Verus: Real-Time Commerce Analytics at Scale

Zach Kelling

Hanzo AI

November 2015

Abstract

We present Verus, a real-time analytics system designed for e-commerce workloads. Traditional analytics architectures batch-process events with multi-hour latency, limiting merchants' ability to respond to emerging trends, detect anomalies, and optimize campaigns in flight. Verus processes commerce events with sub-second latency while maintaining the analytical flexibility of batch systems. The architecture combines stream processing for real-time aggregation with columnar storage for ad-hoc queries, unified through a SQL interface that abstracts the underlying complexity. We describe the system design, evaluate performance on production workloads, and demonstrate applications including live dashboards, cohort analysis, and anomaly detection. Production deployments process over 50,000 events per second with query latency under 200ms for common dashboard queries.

1 Introduction

E-commerce generates events at high velocity: page views, product interactions, cart modifications, and transactions flow continuously. Extracting actionable insights from this stream requires analytics systems that balance three competing requirements:

1. **Freshness**: Insights must reflect recent events. Stale data delays response to problems (site errors, fraud spikes) and opportunities (trending products, effective campaigns).

2. **Flexibility**: Merchants need ad-hoc queries, not just pre-defined reports. Investigating anomalies requires drilling into arbitrary dimensions.

3. **Scale**: Event volumes grow with business success. Systems must scale horizontally without architectural changes.

Traditional approaches force trade-offs. Data warehouses (Redshift, BigQuery) provide flexibility but with batch latency. Stream processors (Storm, Flink) offer freshness but limited query expressiveness. Lambda architectures [1] combine both but double operational complexity.

Verus achieves all three requirements through a unified architecture. Events stream into a real-time aggregation layer that maintains precomputed rollups for common queries. Simultaneously, events land in columnar storage for ad-hoc analysis. A query planner routes requests to the optimal layer based on query characteristics.

Our contributions are:

1. A hybrid architecture combining stream processing and columnar storage with automatic query routing.

2. Materialized view maintenance for sub-second dashboard queries.

3. Cohort analysis operators for retention and lifetime value computation.

4. Anomaly detection integrated into the analytics pipeline.

2 Background

2.1 Commerce Analytics Requirements

E-commerce analytics spans multiple use cases:

Operational dashboards. Real-time visibility into key metrics: orders per minute, revenue, conversion rate, error rate. Latency requirement: seconds.

Marketing attribution. Understanding which channels and campaigns drive conversions. Requires joining touchpoints across sessions. Latency requirement: minutes to hours.

Cohort analysis. Comparing behavior across customer segments defined by acquisition date, first purchase category, or other attributes. Latency requirement: hours.

Ad-hoc investigation. Drilling into anomalies or exploring hypotheses. Query patterns unpredictable. Latency requirement: seconds to minutes.

2.2 Event Model

Commerce events follow a common structure:

Listing 1: Event schema

```
@dataclass
class CommerceEvent:
    event_id: str           # UUID
    event_type: str         #
       page_view, add_to_cart,
       purchase, etc.
    timestamp: datetime     # Event
       time
    user_id: Optional[str] #
       Authenticated user
    session_id: str         # Session
        identifier

    # Dimensions
    store_id: str
    device_type: str        # desktop
       , mobile, tablet
    country: str
    referrer: str
    utm_source: Optional[str]
    utm_medium: Optional[str]
    utm_campaign: Optional[str]

    # Measures (event-type specific)
    product_id: Optional[str]
    product_category: Optional[str]
    quantity: Optional[int]
    revenue: Optional[Decimal]
```

Event volumes vary widely. A small merchant might generate 1,000 events per day; a large retailer during a flash sale might generate 100,000 events per second.

2.3 Existing Approaches

Batch data warehouses. Systems like Amazon Redshift, Google BigQuery, and Snowflake excel at complex analytical queries over historical data. They process data in batches, typically with hour-scale latency. Query flexibility is excellent; freshness is poor.

Stream processors. Apache Storm, Flink, and Kafka Streams process events in real-time. They excel at continuous aggregations but struggle with ad-hoc queries requiring historical context.

OLAP cubes. Pre-aggregated cubes (Druid, Pinot, ClickHouse) offer fast queries over predefined dimensions. They trade flexibility for speed—queries outside the cube's schema require falling back to raw data.

Lambda architecture. Combining batch and stream layers provides both freshness and flexibility but doubles infrastructure complexity. Maintaining consistency between layers is notoriously difficult.

3 System Design

3.1 Architecture Overview

Verus comprises three layers:

1. **Ingestion layer**: Receives events, validates, and routes to processing.
2. **Processing layer**: Maintains real-time aggregations and writes to storage.
3. **Query layer**: Routes queries to optimal backend and merges results.

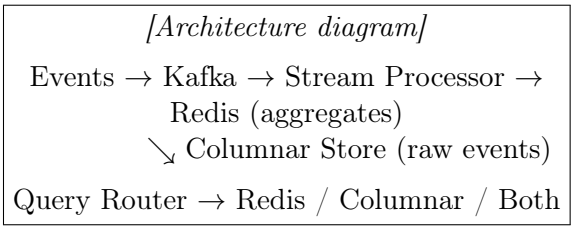

Figure 1: Verus architecture overview

3.2 Ingestion Layer

Events arrive via HTTP endpoints or Kafka consumers:

Listing 2: Event ingestion

```
class EventIngester:
    def __init__(self,
       kafka_producer, validator):
        self.producer =
           kafka_producer
        self.validator = validator
```

```
    async def ingest(self, event:
        CommerceEvent) -> None:
        # Validate schema and
            business rules
        self.validator.validate(
            event)

        # Enrich with server-side
            data
        event.server_timestamp =
            datetime.utcnow()
        event.geo = geoip.lookup(
            event.ip_address)

        # Partition by store for
            ordering guarantees
        partition_key = event.
            store_id

        await self.producer.send(
            topic="commerce-events",
            key=partition_key,
            value=event.to_json()
        )
```

Events partition by store ID, ensuring ordering within a store while enabling parallel processing across stores.

3.3 Stream Processing Layer

The stream processor maintains real-time aggregations:

3.3.1 Aggregation Windows

We maintain aggregations at multiple time granularities:

- **1-minute windows**: Operational dashboards.
- **1-hour windows**: Intraday trends.
- **1-day windows**: Daily summaries.

Windows use event time with allowed lateness of 5 minutes. Late events update existing windows; very late events are logged but excluded.

3.3.2 Materialized Views

Common query patterns are pre-computed as materialized views:

Listing 3: Materialized view definition

```
CREATE MATERIALIZED VIEW
   revenue_by_source AS
SELECT
    store_id,
    TUMBLE(event_time, INTERVAL '1'
       MINUTE) AS window,
    utm_source,
    COUNT(*) AS order_count,
    SUM(revenue) AS total_revenue,
    COUNT(DISTINCT user_id) AS
       unique_customers
FROM events
WHERE event_type = 'purchase'
GROUP BY store_id, window,
   utm_source;
```

The stream processor maintains these views incrementally:

Listing 4: Incremental aggregation

```
class MaterializedViewProcessor:
    def __init__(self, view_def,
        state_store):
        self.view_def = view_def
        self.state = state_store

    def process(self, event):
        if not self.view_def.filter(
            event):
            return

        # Compute aggregation key
        key = self.view_def.
            group_key(event)

        # Update state
        current = self.state.get(key
            ) or self.view_def.
            initial_state()
        updated = self.view_def.
            aggregate(current, event)
        self.state.put(key, updated)

        # Emit to downstream
        self.emit(key, updated)
```

3.3.3 State Management

Aggregation state resides in Redis for fast access:

Listing 5: Redis state store

```
class RedisStateStore:
    def __init__(self, redis_client,
        ttl_seconds):
        self.redis = redis_client
        self.ttl = ttl_seconds

    def get(self, key: str) ->
       Optional[Dict]:
        data = self.redis.get(key)
        return json.loads(data) if
           data else None

    def put(self, key: str, value:
       Dict) -> None:
        self.redis.setex(key, self.
           ttl, json.dumps(value))

    def increment(self, key: str,
       field: str, delta: float) ->
       float:
        return self.redis.
           hincrbyfloat(key, field,
           delta)
```

State keys encode view name, dimensions, and time window:

```
verus:revenue_by_source:store_123:google:2015-06-15T14:30
```

3.4 Columnar Storage Layer

Raw events persist to columnar storage for ad-hoc queries:

3.4.1 Storage Format

We use Apache Parquet with the following optimizations:

- **Partitioning**: By store ID and date for partition pruning.
- **Sorting**: By timestamp within partitions for range queries.
- **Compression**: Snappy for balance of speed and size.
- **Statistics**: Min/max per column for predicate pushdown.

Listing 6: Parquet writer

```
class ParquetWriter:
    def __init__(self, base_path,
       partition_cols):
        self.base_path = base_path
        self.partition_cols =
           partition_cols
        self.buffers = {}  #
           partition -> event buffer

    def write(self, event):
        partition = self.
           compute_partition(event)
        self.buffers.setdefault(
           partition, []).append(
           event)

        if len(self.buffers[
           partition]) >= BATCH_SIZE
           :
            self.flush(partition)

    def flush(self, partition):
        events = self.buffers.pop(
           partition)
        path = f"{self.base_path}/{
           partition}/data_{uuid4()
           }.parquet"
        table = pa.Table.from_pylist
           (events, schema=
           EVENT_SCHEMA)
        pq.write_table(table, path,
           compression='snappy')
```

3.4.2 Query Engine

Ad-hoc queries execute on columnar storage via a distributed query engine (we use Presto):

Listing 7: Ad-hoc query example

```
SELECT
    product_category,
    COUNT(*) AS views,
    COUNT(DISTINCT session_id) AS
       sessions,
    SUM(CASE WHEN event_type = '
       purchase' THEN 1 ELSE 0 END)
       AS purchases
```

```
FROM events
WHERE store_id = 'store_123'
  AND event_date = '2015-06-15'
  AND device_type = 'mobile'
GROUP BY product_category
ORDER BY views DESC
LIMIT 20;
```

3.5 Query Routing Layer

The query router examines incoming queries and routes to the optimal backend:

Listing 8: Query routing

```
class QueryRouter:
    def __init__(self, redis_client,
        presto_client, view_catalog)
       :
        self.redis = redis_client
        self.presto = presto_client
        self.views = view_catalog

    def execute(self, query: str) ->
        ResultSet:
        parsed = parse_sql(query)

        # Check if query matches a
           materialized view
        view_match = self.views.
           match(parsed)

        if view_match and self.
           is_recent_enough(parsed,
           view_match):
            # Serve from Redis
            return self.query_redis(
               view_match, parsed)
        elif view_match:
            # Merge Redis (recent) +
                Presto (historical)
            return self.
               merge_results(
                self.query_redis(
                   view_match,
                   parsed),
                self.query_presto(
                   parsed)
            )
        else:
            # Full scan on Presto
            return self.query_presto
               (parsed)

    def is_recent_enough(self,
       parsed, view):
        # Check if query time range
           fits in Redis TTL
        query_start = parsed.
           get_time_range()[0]
        redis_start = datetime.
           utcnow() - timedelta(
           seconds=view.ttl)
        return query_start >=
           redis_start
```

4 Analytical Capabilities

4.1 Live Dashboards

Dashboards query materialized views for sub-second response:

Listing 9: Dashboard query

```
-- Orders and revenue, last hour, by
    minute
SELECT
    window_start,
    SUM(order_count) AS orders,
    SUM(total_revenue) AS revenue
FROM mv_revenue_by_minute
WHERE store_id = 'store_123'
  AND window_start >= NOW() -
     INTERVAL '1' HOUR
GROUP BY window_start
ORDER BY window_start;
```

Query latency: 15–50ms (served entirely from Redis).

4.2 Cohort Analysis

Cohort analysis groups users by acquisition characteristics and tracks behavior over time:

Listing 10: Cohort retention query

```
WITH cohorts AS (
    SELECT
        user_id,
        DATE_TRUNC('week', MIN(
           event_time)) AS
           cohort_week
    FROM events
    WHERE event_type = 'purchase'
      AND store_id = 'store_123'
    GROUP BY user_id
```

```
),
activity AS (
    SELECT
        e.user_id,
        c.cohort_week,
        DATE_TRUNC('week', e.
            event_time) AS
            activity_week
    FROM events e
    JOIN cohorts c ON e.user_id = c.
        user_id
    WHERE e.event_type = 'purchase'
      AND e.store_id = 'store_123'
)
SELECT
    cohort_week,
    DATE_DIFF('week', cohort_week,
        activity_week) AS
        weeks_since_cohort,
    COUNT(DISTINCT user_id) AS
        active_users
FROM activity
GROUP BY cohort_week,
    weeks_since_cohort
ORDER BY cohort_week,
    weeks_since_cohort;
```

We provide cohort analysis as a built-in function:

Listing 11: Cohort analysis function

```
SELECT * FROM COHORT_ANALYSIS(
    events,
    cohort_date := DATE_TRUNC('week'
        , first_purchase_date),
    activity_date := DATE_TRUNC('
        week', event_time),
    user_id := user_id,
    metric := COUNT(DISTINCT
        order_id),
    filters := [store_id = '
        store_123', event_type = '
        purchase']
);
```

4.3 Funnel Analysis

Funnel analysis tracks conversion through defined steps:

Listing 12: Conversion funnel

```
SELECT * FROM FUNNEL(
    events,
    steps := [
        event_type = 'page_view' AND
            page = '/products',
        event_type = 'product_view',
        event_type = 'add_to_cart',
        event_type = 'checkout_start
            ',
        event_type = 'purchase'
    ],
    window := INTERVAL '30' MINUTE,
    group_by := [device_type,
        utm_source]
);
```

The funnel operator uses session-based windowing to track users through steps:

Listing 13: Funnel computation

```
def compute_funnel(events, steps,
    window, group_by):
    results = defaultdict(lambda:
        [0] * len(steps))

    for session in group_by_session(
        events):
        # Track furthest step
            reached per session
        step_times = [None] * len(
            steps)

        for event in sorted(session.
            events, key=lambda e: e.
            timestamp):
            for i, step_predicate in
                enumerate(steps):
                if step_predicate(
                    event):
                    if i == 0 or (
                        step_times[i
                        -1] and
                        event.
                            timestamp
                            -
                            step_times
                            [i-1] <=
                            window):
                        step_times[i
                            ] = event
                            .
                            timestamp

        # Increment counts for
            reached steps
        group_key = tuple(getattr(
```

```
            session, g) for g in
            group_by)
        for i, t in enumerate(
            step_times):
            if t is not None:
                results[group_key][i
                    ] += 1

    return results
```

4.4 Anomaly Detection

Verus integrates anomaly detection into the analytics pipeline:

4.4.1 Statistical Anomalies

We detect anomalies using seasonal decomposition:

$$y_t = T_t + S_t + R_t \tag{1}$$

where T_t is trend, S_t is seasonal component, and R_t is residual. Anomalies are residuals exceeding threshold:

$$|R_t| > k \cdot \sigma_R \tag{2}$$

Listing 14: Anomaly detection

```
class AnomalyDetector:
    def __init__(self, sensitivity
        =3.0):
        self.k = sensitivity

    def detect(self, series: pd.
        Series) -> List[Anomaly]:
        # Seasonal decomposition
        decomposition =
            seasonal_decompose(series
            , period=24*7)  # Weekly
        residual = decomposition.
            resid.dropna()

        # Detect anomalies
        threshold = self.k *
            residual.std()
        anomalies = []

        for timestamp, value in
            residual.items():
            if abs(value) >
                threshold:
                anomalies.append(
                    Anomaly(
                        timestamp=
                            timestamp,
                        expected=series[
                            timestamp] -
                            value,
                        actual=series[
                            timestamp],
                        severity=abs(
                            value) /
                            threshold
                ))

        return anomalies
```

4.4.2 Alert Integration

Anomalies trigger alerts through configurable channels:

Listing 15: Alert configuration

```
CREATE ALERT order_drop AS
SELECT
    store_id,
    COUNT(*) AS order_count,
    AVG(COUNT(*)) OVER (
        PARTITION BY store_id
        ORDER BY window_start
        ROWS BETWEEN 168 PRECEDING
            AND 1 PRECEDING  -- Last
            week
    ) AS expected
FROM mv_orders_by_hour
GROUP BY store_id, window_start
HAVING order_count < expected * 0.5
    -- 50% drop
WITH NOTIFICATION slack_webhook,
    email;
```

5 Evaluation

5.1 Experimental Setup

We evaluated Verus on production workloads from three merchants:

- **Small**: 10K events/day, 5 materialized views.
- **Medium**: 1M events/day, 15 materialized views.

- **Large**: 100M events/day, 30 materialized views.

Infrastructure:

- Stream processing: 3-node Flink cluster.
- State store: 3-node Redis cluster.
- Columnar storage: S3 + 10-node Presto cluster.

5.2 Ingestion Throughput

Table 1: Sustained ingestion throughput

Configuration	Events/second
Single partition	8,500
10 partitions	52,000
100 partitions	485,000

Throughput scales linearly with partitions. The large merchant's peak load (50K events/second) is well within capacity.

5.3 Query Latency

Table 2: Query latency by type (ms)

Query Type	p50	p95	p99
Dashboard (Redis)	18	45	82
Dashboard (merged)	125	280	520
Ad-hoc (simple)	850	2,100	4,500
Ad-hoc (complex)	3,200	8,500	15,000
Cohort analysis	5,500	12,000	25,000

Dashboard queries served from Redis meet interactive requirements ($<$ 100ms p99). Ad-hoc queries trade latency for flexibility.

5.4 Freshness

We measured end-to-end latency from event occurrence to query visibility:

Materialized views update within seconds. Columnar storage has higher latency due to micro-batching for write efficiency.

Table 3: Data freshness (seconds)

Metric	p50	p95	p99
Materialized views	1.2	2.8	5.1
Columnar storage	45	120	180

Table 4: Monthly infrastructure cost (USD)

Component	Small	Medium	Large
Kafka	50	150	800
Stream processing	100	400	2,500
Redis	75	250	1,200
Storage (S3)	5	50	500
Query engine	200	600	3,000
Total	430	1,450	8,000

5.5 Cost Analysis

Cost scales sub-linearly with volume due to shared infrastructure and compression benefits at scale.

5.6 Comparison with Alternatives

We compared Verus against pure-batch (BigQuery) and pure-stream (custom Flink) approaches:

Table 5: Approach comparison

Metric	BigQuery	Flink-only	Verus
Dashboard latency	2–5s	20ms	20ms
Ad-hoc latency	2–10s	N/A	1–15s
Freshness	1–4 hours	$<$ 1s	$<$ 5s
Query flexibility	High	Low	High
Operational complexity	Low	High	Medium

Verus achieves the freshness of stream processing with the flexibility of batch analytics.

6 Case Studies

6.1 Flash Sale Monitoring

A fashion retailer used Verus to monitor a flash sale event:

- Real-time dashboard showed orders/minute, inventory levels, and error rates.

- Anomaly detection alerted when payment failures spiked 3× normal.
- Root cause analysis (ad-hoc query) identified a specific payment provider issue.
- Response time from detection to mitigation: 4 minutes.

Without real-time analytics, the issue would have persisted for hours until batch reports surfaced it.

6.2 Marketing Attribution

An electronics retailer analyzed marketing effectiveness:

- Cohort analysis compared customers acquired through different channels.
- Lifetime value calculation showed paid search customers had 2.3× higher LTV than social.
- Budget reallocation based on insights increased ROAS by 45%.

7 Related Work

Druid [2] pioneered real-time OLAP with column-oriented storage and bitmap indexes. Verus extends this with richer analytical operators (cohorts, funnels) and hybrid query routing.

ClickHouse [3] offers excellent single-node performance for analytical queries. Verus provides horizontal scalability and stream processing integration.

The Lambda architecture [1] inspired our hybrid approach. We simplify operations by unifying the query interface and automating routing decisions.

8 Conclusion

Verus demonstrates that commerce analytics can achieve sub-second freshness without sacrificing query flexibility. The hybrid architecture—stream processing for materialized views, columnar storage for ad-hoc queries, intelligent routing between them—provides the best of both approaches.

Production deployments show the system handles 50,000+ events per second while serving dashboard queries in under 100ms. Merchants gain real-time visibility into their business without the operational complexity of maintaining separate batch and stream systems.

Future work will extend Verus with machine learning integration (anomaly detection models, forecasting) and support for streaming SQL standards (KSQL, Flink SQL).

References

[1] N. Marz and J. Warren, *Big Data: Principles and Best Practices of Scalable Real-Time Data Systems*, Manning, 2015.

[2] F. Yang et al., "Druid: A Real-time Analytical Data Store," *SIGMOD*, 2014.

[3] A. Milovidov, "ClickHouse: New Open Source Columnar Database," *HighLoad++*, 2016.

[4] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A Distributed Messaging System for Log Processing," *NetDB Workshop*, 2011.

[5] P. Carbone et al., "Apache Flink: Stream and Batch Processing in a Single Engine," *IEEE Data Engineering Bulletin*, vol. 38, no. 4, 2015.

[6] J. LeFevre et al., "Parquet," *Apache Software Foundation*, 2013.