callProcedureSync or callProcedureAsync in the Java Client2 API, or callProcedure in some other APIs.

This appendix describes the following system procedures.

- @AdHoc
- @AdjustLogging
- @Explain
- @ExplainProc
- @ExplainView
- @GetPartitionKeys
- @GetYamlConfiguration
- @Metrics
- @Note
- @Pause
- @Ping
- @Quiesce
- @Resume
- @SetYamlConfiguration
- @Shutdown
- @SnapshotDelete
- @SnapshotRestore
- @SnapshotSave
- @SnapshotScan
- @Statistics
- @StopNode
- @SwapTables
- @SystemCatalog
- @SystemInformation
- @UpdateClasses
- @UpdateLogging

@AdHoc

@AdHoc — Executes an SQL statement specified at runtime.

Syntax

@AdHoc String *SQL-statement* [, *statement-parameter* ...]

Description

The @AdHoc system procedure lets you perform arbitrary SQL statements on a running VoltDB database.

You can execute multiple SQL statements — either queries or data definition language (DDL) statements — in a single call to @AdHoc by separating the individual statements with semicolons. When you do this, the statements are performed as a single transaction. That is, the statements all succeed as a group or they all roll back if any of them fail. You cannot mix SQL queries and DDL in a single @AdHoc call.

You can use question marks in the SQL statement as placeholders that are replaced by parameters you provide as additional arguments to the call. For example:

```
sql = "SELECT * FROM Products WHERE partnumber=? AND vendor=?;"
results = client.callProcedureSync("@AdHoc",sql, productid, vendorid);
```

Performance of ad hoc queries is optimized, where possible. However, it is important to note that ad hoc queries are not pre-compiled, like queries in stored procedures. Therefore, use of stored procedures is recommended over @AdHoc for frequent, repetitive, or performance-sensitive queries.

Return Values

Returns one VoltTable for each statement, with as many rows as there are records returned by the statement. The column names and datatypes match the names and datatypes of the fields returned by the query.

Examples

The following program example uses @AdHoc to execute an SQL SELECT statement and display the number of reservations for a specific customer in the flight reservation database.

```
try {
    VoltTable[] results = client.callProcedureSync("@AdHoc",
       "SELECT COUNT(*) FROM RESERVATION " +
       "WHERE CUSTOMERID=" + custid).getResults();
    System.out.printf("%d reservations found.\n",
        results[0].fetchRow(0).getLong(0));
}
catch (Exception e) {
    e.printStackTrace();
}
```

Note that you do not need to explicitly invoke @AdHoc when using sqlcmd. You can type your statement directly into the sqlcmd prompt, like so:

```
$ sqlcmd
```

```
1> SELECT COUNT(*) FROM RESERVATION WHERE CUSTOMERID=12345;
```

@AdjustLogging

@AdjustLogging — Changes the logging level for a Volt component.

Syntax

```
@AdjustLogging String component, String level
```

Description

The @AdjustLogging system procedure lets you change the logging level for a specific logging component in VoltDB. The first argument, *component*, is the Volt component you want to modify and the second argument, *level*, is the new logging level. You can also adjust the logging level using the **voltadmin log4j** command with the syntax *component=level* as a single text string with no spaces. For example:

```
$ voltadmin log4j snapshot=warn
```

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

From the command line, the recommended way to adjust the logging level is to use the **voltadmin log4j** command, specifying the logging component and new logging level as arguments. For example, the following command changes the logging level of export messages to debug or higher.

```
$ voltadmin log4j export=debug
```

The following program example demonstrates how to make the same adjustment to the logging level using the system procedure in Java:

```
try {
    client.callProcedureSync("@AdjustLogging","export","debug");
}
catch (Exception e) {
    e.printStackTrace();
}
```

@Explain

@Explain — Returns the execution plan for the specified SQL query.

Syntax

```
@Explain String SQL-statement
```

Description

The @Explain system procedure evaluates the specified SQL query and returns the resulting execution plan. Execution, or explain, plans describe how VoltDB expects to execute the query at runtime, including what indexes are used, the order the tables are joined, and so on. Execution plans are useful for identifying performance issues in query design. See the chapter on execution plans in the *VoltDB Guide to Performance and Customization* for information on how to interpret the plans.

Return Values

Returns one VoltTable with one row and one column.

Name	Datatype	Description
EXECUTION_PLAN	VARCHAR	The execution plan as text.

Examples

The following program example uses @Explain to evaluate an ad hoc SQL SELECT statement against the voter sample application.

```
try {
    String query = "SELECT COUNT(*) FROM CONTESTANTS;";
    VoltTable[] results = client.callProcedureSync("@Explain",
       query ).getResults();
    System.out.printf("Query: %d\nPlan:\n%d",
        query, results[0].fetchRow(0).getString(0));
}
catch (Exception e) {
    e.printStackTrace();
}
```

In the sqlcmd utility, the "explain" command is a shortcut for "exec @Explain". So the following two commands are equivalent:

```
$ sqlcmd
1> exec @Explain 'SELECT COUNT(*) FROM CONTESTANTS';
2> explain SELECT COUNT(*) FROM CONTESTANTS;
```

@ExplainProc

@ExplainProc — Returns the execution plans for all SQL queries in the specified stored procedure.

Syntax

```
@ExplainProc String procedure-name
```

Description

The @ExplainProc system procedure returns the execution plans for all of the SQL queries within the specified stored procedure. Execution, or explain, plans describe how VoltDB expects to execute the queries at runtime, including what indexes are used, the order the tables are joined, and so on. Execution plans are useful for identifying performance issues in query and stored procedure design. See the chapter on execution plans in the *VoltDB Guide to Performance and Customization* for information on how to interpret the plans.

Return Values

Returns one VoltTable with one row for each query in the stored procedure.

Name	Datatype	Description
SQL_STATEMENT	VARCHAR	The SQL query.
EXECUTION_PLAN	VARCHAR	The execution plan as text.

Examples

The following example uses @ExplainProc to evaluate the execution plans associated with the ContestantWinningStates stored procedure in the voter sample application.

```
try {
    VoltTable[] results = client.callProcedureSync("@ExplainProc",
       "ContestantWinningStates" ).getResults();
    results[0].resetRowPosition();
    while (results[0].advanceRow()) {
         System.out.printf("Query: %d\nPlan:\n%d",
         results[0].getString(0),results[0].getString(1));
    }
}
catch (Exception e) {
    e.printStackTrace();
}
```

In the sqlcmd utility, the "explainproc" command is a shortcut for "exec @ExplainProc". So the following two commands are equivalent:

```
$ sqlcmd
1> exec @ExplainProc 'ContestantWinningStates';
2> explainproc ContestantWinningStates;
```

@ExplainView

@ExplainView — Returns the execution plans for the components of the specified view.

Syntax

```
@ExplainView String view-name
```

Description

The @ExplainView system procedure returns the execution plans for certain components of the specified view. Execution plans describe how VoltDB expects to calculate the component values as the referenced tables or streams are updated. The plans include what indexes are used, the order the tables are joined, and so on. Execution plans are useful for identifying performance issues in the design of the view statement. See the chapter on execution plans in the *VoltDB Guide to Performance and Customization* for information on how to interpret the plans.

For views, execution plans are listed for the calculation of MIN() and MAX() functions and multi-table joins only. For simple views — that is, views on a single table with aggregate functions other than MIN() or MAX() — the system procedure returns no rows.

Return Values

Returns one VoltTable with one row for each component of the view.

Name	Datatype	Description
TASK	VARCHAR	The function or join statement
EXECUTION_PLAN	VARCHAR	The execution plan as text.

Examples

The following example uses @ExplainView to evaluate the execution plans associated with a view that joins two tables.

```
try {
    VoltTable[] results = client.callProcedureSync("@ExplainView",
       "stats_by_city_and_state" ).getResults();
    results[0].resetRowPosition();
    while (results[0].advanceRow()) {
         System.out.printf("Task: %d\nPlan:\n%d",
         results[0].getString(0),results[0].getString(1));
    }
}
catch (Exception e) {
    e.printStackTrace();
}
```

In the sqlcmd utility, the "explainview" command is a shortcut for "exec @ExplainView". So the following two commands are equivalent:

```
$ sqlcmd
```

```
1> exec @ExplainView 'stats_by_city_and_state';
2> explainview stats_by_city_and_state;
```

@GetPartitionKeys

@GetPartitionKeys — Returns a list of partition values, one for every partition in the database.

Syntax

```
@GetPartitionKeys String datatype
```

Description

The @GetPartitionKeys system procedure returns a set of partition values that you can use to reach every partition in the database. This procedure is useful when you want to run a stored procedure in every partition but you do not want to use a multi-partition procedure. By running multiple single-partition procedures, you avoid the impact on latency and throughput that can result from a multi-partition procedure. This is particularly true for longer running procedures. Using multiple, smaller procedures can also help for queries that modify large volumes of data, such as large deletes.

When you call @GetPartitionKeys you specify the datatype of the keys to return as the second parameter. You specify the datatype as a case-insensitive string. Valid options are "INTEGER", "STRING", and "VARCHAR" (where "STRING" and "VARCHAR" are synonyms).

Note that the results of the system procedure are valid at the time they are generated. If the cluster is static (that is, no nodes are being added and any rebalancing is complete), the results remain valid until the next elastic event. However, during rebalancing, the distribution of partitions is likely to change. So it is a good idea to call @GetPartitionKeys once to get the keys, act on them, then call the system procedure again to verify that the partitions have not changed.

Return Values

Returns one VoltTable with a row for every unique partition in the cluster.

Name	Datatype	Description
PARTITION_ID	INTEGER	The numeric ID of the partition.
PARTITION_KEY	INTEGER or STRING	A valid partition key for the partition. The datatype of the key matches the type requested in the procedure call.

Examples

The following example shows the use of sqlcmd to get integer key values from @GetPartitionKeys:

```
$sqlcmd
1> exec @GetPartitionKeys integer;
```

The next example shows a Java program using @GetPartitionKeys to execute a stored procedure to clear out old records, one partition at a time.

```
VoltTable[] results = client.callProcedureSync("@GetPartitionKeys",
    "INTEGER").getResults();
VoltTable keys = results[0];
for (int k=0;k<keys.getRowCount();k++) {
    long key = keys.fetchRow(k).getLong(1);
```

```
            client.callProcedureSync("PurgeOldData", key);
        }
```

@GetYamlConfiguration

@GetYamlConfiguration — Returns current configuration settings for one or more properties

Syntax

@GetYamlConfiguration String *property-list*

Description

The @GetYamlConfiguration system procedure returns the current values for one or more configuration properties as a text string in YAML format. You specify the properties in dot notation, separating multiple properties on individual lines. For example, the following input argument would return the current values for `deployment.cluster.sitesperhost`, `deployment.cluster.kfactor`, and `deployment.snapshot`:

```
deployment.cluster.sitesperhost
deployment.cluster.kfactor
deployment.snapshot
```

In the case of complex properties — those where the value is an array or sequence of subproperties, like `deployment.snapshot` in the preceding example — the results include all of the subproperties. For example, the preceding input would result in the following output on a database started with the default configuration:

```
---
deployment:
  cluster:
    sitesperhost: 8
    kfactor: 0
  snapshot:
    frequency: "24h"
    retain: 2
    prefix: "AUTOSNAP"
    enabled: false
```

Specifying an empty string or the string "null" as the input argument returns all of the configuration properties.

Return Values

Returns a string containing the YAML representation of the property or properties requested.

Examples

The following program example uses @GetYamlConfiguration to retrieve all import and export configuration settings.

```
try {
    VoltTable[] results = client.callProcedureSync("@GetYamlConfiguration",
       "deployment.import\ndeployment.export").getResults();
    String yamloutput = results[0].fetchRow(0).getString(0));
```

```
}
catch (Exception e) {
    e.printStackTrace();
}
```

For interactive use, the **voltadmin get** command performs the same function as @GetYamlConfiguration, except you separate multiple property names with spaces instead of line breaks. For example, the following is the interactive equivalent of the preceding program example:

```
$ voltadmin get deployment.import deployment.export
```

@Metrics

@Metrics — Returns information about the performance and current status of the database server

Syntax

```
@Metrics
```

Description

The @Metrics system procedure returns information about the status and performance of the database server. This information is only available if metrics are enabled when the database is started by setting `deployment.metrics.enabled` to *true* in the configuration file.

Metrics are normally accessed through the metrics port via a network request. The format and request method when using the metrics port are compatible with Prometheus, an industry standard metrics collection and reporting system. The @Metrics system procedure returns the same information, however, structured as a series of VoltTables, appropriate for a client application.

Return Values

The procedure returns one VoltTable summarizing the amount of data being returned, then a series of VoltTables grouping the metrics by topic, with specific columns and number of rows dependent on the information required by each topic.

Example

The following program example filters the results by the metric name MEMORY_TUPLEDATA and prints out the total number of bytes currently used to store tuples.

```
try {
  VoltTable[] results = client.callProcedureSync("@Metrics").getResults();
  String metric = "MEMORY_TUPLEDATA";
  int count = 0;
  int tables = 0;
  for (VoltTable node : results) {
    if (tables++ ==1) continue; // Skip summary row
    node.resetRowPosition();
    while(node.advanceRow()) {
      if (node.getString("NAME").equals(metric)) {
      count += node.getLong("VALUE");
    }
  }
  System.out.println("Tuple Memory in use: " + String.valueOf(count));
} catch (Exception e) { e.printStackTrace(); }
```

@Note

@Note — Writes a message into the VoltDB log file.

Syntax

```
@Note String message
```

Description

The @Note system procedure lets you write arbitrary text into the VoltDB log file as an INFO level message. Adding messages to the log file can be useful when debugging issues with your application. For example, you can write a message when a new process starts and again when it ends, to see if it generates any unusual error messages between the two notes. You can also use this procedure to document administrative actions taken on the system, such as updating the configuration or changing the schema.

When security is enabled, the process or user must have admin permissions to be able to write to the log file. Otherwise, the procedure call returns an error indicating you do not have permission to complete the request.

The @Note system procedure lets you write to the log from within your application. To perform the same task interactively, you can execute @Note from the **sqlcmd** prompt or by using the **voltadmin note** command.

Return Values

Returns one VoltTable with one row

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Example

The following program example uses @Note to write a message into the log file before and after performing a large series of delete operations.

```
client.callProcedureSync("@Note", "Starting database cleanup...");

   /* Code to delete obsolete records */

    [ . . . ]

client.callProcedureSync("@Note", "Database cleanup completed.");
```

The following example uses the **voltadmin note** command to perform the same function when running a cleanup program interactively. (Note the use of single quotes to enclose the message text to avoid punctuation being accidentally interpreted by the command shell).

```
$ voltadmin note 'Starting database cleanup...'
$ java MyCleanupProgram
$ voltadmin note 'Database cleanup completed.'
```

@Pause

@Pause — Initiates read-only mode on the cluster.

Syntax

```
@Pause
```

Description

The @Pause system procedure initiates admin mode on the cluster. Admin mode puts the database into read-only mode and ensures no further changes to the database can be made through the client port when performing sensitive administrative operations, such as taking a snapshot before shutting down.

While in admin mode, any write transactions on the client port are rejected and return an error status. Read-only transactions, including system procedures, are allowed. However, write transactions such as inserts, deletes, or schema changes are only allowed through the admin port.

Several important points to consider concerning @Pause are:

- @Pause must be called through the admin port, not the standard client port.
- Although write transactions on the client port are rejected in admin mode, existing connections from client applications are not removed.
- To return to normal database operation, you must call the system procedure @Resume on the admin port.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

It is possible to call @Pause using the **sqlcmd** utility. However, you must explicitly connect to the admin port when starting **sqlcmd** to do this. Also, it is often easier to use the **voltadmin** utility, which connects to the admin port by default. For example, the following commands demonstrate pausing and resuming the database using both **sqlcmd** and **voltadmin**:

```
$ sqlcmd --port=21211
1> exec @Pause;
2> exec @Resume;

$ voltadmin pause
$ voltadmin resume
```

The following program example, if called through the admin port, initiates admin mode on the database cluster.

```
client.callProcedureSync("@Pause");
```

@Ping

@Ping — Indicates whether the database is currently running.

Syntax

```
@Ping
```

Description

The @Ping system procedure returns a value of zero (0) indicating that the database is up and running. The system procedure does not respond until the database completes its startup process.

The @Ping system procedure is a lightweight procedure and does not require any interaction between cluster nodes, which makes it a better choice than other system procedures (such as @Statistics) if all you need to do is check if the database is running.

Return Values

Returns one VoltTable with one row

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

The following program fragment calls @Ping to determine if the database is still available. For example, if the client application has not sent any data for awhile, it can check to see if the server is running before sending any new transactions. Note that you can either check the response value or the status value to determine if the call succeeded or not.

```
ClientResponse response = client.callProcedureSync("@Ping");
if (response.getStatus() == ClientResponse.SUCCESS) {
    /* We can continue */
} else {
    /* Server is not ready */
}
```

@Quiesce

@Quiesce — Waits for all queued export and DR data to be processed or saved to disk

Syntax

```
@Quiesce
```

Description

The @Quiesce system procedure waits for any pending export or database replication (DR) data to be cleared before returning to the calling application. @Quiesce waits for export data to either be written to the export connector or to the export overflow on disk. Similarly, it waits for pending DR data to be written to DR overflow then does an fsync to ensure all disk I/O is completed. Note that a graceful shutdown (using the **voltadmin shutdown** command without the **--force** argument) performs a quiesce as part of the shutdown process.

If export and DR are not enabled, the procedure returns immediately.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

The following example calls @Quiesce using sqlcmd:

```
$ sqlcmd
1> exec @Quiesce;
```

The following program example uses drain and @Quiesce to complete any asynchronous transactions and clear the export and DR queues before shutting down the database.

```
     // Complete all outstanding activities
try {
    client.drain();
    client.callProcedureSync("@Quiesce");
}
catch (Exception e) {
    e.printStackTrace();
}

     // Shutdown the database.
try {
    client.callProcedureSync("@Shutdown");
}
     // We expect an exception when the connection drops.
     // Report any other exception.
```

```
catch (org.voltdb.client.ProcCallException e) { }
catch (Exception e) { e.printStackTrace(); }
```

@Resume

@Resume — Returns a paused database to normal operating mode.

Syntax

```
@Resume
```

Description

The @Resume system procedure switches all nodes in a database cluster from admin mode to normal operating mode. In other words, @Resume is the opposite of @Pause.

After calling this procedure, the cluster returns to accepting read/write ad hoc queries and stored procedure invocations from clients connected to the standard client port.

@Resume must be invoked from a connection to the admin port.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

You can call @Resume using the **sqlcmd** utility. However, you must explicitly connect to the admin port when starting **sqlcmd** to do this. It is often easier to use the **voltadmin resume** command, which connects to the admin port by default. For example, the following commands are equivalent:

```
$ sqlcmd --port=21211
1> exec @Resume;

$ voltadmin resume
```

The following program example uses @Resume to return the cluster to normal operation.

```
client.callProcedureSync("@Resume");
```

@SetYamlConfiguration

@SetYamlConfiguration — changes configuration settings for one or more properties

Syntax

```
@SetYamlConfiguration String yaml, String dot-notation
```

Description

The @SetYamlConfiguration system procedure sets new values for one or more configuration properties. You specify the properties to change in either YAML or dot notation. You provide a YAML definition as the first argument and one or more property assignments in dot notation as the second argument. If either argument is an empty string or null it is ignored.

YAML natively supports defining multiple properties. To specify multiple values in dot notation, place each property name and value on a separate line.

The @SetYamlConfiguration procedure only changes the specified properties, all other configuration options remain unchanged. It is also important to note that only certain properties can be changed while the database is running. See Table E.1, “Complete List of Volt Configuration Properties” for a list of all configuration options. Those properties with a checkmark next to their names can be changed "on the fly". All other options are static and will generate an error if you try to change them with this system procedure.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Returns a value zero (0) indicating success or two (2) indicating success but no changes required

.Examples

The following program example uses @SetYamlConfiguration and YAML to enable automatic snapshots and set the retention limit to 2 snapshots.

```
String yaml = """---
deployment:
  snapshot:
    enabled: true
    retain: 2""";
String dots = """deployment.snapshot.frequency=60m
deployment.snapshot.prefix=backup""";

try {
    VoltTable[] results = client.callProcedureSync("@SetYamlConfiguration",
        yaml,""
    ).getResults();
}
catch (Exception e) {
```

```
    e.printStackTrace();
}
```

To set the snapshot frequency and prefix using dot notation, you can change the procedure call in the preceding example to the following:

```
try {
    VoltTable[] results = client.callProcedureSync("@SetYamlConfiguration",
        "",dots
]   ).getResults();
}
```

Or you can specify both YAML and dot notation at the same time to set all four properties:

```
try {
    VoltTable[] results = client.callProcedureSync("@SetYamlConfiguration",
        yaml,dots
]   ).getResults();
}
```

For interactive use, the **voltadmin set** command performs the same function as @SetYamlConfiguration, except you separate multiple property names in dot notation with spaces instead of line breaks. The following is the interactive equivalent of the second program example:

```
$ voltadmin set \
    deployment.snapshot.frequency=60m \
    deployment.snapshot.prefix=backup
```

@Shutdown

@Shutdown — Shuts down the database.

Syntax

```
@Shutdown
```

Description

The @Shutdown system procedure performs an immediate shut down of a VoltDB database on all nodes of the cluster.

Note

The @Shutdown system procedure does not wait for running transactions to complete or queued data to be written to disk before stopping the database process, which can result in loss of data. Therefore, using the **voltadmin shutdown** command to perform an orderly shutdown, making all data durable, is the recommended method for stopping a VoltDB database.

Note that once the database shuts down, the client connection is lost and the calling program cannot make any further requests to the server.

Examples

The first example shows the recommended way to shutdown a VoltDB database, using the **voltadmin shutdown** command to perform an orderly shutdown:

```
$ voltadmin shutdown
```

The next example shows calling @Shutdown from **sqlcmd**. This is equivalent to the **voltdb shutdown --force** command:

```
$ sqlcmd
1> exec @Shutdown;
```

The following program example uses @Shutdown to stop the database cluster programmatically. Note the use of catch to separate out a VoltDB call procedure exception (which is expected) from any other exception.

```
try {
    client.callProcedureSync("@Shutdown");
}

        // we expect an exception when the connection drops.
catch (org.voltdb.client.ProcCallException e) {
    System.out.println("Database shutdown initiated.");
}
        // report any other exception.
catch (Exception e) {
    e.printStackTrace();
}
```

@SnapshotDelete

@SnapshotDelete — Deletes one or more native snapshots.

Syntax

```
@SnapshotDelete String[] directory-paths, String[] Unique-IDs
```

Description

The @SnapshotDelete system procedure deletes native snapshots from the database cluster. This is a cluster-wide operation and a single invocation will remove the snapshot files from all of the nodes.

The procedure takes two parameters: a String array of directory paths and a String array of unique IDs (prefixes).

The two arrays are read as a series of value pairs, so that the first element of the directory path array and the first element of the unique ID array will be used to identify the first snapshot to delete. The second element of each array will identify the second snapshot to delete. And so on.

@SnapshotDelete can delete native format snapshots only. The procedure cannot delete CSV format snapshots.

Return Values

Returns one VoltTable with a row for every snapshot file affected by the operation.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PATH	STRING	The directory path where the snapshot file resides.
PATHTYPE	STRING	The type of snapshot — whether manual, command log, or auto — specified as "SNAP_PATH", "SNAP_CL", or 'SNAP_AUTO".
NONCE	STRING	The unique identifier for the snapshot.
NAME	STRING	The file name.
SIZE	BIGINT	The total size, in bytes, of the file.
DELETED	STRING	String value indicating whether the file was successfully deleted ("TRUE") or not ("FALSE").
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Example

The following example uses @SnapshotScan to identify all of the snapshots in the directory `/tmp/voltdb/backup/`. This information is then used by @SnapshotDelete to delete those snapshots.

```
try {
     results = client.callProcedureSync("@SnapshotScan",
                                        "/tmp/voltdb/backup/").getResults();
}
catch (Exception e) { e.printStackTrace(); }

VoltTable table = results[0];
int numofsnapshots = table.getRowCount();
int i = 0;

if (numofsnapshots > 0) {
   String[] paths  = new String[numofsnapshots];
   String[] prefixes = new String[numofsnapshots];
   for (i=0;i<numofsnapshots;i++) { paths[i] = "/etc/voltdb/backup/"; }
   table.resetRowPosition();
   i = 0;
   while (table.advanceRow()) {
       prefixes[i] = table.getString("NONCE");
       i++;
   }

   try {
       client.callProcedureSync("@SnapshotDelete",paths,prefixes);
   }
  catch (Exception e) { e.printStackTrace(); }
}
```

@SnapshotRestore

@SnapshotRestore — Restores a database from disk using a native format snapshot.

Syntax

@SnapshotRestore String *directory-path*, String *unique-ID*

@SnapshotRestore String *json-encoded-options*

Description

The @SnapshotRestore system procedure restores a previously saved database from disk to memory. The snapshot must be in native format. (You cannot restore a CSV format snapshot using @SnapshotRestore.) The restore request is propagated to all nodes of the cluster, so a single call to @SnapshotRestore will restore the entire database cluster.

If the database is empty — that is, there is no data or schema defined in the database — the restore operation restores the schema and the requested data. If a schema is already defined, the restore operation restores data only for those tables defined in the current schema.

There are two forms of the @SnapshotRestroe procedure, as described below. See Chapter 13, *Saving & Restoring a VoltDB Database* for more information about saving and restoring VoltDB databases.

Individual Arguments

When you specify the arguments as individual parameters, you must specify two arguments:

1. The directory path where the snapshot files are stored
2. An identifier that uniquely identifies the snapshot

JSON-Encoded Arguments

When you specify the arguments as a JSON-encoded string, you can specify not only what snapshot to use, but what data to restore from that snapshot. Table G.1, "@SnapshotRestoreOptions" describes the possible options when restoring a snapshot using JSON-encoded arguments.

Table G.1. @SnapshotRestoreOptions

Option	Description
path	Specifies the path where the snapshot files are stored.
nonce	Specifies the unique identifier for the snapshot.
skiptables	Specifies tables to leave out when restoring the snapshot. Use of tables or skiptables allows you to restore part of a snapshot. Specify the list of tables as a JSON array. For example, the following JSON argument excludes the Areacode and Country tables from the restore operation: `"skiptables":["areacode","country"]`
tables	Specifies tables to include when restoring the snapshot. Use of tables or skiptables allows you to restore part of a snapshot. Specify the list of tables as a JSON array. For

Option	Description
	example, the following JSON argument includes only the Employee and Company tables in the restore operation: `"tables":["employee","company"]`

For example, the JSON-encoded arguments to restore the tables Employee and Company from the "mydb" snapshot in the /tmp directory is the following:

```
{path:"/tmp",nonce:"mydb",tables:["employee","company"]}
```

Return Values

Returns one VoltTable with a row for every table restored at each execution site.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
TABLE	STRING	The name of the table being restored.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Examples

The following example uses @SnapshotRestore to restore previously saved database content from the path `/tmp/voltdb/backup/` using the unique identifier *flight*.

```
$ sqlcmd
1> exec @SnapshotRestore '/tmp/voltdb/backup/', 'flight';
```

Alternately, you can use the **voltadmin restore** command to perform the same function:

```
$ voltadmin restore /tmp/voltdb/backup/ flight
```

Since there are a number of situations that impact what data is restored, it is a good idea to review the return values to see what tables and partitions were affected. In the following program example, the contents of the VoltTable array is written to standard output so the operator can confirm that the restore completed as expected.

```
VoltTable[] results = null;

try {
     results = client.callProcedureSync("@SnapshotRestore",
                                        "/tmp/voltdb/backup/",
                                        "flight").getResults();
```

```
}
catch (Exception e) {
    e.printStackTrace();
}

for (int t=0; t<results.length; t++) {
    VoltTable table = results[t];
    for (int r=0;r<table.getRowCount();r++) {
        VoltTableRow row = table.fetchRow(r);
        System.out.printf("Node %d Site %d restoring " +
              "table %s partition %d.\n",
              row.getLong("HOST_ID"), row.getLong("SITE_ID"),
              row.getString("TABLE"),row.getLong("PARTITION"));
     }
}
```

@SnapshotSave

@SnapshotSave — Saves the current database contents to disk.

Syntax

```
@SnapshotSave String directory-path, String unique-ID, Integer blocking-flag

@SnapshotSave String json-encoded-options

@SnapshotSave
```

Description

The @SnapshotSave system procedure saves the contents of the current in-memory database to disk. Each node of the database cluster saves its portion of the database locally.

There are three forms of the @SnapshotSave stored procedure: a procedure call with individual argument parameters, with all arguments in a single JSON-encoded string, or with no arguments. When you specify the arguments as individual parameters, VoltDB creates a native mode snapshot that can be used to recover or restore the database. When you specify the arguments as a JSON-encoded string, you can request a different format for the snapshot, including CSV (comma-separated value) files that can be used for import into other databases or utilities. When you specify no arguments a full, native snapshot is saved into the default snapshots directory in the database root directory.

Individual Arguments

When you specify the arguments as individual parameters, you must specify three arguments:

1. The directory path where the snapshot files are stored
2. An identifier that is included in the file names to uniquely identify the files that make up a single snapshot
3. A flag value indicating whether the snapshot should block other transactions until it is complete or not

The resulting snapshot consists of multiple files saved to the directory specified by *directory-path* using *unique-ID* as a filename prefix. The third argument, *blocking-flag*, specifies whether the save is performed synchronously (thereby blocking any following transactions until the save completes) or asynchronously. If this parameter is set to any non-zero value, the save operation will block any following transactions. If it is zero, others transactions will be executed in parallel.

The files created using this invocation are in native VoltDB snapshot format and can be used to restore or recover the database at some later time. This is the same format used for automatic snapshots. See Chapter 13, *Saving & Restoring a VoltDB Database* for more information about saving and restoring VoltDB databases.

JSON-Encoded Arguments

When you specify the arguments as a JSON-encoded string, you can specify what snapshot format you want to create. Table G.2, “@SnapshotSave Options” describes all possible options when creating a snapshot using JSON-encoded arguments.

Table G.2. @SnapshotSave Options

Option	Description
uripath	Specifies the path where the snapshot files are created. Note that, as a JSON-encoded argument, the path must be specified as a URI, not just a system directory path. Therefore, a local directory must be specified using the `file://` identifier, such as "`file:///tmp`", and the path must exist on all nodes of the cluster.
nonce	Specifies the unique identifier for the snapshot.
block	Specifies whether the snapshot should be synchronous (true) and block other transactions or asynchronous (false).
format	Specifies the format of the snapshot. Valid formats are "csv" and "native". When you save a snapshot in CSV format, the resulting files are in standard comma-separated value format, with only one file for each table. In other words, duplicates (from replicated tables or duplicate partitions due to K-safety) are eliminated. CSV formatted snapshots are useful for import or reuse by other databases or utilities. However, they *cannot* be used to restore or recover a VoltDB database. When you save a snapshot in native format, each node and partition saves its contents to separate files. These files can then be used to restore or recover the database. It is also possible to later convert native format snapshots to CSV using the snapshot utilities described in the *VoltDB Administrator's Guide*.
skiptables	Specifies tables to leave out of the snapshot. Use of tables or skiptables allows you to create a partial snapshot of the larger database. Specify the list of tables as a JSON array. For example, the following JSON argument excludes the Areacode and Country tables from the snapshot: `"skiptables":["areacode","country"]`
tables	Specifies tables to include in the snapshot. Use of tables or skiptables allows you to create a partial snapshot of the larger database. Specify the list of tables as a JSON array. For example, the following JSON argument includes only the Employee and Company tables in the snapshot: `"tables":["employee","company"]`

For example, the JSON-encoded arguments to synchronously save a CSV formatted snapshot to /tmp using the unique identifier "mydb" is the following:

```
{uripath:"file:///tmp",nonce:"mydb",block:true,format:"csv"}
```

The tables and skiptables arguments are logically mutually exclusive. Because the unique identifier is used in the resulting filenames, the identifier can contain only characters that are valid for Linux file names. In addition, hyphens ("-") and commas (",") are not permitted.

Note that it is normal to perform manual saves synchronously, to ensure the snapshot represents a known state of the database. However, automatic snapshots are performed asynchronously to reduce the impact on ongoing database activity.

Return Values

The @SnapshotSave system procedure returns two different VoltTables, depending on the outcome of the request.

Option #1: one VoltTable with a row for every execution site. (That is, the number of hosts multiplied by the number of sites per host.).

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Option #2: one VoltTable with a variable number of rows.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
TABLE	STRING	The name of the database table. The contents of each table is saved to a separate file. Therefore it is possible for the snapshot of each table to succeed or fail independently.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

Examples

The following example uses @SnapshotSave to save the current database content in native snapshot format to the path `/tmp/voltdb/backup/` using the unique identifier *flight* on each node of the cluster.

```
$ sqlcmd
1> exec @SnapshotSave '/tmp/voltdb/backup/', 'flight', 1;
```

Alternately, you can use the **voltadmin save** command to perform the same function. When using the **voltadmin save** command, you use the `--blocking` flag instead of a third parameter to request a blocking save:

```
$ voltadmin save --blocking /tmp/voltdb/backup/ flight
```

Note that the procedure call will return successfully even if the save was not entirely successful. The information returned in the VoltTable array tells you what parts of the operation were successful or not. For example, save may succeed on one node but not on another.

The following code sample performs the same function, but also checks the return values and notifies the operator when portions of the save operation are not successful.

```
VoltTable[] results = null;

try { results = client.callProcedureSync("@SnapshotSave",
                                         "/tmp/voltdb/backup/",
                                         "flight", 1).getResults(); }
```

```
catch (Exception e) { e.printStackTrace(); }

for (int table=0; table<results.length; table++) {
    for (int r=0;r<results[table].getRowCount();r++) {
        VoltTableRow row = results[table].fetchRow(r);
        if (row.getString("RESULT").compareTo("SUCCESS") != 0) {
            System.out.printf("Site %s failed to write " +
                   "table %s because %s.\n",
                   row.getString("HOSTNAME"), row.getString("TABLE"),
                   row.getString("ERR_MSG"));
        }
    }
}
```

@SnapshotScan

@SnapshotScan — Lists information about existing native snapshots in a given directory path.

Syntax

```
@SnapshotScan String directory-path
```

Description

The @SnapshotScan system procedure provides information about any native snapshots that exist within the specified directory path for all nodes on the cluster. The procedure reports the name (prefix) of the snapshot, when it was created, how long it took to create, and the size of the individual files that make up the snapshot(s).

@SnapshotScan does not include CSV format snapshots in its output. Only native format snapshots are listed.

Return Values

On successful completion, this system procedure returns three VoltTables providing the following information:

- A summary of the snapshots found
- Available space in the directories scanned
- Details concerning the Individual files that make up the snapshots

The first table contains one row for every snapshot found.

Name	Datatype	Description
PATH	STRING	The directory path where the snapshot resides.
NONCE	STRING	The unique identifier for the snapshot.
TXNID	BIGINT	The transaction ID of the snapshot.
CREATED	BIGINT	The timestamp when the snapshot was created (in milliseconds).
SIZE	BIGINT	The total size, in bytes, of all the snapshot data.
TABLES_REQUIRED	STRING	A comma-separated list of all the table names listed in the snapshot digest file. In other words, all of the tables that make up the snapshot.
TABLES_MISSING	STRING	A comma-separated list of database tables for which no data can be found. (That is, the corresponding files are missing or unreadable.)
TABLES_INCOMPLETE	STRING	A comma-separated list of database tables with only partial data saved in the snapshot. (That is, data from some partitions is missing.)
COMPLETE	STRING	A string value indicating whether the snapshot as a whole is complete ("TRUE") or incomplete ("FALSE"). If this col-

Name	Datatype	Description
		umn is "FALSE", the preceding two columns provide additional information concerning what is missing.
PATHTYPE	STRING	A string value indicating the type of snapshot and its location, where the type can be "SNAP_PATH" for manual snapshots, "SNAP_CL" for command log snapshots, and "SNAP_AUTO" for automated snapshots.

The second table contains one row for every host.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PATH	STRING	The directory path specified in the call to the procedure.
TOTAL	BIGINT	The total space (in bytes) on the device.
FREE	BIGINT	The available free space (in bytes) on the device.
USED	BIGINT	The total space currently in use (in bytes) on the device.
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

The third table contains one row for every file in the snapshot collection.

Name	Datatype	Description
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PATH	STRING	The directory path where the snapshot file resides.
NAME	STRING	The file name.
TXNID	BIGINT	The transaction ID of the snapshot.
CREATED	BIGINT	The timestamp when the snapshot was created (in milliseconds).
TABLE	STRING	The name of the database table the data comes from.
COMPLETED	STRING	A string indicating whether all of the data was successfully written to the file ("TRUE") or not ("FALSE").
SIZE	BIGINT	The total size, in bytes, of the file.
IS_REPLICATED	STRING	A string indicating whether the table in question is replicated ("TRUE") or partitioned ("FALSE").
PARTITIONS	STRING	A comma-separated string of partition (or site) IDs from which data was taken during the snapshot. For partitioned tables where there are multiple sites per host, there can be data from multiple partitions in each snapshot file. For replicated tables, data from only one copy (and therefore one partition) is required.
TOTAL_PARTITIONS	BIGINT	The total number of partitions from which data was taken.

Name	Datatype	Description
READABLE	STRING	A string indicating whether the file is accessible ("TRUE") or not ("FALSE").
RESULT	STRING	String value indicating the success ("SUCCESS") or failure ("FAILURE") of the request.
ERR_MSG	STRING	If the result is FAILURE, this column contains a message explaining the cause of the failure.

If the system procedure fails because it cannot access the specified path, it returns a single VoltTable with one row and one column.

Name	Datatype	Description
ERR_MSG	STRING	A message explaining the cause of the failure.

Examples

The following example uses @SnapshotScan to list information about the snapshots in the directory `/tmp/voltdb/backup/`.

```
$ sqlcmd
1> exec @SnapshotScan /tmp/voltdb/backup/;
```

The following program example performs the same function, using the VoltTable `toString()` method to display the results of the procedure call:

```
VoltTable[] results = null;

try { results = client.callProcedureSync("@SnapshotScan",
                        "/tmp/voltdb/backup/").getResults();
}
catch (Exception e) { e.printStackTrace(); }

for (VoltTable t: results) {
   System.out.println(t.toString());
}
```

In the return value, the first VoltTable in the array lists the snapshots and certain status information. The second element of the array provides information about the directory itself (such as used, free, and total disk space). The third element of the array lists specific information about the individual files in the snapshot(s).

@Statistics

@Statistics — Returns statistics about the usage of the VoltDB database.

Syntax

```
@Statistics String component, Integer delta-flag
```

Description

The @Statistics system procedure returns information about the VoltDB database. The first argument, *component*, specifies what aspect of VoltDB to return statistics about. The second argument, *delta-flag*, specifies whether statistics are reported from when the database started or since the last call to @Statistics where the flag was set.

Some components report statistics at the moment, collected when you make the procedure call. While other components report statistics over time. For cumulative components, if the delta flag is set to zero, the system procedure returns statistics since the database started. If the delta flag is non-zero, the system procedure returns statistics for the interval since the last time @Statistics was called with a non-zero flag. (If @Statistics has not been called with a non-zero flag before, the first call with the flag set returns statistics since startup.) The statistics that are affected by the delta flag are DRCONFLICTS, EXPORT, GC, IDLETIME, IMPORT, INITIATOR, IOSTATS, PLANNER, and the compound procedure and stored procedure statistics (COMPOUNDPROC, COMPOUNDPROCSUMMARY, PROCEDURE, PROCEDUREDETAIL, PROCEDUREINPUT, PROCEDUREOUTPUT, and PROCEDUREPROFILE).

VoltDB maintains separate statistics for each selector. As a result, when making two calls to @Statistics with a given selector and the delta flag set, the second call returns results since the first call, even if there was another call to @Statistics with a different selector in between. The one exception to this rule is the procedure statistics, where there is a single set of statistics maintained for all of the procedure selectors. Calling @Statistics with any of the procedure selectors resets the delta values for all of the procedure selectors.

Note that in a cluster with K-safety, if a node fails, the statistics reported by this procedure are reset to zero for the node when it rejoins the cluster.

The following are the allowable values of *component*:

"CLOCKSKEW"	Returns information about clock skew (that is, the difference in system clock time) between the nodes of the cluster. Note that the calculation of clock skew can be affected by other system activity, such as database transactions and Java garbage collection. This information should be considered only an approximation on heavily loaded servers.
"COMMANDLOG"	Returns information about the progress of command logging, including the number of segment files in use and the amount of command log data waiting to be written to disk.
"COMPACTION"	Returns information about the performance of periodic defragmentation of persistent memory used for storing tuple data. See the section on "How the Database Manages Memory" in the *Volt Guide to Performance and Customization* for more information on memory compaction.

"COMPOUNDPROC"	Returns information about the performance of compound procedures, per host per procedure. The information includes the names of the procedure and host, the number of invocations, and selected performance information on minimum, maximum, and average elapsed time.
"COMPOUNDPROCCALLS"	Returns information about which stored procedures the compound procedures invoke and how often.
"COMPOUNDPROCSUMMARY"	Returns a rollup of information about the performance of each compound procedure across all of the hosts. This is the same information provided by the COMPOUNDPROC selector, except aggregated per compound procedure.
"CPU"	Returns information about the amount of CPU used by each VoltDB server process. CPU usage is returned as a number between 0 and 100 representing the amount of CPU used by the VoltDB process out of the total CPU available for that server.
"DEPLOYMENTUPDATE"	Returns information about the number of configuration changes since the cluster started, including metrics on how long the update took, in microseconds.
"DRCONFLICTS"	Returns information about the type and number of XDCR conflicts occurring on the cluster. This information is available only if the database is licensed for and operating as an active XDCR database.
"DRCONSUMER"	Returns information about the status of database replication on a DR consumer, including the status and data replication rate of each partition. This information is available only if the database is licensed for database replication and operating as an active XDCR database.
"DRPRODUCER"	Returns information about the status of database replication on a producer database, including how much data is waiting to be sent to the consumer. This information is available only if the database is licensed for database replication and is operating as an active XDCR database.
"DRROLE"	Returns information about the current state of database replication (DR), including the role of the cluster (XDCR) and whether DR has started, is running, stopped, or been disabled.
"DRSCHEMA"	Returns information about the current state of the database schema for DR tables and, specifically, whether the schema match between the local and the remote clusters.
"EXPORT"	Returns statistics on the export streams and targets, including how many records have be written, how many are pending, and the status of the export connection.
"GC"	Returns statistics on Java garbage collection associated with the server process on each host.
"HOST"	Returns information about each host, including the assignment of partitions.
"IDLETIME"	Returns statistics on how busy the partitions are. For each execution site, the results provide a minimum, maximum, and average amount of time the site waited without any transactions to process, as well as the

	overall percentage of time the site was waiting (that is, the partition was "idle").
"IMPORT"	Returns statistics on the import streams, including how many import transactions have succeeded, failed, and been retried and how many rows have been read but not applied yet.
"INDEX"	Returns information about the indexes in the database, including the number of keys for each index and the estimated amount of memory used to store those keys. Separate information is returned for each partition in the database.
"INITIATOR"	Returns information on the number of procedure invocations for each stored procedure (including system and import procedures). The count of invocations is reported for each connection to the database.
"IOSTATS"	Returns information on the number of messages and amount of data (in bytes) sent to and from each connection to the database.
"LATENCY"	Returns statistics on the latency of transactions. The information reports on median, percentage (99% through 99.999%), and maximum latency over the most recent five second sampling period.
"LIMITS"	Returns information about limits on the number file descriptors and, as a consequence, the number of client connections on the servers.
"LIVECLIENTS"	Returns information about the number of outstanding requests per client. You can use this information to determine how much work is waiting in the execution queues.
"MANAGEMENT"	Returns the same information as MEMORY, INITIATOR, PROCEDURE, IOSTATS, TABLE, INDEX, IDLETIME, QUEUE, and CPU, except all in a single procedure call.
"MEMORY"	Returns statistics on the use of memory for each node in the cluster. MEMORY statistics include the current resident set size (RSS) of the VoltDB server process; the amount of memory used for Java temporary storage, database tables, indexes, and string (including varbinary) storage; as well as other information.
"PARTITIONCOUNT"	Returns information on the number of unique partitions in the cluster. The VoltDB cluster creates multiple partitions based on the number of servers and the number of sites per host requested. So, for example, a 2 node cluster with 4 sites per host will have 8 partitions. However, when you define a cluster with K-safety, there are duplicate partitions. PARTITIONCOUNT only reports the number of unique partitions available in the cluster.
"PLANNER"	Returns information on the use of cached plans within each partition. Queries in stored procedures are planned when the procedure is declared in the schema. However, ad hoc queries must be planned at runtime. To improve performance, VoltDB caches plans for ad hoc queries so they can be reused when a similar query is encountered later. There are two caches: the level 1 cache performs exact matches on queries and the level 2 cache parameterizes constants so it can match queries with the same plan but different input. The planner statistics

	provide information about the size of each cache, how frequently it is used, and the minimum, maximum, and average execution time of ad hoc queries as a result.
"PROCEDURE"	Returns information on the usage of stored procedures for each site within the database cluster sorted by partition. The information includes the name of the procedure, the number of invocations (for each site), and selected performance information on minimum, maximum, and average execution time.
"PROCEDUREDETAIL"	Returns detailed performance information about the individual statements within each stored procedure. PROCEDUREDETAIL returns information for each statement in each procedure, grouped by site and partition within the database cluster. The information includes the name of the procedure, the name of the statement, the number of invocations (for each site), and selected performance information on minimum, maximum, and average execution time.
"PROCEDUREINPUT"	Returns summary information on the size of the input data submitted with stored procedure invocations. PROCEDUREINPUT uses information from PROCEDURE, except it focuses on the input parameters and aggregates data for the entire cluster.
"PROCEDUREOUTPUT"	Returns summary information on the size of the result sets returned by stored procedure invocations. PROCEDUREOUTPUT uses information from PROCEDURE, except it focuses on the result sets and aggregates data for the entire cluster.
"PROCEDUREPROFILE"	Returns summary information on the usage of stored procedures averaged across all partitions in the cluster. The information from PROCEDUREPROFILE is similar to the information from PROCEDURE, except it focuses on the performance of the individual procedures rather than on procedures by partition. The weighted average across partitions is helpful for determining which stored procedures the application is spending most of its time in.
"QUEUE"	Returns statistics on the number of tasks in each partition's process queue and the average and maximum time tasks were waiting in the queue.
"QUEUEPRIORITY"	Returns statistics on the number of tasks in each priority queue for each partition and the average and maximum time tasks were waiting in the queue.
"REBALANCE"	Returns information on the current progress of rebalancing on the cluster. Rebalancing occurs when one or more nodes are added "on the fly" to an elastic cluster. If no rebalancing is occurring, no data is returned. During a rebalance, this selector returns information about the speed of migration of the data, the latency of rebalance tasks, and the estimated time until completion. All rebalance statistics are cumulative for the current rebalance activity.
"SNAPSHOTAUTOTUNING"	Returns information about the effect of autotuning on snapshot processing. The results include the snapshot prefix, the number of individual snapshot tasks run, how many were postponed to allow user transactions to take precedence, and statistics on the size of the individual

	tasks. While autotuning is enabled the number of preempted tasks and the size of tasks will vary as needed to accommodate the current workload. When autotuning is not enabled, there will be no preempted tasks and the size of the tasks will be fixed.
"SNAPSHOTSTATUS"	Returns information about the individual files of up to ten recent snapshots performed by the database. The results include the directory path and prefix for the snapshot, when it occurred, how long it took, and whether the snapshot was completed successfully or not. The results report on both native and CSV snapshots, as well as manual, automated, and command log snapshots. Note that this selector does not tell you whether the snapshot files still exist, only that the snapshot was performed. Use the @SnapshotScan procedure to determine what snapshots are available.
"SNAPSHOTSUMMARY"	Returns information about up to ten recent snapshots performed by the database. Unlike SNAPSHOTSTATUS, which reports on individual files, SNAPSHOTSUMMARY provides a one row summary for each snapshot. The results include the directory path and prefix, when the snapshot started, how long it took, and whether the snapshot was completed successfully or not. If the snapshot is still in progress, the results include the percentage complete. Note that this selector does not tell you whether the snapshot files still exist, only that the snapshot was started. Use the @SnapshotScan procedure to determine what snapshot files are available.
"TABLE"	Returns information about the database tables, including the number of rows per site for each table. This information can be useful for seeing how well the rows are distributed across the cluster for partitioned tables.
"TASK"	Returns information about scheduled tasks and their current status. There are separate queues for the task schedulers and the task procedures, which run on the standard transactional partition queues. The TASK selector reports on both. Separate selectors for TASK_SCHEDULER and TASK_PROCEDURE report on a corresponding subset of the columns.
"TLS"	Returns statistics about the time it takes to encrypt and decrypt messages for TLS/SSL transmission.
"TOPIC"	Returns statistics about the queues for outbound topics (that is, data fetched by consumers) including throughput, offsets, and retention policy, as well as how much data is currently stored on disk.
"TTL"	Returns information about the processing of expired data in "time to live" (TTL) tables, including how recently and how many records have been deleted.
"XDCR"	Returns information about the current state of XDCR communication, including whether the cluster supports dynamic schema changes.

Note that INITIATOR and PROCEDURE report information on both user-declared stored procedures and system procedures. These include certain system procedures that are used internally by VoltDB and are not intended to be called by client applications. Only the system procedures documented in this appendix are intended for client invocation.

Return Values

Returns different VoltTables depending on which component is requested. The following tables identify the structure of the return values for each component. (Note that the MANAGEMENT component returns seven VoltTables.)

CLOCKSKEW — Returns rows for every server in the cluster showing clock skew between it and every other server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was reported (in milliseconds). Note that the clock skew is calculated and cached periodically (once an hour, by default). It is not evaluated when the procedure is called.
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SKEW_TIME	INTEGER	The number of milliseconds difference between the system clock time of the current host and the remote host.
REMOTE_HOST_ID	INTEGER	Numeric ID for the remote host node.
REMOTE_HOSTNAME	STRING	Server name of the remote host node.

COMMANDLOG — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
OUTSTANDING_BYTES	BIGINT	The size, in bytes, of pending command log data. That is, data for transactions that have been initiated but the log has yet to be written to disk. For synchronous logging, this value is always zero.
OUTSTANDING_TXNS	BIGINT	The size, in number of transactions, of pending command log data. That is, the number of transactions that have been initiated for which the log has yet to be written to disk. For synchronous logging, this value is always zero.
IN_USE_SEGMENT_COUNT	INTEGER	The total number of segment files currently in use for command logging.
SEGMENT_COUNT	INTEGER	The number of segment files allocated, including currently unused segments.
FSYNC_INTERVAL	INTEGER	The average interval, in milliseconds, between the last 10 fsync system calls.
AVG_SEGMENT_ALLOC_TIME	BIGINT	The average time, in milliseconds, it took to allocate a new command log segment.
MAX_SEGMENT_ALLOC_TIME	BIGINT	The maximum time, in milliseconds, it took to allocate a new command log segment.

Name	Datatype	Description
BACK-PRESSURE_COUNT	BIGINT	The number of times back pressure was applied to transactions waiting to write to the command log .
LAST_BACK-PRESSURE_TIMESTAMP	BIGINT	The most recent timestamp when back pressure was applied.

COMPACTION — Returns rows for every table in every partition, per host.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	BIGINT	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
TABLE_NAME	STRING	The name of the database table.
INVOCATIONS	BIGINT	The total number of invocations of the compaction procedure for this table and partition
MIN_EXECUTION_TIME	BIGINT	The minimum length of time (in milliseconds) it took to compact the table data for this partition.
MAX_EXECUTION_TIME	BIGINT	The maximum length of time (in milliseconds) it took to compact the table data for this partition.
AVG_EXECUTION_TIME	BIGINT	The average length of time (in milliseconds) it took to compact the table data for this partition.
MIN_RELOCAT-ED_COUNT	BIGINT	The minimum number of tuples moved during a compaction event.
MAX_RELOCAT-ED_COUNT	BIGINT	The maximum number of tuples moved during a compaction event.
AVG_RELOCAT-ED_COUNT	BIGINT	The average number of tuples moved during a compaction event.
MIN_FRAGMENT-ED_PERCENT	FLOAT	The minimum percentage of fragmentation found during compaction.
MAX_FRAGMENT-ED_PERCENT	FLOAT	The maximum percentage of fragmentation found during compaction.
FRAGMENTED_PER-CENT	FLOAT	The percentage of fragmentation found after the most recent compaction event.

COMPOUNDPROC — Returns rows for every compound procedure that has been executed on the cluster, per procedure per host.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.

Name	Datatype	Description
HOSTNAME	STRING	Server name of the host node.
PROCEDURE	STRING	The name of the compound procedure.
INVOCATIONS	BIGINT	The total number of invocations of this procedure on this host.
MIN_ELAPSED_TIME	BIGINT	The minimum elapsed length of time (in nanoseconds) to run the compound procedure.
MAX_ELAPSED_TIME	BIGINT	The maximum elapsed length of time (in nanoseconds) to run the compound procedure.
AVG_ELAPSED_TIME	BIGINT	The average elapsed length of time (in nanoseconds) to run the compound procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly (as opposed to user aborts).

COMPOUNDPROCCALLS — Returns a row for every compound procedure and each stored procedure it has invoked, separated by host server.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PROCEDURE	STRING	The name of the compound procedure.
-ECORP_ CALLED # DURE	STRING	The name of the stored procedure invoked by the compound procedure.
INVOCATIONS	BIGINT	The total number of invocations of the stored procedure by the compound procedure on this host.

COMPOUNDPROCSUMMARY — Returns a row for every compound procedure that has been executed on the cluster, aggregated across all nodes of the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PROCEDURE	STRING	The name of the compound procedure.
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
MIN_ELAPSED_TIME	BIGINT	The minimum elapsed length of time (in nanoseconds) to run the compound procedure.
MAX_ELAPSED_TIME	BIGINT	The maximum elapsed length of time (in nanoseconds) to run the compound procedure.
AVG_ELAPSED_TIME	BIGINT	The average elapsed length of time (in nanoseconds) to run the compound procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly (as opposed to user aborts).

CPU — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PERCENT_USED	BIGINT	The percentage of total CPU available used by the database server process.

DEPLOYMENTUPDATE — Returns a row for every server in the cluster. The statistics return two sets of values: the minimum, maximum, and average of the overall elapsed time taken to complete configuration updates and the same calculations for that portion of the overall elapsed time during which other transactions are blocked.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
UPDATE_COUNT	BIGINT	The number of configuration updates since the database cluster first started.
MIN_ELAPSED	BIGINT	The minimum elapsed time (in microseconds) to complete a configuration update.
MAX_ELAPSED	BIGINT	The maximum elapsed time (in microseconds) to complete a configuration update.
AVG_ELAPSED	BIGINT	The average elapsed time (in microseconds) to complete a configuration update.
MIN_BLOCKED	BIGINT	The minimum elapsed time (in microseconds) other transactions were blocked by an update.
MAX_BLOCKED	BIGINT	The maximum elapsed time (in microseconds) other transactions were blocked by an update.
AVG_BLOCKED	BIGINT	The average elapsed time (in microseconds) other transactions were blocked by an update.

DRCONFLICTS — Returns a row per table per partition per server.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the remote cluster.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.

Name	Datatype	Description
TABLE_NAME	STRING	The name of the database table where the conflicts are occurring.
LAST_CONFLICT_TIMESTAMP	TIMESTAMP	The timestamp of the last conflict.
TOTAL_CONFLICT_COUNT	BIGINT	The total number of conflicts that have been recorded for this table in this partition. This is the sum of the missing row, timestamp mismatch, and constraint violation counts.
DIVERGENCE_COUNT	BIGINT	The number of conflicts that may have resulted in divergence between the clusters, which is a subset of the total conflicts.
MISSING_ROW_COUNT	BIGINT	The number of missing row conflicts that occurred.
TIMESTAMP_MISMATCH_COUNT	BIGINT	The number of timestamp mismatch conflicts that occurred.
CONSTRAINT_VIOLATION_COUNT	BIGINT	The number of constraint violation conflicts that occurred.

DRCONSUMER — Returns three VoltTables. The first table returns a row for every host in the cluster, showing whether a replication snapshot is in progress and if it is, the status of transmission to the consumer.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the producer cluster.
STATE	STRING	A text string indicating the current state of replication. Possible values are: • UNINITIALIZED — DR has not begun yet or has stopped • INITIALIZE — DR is enabled and the consumer is attempting to contact the producer • SYNC — DR has started and the consumer is synchronizing by receiving snapshots of existing data from the producer • RECEIVE — DR is underway and the consumer is receiving binary logs from the producer • DISABLE — DR has been canceled for some reason and the consumer is stopping DR
REPLICATION_RATE_1M	BIGINT	The average rate of replication over the past minute. The data rate is measured in bytes per second.
REPLICATION_RATE_5M	BIGINT	The average rate of replication over the past five minutes. The data rate is measured in bytes per second.
REMOTE_CREATION_TIMESTAMP	TIMESTAMP	The timestamp when the remote cluster started for the first time.

Name	Datatype	Description
QUEUE_DEPTH	BIGINT	The number of tasks currently in the execution queue.
TASKS	STRING	A comma-separated list of the names of the tasks at the head of the execution queue, including the currently executing transaction.

The second table contains information about the replication streams, which consist of a row per partition for each server. The data shows the current state of replication and how much data has been received by the consumer from each producer.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the producer cluster.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.
IS_COVERED	STRING	A text string of "true" or "false" indicating whether this partition is currently connected to and receiving data from a matching partition on the producer cluster.
COVERING_HOST	STRING	The host name of the server in the producer cluster that is providing DR data to this partition. If IS_COVERED is "false", this field is empty.
LAST_RECEIVED_TIMESTAMP	TIMESTAMP	The timestamp of the last transaction received from the producer.
LAST_APPLIED_TIMESTAMP	TIMESTAMP	The timestamp of the last transaction successfully applied to this partition on the consumer.
IS_PAUSED	STRING	A text string of "true" or "false" indicating whether this partition is paused. A partition "pauses" when the schema of the DR tables on the producer change to no longer match the consumer and all binary logs prior to the change have been processed.
DUPLICATE_BUFFERS	BIGINT	The number of repeated buffers received after the initial packets were dropped because the queue was full.
IGNORED_BUFFERS	BIGINT	The number of buffers received but dropped because the queue was full.
AVAILABLE_BYTES	INTEGER	The number of free bytes left in the DR queue.
AVAILABLE_BUFFERS	INTEGER	The number of free buffers left in the DR queue.
CONSUMER_LIMIT_TYPE	STRING	The type of limit on the DR queue. The response is either BYTES or BUFFERS.
LAST_APPLIED_PROTOCOL_VERSION	INTEGER	The current DR protocol version of binary logs being received and applied for this partition.
PARTITIONS_MATCH	STRING	A text string of "true" or "false" indicating whether the two clusters have the same number of unique partitions or not.

The third table is primarily for use by support. It returns a row for every host in the cluster with information for use in debugging XDCR issues.

Name	Datatype	Description
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the consumer cluster.
STATE	STRING	A text string indicating the current state of replication. Possible values are UNINITIALIZED, INITIALIZE, SYNC, RECEIVE, and DISABLE.
LAST_FAILURE_CODE	INTEGER	A numeric code reserved for use by Volt Active Data support.

DRPRODUCER — Returns three VoltTables. The first table contains information about the replication streams, which consist of a row per partition for each server. The data shows the current state of replication and how much data is currently queued for each consumer.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the consumer cluster.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.
STREAMTYPE	STRING	The type of stream, which can either be "TRANSACTIONS" or "SNAPSHOT".
TOTALBYTES	BIGINT	The total number of bytes currently queued for transmission to the consumer.
TOTALBYTESIN MEMORY	BIGINT	The total number of bytes of queued data currently held in memory. If the amount of total bytes is larger than the amount in memory, the remainder is kept in overflow storage on disk.
TOTALBUFFERS	BIGINT	The total number of buffers in this partition currently waiting for acknowledgement from the consumer. The partitions buffer the binary logs to reduce overhead and optimize network transfers.
LASTQUEUEDDRID	BIGINT	The ID of the last transaction queued for transmission to the consumer.
LASTACKDRID	BIGINT	The ID of the last transaction acknowledged by the consumer.
LASTQUEUEDTIMESTAMP	TIMESTAMP	The timestamp of the last transaction queued for transmission to the consumer.
LASTACKTIMESTAMP	TIMESTAMP	The timestamp of the last transaction acknowledged by the consumer.
ISSYNCED	STRING	A text string indicating whether the database is currently being replicated. If replication has not started, or the overflow

Name	Datatype	Description
		capacity has been exceeded (that is, replication has failed), the value of ISSYNCED is "false". If replication is currently in progress, the value is "true".
MODE	STRING	A text string indicating whether this particular partition is replicating data to the consumer ("NORMAL") or not ("PAUSED"). Only one copy of each logical partition actually sends data during replication. So for clusters with a K-safety value greater than zero, not all physical partitions will report "NORMAL" even when replication is in progress.
QUEUE_GAP	BIGINT	The number of missing transactions between those already acknowledged by the consumer and the next available for transmission. Under normal operating conditions, this value is zero.
CONNECTION_STATUS	STRING	A text string indicating whether the connection to the consumer is operational ("UP") or not ("DOWN"). If the connection between the producer and consumer is broken or if the producer does not hear from the consumer for more than 30 seconds, the connection is marked as "DOWN".
AVAILABLE_BYTES	INTEGER	The number of bytes waiting to be sent to the consumer
AVAILABLE_BUFFERS	INTEGER	The number of buffers waiting to be sent to the consumer.
CONSUMER_LIMIT_TYPE	STRING	The type of limit on the DR queue. The response is either BYTES or BUFFERS.
DR_ROUNDTRIPTIME_1MINUTE_MAX	BIGINT	The maximum time in milliseconds it took to receive acknowledgement from the consumer, over the past minute.
DR_ROUNDTRIPTIME_1MINUTE_AVG	BIGINT	The average time in milliseconds it took to receive acknowledgement from the consumer, over the past minute.
DR_ROUNDTRIPTIME_51MINUTE_MAX	BIGINT	The maximum time in milliseconds it took to receive acknowledgement from the consumer, over the past five minutes.
DR_ROUNDTRIPTIME_5MINUTE_AVG	BIGINT	The average time in milliseconds it took to receive acknowledgement from the consumer, over the past five minutes.
CURRENT_DR_PROTOCOL	SMALLINT	The DR protocol version currently in use when sending data to consumers.
SUPPORTED_DR_PROTOCOL	SMALLINT	The highest version of DR protocol this cluster is capable of using to send data to consumers.

The second table returns a row for every host in the cluster, showing whether a replication snapshot is in progress and if it is, the status of transmission to the consumer.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the consumer cluster.

Name	Datatype	Description
STATE	STRING	A text string indicating the current state of replication. Possible values are "OFF" (replication is not enabled), "PENDING" (replication is enabled but not occurring), and "ACTIVE" (replication is enabled and a remote database has initiated DR).
SYNCSNAPSHOTSTATE	STRING	A text string indicating the current state of the synchronization snapshot that begins replication. During normal operation, this value is "NONE" indicating either that replication is not active or that transactions are actively being replicated. If a synchronization snapshot is in progress, this value provides additional information about the specific activity underway.
ROWSINSYNC SNAPSHOT	BIGINT	Reserved for future use.
ROWSACKEDFORSYNC SNAPSHOT	BIGINT	Reserved for future use.
QUEUEDEPTH	BIGINT	The number of DR tasks waiting to be processed.
REMOTECREATION-TIMESTAMP	BIGINT	The timestamp when the remote cluster started for the first time.
TASKS	STRING	A comma-separated list of the names of the tasks at the head of the execution queue, including the currently executing transaction.

The third table returns a row for every other cluster in the XDCR environment, describing its current state and most recent error code (for debugging purposes).

Name	Datatype	Description
CLUSTER_ID	INTEGER	The numeric ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The numeric ID of the producer cluster.
STATE	STRING	A text string indicating the current state of replication. Possible values are UNINITIALIZED, INITIALIZE, SYNC, RECEIVE, and DISABLE.
LAST_FAILURE_CODE	INTEGER	A numeric code reserved for use by Volt Active Data support.
SUPPORTED_DR_PROTOCOL	SMALLINT	The highest version of DR protocol this cluster is capable of using to send data to consumers.

DRROLE — Returns one row per connection showing the current status of DR for that cluster.

Name	Datatype	Description
ROLE	STRING	The role of the current cluster in a DR relationship. Possible values are NONE and XDCR. (None indicates that no DR ID is defined and the cluster cannot participate in DR.)
STATE	STRING	The current state of the DR relationship. Possible values are the following: • DISABLED — DR is not enabled for the cluster

Name	Datatype	Description
		• PENDING — DR is enabled but communication with the other cluster has not begun • ACTIVE — Communication with the other cluster has begun • STOPPED — Communication with the other cluster has stopped due to a failure of some kind Note that if DR stops, issuing the **voltadmin dr reset** command will return the cluster to the PENDING state.
REMOTE_CLUSTER_ID	INTEGER	The DR ID of the other DR cluster, or -1 if not available (for example, when DR is disabled or communication has not begun).
SUPPORTED_DR_PROTOCOL	SMALLINT	The highest version of DR protocol this cluster is capable of using to send data to consumers.
PARTITIONS_MATCH	STRING	A text string of "true" or "false" indicating whether the two clusters have the same number of unique partitions or not.

DRSCHEMA — Returns a row for every table, per XDCR connection. The data describes the current state of the table's schema and whether it matches the schema on the remote XDCR cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
TABLE_NAME	STRING	The name of the database table.
CLUSTER_ID	INTEGER	The DR ID of the current cluster.
REMOTE_CLUSTER_ID	INTEGER	The DR ID of the remote cluster.
SCHEMA_MATCH	STRING	A text string of "true" or "false" indicating whether the schema for the table matches on the two clusters.
SCHEMACHANGE_TUPLE_COUNT	BIGINT	The total number of tuples exchanged for this tuple while the schema did not match.
LAST_APPLIED_DR_PROTOCOL	INTEGER	For internal use only.

EXPORT — Returns a separate row for each export stream per partition.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.
SOURCE	STRING	The name of the export stream.
TARGET	STRING	The name of the export target.
ACTIVE	STRING	Whether this site is currently actively exporting data. For normal export in a K-safe cluster, only one copy of each

Name	Datatype	Description
		partition actively exports data at any given time. Possible values for user export are "TRUE" and "FALSE". Note that cross datacenter replication (XDCR) uses the export infrastructure to write the DR conflict logs. In this case, all copies of the partition write the logs and the ACTIVE column is marked as "XDCR" to distinguish it from user-defined export streams.
TUPLE_COUNT	BIGINT	The total number of records queued to the export target since the queue was created. (Not affected by the delta flag.)
TUPLE_PENDING	BIGINT	The number of records out of TUPLE_COUNT still waiting to be written to or acknowledged by the target.
LAST_QUEUED_TIMESTAMP	TIMESTAMP	The timestamp when the most recent tuple was added to the export queue for this partition (in milliseconds).
LAST_ACKED_TIMESTAMP	TIMESTAMP	The timestamp when the last tuple was acknowledged as received by the target (in milliseconds).
AVERAGE_LATENCY	BIGINT	The average time between when records are inserted and they are acknowledged by the target.
MAX_LATENCY	BIGINT	The maximum time between when a record was inserted and it was acknowledged by the target.
QUEUE_GAP	BIGINT	The number of records missing from the queue for the current stream and partition.
STATUS	STRING	The current status of the export connection. Possible values are the following: • ACTIVE — Queue is currently exporting to the target • BLOCKED — There is a gap in the queue and export is waiting to see if the missing records become available when a missing node rejoins • DROPPED — either the source stream has been dropped from the schema or the export configuration has been removed from the configuration and queue is draining any remaining records Note that if the queue is blocked, the **voltadmin export release** command returns the queue to the ACTIVE state.

GC — Returns a separate row for each host.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
NEWGEN_GC_COUNT	INTEGER	The number of times "young generation" garbage collection was performed.
NEWGEN_AVG_GC_TIME	BIGINT	The average time (in milliseconds) taken by young generation collections.

Name	Datatype	Description
OLDGEN_GC_COUNT	INTEGER	The number of times "old generation" garbage collection was performed.
OLD-GEN_AVG_GC_TIME	BIGINT	The average time (in milliseconds) taken by old generation collections.

HOST — Returns two tables. The first table returns a separate row for each host in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PARTITIONS	STRING	Comma-separated list of partition IDs assigned to this host.
LEADERS	STRING	Comma-separated list of partitions for which this host is the leader.
PLACEMENTGROUP	STRING	The placement group to which this host is assigned.
SAFETOSTOP	STRING	A text string of "true" or "false" indicating whether it is safe to stop this node. That is, the database is K-safe and will continue to run if this host leaves the cluster.
REGION	STRING	For Kubernetes, the region in which the server is running.
ZONE	STRING	For Kubernetes, the availability zone in which the server is running.

The second table returns one row for each placement, indicating whether K-safety allows the database to continue running if the servers in the placement group are shut down.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PLACEMENTGROUP	STRING	The name of the placement group
SAFETOSTOP	STRING	A text string of "true" or "false" indicating whether it is safe to stop all nodes in this placement group. That is, the database is K-safe and will continue to run if this placement group leaves the cluster.

IDLETIME — Returns a separate row for each execution site and host.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
COUNT	BIGINT	The number of times the execution site had to wait for a new task (that is, the queue was empty).

Name	Datatype	Description
PERCENT	FLOAT	The percentage of time the execution site was waiting for a new task (that is, the site was "idle").
AVG	BIGINT	The average amount of time the execution site had to wait for a new task (in microseconds).
MIN	BIGINT	The minimum amount of time the execution site had to wait for a new task (in nanoseconds).
MAX	BIGINT	The maximum amount of time the execution site had to wait for a new task (in nanoseconds).
STDDEV	BIGINT	The standard deviation of the waiting time (in microseconds).

IMPORT — Returns a separate row for each import stream and each server.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
IMPORTER_NAME	STRING	The name of the import stream.
PROCEDURE_NAME	STRING	The name of the stored procedure invoked by the import stream to insert the incoming data.
SUCCESSES	BIGINT	The number of import transactions that succeeded.
FAILURES	BIGINT	The number of import transactions that failed.
OUTSTANDING_REQUESTS	BIGINT	The number of records read from the import stream and waiting to be inserted into the database.
RETRIES	BIGINT	The number of attempts to replay failed transactions.

INDEX — Returns a row for every index in every execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	BIGINT	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
INDEX_NAME	STRING	The name of the index.
TABLE_NAME	STRING	The name of the database table to which the index applies.
INDEX_TYPE	STRING	A text string identifying whether the index is unique or not. Possible values include the following: CompactingTreeMultiMapIndex

Name	Datatype	Description
		CompactingTreeUniqueIndex
IS_UNIQUE	TINYINT	A byte value specifying whether the index is unique (1) or not (0).
IS_COUNTABLE	TINYINT	A byte value specifying whether the index maintains a counter to optimize COUNT(*) queries.
ENTRY_COUNT	BIGINT	The number of index entries currently in the partition.
MEMORY_ESTIMATE	BIGINT	The estimated amount of memory (in kilobytes) consumed by the current index entries.

INITIATOR — Returns a separate row for each connection and the stored procedures initiated by that connection.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CONNECTION_HOST NAME	STRING	The server name of the node from which the client connection originates. In the case of import procedures, the name of the importer is reported here.
PROCEDURE_NAME	STRING	The name of the stored procedure. If import is enabled, import procedures are included as well.
INVOCATIONS	BIGINT	The number of times the stored procedure has been invoked by this connection on this host node.
AVG_EXECUTION_TIME	INTEGER	The average length of time (in milliseconds) it took to execute the stored procedure.
MIN_EXECUTION_TIME	INTEGER	The minimum length of time (in milliseconds) it took to execute the stored procedure.
MAX_EXECUTION_TIME	INTEGER	The maximum length of time (in milliseconds) it took to execute the stored procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

IOSTATS — Returns one row for every client connection on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.

Name	Datatype	Description
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CONNECTION_HOST NAME	STRING	The server name of the node from which the client connection originates.
BYTES_READ	BIGINT	The number of bytes of data sent from the client to the host.
MESSAGES_READ	BIGINT	The number of individual messages sent from the client to the host.
BYTES_WRITTEN	BIGINT	The number of bytes of data sent from the host to the client.
MESSAGES_WRITTEN	BIGINT	The number of individual messages sent from the host to the client.

LATENCY — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp, in milliseconds, when the data was collected (not when the call was processed). If two calls to this selector return the same timestamp, the data being returned is identical.
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
INTERVAL	INTEGER	The length of the measurement interval, in milliseconds. The interval is five seconds (5000).
COUNT	INTEGER	The total number of transactions during the interval.
TPS	INTEGER	The number of transactions per second during the interval.
P50	BIGINT	The 50th percentile latency, in microseconds. This value measures the median latency.
P95	BIGINT	The 95th percentile latency, in microseconds.
P99	BIGINT	The 99th percentile latency, in microseconds.
P99.9	BIGINT	The 99.9th percentile latency, in microseconds.
P99.99	BIGINT	The 99.99th percentile latency, in microseconds.
P99.999	BIGINT	The 99.999th percentile latency, in microseconds.
MAX	BIGINT	The maximum latency during the interval, in microseconds.

LIMITS — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was reported (in milliseconds). Note that the number of file descriptors (maximum and currently open) is calculated periodically and cached rather than at the time of each call.
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
FILE_DESCRIPTORS_LIMIT	INTEGER	The maximum number of file descriptors allowed for the process running the server.

Name	Datatype	Description
FILE_DESCRIP-TORS_OPEN	INTEGER	The number of file descriptors currently open in the process.
CLIENT_CONNEC-TIONS_LIMIT	INTEGER	The maximum number of client connections allowed for the server.
CLIENT_CONNEC-TIONS_OPEN	INTEGER	The number of client connections currently open on the server.
ACCEPTED_CONNEC-TIONS	INTEGER	The total number of client connections opened since the server started, including connections that are now closed.
DROPPED_CONNEC-TIONS	INTEGER	The total number of connections that were rejected because the connection limit had been reached.

LIVECLIENTS — Returns two VoltTables. The first contains one row for every client connection currently active on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection invoking the procedure.
CLIENT_HOSTNAME	STRING	The server name of the node from which the client connection originates.
ADMIN	TINYINT	A byte value specifying whether the connection is to the client port (0) or the admin port (1).
OUTSTANDING_ REQUEST_BYTES	BIGINT	The number of bytes of data sent from the client currently pending on the host.
OUTSTANDING_ RESPONSE_MESSAGES	BIGINT	The number of messages on the host queue waiting to be retrieved by the client.
OUTSTANDING_ TRANSACTIONS	BIGINT	The number of transactions (that is, stored procedures) initiated on behalf of the client that have yet to be completed.

The second VoltTable returns one row for every server in the cluster. Note this second table repeats the information returned by the LIMITS selector.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was reported (in milliseconds). Note that due to the resources needed to determine the number of file descriptors (maximum and currently open), this information is calculated periodically and cached rather than at the time of each call.
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
FILE_DESCRIP-TORS_LIMIT	INTEGER	The maximum number of file descriptors allowed for the process running the server.

Name	Datatype	Description
FILE_DESCRIP-TORS_OPEN	INTEGER	The number of file descriptors currently open in the process
CLIENT_CONNEC-TIONS_LIMIT	INTEGER	The maximum number of client connections allowed for the server.
CLIENT_CONNEC-TIONS_OPEN	INTEGER	The number of client connections currently open on the server.
ACCEPTED_CONNEC-TIONS	INTEGER	The total number of client connections opened since the server started, including connections that are now closed.
DROPPED_CONNEC-TIONS	INTEGER	The total number of connections that were rejected because the connection limit had been reached.

MEMORY — Returns a row for every server in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
RSS	INTEGER	The current resident set size. That is, the total amount of memory allocated to the VoltDB processes on the server.
JAVAUSED	INTEGER	The amount of memory (in kilobytes) allocated by Java and currently in use by VoltDB.
JAVAUNUSED	INTEGER	The amount of memory (in kilobytes) allocated by Java but unused. (In other words, free space in the Java heap.)
TUPLEDATA	BIGINT	The amount of memory (in kilobytes) currently in use for storing database records.
TUPLEALLOCATED	BIGINT	The amount of memory (in kilobytes) allocated for the storage of database records (including free space).
INDEXMEMORY	BIGINT	The amount of memory (in kilobytes) currently in use for storing database indexes.
STRINGMEMORY	BIGINT	The amount of memory (in kilobytes) currently in use for storing string, binary, and geospatial data that is not stored "in-line" in the database record.
TUPLECOUNT	BIGINT	The total number of database records currently in memory.
POOLEDMEMORY	BIGINT	The total size of memory (in kilobytes) allocated for tasks other than database records, indexes, and strings. (For example, pooled memory is used for temporary tables while processing stored procedures.)
PHYSICALMEMORY	BIGINT	The total size of physical memory (in kilobytes) on the server.
JAVAMAXHEAP	INTEGER	The maximum heap size (in kilobytes) of the Java runtime environment.
UNDO_POOL_SIZE	BIGINT	The total size of memory (in kilobytes) allocated for the undo pool — memory used to store information needed to "undo" database changes if a transaction needs to rollback.

Name	Datatype	Description
UNDO_LOG_SIZE	BIGINT	The total size of memory (in kilobytes) allocated for the undo log.
NIO_TOTAL_BUFER_COUNT	BIGINT	The total number of buffers allocated for Java NIO direct memory.
NIO_TOTAL_SIZE	BIGINT	The total size of memory (in kilobytes) allocated for Java NIO direct memory.
NIO_USED	BIGINT	The total size of memory (in kilobytes) in use for Java NIO direct memory.

PARTITIONCOUNT — Returns one row identifying the total number of partitions and the host that provided that information.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PARTITION_COUNT	INTEGER	The number of unique or logical partitions on the cluster. When using a K value greater than zero, there are multiple copies of each logical partition.

PLANNER — Returns a row for every planner cache. That is, one cache per execution site, plus one global cache per server. (The global cache is identified by a site and partition ID of minus one.)

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
CACHE1_LEVEL	INTEGER	The number of query plans in the level 1 cache.
CACHE2_LEVEL	INTEGER	The number of query plans in the level 2 cache.
CACHE1_HITS	BIGINT	The number of queries that matched and reused a plan in the level 1 cache.
CACHE2_HITS	BIGINT	The number of queries that matched and reused a plan in the level 2 cache.
CACHE_MISSES	BIGINT	The number of queries that had no match in the cache and had to be planned from scratch
PLAN_TIME_MIN	BIGINT	The minimum length of time (in nanoseconds) it took to complete the planning of ad hoc queries.
PLAN_TIME_MAX	BIGINT	The maximum length of time (in nanoseconds) it took to complete the planning of ad hoc queries.

Name	Datatype	Description
PLAN_TIME_AVG	BIGINT	The average length of time (in nanoseconds) it took to complete the planning of ad hoc queries.
FAILURES	BIGINT	The number of times planning for an ad hoc query failed.

PROCEDURE — Returns a row for every stored procedure that has been executed on the cluster, grouped by execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
PROCEDURE	STRING	The class name of the stored procedure.
INVOCATIONS	BIGINT	The total number of invocations of this procedure at this site.
TIMED_INVOCATIONS	BIGINT	The number of invocations used to measure the minimum, maximum, and average execution time.
MIN_EXECUTION_TIME	BIGINT	The minimum length of time (in nanoseconds) it took to execute the stored procedure.
MAX_EXECUTION_TIME	BIGINT	The maximum length of time (in nanoseconds) it took to execute the stored procedure.
AVG_EXECUTION_TIME	BIGINT	The average length of time (in nanoseconds) it took to execute the stored procedure.
MIN_RESULT_SIZE	INTEGER	The minimum size (in bytes) of the results returned by the procedure.
MAX_RESULT_SIZE	INTEGER	The maximum size (in bytes) of the results returned by the procedure.
AVG_RESULT_SIZE	INTEGER	The average size (in bytes) of the results returned by the procedure.
MIN_PARAMETER_SET_SIZE	INTEGER	The minimum size (in bytes) of the parameters passed as input to the procedure.
MAX_PARAMETER_SET_SIZE	INTEGER	The maximum size (in bytes) of the parameters passed as input to the procedure.
AVG_PARAMETER_SET_SIZE	INTEGER	The average size (in bytes) of the parameters passed as input to the procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)
TRANSACTIONAL	TINYINT	0 or 1. Reserved for future use.

Name	Datatype	Description
COMPOUND	TINYINT	0 or 1. Whether this is a compound procedure (1) or not (0).

PROCEDUREDETAIL — Returns a row for every statement in every stored procedure that has been executed on the cluster, grouped by execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PARTITION_ID	INTEGER	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
PROCEDURE	STRING	The class name of the stored procedure.
STATEMENT	STRING	The name of the statement in the stored procedure. Cumulative statistics for all statements in the procedure are included in a separate row labeled "<ALL>".
INVOCATIONS	BIGINT	The total number of invocations of the statement at this site.
TIMED_INVOCATIONS	BIGINT	The number of invocations used to measure the minimum, maximum, and average execution time.
MIN_EXECUTION_TIME	BIGINT	The minimum length of time (in nanoseconds) it took to execute the statement.
MAX_EXECUTION_TIME	BIGINT	The maximum length of time (in nanoseconds) it took to execute the statement.
AVG_EXECUTION_TIME	BIGINT	The average length of time (in nanoseconds) it took to execute the statement.
MIN_RESULT_SIZE	INTEGER	The minimum size (in bytes) of the results returned by the statement.
MAX_RESULT_SIZE	INTEGER	The maximum size (in bytes) of the results returned by the statement.
AVG_RESULT_SIZE	INTEGER	The average size (in bytes) of the results returned by the statement.
MIN_PARAMETER _SET_SIZE	INTEGER	The minimum size (in bytes) of the parameters passed as input to the statement.
MAX_PARAMETER _SET_SIZE	INTEGER	The maximum size (in bytes) of the parameters passed as input to the statement.
AVG_PARAMETER _SET_SIZE	INTEGER	The average size (in bytes) of the parameters passed as input to the statement.
ABORTS	BIGINT	In the cumulative row for each procedure ("<ALL>"), the number of times the procedure was aborted. For individual statements, this column is set to zero.
FAILURES	BIGINT	The number of times the statement failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

PROCEDUREINPUT — Returns a row for every stored procedure that has been executed on the cluster, summarized across the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PROCEDURE	STRING	The class name of the stored procedure.
WEIGHTED_PERC	BIGINT	A weighted average expressed as a percentage of the parameter set size for invocations of this stored procedure compared to all stored procedure invocations.
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
MIN_PARAMETER _SET_SIZE	BIGINT	The minimum parameter set size in bytes.
MAX_PARAMETER _SET_SIZE	BIGINT	The maximum parameter set size in bytes.
AVG_PARAMETER _SET_SIZE	BIGINT	The average parameter set size in bytes.
TOTAL_PARAMETER _SET_SIZE_MB	BIGINT	The total input for all invocations of this stored procedure measured in megabytes.

PROCEDUREOUTPUT — Returns a row for every stored procedure that has been executed on the cluster, summarized across the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PROCEDURE	STRING	The class name of the stored procedure.
WEIGHTED_PERC	BIGINT	A weighted average expressed as a percentage of the result set size returned by invocations of this stored procedure compared to all stored procedure invocations.
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
MIN_RESULT_SIZE	BIGINT	The minimum result set size in bytes.
MAX_RESULT_SIZE	BIGINT	The maximum result set size in bytes.
AVG_RESULT_SIZE	BIGINT	The average result set size in bytes.
TOTAL_RESULT _SIZE_MB	BIGINT	The total output returned by all invocations of this stored procedure measured in megabytes.

PROCEDUREPROFILE — Returns a row for every stored procedure that has been executed on the cluster, summarized across the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
PROCEDURE	STRING	The class name of the stored procedure.
WEIGHTED_PERC	BIGINT	A weighted average expressed as a percentage of the execution time for this stored procedure compared to all stored procedure invocations.

Name	Datatype	Description
INVOCATIONS	BIGINT	The total number of invocations of this procedure.
AVG	BIGINT	The average length of time (in nanoseconds) it took to execute the stored procedure.
MIN	BIGINT	The minimum length of time (in nanoseconds) it took to execute the stored procedure.
MAX	BIGINT	The maximum length of time (in nanoseconds) it took to execute the stored procedure.
ABORTS	BIGINT	The number of times the procedure was aborted.
FAILURES	BIGINT	The number of times the procedure failed unexpectedly. (As opposed to user aborts or expected errors, such as constraint violations.)

QUEUE — Returns a separate row for each partition and host listing the current state of the process queue for that execution site.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
CURRENT_DEPTH	INTEGER	The number of tasks currently in the queue.
POLL_COUNT	BIGINT	The number of tasks that left the queue (and started executing) in the past five seconds.
AVG_WAIT	BIGINT	The average length of time (in microseconds) tasks were waiting in the queue in the last five seconds.
MAX_WAIT	BIGINT	The maximum length of time (in microseconds) tasks were waiting in the queue in the last five seconds.

QUEUEPRIORITY — Returns a separate row for each partition and host listing the current state of the priority queues for that execution site (if prioritization is enabled).

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
PRIORITY	INTEGER	The priority of the tasks in the queue.
CURRENT_DEPTH	INTEGER	The number of tasks currently in the queue.
POLL_COUNT	BIGINT	The number of tasks that left the queue (and started executing) in the past five seconds.
AVG_WAIT	BIGINT	The average length of time (in microseconds) tasks were waiting in the queue in the last five seconds.

Name	Datatype	Description
MAX_WAIT	BIGINT	The maximum length of time (in microseconds) tasks were waiting in the queue in the last five seconds.

REBALANCE — Returns one row if the cluster is rebalancing. No data is returned if the cluster is not rebalancing.

Name	Datatype	Description
TOTAL_RANGES	BIGINT	The total number of partition segments to be migrated.
PERCENTAGE_MOVED	FLOAT	The percentage of the total segments that have already been moved.
MOVED_ROWS	BIGINT	The number of rows of data that have been moved.
ROWS_PER_SECOND	FLOAT	The average number of rows moved per second.
ESTIMATED _REMAINING	BIGINT	The estimated time remaining until the rebalance is complete, in milliseconds.
MEGABYTES_PER _SECOND	FLOAT	The average volume of data moved per second, measured in megabytes.
CALLS_PER_SECOND	FLOAT	The average number of rebalance work units, or transactions, executed per second.
CALLS_LATENCY	FLOAT	The average total time between start and finish of rebalance operations, in milliseconds.
CALLS_TIME	FLOAT	The average execution time for rebalance transactions, in milliseconds.
CAL-LS_TRANSFER_TIME	FLOAT	The average time spent transferring data during rebalance transactions, in milliseconds.

SNAPSHOTAUTOTUNING — Returns a row for every table per execution site, reporting the effects of autotuning on the execution of recent snapshots.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	BIGINT	Numeric ID of the execution site on the host node.
TABLE	STRING	The name of the database table being saved to a snapshot.
NONCE	STRING	The unique identifier for the snapshot.
IN_PROGRESS	STRING	String indicating whether the snapshot is still under way (TRUE) or has been completed (FALSE).
COLLECTION_ITER-ATIONS	BIGINT	The number of individual snapshot tasks completed
COLLECTION_DU-RATION	BIGINT	The total length of time in nanoseconds spent collecting data for this table and snapshot.
WRITE_DURATION	BIGINT	The total length of time in nanoseconds spent writing data to disk for this table and snapshot.

Name	Datatype	Description
PREEMPT_PERCENT	BIGINT	The number of snapshot tasks that were requeued due to transaction workload superseding them.
MAX_BUFFER_SIZE	BIGINT	The maximum amount of data (in bytes) processed during a snapshot task.
MIN_BUFFER_SIZE	BIGINT	The minimum amount of data (in bytes) processed during a snapshot task.

SNAPSHOTSTATUS — Returns a row for every snapshot file in the recent snapshots performed on the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
TABLE	STRING	The name of the database table whose data the file contains.
PATH	STRING	The directory path where the snapshot file resides.
FILENAME	STRING	The file name.
NONCE	STRING	The unique identifier for the snapshot.
TXNID	BIGINT	The transaction ID of the snapshot.
START_TIME	BIGINT	The timestamp when the snapshot began (in milliseconds).
END_TIME	BIGINT	The timestamp when the snapshot was completed (in milliseconds).
SIZE	BIGINT	The total size, in bytes, of the file.
DURATION	BIGINT	The length of time (in seconds) it took to complete the snapshot.
THROUGHPUT	FLOAT	The average number of bytes per second written to the file during the snapshot process.
RESULT	STRING	String value indicating whether the writing of the snapshot file was successful ("SUCCESS") or not ("FAILURE").
TYPE	STRING	String value indicating how the snapshot was initiated. Possible values are: • AUTO — an automated snapshot as defined by the configuration • COMMANDLOG — a command log snapshot • MANUAL — a manual snapshot initiated by a user
PREEMPT_PERCENT	BIGINT	The number of snapshot tasks that were requeued due to transaction workload superseding them.

SNAPSHOTSUMMARY — Returns a row for every snapshot performed by the cluster, including up to ten snapshots

Name	Datatype	Description
NONCE	STRING	The unique identifier for the snapshot.
TXNID	BIGINT	The transaction ID of the snapshot.

Name	Datatype	Description
TYPE	STRING	String value indicating how the snapshot was initiated. Possible values are: • AUTO — an automated snapshot as defined by the configuration • COMMANDLOG — a command log snapshot • MANUAL — a manual snapshot initiated by a user
PATH	STRING	The target directory path for the snapshot files.
START_TIME	BIGINT	The timestamp when the snapshot began (in milliseconds).
END_TIME	BIGINT	The timestamp when the snapshot was completed (in milliseconds).
DURATION	BIGINT	The length of time (in seconds) it took to complete the snapshot.
PROGRESS_PCT	FLOAT	For snapshots currently in progress, the percent complete at the time of the call.
RESULT	STRING	String value indicating whether the writing of the snapshot was successful ("SUCCESS") or not ("FAILURE").

TABLE — Returns a row for every table, per partition. In other words, the number of tables, multiplied by the number of sites per host and the number of hosts.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	BIGINT	Numeric ID of the execution site on the host node.
PARTITION_ID	BIGINT	The numeric ID for the logical partition that this site represents. When using a K value greater than zero, there are multiple copies of each logical partition.
TABLE_NAME	STRING	The name of the database table.
TABLE_TYPE	STRING	The type of the table. Values returned include "PersistentTable" for normal data tables and views and "StreamedTable" for streams.
TUPLE_COUNT	BIGINT	The number of rows currently stored for this table in the current partition. For streams, the cumulative total number of rows inserted into the stream.
TUPLE_ALLOCATED_MEMORY	BIGINT	The total size of memory, in kilobytes, allocated for storing inline data associated with this table in this partition. The allocated memory can exceed the currently used memory (TUPLE_DATA_MEMORY). For streams, this field identifies the amount of memory currently in use to queue export data (both in memory and as export overflow) prior to its being passed to the export target.
TUPLE_DATA_MEMORY	BIGINT	The total memory, in kilobytes, used for storing inline data associated with this table in this partition. The total memo-

Name	Datatype	Description
		ry used for storing data for this table is the combination of memory used for inline (tuple) and non-inline (string) data.
STRING_DATA _MEMORY	BIGINT	The total memory, in kilobytes, used for storing non-inline variable length data (VARCHAR, VARBINARY, and GEOGRAPHY) associated with this table in this partition. The total memory used for storing data for this table is the combination of memory used for inline (tuple) and non-inline (string) data.
TUPLE_LIMIT	INTEGER	Unused. Returns zero.
PERCENT_FULL	INTEGER	Unused. Returns zero.
DR	STRING	A text string of "true" or "false" indicating whether the table is a DR table or not.
EXPORT	STRING	If the table or stream is configured with MIGRATE or EXPORT TO TARGET, the name of the associated export target.
TUPLES_PER_BLOCK	BIGINT	The number of tuples that fit into each block of memory.
FRAGMENTED_PERCENT	FLOAT	The percentage of memory for this table and partition that is currently unused due to fragmentation.

TASK — Returns a row for every task, per logical partition.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
PARTITION_ID	BIGINT	The numeric ID for the logical partition running the task procedure. Directed procedures run on each logical partition. Multi-partition procedures run on the multi-partition initiator.
TASK_NAME	STRING	The name of the task.
STATE	STRING	The current status of the task. Possible values include: • RUNNING — The task is enabled and running normally. • DISABLED — The task is disabled and not running. • ERROR — The task returned an error and was stopped due to the ON ERROR STOP attribute. • PAUSED — The database is paused, so the task is not currently running but will restart when the database resumes.
SCOPE	STRING	The execution scope of the task, which matches the RUN ON attribute. Possible values are DATABASE, HOSTS, or PARTITIONS.
SCHEDULER_INVOCATIONS	BIGINT	The total number of invocations of the task's schedule. For example, if the task is scheduled for every 5 minutes, after 20 minutes of normal operation you would expect 4 invocations.

Name	Datatype	Description
SCHEDULER_TOTAL_EXECUTION	BIGINT	The total time, in nanoseconds, consumed by the scheduler for scheduling the task.
SCHEDULER_MIN_EXECUTION	BIGINT	The minimum amount of time, in nanoseconds, taken by the scheduler to schedule an instance of the task.
SCHEDULER_MAX_EXECUTION	BIGINT	The maximum amount of time, in nanoseconds, taken by the scheduler to schedule an instance of the task.
SCHEDULER_AVG_EXECUTION	BIGINT	The average amount of time, in nanoseconds, taken by the scheduler to schedule instances of the task.
SCHEDULER_TOTAL_WAIT_TIME	BIGINT	The total time, in nanoseconds, between the task's scheduled start and when the scheduler was invoked.
SCHEDULER_MIN_WAIT_TIME	BIGINT	The minimum difference, in nanoseconds, between when the task was scheduled to run and when the scheduler was invoked.
SCHEDULER_MAX_WAIT_TIME	BIGINT	The maximum difference, in nanoseconds, between when the task was scheduled to run and when the scheduler was invoked.
SCHEDULER_AVG_WAIT_TIME	BIGINT	The average difference, in nanoseconds, between when the task was scheduled to run and when the scheduler was invoked.
SCHEDULER_STATUS	STRING	For future use.
PROCEDURE_INVOCATIONS	BIGINT	The total number of invocations of the task's procedure.
PROCEDURE_TOTAL_EXECUTION	BIGINT	The total time, in nanoseconds, consumed by the task for all of its invocations.
PROCEDURE_MIN_EXECUTION	BIGINT	The minimum amount of time, in nanoseconds, an instance of the task took to execute.
PROCEDURE_MAX_EXECUTION	BIGINT	The maximum amount of time, in nanoseconds, an instance of the task took to execute.
PROCEDURE_AVG_EXECUTION	BIGINT	The average amount of time, in nanoseconds, instances of the task took to execute.
PROCEDURE_TOTAL_WAIT_TIME	BIGINT	The total time, in nanoseconds, between when the procedure was scheduled to run and when it was invoked.
PROCEDURE_MIN_WAIT_TIME	BIGINT	The minimum difference, in nanoseconds, between when the procedure was scheduled to run and when it was invoked.
PROCEDURE_MAX_WAIT_TIME	BIGINT	The maximum difference, in nanoseconds, between when the procedure was scheduled to run and when it was invoked.
PROCEDURE_AVG_WAIT_TIME	BIGINT	The average difference, in nanoseconds, between when the procedure was scheduled to run and when it was invoked.
PROCEDURE_FAILURES	BIGINT	The number of times the procedure generated an error when run.

TLS — Returns a separate row for each topic and partition.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
CONNECTION_ID	BIGINT	Numeric ID of the client connection.
CONNECTION_HOSTNAME	STRING	The server name of the node from which the client connection originates.
MESSAGES_ENCRYPTED	BIGINT	The total number of messages encrypted for output.
AVERAGE_ENCRYPT_LATENCY	BIGINT	The average time, in nanoseconds, to encrypt the messages.
MESSAGES_DECRYPTED	BIGINT	The total number of messages decrypted on input.
AVERAGE_DECRYPT_LATENCY	BIGINT	The average time, in nanoseconds, to decrypt the messages.

TOPIC — Returns a separate row for each topic and partition.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
HOST_ID	INTEGER	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
SITE_ID	INTEGER	Numeric ID of the execution site on the host node.
TOPIC	STRING	The name of the topic.
PARTITION_ID	INTEGER	The numeric ID for the logical partition.
FIRST_OFFSET	BIGINT	The value of the first offset currently available in the topic.
LAST_OFFSET	BIGINT	The value of the last offset in the topic.
FIRST_OFFSET_TIMESTAMP	TIMESTAMP	The timestamp when the first offset was inserted into the queue.
LAST_OFFSET_TIMESTAMP	TIMESTAMP	The timestamp when the most recent message (the last offset) was inserted into the queue.
BYTES_ON_DISK	BIGINT	The size, in bytes, of data stored on disk for this partition and topic.
BYTES_FETCHED	BIGINT	The size, in bytes, of data sent to consumers for this partition and topic.
STATE	STRING	The current level of completeness for this topic in this partition. If the server was down at any point, it may be missing records that were queued while the partition was offline. Possible values are: • STABLE — the queue is complete. • BACKFILLING — records are missing but are being retrieved from other servers.

Name	Datatype	Description
		• BLOCKED — records are missing from all copies of the partition. • ORPHANED — the queue is no longer being served by this partition, but is saved in case other copies of the queue are blocked or backfilling and need the data. This is a transitional state and the queue is deleted as soon as no other copies need its records. If a topic becomes blocked and the cluster is complete (has no missing nodes), you can use the **voltadmin topic release** command to "jump" past the missing offsets.
MASTER	STRING	A text string of "true" or "false" indicating whether the current site is the master for the logical partition.
RETENTION_POLICY	STRING	The retention policy for this topic.
ROWS_SKIPPED	BIGINT	Reserved for future use.
ERROR_OFFSET	BIGINT	If an error occurs while encoding a message for consumers, an error is returned to the consumer, the offset of the message is recoded here, and a description of the error stored in the next column.
ERROR_MESSAGE	STRING	A description of the last error that occurred while encoding messages for consumers.

TTL — Returns a separate row for each table in the database where TTL processing is currently active. It does not list tables that do not have TTL defined or where TTL processing has been cancelled due to an error or lack of a suitable index.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).
TABLE_NAME	STRING	The name of the table.
ROWS_DELETED	BIGINT	The total number of rows expired and deleted by the TTL attribute.
ROWS_DELETED_LAST_ROUND	BIGINT	The number of rows expired and deleted during the last TTL processing.
ROWS_REMAINING	BIGINT	The number of expired rows *not* deleted during the last TTL processing due to batch size limits. If TTL processing is keeping up with the throughput, this value should tend towards zero.
LAST_DELETE_TIMESTAMP	TIMESTAMP	The timestamp when the last round of TTL processing occurred (in milliseconds).

XDCR — Returns a row for each host in the cluster.

Name	Datatype	Description
TIMESTAMP	BIGINT	The timestamp when the information was collected (in milliseconds).

Name	Datatype	Description
HOST_ID	BIGINT	Numeric ID for the host node.
HOSTNAME	STRING	Server name of the host node.
IS_READY	STRING	A text string of "true" or "false" indicating that consumers and producers for all partitions are active and able to send and receive data from participating XDCR clusters.
DRROLE_STATE	STRING	The current state of the DR relationship. Possible values are the following: • DISABLED — DR is not enabled for the cluster • PENDING — DR is enabled but communication with the other cluster has not begun • ACTIVE — Communication with the other cluster has begun • STOPPED — Communication with the other cluster has stopped due to a failure of some kind Note that if DR stops, issuing the **voltadmin dr reset** command will return the cluster to the PENDING state.
DRPROD_STATE	STRING	The current state of the cluster's DR producers. The return value is "ACTIVE" if the producers are creating binary logs to send to the remote cluster, otherwise the value is null.
DRPROD_ISSYNCED	STRING	Unused.
DRPROD_CNXTS	STRING	The value is "UP" if all the cluster's partitions are covered and connected to consumers on the remote cluster. Otherwise the value is "DOWN".
DRCONS_STATE	STRING	The current state of the cluster's DR consumers. The return value is "RECEIVE" if the partitions are receiving binary logs from the remote cluster, otherwise the value is null.
DRCONS_ISCOVERED	STRING	The value is "UP" if all the cluster's consumers are covered and connected to producers on the remote cluster. Otherwise the value is "DOWN".
DRCONS_ISPAUSED	STRING	A text string of "true" or "false" indicating whether the consumer connections are paused or not.
DYNAMIC_SCHEMA	STRING	A text string of "true" or "false" indicating whether dynamic schema change is supported and enabled for the clusters.

Examples

The following example uses @Statistics to gather information about the distribution of table rows within the cluster:

```
$ sqlcmd
1> exec @Statistics TABLE, 0;
```

The next program example shows a procedure that collects and displays the number of transactions (i.e. stored procedures) during a given interval, by setting the delta flag to a non-zero value. By calling this procedure iteratively (for example, every five minutes), it is possible to identify fluctuations in the database workload over time (as measured by the number of transactions processed).

```
void measureWorkload() {
  VoltTable[] results = null;
  String procName;
  int procCount = 0;
  int sysprocCount = 0;

  try { results = client.callProcedureSync("@Statistics",
                  "INITIATOR",1).getResults(); }
  catch (Exception e) { e.printStackTrace(); }

  for (VoltTable t: results) {
     for (int r=0;r<t.getRowCount();r++) {
         VoltTableRow row = t.fetchRow(r);
         procName = row.getString("PROCEDURE_NAME");
                /* Count system procedures separately */
         if (procName.substring(0,1).compareTo("@") == 0)
           { sysprocCount += row.getLong("INVOCATIONS"); }
         else
           { procCount += row.getLong("INVOCATIONS"); }
     }
  }
  System.out.printf("System procedures: %d\n" +
                    "User-defined procedures: %d\n",+
                    sysprocCount,procCount);
}
```

@StopNode

@StopNode — Stops a VoltDB server process, removing the node from the cluster.

Syntax

```
@StopNode Integer host-ID
```

Description

The @StopNode system procedure lets you stop a specific server in a K-safe cluster. You specify which node to stop using the host ID, which is the unique identifier for the node assigned by VoltDB when the server joins the cluster.

Note that by calling the @StopNode procedure on a node other than the node being stopped, you will receive a return status indicating the success or failure of the call. If you call the procedure on the node that you are requesting to stop, the return status can only indicate that the call was interrupted (by the VoltDB process on the node stopping), not whether it was successfully completed or not.

If you call @StopNode on a node or cluster that is not K-safe — either because it was started with a K-safety value of zero or one or more nodes have failed so any further failure could crash the database — the @StopNode procedure will not be executed. You can only stop nodes on a cluster that will remain viable after the node stops. To stop the entire cluster, please use the **voltadmin shutdown** command.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

The following program example uses **grep**, **sqlcmd**, and the @SystemInformation stored procedure to identify the host ID for a specific node (doodah) of the cluster. The example then uses that host ID (2) to call @StopNode and stop the desired node.

```
$ echo "exec @SystemInformation overview;" | sqlcmd | grep "doodah"
       2 HOSTNAME                doodah
$ sqlcmd
1> exec @StopNode 2;
```

The next example uses the **voltadmin stop** command to perform the same action. Note that **voltadmin stop** performs the translation from a network name to a host ID for you.

```
$ voltadmin stop doodah
```

The following Java code fragment performs the same function programmatically.

```
try {
     results = client.callProcedureSync("@SystemInformation",
```

```
                                            "overview").getResults();
}
catch (Exception e) { e.printStackTrace(); }

VoltTable table = results[0];
table.resetRowPosition();
int targetHostID = -1;

while (table.advanceRow() && targetHostId < 0) {
    if ( (table.getString("KEY") == "HOSTNAME") &&
            (table.getString("VALUE") == targetHostName) ) {
         targetHostId = (int) table.getLong("HOST_ID");
    }
}

try {
      client.callProcedureSync("@StopNode",
                               targetHostId).getResults();
   }
  catch (Exception e) { e.printStackTrace(); }
```

@SwapTables

@SwapTables — Swaps the contents of one table for another

Syntax

```
@SwapTables String[] table-name, String[] table-name
```

Description

The @SwapTables system procedure swaps the contents of one table for another. So, for example, if table A has 2 rows and table B has 10 rows, after executing the following system procedure call table A would have 10 rows and table B would have 2 rows:

```
sqlcmd> exec @SwapTables 'A' 'B';
```

The tables being swapped must have identical schema. That is the names, datatype, and order of the columns must be the same and the tables must have the same indexes and other constraints. Also there cannot be any views on either table. If these requirements are not met, or if either of the named tables does not exist, the system procedure returns an error.

The system procedure provides a significant performance advantage over any comparable SQL statements when swapping large tables because the operation does not actually move any data. The pointers for the two tables are switched, eliminating any need for excessive temporary storage or data movement.

When using database replication (DR), you must pause the database before using the @SwapTables procedure, or the system procedure returns an error. The process for using @SwapTables in an XDCR environment is the following:

1. Pause *all* of the clusters in the XDCR relationshp with **voltadmin pause --wait**.
2. Invoke the same @SwapTables call on all of the databases.
3. Resume all the databases with **voltadmin resume**.

Return Values

Returns one VoltTable with one row and one column.

Name	Datatype	Description
MODIFIED_TUPLES	BIGINT	The number of tuples affected by the swap. In other words, the sum of the tuples in both tables.

Examples

The following example uses the @SwapTables system procedure to replace a lookup table of hot topics with an updated list in a single statement.

```
sqlcmd> exec @SwapTables Hot_Topics Hot_Topics_Update;
```

@SystemCatalog

@SystemCatalog — Returns metadata about the database schema.

Syntax

```
@SystemCatalog String component
```

Description

The @SystemCatalog system procedure returns information about the schema of the VoltDB database, depending upon the component keyword you specify. The following are the allowable values of *component*:

"COLUMNS"	Returns a list of columns for all of the tables in the database.
"FUNCTIONS"	Returns information about user-defined functions in the database.
"INDEXINFO"	Returns information about the indexes in the database schema. Note that the procedure returns information for each column in the index. In other words, if an index is composed of three columns, the result set will include three separate entries for the index, one for each column.
"PRIMARYKEYS"	Returns information about the primary keys in the database schema. Note that the procedure returns information for each column in the primary key. If an primary key is composed of three columns, the result set will include three separate entries.
"PROCEDURECOLUMNS"	Returns information about the arguments to the stored procedures.
"PROCEDURES"	Returns information about the stored procedures defined in the database.
"ROLES"	Returns information about the roles defined in the database and their associated permissions.
"TABLES"	Returns information about the tables in the database.
"TASKS"	Returns information about the tasks defined in the database.
"USERS"	Returns information about the users defined in the database and their associated roles.

Return Values

Returns a different VoltTable for each component. The layout of the VoltTables is designed to match the corresponding JDBC data structures. Columns are provided for all JDBC properties, but where VoltDB has no corresponding element the column is unused and a null value is returned.

For the COLUMNS component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.

Name	Datatype	Description
TABLE_NAME	STRING	The name of the database table the column belongs to.
COLUMN_NAME	STRING	The name of the column.
DATA_TYPE	INTEGER	An enumerated value specifying the corresponding Java SQL datatype of the column.
TYPE_NAME	STRING	A string value specifying the datatype of the column.
COLUMN_SIZE	INTEGER	The length of the column in bits, characters, or digits, depending on the datatype.
BUFFER_LENGTH	INTEGER	Unused.
DECIMAL_DIGITS	INTEGER	The number of fractional digits in a DECIMAL datatype column. (Null for all other datatypes.)
NUM_PREC_RADIX	INTEGER	Specifies the radix, or numeric base, for calculating the column size. A radix of 2 indicates the column size is measured in bits while a radix of 10 indicates a measurement in bytes or digits.
NULLABLE	INTEGER	Indicates whether the column value can be null (1) or not (0).
REMARKS	STRING	Contains the string "PARTITION_COLUMN" if the column is the partitioning key for a partitioned table. Otherwise null.
COLUMN_DEF	STRING	The default value for the column.
SQL_DATA_TYPE	INTEGER	Unused.
SQL_DATETIME_SUB	INTEGER	Unused.
CHAR_OCTET_LENGTH	INTEGER	For variable length columns (VARCHAR and VARBINARY), the maximum length of the column. Null for all other datatypes.
ORDINAL_POSITION	INTEGER	An index specifying the position of the column in the list of columns for the table, starting at 1.
IS_NULLABLE	STRING	Specifies whether the column can contain a null value ("YES") or not ("NO").
SCOPE_CATALOG	STRING	Unused.
SCOPE_SCHEMA	STRING	Unused.
SCOPE_TABLE	STRING	Unused.
SOURCE_DATE_TYPE	SMALLINT	Unused.
IS_AUTOINCREMENT	STRING	Specifies whether the column is auto-incrementing or not. (Always returns "NO").

For the FUNCTIONS component, the VoltTable has the following columns:

Name	Datatype	Description
FUNCTION_TYPE	STRING	The function type is always "scalar".
FUNCTION_NAME	STRING	The name of the user-defined function.
CLASS_NAME	STRING	The Java class name that contains the user-defined function method.

Name	Datatype	Description
METHOD_NAME	STRING	The name of the method that implements the user-defined function.

For the INDEXINFO component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table the index applies to.
NON_UNIQUE	TINYINT	Value specifying whether the index is unique (0) or not (1).
INDEX_QUALIFIER	STRING	Unused.
INDEX_NAME	STRING	The name of the index that includes the current column.
TYPE	SMALLINT	A value indicating the type of index, which is always three (3).
ORDINAL_POSITION	SMALLINT	An index specifying the position of the column in the index, starting at 1.
COLUMN_NAME	STRING	The name of the column.
ASC_OR_DESC	STRING	A string value specifying the sort order of the index. Possible values are "A" for ascending or null for unsorted indexes.
CARDINALITY	INTEGER	Unused.
PAGES	INTEGER	Unused.
FILTER_CONDITION	STRING	Unused.

For the PRIMARYKEYS component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table.
COLUMN_NAME	STRING	The name of the column in the primary key.
KEY_SEQ	SMALLINT	An index specifying the position of the column in the primary key, starting at 1.
PK_NAME	STRING	The name of the primary key.

For the PROCEDURECOLUMNS component, the VoltTable has the following columns:

Name	Datatype	Description
PROCEDURE_CAT	STRING	Unused.
PROCEDURE_SCHEM	STRING	Unused.
PROCEDURE_NAME	STRING	The name of the stored procedure.
COLUMN_NAME	STRING	The name of the procedure parameter.
COLUMN_TYPE	SMALLINT	An enumerated value specifying the parameter type. Always returns 1, corresponding to procedureColumnIn.

Name	Datatype	Description
DATA_TYPE	INTEGER	An enumerated value specifying the corresponding Java SQL datatype of the column.
TYPE_NAME	STRING	A string value specifying the datatype of the parameter.
PRECISION	INTEGER	The length of the parameter in bits, characters, or digits, depending on the datatype.
LENGTH	INTEGER	The length of the parameter in bytes. For variable length datatypes (VARCHAR and VARBINARY), this value specifies the maximum possible length.
SCALE	SMALLINT	The number of fractional digits in a DECIMAL datatype parameter. (Null for all other datatypes.)
RADIX	SMALLINT	Specifies the radix, or numeric base, for calculating the precision. A radix of 2 indicates the precision is measured in bits while a radix of 10 indicates a measurement in bytes or digits.
NULLABLE	SMALLINT	Unused.
REMARKS	STRING	If this column contains the string "PARTITION_PARAMETER", the parameter is the partitioning key for a single-partitioned procedure. If the column contains the string "ARRAY_PARAMETER" the parameter is a native Java array. Otherwise this column is null.
COLUMN_DEF	STRING	Unused.
SQL_DATA_TYPE	INTEGER	Unused.
SQL_DATETIME_SUB	INTEGER	Unused.
CHAR_OCTET_LENGTH	INTEGER	For variable length columns (VARCHAR and VARBINARY), the maximum length of the column. Null for all other datatypes.
ORDINAL_POSITION	INTEGER	An index specifying the position in the parameter list for the procedure, starting at 1.
IS_NULLABLE	STRING	Unused.
SPECIFIC_NAME	STRING	Same as COLUMN_NAME

For the PROCEDURES component, the VoltTable has the following columns:

Name	Datatype	Description
PROCEDURE_CAT	STRING	Unused.
PROCEDURE_SCHEM	STRING	Unused.
PROCEDURE_NAME	STRING	The name of the stored procedure.
RESERVED1	STRING	Unused.
RESERVED2	STRING	Unused.
RESERVED3	STRING	Unused.
REMARKS	STRING	Unused.
PROCEDURE_TYPE	SMALLINT	An enumerated value that specifies the type of procedure. Always returns zero (0), indicating "unknown".
SPECIFIC_NAME	STRING	Same as PROCEDURE_NAME.

For the ROLES component, the VoltTable has the following columns:

Name	Datatype	Description
ROLE	STRING	The name of the role.
PERMISSIONS	STRING	A comma-separated list of permissions associated with the role.

For the TABLES component, the VoltTable has the following columns:

Name	Datatype	Description
TABLE_CAT	STRING	Unused.
TABLE_SCHEM	STRING	Unused.
TABLE_NAME	STRING	The name of the database table.
TABLE_TYPE	STRING	Specifies whether the table is a data table ("TABLE"), a materialized view ("VIEW"), or a stream ('EXPORT").
REMARKS	STRING	Unused.
TYPE_CAT	STRING	Unused.
TYPE_SCHEM	STRING	Unused.
TYPE_NAME	STRING	Unused.
SELF_REFERENCING _COL_NAME	STRING	Unused.
REF_GENERATION	STRING	Unused.

For the TASKS component, the VoltTable has the following columns:

Name	Datatype	Description
TASK_NAME	STRING	The name of the task.
SCHEDULER_CLASS	STRING	The class name of the scheduler.
SCHEDULER_PARAMETERS	STRING	The parameters to the scheduler.
ACTIONS_CLASS	STRING	The class name of the task itself.
ACTIONS_PARAMETERS	STRING	The parameters to the task procedure.
SCHEDULE_CLASS	STRING	The class name of the schedule procedure.
SCHEDULE_PARAMETERS	STRING	The parameters to the schedule procedure
ON_ERROR	STRING	Specifies the ON ERROR condition ("LOG", "IGNORE", or 'STOP").
RUN_LOCATION	STRING	Specifies where the task runs ("DATABASE", "HOSTS", or "PARTITIONS").
USER	STRING	Identifies the username of the account under which the procedures runs.
ENABLED	STRING	Specifies whether the task is currently enabled or disabled.

For the USERS component, the VoltTable has the following columns:

Name	Datatype	Description
USER	STRING	The name of the user.
ROLES	STRING	A comma-separated list of roles assigned to the user.

Examples

The following example calls @SystemCatalog to list the stored procedures in the active database schema:

```
$ sqlcmd
1> exec @SystemCatalog procedures;
```

The next program example uses @SystemCatalog to display information about the tables in the database schema.

```
VoltTable[] results = null;
try {
     results = client.callProcedureSync("@SystemCatalog",
               "TABLES").getResults();
     System.out.println("Information about the database schema:");
     for (VoltTable node : results) System.out.println(node.toString());
}
catch (Exception e) {
     e.printStackTrace();
}
```

@SystemInformation

@SystemInformation — Returns configuration information about VoltDB and the individual nodes of the database cluster.

Syntax

```
@SystemInformation

@SystemInformation String component
```

Description

The @SystemInformation system procedure returns information about the configuration of the VoltDB database or the individual nodes of the database cluster, depending upon the component keyword you specify. The following are the allowable values of *component*:

"DEPLOYMENT"	Returns information about the configuration of the database. In particular, this keyword returns information about the various features and settings, such as export, snapshots, K-safety, and so on. These properties are returned in a single VoltTable of name/value pairs.
"ENV"	Returns information about the environment variables defined for each host, returning a row containing the variable name and value per host.
"LICENSE"	Returns information about the license in use by the database. The license properties are returned in a single VoltTable of name/value pairs.
"OVERVIEW"	Returns information about the individual servers in the database cluster, including the host name, the IP address, the version of VoltDB running on the server, as well as the path to the configuration file in use. The overview also includes entries for the start time of the server and length of time the server has been running.

If you do not specify a component, @SystemInformation returns the results of the OVERVIEW component (to provide compatibility with previous versions of the procedure).

Return Values

Returns different VoltTables depending on which component is requested.

DEPLOYMENT — returns one row for each configuration property.

Name	Datatype	Description
PROPERTY	STRING	The name of the configuration property. The system procedure reports the following properties, depending on what features are enabled in the database configuration: • adminport — admin port number • commandlogenabled — whether command logging is enabled or not • commandlogfreqtime — frequency of command logging in milliseconds • commandlogfreqtxns — frequency of command logging in number of transactions

Name	Datatype	Description
		• commandlogmode — command logging mode, sync or async • commandlogpath — directory path to command log segments • commandlogsnapshotpath — directory path to command log snapshots • droverflowpath — directory path for DR overflow • drpriority — priority of DR transactions, when priorities are enabled • drrole — DR role (xdcr or none) • drschemachangeenabled — whether dynamic schema change is enabled • elasticduration — target duration of rebalance transactions in milliseconds • elasticthroughput — target throughput of rebalance transactions in MB/second • export — whether export is enabled or not • exportcursorpath — directory path for storing the current cursor location for export queues • exportoverflowpath — directory path to export overflow • heartbeattimeout — heartbeat timeout setting in seconds • hostcount — full number of servers in the cluster • kfactor — K-safety value • largequeryswappath — directory path used as temporary storage for processing large queries • metricsenabled — whether the metrics port is open and responding • partitiondetection — whether network partition detection is enabled or not • prioritiesenabled — whether queue prioritization is enabled (true or false) • querytimeout — default query timeout in seconds • sitesperhost — number of sites per host • snapshotenabled — whether automatic snapshots are enabled or not • snapshotfrequency — frequency of automatic snapshots • snapshotpath — directory path to automatic snapshot files • snapshotprefix — unique file prefix for automatic snapshots • snapshotpriority — system priority of automatic snapshots • snapshotretain — number of automatic snapshots to retain • temptablesmaxsize — maximum size of the temp tables • users — list of user names and their roles • voltdbroot — path of database root directory
VALUE	STRING	The corresponding value of that property in the configuration (either explicitly or by default).

ENV — returns a row for every environment variable defined on every host in the cluster.

Name	Datatype	Description
HOST_ID	INTEGER	A numeric identifier for the host node.
ENV_NAME	STRING	The name of the environment variable.
ENV_VALUE	STRING	The value of the variable on the specified host.

LICENSE — returns one row for each property of the currently active license. (Not all properties apply to all licenses.)

Name	Datatype	Description
PROPERTY	STRING	The name of the license property. The system procedure reports the following properties, depending on what license features are enabled: • PERMIT_VERSION — for internal use • PERMIT_SCHEME — for internal use • TYPE — the type of license in use • ISSUER_COMPANY — the company that issued the license • ISSUER_EMAIL — the email address of the company that issued the license • ISSUER_URL — the website of the company that issued the license • ISSUER_PHONE — the telephone number of the company that issued the license • ISSUE_DATE — the date the license was issued • LICENSEE — who the license was issued to • EXPIRATION — the date the license expires • HOSTCOUNT_MAX — the maximum number of hosts supported by the license • CORECOUNT_MAX — the maximum number of processor cores supported by the license • FEATURES_* — there may be multiple "feature" entries, each identifying a separately licensable function enabled by the license • NOTE — notes regarding the license • SIGNATURE — an encrypted signature identifying the license (for internal use)
VALUE	STRING	The corresponding value of the property (either set explicitly by the license or by default).

OVERVIEW — returns a row for every system attribute on every node of the cluster. The rows contain an additional column to identify the host node associated with the attribute.

Name	Datatype	Description
HOST_ID	INTEGER	A numeric identifier for the host node.
KEY	STRING	The name of the system attribute.
VALUE	STRING	The corresponding value of that attribute for the specified host. The system procedure reports the following properties: • ADMININTERFACE — admin network interface • ADMINPORT — admin port number • BUILDSTRING — VoltDB software build string including version number • CATALOG — for internal use • CATALOGCRC — for internal use • CLIENTINTERFACE — client network interface • CLIENTPORT — client port number • CLUSTERID — DR cluster ID • CLUSTERSAFETY — whether the database is running in normal K-safe mode (FULL), or if it has reduced k-safety due to non-deterministic results from a transaction (REDUCED) • CLUSTERSTATE — whether the database is running normally or paused (that is, in admin mode)

Name	Datatype	Description
		• CPUCORES — number of CPU cores • CPUTHREADS — number of processor threads • CREATIONTIME — timestamp when the database was started for the first time • DEPLOYMENT — path to current configuration in root directory • DRINTERFACE — database replication (DR) network interface • DRPORT — database replication (DR) port number • DRPUBLICINTERFACE — network interface advertised to member clusters as DR interface • DRPUBLICPORT — port advertised to member clusters as DR port • FULLCLUSTERSIZE — initial cluster size • HOSTNAME — server host name • HOSTSTARTTIME — timestamp when the current host process started • HTTPINTERFACE — httpd network interface • HTTPPORT — httpd port number • INITIALIZED — for internal use • INTERNALINTERFACE — internal network interface • INTERNALPORT — internal port number • IPADDRESS — server IP address • IV2ENABLED — for internal use • KUBERNETES — whether the database is running under Kubernetes (true or false) • JAVAAVAILABLEPROCESSORS — the number of processors available to Java, which may be lower than the number of CPU cores • LAST_UPDATECORE_DURATION — length of last schema change transaction • LICENSE — product license information • LOG4JPORT — log4J port number • MEMORY — total size of memory of current server • METRICSINTERFACE —metrics network interface • METRICSPORT — metrics port number • PARTITIONGROUP — for internal use • PLACEMENTGROUP — name of the placement group the server belongs to • PUBLICINTERFACE — network interface advertised by Volt Management Center as the external interface • REPLICATIONROLE — the DR role of the cluster (xdcr or none) • STARTTIME — timestamp when cluster started • TOPICSPUBLICINTERFACE — network interface advertised to clients as the topic broker interface • TOPICSPUBLICPORT — network port advertised to clients as the topic broker port • TOPICSPORT — Topic broker port number • UPTIME — how long the cluster has been running • VERSION — VoltDB software version number • VOLTDBROOT — path of database root directory • ZKINTERFACE — ZooKeeper network interface • ZKPORT — ZooKeeper port number

Examples

The first example displays information about the individual servers in the database cluster:

```
$ sqlcmd
1> exec @SystemInformation overview;
```

The following program example uses @SystemInformation to display information about the nodes in the cluster and then about the database itself.

```
VoltTable[] results = null;
try {
     results = client.callProcedureSync("@SystemInformation",
               "OVERVIEW").getResults();
     System.out.println("Information about the database cluster:");
     for (VoltTable node : results) System.out.println(node.toString());

     results = client.callProcedureSync("@SystemInformation",
               "DEPLOYMENT").getResults();
     System.out.println("Information about the database configuration:");
     for (VoltTable node : results) System.out.println(node.toString());

}
catch (Exception e) {
     e.printStackTrace();
}
```

@UpdateClasses

@UpdateClasses — Adds and removes Java classes from the database.

Syntax

@UpdateClasses byte[] *JAR-file*, String *class-selector*

Description

The @UpdateClasses system procedure performs two functions:

- Loads into the database any Java classes and resources in the JAR file passed as the first parameter
- Removes any classes matching the class selector string passed as the second parameter

You need to compile and pack your stored procedure classes into a JAR file and load them into the database using @UpdateClasses before entering the CREATE PROCEDURE statements that define those classes as VoltDB stored procedures. Note that, for interactive use, the sqlcmd utility has two directives, **load classes** and **remove classes**, that perform these actions in separate steps.

To remove classes, you specify the class names in the second parameter, the class selector. You can include multiple class selectors using a comma-separated list. You can also use Ant-style wildcards in the class specification to identify multiple classes. For example, the following command deletes all classes that are children of org.mycompany.utils as well as *.DebugHandler:

```
sqlcmd
1> exec @UpdateClasses NULL "org.mycompany.utils.*,*.DebugHandler";
```

You can also use the @UpdateClasses system procedure to include reusable code that is accessed by multiple stored procedures. Reusable code includes both resource files, such as XML or other data files, as well as classes accessed by stored procedures.

- Resource files must be stored in a subfolder within the JAR. Resources in the root directory are ignored.
- Any classes and methods called by stored procedures must follow the same rules for deterministic behavior that stored procedures follow, as described in Section 5.1.2, “VoltDB Stored Procedures are Deterministic”.
- Use of @UpdateClasses is not recommended for large, established libraries of classes used by stored procedures. For larger, static libraries that do not need to be modified on the fly, the preferred approach is to include the code by placing JAR files in the /lib directory where VoltDB is installed on the database servers.

Classes can be overwritten by loading a new class with the same path. Similarly, resource files can be updated by reloading a file with the same name and location. Classes can be removed using the second argument to the system procedure (or the **remove classes** directive). However, there is no mechanism for removing resources files other than classes once they are loaded.

Examples

The following example compiles and packs Java stored procedures into the file myapp.jar. The example then uses @UpdateClasses to load the classes from the JAR file, then defines and partitions a stored procedure based on the uploaded classes.

```
$ javac -cp "/opt/voltdb/voltdb/*" -d obj src/myapp/*.java
$ jar cvf myapp.jar -C obj .
$ sqlcmd
1> exec @UpdateClasses myapp.jar "";
2> CREATE PROCEDURE
3>    PARTITION ON TABLE Customer COLUMN CustomerID
4>    FROM CLASS myapp.procedures.AddCustomer;
```

The second example removes the class added and defined in the preceding example. Note that you must drop the procedure definition first; you cannot delete classes that are referenced by defined stored procedures.

```
$ sqlcmd
1> DROP PROCEDURE AddCustomer;
2> exec @UpdateClasses NULL "myapp.procedures.AddCustomer";
```

As an alternative, the loading and removing of classes can be performed using native sqlcmd directives **load classes** and **remove classes**. So the previous tasks can be performed using the following commands:

```
$ sqlcmd
1> load classes myapp.jar "";
2> CREATE PROCEDURE
3>    PARTITION ON TABLE Customer COLUMN CustomerID
4>    FROM CLASS myapp.procedures.AddCustomer;

      [ . . . ]

1> DROP PROCEDURE AddCustomer;
2> remove classes "myapp.procedures.AddCustomer";
```

@UpdateLogging

@UpdateLogging — Changes the logging configuration for a running database.

Syntax

```
@UpdateLogging String configuration
```

Description

The @UpdateLogging system procedure lets you change the logging configuration for VoltDB. The argument, *configuration*, is a text string containing the Log4J 2 configuration definition. Calling @UpdateLogging updates the logging configuration for the entire cluster and can include multiple modifications. If you just want to change the logging level of a single component, you can use the @AdjustLogging system procedure or its corresponding **voltadmin log4j** command.

Return Values

Returns one VoltTable with one row.

Name	Datatype	Description
STATUS	BIGINT	Always returns the value zero (0) indicating success.

Examples

From the command line, the recommended way to update the Log4J configuration is to use the **voltadmin log4j** command, specifying the location of an updated Log4J 2 properties file. For example:

```
$ voltadmin log4j mylog4j.properties
```

The following program example demonstrates another way to update the logging, using the system procedure and passing the contents of an properties file as the argument:

```
try {
    Scanner scan = new Scanner(new File(propertyfilename));
    scan.useDelimiter("\\Z");
    String content = scan.next();
    client.callProcedureSync("@UpdateLogging",content);
}
catch (Exception e) {
    e.printStackTrace();
}
```