PRACTICAL GREMLIN

An Apache TinkerPop Tutorial

Kelvin R. Lawrence, Stephen P. Mallette

Version v2-002-preview, 2026-02-26 16:36:56 UTC

Table of Contents

Chapter 1. INTRODUCTION

This second edition of Practical Gremlin is an evolving guide to the Apache TinkerPop Gremlin graph query and traversal language, built around real examples using real-world graph data. It is written for developers and practitioners who want to learn Gremlin by doing, with a focus on practical patterns, working traversals, and lessons learned from applying graph technology to real problems.

Feedback on this book is very much encouraged and welcomed! Please open GitHub issues as appropriate.

1.1. Welcome to the second edition

Practical Gremlin was first published in 2017. In the years that followed, the Apache TinkerPop graph computing framework has continued to evolve. While it was possible to keep the first edition mostly up to date and to publish periodic updates, there comes a time when so much has changed that it makes more sense to take a step back and release a second edition. The book has been updated to include the most recent new features added to the Gremlin query language and related technologies. Material such as discussions of migrating from old versions of Gremlin, and discussions of how to work around features then lacking, but since added, have been removed. The following paragraph was used to introduce the first edition. It remains as true today as it was then.

> The title of this book could equally well be '"A getting started guide for users of graph databases and the Gremlin query language featuring hints, tips, and sample queries"'. It turns out that is a bit too long to fit on one line for a heading but in a single sentence that describes the focus of this work pretty well.
>
> — Practical Gremlin first edition, October 2017

As with the first edition, I have resisted the urge to cover every single feature of TinkerPop one after the other in a reference manual fashion. Instead, what I have tried to do is capture the learning process that I myself have gone through using what I hope is a sensible flow from getting started to more advanced topics. To get the most from this book, I recommend having the Gremlin Console open, with the air route sample data loaded, as you follow along. I have not assumed that anyone reading this book has any prior knowledge of Apache TinkerPop, the Gremlin query language, or related tools. Everything you need to get started is introduced in Chapter 2.

I hope people continue to find what follows useful. It definitely remains a work in progress as the Apache TinkerPop framework, and in particular, the Gremlin query language, continues to evolve.

The book is available in multiple formats including PDF, HTML, ePub, and MOBI. Those versions, along with sample code and data, can be found at the project's home on GitHub. You will find a summary of everything that is available in the "Introducing the book sources, sample programs, and data" section.

1.2. How this book came to be

I forget exactly when, but sometime early in 2016 I started compiling a list of notes, hints, and tips, initially for my own benefit. My notes were full of things I had found poorly explained elsewhere while using graph databases and especially while using Apache TinkerPop, Gremlin, and JanusGraph. Over time that document continued to grow and had effectively become a book in all but name. After some encouragement from colleagues, I decided to release my notes as a 'living book' in an open source venue so that anyone who is interested can read it. It is definitely aimed at programmers and data scientists, but I hope it is also consumable by anyone using the Gremlin graph query and traversal language to work with graph databases.

I have included a large number of code examples and sample queries along with discussions of best practices and more than a few lessons I learned the hard way, that I hope you will find informative. I call it a 'living book' as my goal is to regularly make updates as I discover things that need adding while also trying to keep the content as up to date as possible as Apache TinkerPop itself evolves.

I remain extremely grateful to all those who have encouraged me to keep going with this adventure. Keeping up with a moving target requires a fair bit of work, but it remains a lot of fun!

Kelvin R. Lawrence
First draft (first edition): October 5th, 2017
Final draft (first edition): May 4th, 2022
First draft (second edition): January 1, 2026 Current draft (second edition): 2026-02-26 16:36:56 UTC

1.3. Providing feedback

Please let me know about any mistakes you find in this material, and also, please feel free to provide feedback of any sort. Suggested improvements are especially welcome. A good way to provide feedback is by opening an issue in the GitHub repository located at https://github.com/krlawrence/graph. You are currently reading revision v2-002-preview of the book.

The change history contains details of everything that has been added over time and can be found at this location: https://github.com/krlawrence/graph/blob/main/ChangeHistory.md

I am grateful to those who have already taken the time to review the manuscript and open issues or submit pull requests.

1.4. Some words of thanks

No open source project can succeed without dedicated contributors and equally dedicated users. Apache TinkerPop continues to be not just a technology, but a vibrant community as well. This book would have no audience but for the continued hard work, and interest, of that community.

As always, special thanks should go to, Marko Rodriguez, Daniel Kuppitz, Stephen Mallette, and others, who created TinkerPop and Gremlin, and drove its evolution for many years. I'm also grateful to Stephen for his help in putting this second edition of the book together.

Inspiration as to what topics people are interested in often comes from seeing the active discussions on-line at venues such as StackOverflow and the Gremlin Users Google Group. Since the first edition of this book was released, Apache TinkerPop now also has an active Discord server where many exciting topics get discussed daily.

I continue to be grateful for the contributions of my former colleagues, Graham Wallis, Jason Plurad, and Adam Holley, who helped refine and improve several of the example queries contained in the first edition of this book. Gremlin is definitely a bit of a team sport. We spent many fun hours discussing the best way to handle different types of queries and traversals!

Lastly, I would like to thank everyone who has submitted feedback and ideas via e-mail or GitHub issues, and pull requests. That is the best part about this being a 'living book' we can continue to improve and evolve it just as the technology it is about continues to evolve. Your help and support is very much appreciated.

1.5. Thoughts on the Second Edition

Except for this section, references to "I" in this book refer to the book's original author, Kelvin Lawrence. As the editor and co-author of the Second Edition, I didn't feel as though "I" should become "We" anywhere, as I believe that much of the appeal of this book lies in Kelvin relating his personal discoveries on his journey with Gremlin. I've tried to preserve that feeling and voice as much as possible in my many additions and edits.

As I write this section, I realize that I've authored many lines of TinkerPop's Reference Documentation, Tutorials, and Recipes over the years and, as a result, I tend to think there is a certain completeness to the documentation that is officially offered. I think that's a valuable aspect of the TinkerPop project, yet I've also always recognized the importance and impact of this book as a resource for Gremlin users. My personal opinion is that its approach and organization is what allows it to make that helpful impact. Where TinkerPop Documentation has its completeness presented all at once, this book gives you a clear, ramped process to learning Gremlin so that you can make use of that completeness in the official documentation.

In 2023, TinkerPop was in the midst of releasing 3.6.x and 3.7.x release lines which had introduced a number of important new features. Kelvin and I started realizing that Practical Gremlin was falling behind TinkerPop's continued evolution significantly. With even bigger sets of changes expected on the horizon with 4.0, it felt as though it was time to make an organized effort to produce an updated Second Edition.

We officially announced that work had started on it in September 2023 and with the big release of 3.8.0 in November 2025, we believed it time to polish up the draft and make an official publication of the new edition. As with the First Edition, we expect the book to continue to be a work-in-progress and will make minor publications as new TinkerPop releases appear.

As you read this edition, please keep in mind that it reflects the state of Gremlin and TinkerPop at a particular point in time, but it is also part of an ongoing conversation with the community. Your questions, suggestions, and examples help shape where the book goes next. If you find places where it could be clearer, more helpful, or more complete, your feedback is welcome and will help guide future updates.

Stephen Mallette
Second edition editor and co-author

1.6. What is this book about?

This book is about learning to think in Gremlin by working through practical examples. It focuses on how to express real graph questions as traversals, how to read and reason about Gremlin code, and how to apply common patterns to your own graphs. As the book evolves, new examples and refinements continue to be added, but the core goal remains the same: to help you build an intuitive understanding of how Gremlin works.

In this book you will find real examples featuring real-world graph data. That data, along with sample code and example applications, is available for download from the GitHub project as well as many other items. The graph, 'air-routes', is a model of the world airline route network between 3,504 airports, including 50,637 routes. The examples presented will work unmodified with the `air-routes.graphml` file loaded into the Gremlin console running with a TinkerGraph. How to set that environment up is covered in the "Download, install, and launch the Gremlin Console" section below.

The examples in this book have been tested using Apache TinkerPop release 3.8.0.

TinkerGraph is an 'in-memory' graph, meaning nothing gets saved to disk automatically. It is included as part of the Apache TinkerPop download. The goal of this tutorial is to allow someone with little to no prior knowledge to get up and going quickly using the Gremlin Console and the 'air-routes' graph. Later in the book we discuss writing standalone applications in other programming languages such as Java, Groovy, and Python.

The first few sections of the book focus on showing some basic Gremlin queries that are both useful and yet easy to understand. By the end of Chapter 3 you should have a basic understanding of how to explore the air-routes graph using commonly used Gremlin steps. Chapters 4, 5, and 6 explore Gremlin in more depth.

How this book is organized

Chapter 1 – INTRODUCTION

- We start our journey with a brief introduction to Apache Tinkerpop and a quick look at why Graph databases are of interest to us. We also discuss how the book is organized and where to find additional materials, such as sample code and data sets.

Chapter 2 – GETTING STARTED

- Many of the examples throughout the book use the Gremlin Console and TinkerGraph, and both are introduced in this chapter. We also introduce the air-routes example graph - `air-routes.graphml` - used throughout the book.

Chapter 3 – WRITING GREMLIN QUERIES

- Now that the basics have been covered, things start to get a lot more interesting! It's time to start writing Gremlin queries. We briefly explore how we could have built the 'air-routes' graph using a relational database, and then look at how SQL and Gremlin are both similar in

some way,s and very different in others. We then introduce several of the key Gremlin query language '"steps"'. We focus on exploring the graph rather than changing it in this chapter.

Chapter 4 – BEYOND BASIC QUERIES

- Having now introduced Gremlin in some detail, we introduce the Gremlin steps that can be used to create, modify, and delete, data. We present a selection of best practices and start to explore some more advanced query writing.

Chapter 5 – MISCELLANEOUS QUERIES AND THE RESULTS THEY GENERATE

- Using the Gremlin steps introduced in Chapters 3 and 4, we are now ready to use what we have learned so far and write queries that analyze the air-routes graph in more depth and answer more complicated questions. The material presented includes a discussion of analyzing distances, route distribution, and writing geospatial queries.

Chapter 6 – MOVING BEYOND THE GREMLIN CONSOLE

- The next step in our journey is to move beyond the Gremlin Console and take a look at interacting with a TinkerGraph using Java and Groovy applications.

Chapter 7 – INTRODUCING GREMLIN SERVER

- Our journey so far has focused on working with graphs in a "directly attached" fashion. We now introduce Gremlin Server as a way to deploy and interact with remotely hosted graphs.

Chapter 8 – COMMON GRAPH SERIALIZATION FORMATS

- Having introduced Gremlin Server, we take a look at some common Graph serialization file formats along with coverage of how to use them in the context of TinkerPop enabled graphs. We take a close look at the TinkerPop GraphSON (JSON) format, which is used extensively when using Gremlin queries in conjunction with a Gremlin Server.

Chapter 9 – FURTHER READING

- Our journey to explore Apache TinkerPop and Gremlin concludes with a look at useful sources of further reading. We present links to useful websites where you can find tools and documentation for many of the topics and technologies covered in this book.

1.7. Introducing the book sources, sample programs, and data

All work related to this project is being done in the open at GitHub. A list of where to find the key components is provided below. The examples in this book make use of a sample graph called 'air-routes' which contains a graph based on the world airline route network between over 3,504 airports. The sample graph data, quite a bit of sample code, and some larger demo applications can all be found at the same GitHub location that hosts the book manuscript. You will also find releases of the book in various formats (HTML, PDF, DocBook/XML, MOBI, and EPUB) at the same GitHub location.

The sample programs include standalone Java, Groovy, Python, and Ruby examples as well as many examples that can be run from the Gremlin Console. There are some differences between using Gremlin from a standalone program and from the Gremlin Console. The sample programs demonstrate several of these differences. The sample applications area contains a full example HTML and JavaScript application that lets you explore the 'air-routes' graph visually. The home page for the GitHub project includes a README.md file to help you navigate the site. Below are

some links to various resources included in this book.

Where to find the book, samples, and data

Project home

- https://github.com/krlawrence/graph

Book manuscript in Asciidoc format

- This file can be viewed using the GitHub web interface. It will always represent the very latest updates.
- https://github.com/krlawrence/graph/tree/main/book

Latest PDF and HTML snapshots

- These files are regularly updated to reflect any significant changes. These are the only generated formats that are updated outside the full release cycle. The PDF version includes pagination as well as page numbering and is produced using an A4 page size. The HTML version does not include these features. Otherwise, they are more or less identical.
- https://kelvinlawrence.net/book/PracticalGremlin.pdf
- https://kelvinlawrence.net/book/PracticalGremlin.html

Official book releases in multiple formats

- Official releases include AsciiDoc, HTML, PDF, ePub, MOBI, and DocBook versions as well as snapshots of all the samples and other materials in a single package available through GitHub Releases. The eBook and MOBI versions are really intended to be read using e-reader devices, and for that reason use a white background for all source code highlighting to make it easier to read on monochrome devices.
- I recommend using the PDF version if possible as it has page numbering. If you prefer reading the book as if it were a web page, then by all means use the HTML version. You will just not get any pagination or page numbers. The DocBook format can be read using tools such as Yelp on Linux systems, but is primarily included so that people can use it to generate other formats that I do not already provide. The MOBI and ePub versions may require you to change the font size you use on your device to make things easier to read.
- https://github.com/krlawrence/graph/releases

Sample data (`air-routes.graphml`)

- https://github.com/krlawrence/graph/tree/main/sample-data

Sample code

- https://github.com/krlawrence/graph/tree/main/sample-code

Example applications

- https://github.com/krlawrence/graph/tree/main/demos

Change history

- If you want to keep up with the changes being made, this is the file to keep an eye on.
- https://github.com/krlawrence/graph/blob/main/ChangeHistory.md

1.8. Apache TinkerPop Evolution

Over the last 15 years, TinkerPop, and especially Gremlin, have evolved substantially from their earliest versions. What we now know as Apache TinkerPop is the result of an open source project created in 2009 and moved to the Apache Software Foundation (ASF) in 2015, after the final release of TinkerPop version 2. The first official release of Apache TinkerPop 3.0 came in July 2015, with the project being promoted to Apache's "top-level" status the following year. After a decade of continuous releases for TinkerPop 3.0, the project released a beta version of 4.0 in January 2025 for early evaluation.

We focus this second edition of the book on the semantics of Gremlin as of TinkerPop release 3.8.0.

If you are new to TinkerPop and Gremlin, you can probably skip the next few sections. They appeared in a slightly modified form, as part of the First Edition, and provided a way to highlight the arrival of key new features. These notes have been left in the Second Edition as there are still people using older versions of Gremlin, and it can be useful to have a list like this to cross-reference.

The complete ApacheTinkerPop change history can be found at https://github.com/apache/tinkerpop/blob/master/CHANGELOG.asciidoc

Graph database engines that support Apache TinkerPop often take a while to move up to new releases, and it's always a good idea to verify the exact level the database you are using supports.

This version of the book covers features of Gremlin available as part of the TinkerPop 3.8.0 release. As appropriate, notes and examples have been added that show other ways to perform tasks that new features may simplify. In some cases, notes have been added to point out when more recent features first appeared.

1.8.1. TinkerPop 3.4

A major update to Apache TinkerPop, version 3.4.0, was released in January 2019, and a number of point releases followed.

Full details of all the new features added in the TinkerPop 3.4.x releases can be found at the following link: https://github.com/apache/tinkerpop/blob/3.4-dev/CHANGELOG.asciidoc

1.8.2. TinkerPop 3.5

Apache TinkerPop 3.5.0 was released in May 2021. This update introduced a number of improvements in areas such as Gremlin client drivers, the Gremlin Server, and overall bug fixes. The release also improved the Gremlin query language in some key areas. Some features that had been declared deprecated in earlier releases were finally removed as part of the 3.5.0 update. If you have queries and code that still use these deprecated features, as part of an upgrade to the 3.5.x level, you will need to make the appropriate changes.

The main breaking change to be aware of is that 'Order.incr' and 'Order.decr' were removed from

the Gremlin language. The newer 'Order.asc' and 'Order.desc' must be used instead. The examples in this book and those in the `sample-code` folder have been updated to reflect these changes.

In January 2022, the TinkerPop 3.5.2 release added a native `datetime` operator to the Gremlin language such that dates can be added without needing programming language specific constructs. This is useful when sending Gremlin queries as text strings.

Full details of all the new features added in the TinkerPop 3.5.x releases can be found at the following link: https://github.com/apache/tinkerpop/blob/3.5-dev/CHANGELOG.asciidoc

1.8.3. TinkerPop 3.6

Apache TinkerPop 3.6.0 was released in April 2022. Coming almost exactly a year after the initial 3.5.0 release, this is one of the most significant TinkerPop releases since TinkerPop 3.4.0 appeared in January 2019. The release contains many improvements, including several new Gremlin steps, designed to make commonly performed tasks much easier. Notable improvements include:

- New 'mergeV' and 'mergeE' steps that make "create if not exist" type queries, sometimes referred to as "upserts", much easier to write. Over time, these steps will replace the use of the 'fold...coalesce' pattern and will also replace the various "map injection" patterns that can be used to create multiple vertices and edges in a single query.
- A new 'TextP.regex' predicate that allows regular expressions to be used when comparing strings.
- The 'property' step can now be given a map of key/value pairs so that several properties can be created at once.
- A new 'element' step that can be used to find the parent element (vertex or edge) of a property.
- A new 'call' step that lays the foundation enabling Gremlin queries to call other endpoints. This opens up many types of interesting use cases such as query federation, and looking up values from other services.
- A lot of effort has been put into removing unnecessary exceptions by filtering out parts of traversals instead of failing with an error. This is especially so in the case of 'by' modulators that now filter when a value does not exist rather than throw an exception. This work began as part of the TinkerPop 3.5.2 update and is completed as of TinkerPop 3.6.0.
- A new 'fail' step that can be used to abort a query in a controlled way.

Full details of all the new features added in the TinkerPop 3.6.x releases can be found at the following link: https://github.com/apache/tinkerpop/blob/3.6-dev/CHANGELOG.asciidoc

1.8.4. TinkerPop 3.7

TinkerPop 3.7.0 was released July 2023 and with the follow-on release of 3.7.1 a few months later, introduced a large expansion of the Gremlin language, providing long-awaited features for manipulating strings, collections, and dates. There were other major features as well, such as

TinkerGraph gaining some simple transactional features and the ability for properties to be returned on elements from Gremlin Server, rather than only getting references. Notable improvements include:

- New Gremlin steps for working with strings: 'asString', 'concat', 'length', 'toLower', 'toUpper', 'trim', 'lTrim', 'rTrim', 'reverse', 'replace', 'split', 'substring, and 'format'.
- New Gremlin steps for working with collections: 'any', 'all', 'product', 'merge', 'intersect', 'combine', 'conjoin', 'difference', 'disjunct,' and 'reverse'.
- A new Gremlin steps for working with dates: 'asDate', 'dateAdd' and 'dateDiff'.
- The 'union' step became available as a start step.
- Improved syntax for specifying cardinality directly within a 'Map' for use with 'mergeV'.
- TinkerGraph gained support for simple transactions.
- Graph elements like 'Vertex' and 'Edge' can now be returned from Gremlin Server with their properties attached using the 'materializeProperties' option.

Full details of all the new features added in the TinkerPop 3.7.x releases can be found at the following link: https://github.com/apache/tinkerpop/blob/3.7-dev/CHANGELOG.asciidoc

1.8.5. TinkerPop 3.8

TinkerPop 3.8.0 was released November 2025 and established a wide mix of features and improvements to Gremlin semantics designed to enhance language consistency. Some of these changes do lead to behaviors that break from previous versions of TinkerPop, but are expected to stay consistent with TinkerPop 4.0, which should help ease the migration there when 4.0 eventually releases. Here are some of the important highlights to consider from this release:

- New Gremlin steps for type conversions: 'asBool' and 'asNumber'.
- Added the new 'typeOf' predicate to make it possible to filter traverser objects given their data type.
- The 'none' step was renamed to 'discard', and 'none' has been modified to take 'P' as an argument, whose behavior is now a complement to 'any' and 'all' steps.
- The minimum Java version supported is JDK11.
- The default implementation for the 'date' data type for Java is 'OffsetDateTime' rather than 'java.util.Date', which means that steps like 'asDate' and helpers like the 'datetime' function will return 'OffsetDateTime' whose string representation is an ISO 8601 format.
- Creation of 'g' has been simplified slightly by replacing more verbose construction options of 'traversal().withEmbedded(...)' and 'traversal().withRemote(...)' with a single 'traversal().with(...)'.
- The 'split' method splits a string to characters if given an empty string as a separator argument.
- The Gremlin language removed syntax for 'Vertex', made use of 'new' optional, included support for 'withoutStrategies,' and removed 'Map' key restrictions related to reserved words.
- The semantics for the 'choose' step changed to match the first option only, pass through

traversers when the choice is unproductive or the determined choice unmatched, and introduces a new 'Pick.unproductive' option.

- Gremlin no longer follows Groovy by defaulting float values to 'BigDecimal' – they go to 'double' instead.
- The 'valueMap', 'propertyMap', 'groupCount', 'dedup', 'sack', 'sample', 'aggregate' steps can no longer take 'by' modulators beyond the number each step expects.
- Included the air-routes dataset, as is used in this book, as official sample data in the distribution.
- The form of 'has' that takes a key and a 'Traversal' as an argument has been removed to prevent confusion around its functionality and to prepare for a revised implementation of it in 4.0.0.
- The semantics for 'limit', 'skip,' and 'range' with 'global' scope are tracked per iteration inside 'repeat'.
- The semantics for 'limit', 'skip,' and 'range', with 'local' scope will no longer automatically unwrap a single item in a list if there is only one object present.

Full details of all the new features added in the TinkerPop 3.8.x releases can be found at the following link: https://github.com/apache/tinkerpop/blob/3.8-dev/CHANGELOG.asciidoc

1.8.6. TinkerPop 4.0

TinkerPop 4.0.0 has released a beta version for early evaluation of specific features and remains under active development. We mention it here for awareness but do not intend to cover any features it offers until its full official release. We will only note that the intention of the TinkerPop Community is for 4.0.0 to have the same semantics for Gremlin as 3.8.0. As a result, all the Gremlin examples provided in this second edition of Practical Gremlin will work equally well for both versions.

1.9. So what is a graph database and why should I care?

This book is mainly intended to be a tutorial in working with graph databases and related technology using the Gremlin query language. However, it is worth spending just a few moments to summarize why it is important to understand what a graph database is, what some good use cases for graphs are and why you should care in a world that is already full of all kinds of SQL and NoSQL databases. In this book we are going to be discussing 'directed property graphs'. At the conceptual level, these types of graphs are quite simple to understand. You have three basic building blocks: vertices, edges, and properties. Vertices represent "things" such as people or places. Edges represent connections between those vertices, and properties are information added to the vertices and edges as needed. The 'directed' part of the name means that any edge has a direction. It goes 'out' from one vertex and 'in' to another. You will sometimes hear people use the word 'digraph' as shorthand for 'directed graph'. Consider the relationship "Kelvin knows Jack". This could be modeled as a vertex for each of the people and an edge for the relationship as follows.

Kelvin — knows → Jack

Note the arrow which implies the direction of the relationship. If we wanted to record the fact that Jack also admits to knowing Kelvin, we would need to add a second edge from Jack to Kelvin. Properties could be added to each person to give more information about them. For example, my age might be a property on my vertex.

It turns out that Jack really likes cats. We might want to store that in our graph as well so we could create the relationship:

Jack — likes → Cats

Now that we have a bit more in our graph, we could answer the following question: "who does Kelvin know that likes cats?"

Kelvin — knows → Jack — likes → Cats

This is a simple example, but hopefully you can already see that we are modeling our data the way we think about it in the real world. Armed with this knowledge, you now have all the basic building blocks you need to start thinking about how you might model things you are familiar with as a graph.

So getting back to the question of "why should I care?", well, if something looks like a graph, then wouldn't it be great if we could model it that way. Many things in our everyday lives center around things that can very nicely be represented in a graph. Things such as your social and business networks, the route you take to get to work, the phone network, airline route choices for trips you need to take are all great candidates. There are also many great business applications for graph databases and algorithms. These include recommendation systems, crime prevention, and fraud detection to name but three.

The reverse is also true. If something does not feel like a graph, then don't try to force it to be. Your videos are probably doing quite nicely living in the object store where you currently have them. A sales ledger system built using a relational database is probably doing just fine where it is, and likewise a document store is quite possibly just the right place to be storing your documents. So "using the right tool for the job" remains as valid a phrase here as elsewhere. Where graph databases come into their own is when the data you are storing is intrinsically linked by its very nature, the air-routes network used as the basis for all the examples in this book being a perfect example of such a situation.

Those of you that looked at graphs as part of a computer science course are correct if your reaction was "Surely graphs have been around for ages, why is this considered new?". Indeed, Leonard Euler is credited with demonstrating the first graph problem and inventing the whole concept of "Graph Theory" all the way back in 1763 when he investigated the now famous "Seven Bridges of Koenigsberg" problem.

If you want to read a bit more about graph theory and its present-day application, you can find a lot of good information online. Here's a Wikipedia link to get you started: https://en.wikipedia.org/wiki/Graph_theory

So, given Graph Theory is anything but a new idea, why is it that only recently we are seeing a massive growth in the building and deployment of graph database systems and applications? At least part of the answer is that computer hardware and software have reached the point where you

can build large big data systems that scale well for a reasonable price. In fact, it's even easier than ever to build large systems because you don't have to buy the hardware that your system will run on when you use the cloud.

While you can certainly run a graph database on your laptop—I do just that every day—the reality is that in production, at scale, they are big data systems. Large graphs commonly have many billions of vertices and edges in them, taking up petabytes of data on disk. Graph algorithms can be both compute- and memory-intensive, and it is only fairly recent that deploying the necessary resources for such big data systems has made financial sense for more everyday uses in business, and not just in government or academia. Graph databases are becoming much more broadly adopted across the spectrum, from high-end scientific research to financial networks and beyond.

Another factor that has really helped start this graph database revolution is the availability of high-quality open source technology. There are a lot of great open source projects addressing everything from the databases you need to store the graph data, to the query languages used to traverse them, all the way up to visually displaying graphs as part of the user interface layer. In particular, it is so-called 'property graphs' where we are seeing the broadest development and uptake. In a property graph, both vertices and edges can have properties (effectively, key-value pairs) associated with them. There are many styles of graph that you may end up building, and there have been whole books written on these various design patterns, but the property graph technology we will focus on in this book can support all the most common usage patterns. If you hear phrases such as 'directed graph' and 'undirected graph', or 'cyclic' and 'acyclic' graph, and many more as you work with graph databases, a quick online search will get you to a place where you can get familiar with that terminology. A deep discussion of these patterns is beyond the scope of this book, and it's in no way essential to have a full background in graph theory to get productive quickly.

A third, and equally important, factor in the growth we are seeing in graph database adoption is the low barrier of entry for programmers. As you will see from the examples in this book, someone wanting to experiment with graph technology can download the Apache TinkerPop package and as long as Java 11+ is installed, be up and running with zero configuration (other than unzipping of the files), in as little as five minutes. Graph databases do not force you to define schemas or specify the layout of tables and columns before you can get going and start building a graph. Programmers also seem to find the graph style of programming quite intuitive as it closely models the way they think of the world.

Graph database technology should not be viewed as a "rip and replace" technology, but as very much complementary to other databases that you may already have deployed. One common use case is for the graph to be used as a form of smart index into other data stores. This is sometimes called having a polyglot data architecture.

1.10. A word about terminology

The words 'node' and 'vertex' are synonymous when discussing a graph. This book will prefer the term 'vertex' and 'vertices' as the Apache TinkerPop documentation almost exclusively uses it when discussing Gremlin queries and other concepts. We will only see a shade of the term 'node' when we refer to "supernodes", relating to vertices of an especially high degree, as this term has wide acceptance independently of TinkerPop. By the same token, this book will be consistent with the official TinkerPop documentation when discussing the connections between vertices, where we will

use the term 'edge' or the plural form, 'edges'. In other books and articles you may also see terms like 'relationship' or 'arc' used. Again, these terms are synonymous in the context of graphs.

Chapter 2. GETTING STARTED

In this chapter you will set up the Gremlin Console, load the air-routes sample graph, and run your first Gremlin traversals against it. You will learn how to start and stop the console, adjust a few useful settings, and verify that the sample data has been loaded correctly. By the end of the chapter, you should be comfortable using the console as a workspace for experimenting with the examples used throughout the rest of the book.

2.1. What is Apache TinkerPop?

Apache TinkerPop is a graph computing framework and top-level project hosted by the Apache Software Foundation. The homepage for the project is located at this URL: https://tinkerpop.apache.org/

The project includes the following components:

Gremlin

- A graph traversal (query) language

Gremlin Console

- An interactive shell for working with local or remote graphs.
- https://tinkerpop.apache.org/docs/current/reference/#gremlin-console

Gremlin Server

- Allows hosting of graphs remotely via an HTTP/Web Sockets connection.
- https://tinkerpop.apache.org/docs/current/reference/#gremlin-server

TinkerGraph

- A small in-memory graph implementation that is great for learning.
- https://tinkerpop.apache.org/docs/current/reference/#tinkergraph-gremlin

Programming Interfaces

- A set of programming interfaces written in Java
- https://tinkerpop.apache.org/javadocs/current/full/

Documentation

- A user guide, a tutorial and programming API documentation.
- https://tinkerpop.apache.org/docs/current/
- https://tinkerpop.apache.org/docs/current/reference/

Useful Recipes

- A set of examples or "recipes" showing how to perform common graph-oriented tasks using Gremlin queries.
- https://tinkerpop.apache.org/docs/current/recipes/

The programming interfaces allow providers of graph databases to build systems that are TinkerPop enabled and allow application programmers to write programs that talk to those systems.

Any TinkerPop enabled graph databases can be accessed using the Gremlin query language and corresponding API. We can also use the TinkerPop API to write client code, in languages like Java, that can talk to a TinkerPop enabled graph. For most of this book we will be working within the Gremlin Console with a local graph. However, in Chapters 7 and 8 we take a look at Gremlin Server and some other TinkerPop enabled environments. Most of Apache TinkerPop has been developed using Java, but there are also bindings available for many other programming languages such as Groovy, Python, Go, JavaScript, and C#. These bindings help make Gremlin feel comfortable to you as you can work with Gremlin in the idioms of the programming language that you are most familiar with.

Even though this book focuses on Gremlin written with Groovy, you should remember that whatever examples you see in Groovy can easily be converted to any other supported programming language. You need to understand the idioms of the language you are using, and converting should be straightforward. For example, Python prefers snake case compared to Groovy preferring camel-case formatting. Therefore, a Groovy query of 'g.addV("person")' just converts to 'g.add_v("person")' in Python. You will read more about Gremlin translation in the "Translating Gremlin to different programming languages" section.

The queries used as examples in this book have been tested with Apache TinkerPop version 3.8.0 as well as some prior releases where appropriate. Tests were performed using the TinkerGraph in memory graph and the Gremlin console, as well as other TinkerPop enabled graph stores.

2.2. The Gremlin Console

The Gremlin Console is an interactive shell for running Gremlin traversals against a graph. In this book it serves as the main workspace for experimenting with the air-routes graph, trying out examples, and exploring variations of your own. It is based on the Groovy Shell, and if you have used any of the other console environments such as those found with Scala, Python, and Ruby, you will feel right at home here. The console offers a low overhead (you can set it up in seconds) and a low barrier to entry as a way to start to play with graphs on your local computer. The console can actually work with graphs that are running locally or remotely, but for the majority of this book we will keep things simple and focus on local graphs.

To follow along with this tutorial, you will need to have installed the Gremlin console or have access to a TinkerPop/Gremlin enabled graph store such as TinkerGraph or JanusGraph.

Regardless of the environment you use, if you work with Apache TinkerPop enabled graphs, the console should always be installed on your machine!

2.2.1. Download, install, and launch the Gremlin Console

You can download the Gremlin Console from the official Apache TinkerPop website at https://tinkerpop.apache.org/

It only takes a few minutes to get the console installed and running. You just download the ZIP, 'unzip' it, and you are all set. TinkerPop also requires Java to be at version 11 or higher.

For more information on the compatability of Gremlin with versions of Java and

> other languages, please see the official documentation.

The console download also includes all the JAR files that are needed to write a standalone Java or Groovy TinkerPop application, but that is a topic for later!

When you start the console, you will be presented with a banner/logo and a prompt that will look something like this. Don't worry about the plugin messages, yet we will talk about those a bit later.

```
$ ./gremlin.sh

         \,,,/
         (o o)
-----oOOo-(3)-oOOo-----
plugin activated: tinkerpop.server
plugin activated: tinkerpop.utilities
plugin activated: tinkerpop.tinkergraph
gremlin>
```

You can get a list of the available commands by typing ':help'. Note that a colon prefixes all commands to the console itself. This enables the console to distinguish them as special and different from actual Gremlin and Groovy commands.

```
gremlin> :help

For information about Groovy, visit:
    http://groovy-lang.org

Available commands:
  :help       (:h  ) Display this help message
  ?           (:?  ) Alias to: :help
  :exit       (:x  ) Exit the shell
  :quit       (:q  ) Alias to: :exit
  import      (:i  ) Import a class into the namespace
  :display    (:d  ) Display the current buffer
  :clear      (:c  ) Clear the buffer and reset the prompt counter
  :show       (:S  ) Show variables, classes or imports
  :inspect    (:n  ) Inspect a variable or the last result with the GUI object browser
  :purge      (:p  ) Purge variables, classes, imports or preferences
  :edit       (:e  ) Edit the current buffer
  :load       (:l  ) Load a file or URL into the buffer
  .           (:.  ) Alias to: :load
  :save       (:s  ) Save the current buffer to a file
  :record     (:r  ) Record the current session to a file
  :history    (:H  ) Display, manage and recall edit-line history
  :alias      (:a  ) Create an alias
  :grab       (:g  ) Add a dependency to the shell environment
  :register   (:rc ) Register a new command with the shell
  :doc        (:D  ) Open a browser window displaying the doc for the argument
  :set        (:=  ) Set (or list) preferences
```

```
  :uninstall  (:-  ) Uninstall a Maven library and its dependencies from the Gremlin
Console
  :install    (:+  ) Install a Maven library and its dependencies into the Gremlin
Console
  :plugin     (:pin) Manage plugins for the Console
  :remote     (:rem) Define a remote connection
  :submit     (:>  ) Send a Gremlin script to Gremlin Server
  :bytecode   (:bc ) Gremlin bytecode helper commands
  :cls        (:C  ) Clear the screen.

For help on a specific command type:
    :help command
```

Of all the commands listed above ':clear' (':c' for short) is important to remember. If the console starts acting strangely, or you find yourself stuck with a prompt like "……1>" , typing ':clear' will reset things nicely.

It is worth noting that as mentioned above, the console is based on the Groovy Shell, and as such you can enter valid Groovy code directly into the console. So as well as using it to experiment with graphs and Gremlin, you can use it as, for example, a desktop calculator should you so desire!

```
gremlin> 2+3
==>5

gremlin> a = 5
==>5

gremlin> println "The number is ${a}"
The number is 5

gremlin> for (a in 1..5) {print "${a} "};println()
1 2 3 4 5
```

If you want to see lots of examples of the output from running various queries, you will find plenty in the "MISCELLANEOUS QUERIES AND THEIR RESULTS" section of this book where we have tried to go into more depth on various topics.

Mostly, you will run the console in its interactive mode. However, you can also pass the name of a file as a command line parameter, preceded by the '-e' flag, and Gremlin will execute the file and exit. For example, if you had a file called "mycode.groovy", you could execute it directly from your command line window or terminal window as follows:

```
$ ./gremlin.sh -e mycode.groovy
```

If you want to have the console run your script and not exit afterward, you can use the '-i' option instead of '-e'.

You can get help on all the command line options for the console by typing 'gremlin --help'. You should get back some help text that looks like this

```
$ ./gremlin.sh --help

Usage: gremlin.sh [-CDhlQvV] [-e=<SCRIPT ARG1 ARG2 ...>]... [-i=<SCRIPT ARG1
                  ARG2 ...>...]...
  -C, --color     Disable use of ANSI colors
  -D, --debug     Enabled debug Console output
  -e, --execute=<SCRIPT ARG1 ARG2 ...>
                  Execute the specified script (SCRIPT ARG1 ARG2 ...) and close
                    the console on completion
  -h, --help      Display this help message
  -i, --interactive=<SCRIPT ARG1 ARG2 ...>...
                  Execute the specified script and leave the console open on
                    completion
  -l              Set the logging level of components that use standard logging
                    output independent of the Console
  -Q, --quiet     Suppress superfluous Console output
  -v, --version   Display the version
  -V, --verbose   Enable verbose Console output
```

If you ever want to check which version of TinkerPop you have installed, you can enter the following command from inside the console.

```
// What version of the console am I running?
gremlin>  Gremlin.version()
==>3.8.0
```

One thing that is not at all obvious is that the console quietly imports a large number of Java Classes and Enums on your behalf as it starts up. This makes writing queries within the console simpler. However, as we shall explore in the "Important Classes and Enums to be aware of" section later, once you start writing standalone programs in Java or other languages, you need to actually know what the console did on your behalf. Reading through that section will help familiarize you with the classes you need to import to your application code.

2.2.2. Saving output from the Gremlin Console to a file

Sometimes it is useful to save part or all of a Gremlin Console session to a file so that you can review it later, compare different traversals, or share it with others. The console provides a simple way to start and stop recording, which captures both the commands you type and the results that are displayed.

In the following example, we turn session recording on using ':record start mylog.txt' which will force all commands entered and their output to be written to the file 'mylog.txt' until the command ':record stop' is entered. The command 'g.V().count().next()' just counts how many vertices are in the graph. We will explain the Gremlin graph traversal and query language in detail starting in the

next section.

```
gremlin> :record start mylog.txt
Recording session to: "mylog.txt"

gremlin> g.V().count().next()
==>3749
gremlin> :record stop
Recording stopped; session saved as: "mylog.txt" (157 bytes)
```

If we were to look at the 'mylog.txt' file, this is what it now contains.

```
// OPENED: Tue Sep 12 10:43:40 CDT 2017
// RESULT: mylog.txt
g.V().count().next()
// RESULT: 3618
:record stop
// CLOSED: Tue Sep 12 10:43:50 CDT 2017
```

For the remainder of this book we are not going to show the 'gremlin>' prompt or the '==>' output identifier as part of each example, just to reduce clutter a bit. You can assume that each command was entered and tested using the console, however.

If you want to learn more about the console itself, you can refer to the official TinkerPop documentation and, even better, have a play with the console and the built-in help.

2.2.3. Setting up console preferences

There are a number of preferences that can be established within the Gremlin Console to make it more suitable for your needs. The ':set' command is used to establish various preference values. Let's look at a few helpful configurations.

The first option to know is 'max-iteration'. The console will only display the first 100 lines of output for any command by default. If you'd like to see more, you would need to increase this value.

```
:set max-iteration 1000
```

Set the 'max-iteration' to '-1' to have no limit in the number of lines displayed.

If you are on a system that can display colors, there are a wide range of color options you can modify to suit your needs. The various color settings take a comma-separated combination of a foreground, background and attribute.

```
:set error.color black,bg_black,underline
```

If you'd like to remove console configurations, you can use the ':purge preferences' command.

The full list of available preferences can be found in the Apache TinkerPop reference documentation https://tinkerpop.apache.org/docs/current/reference/#console-preferences

2.3. Introducing TinkerGraph

TinkerGraph is the in-memory reference implementation that ships with Apache TinkerPop and is included with the Gremlin Console download. It runs inside a single JVM process, stores its data in memory, and is designed primarily for learning, experimentation, and small test graphs, but does have use cases for production workloads in some scenarios.

This book was mostly developed using TinkerGraph The nice thing about TinkerGraph is that for learning and testing things you can run everything you need on your laptop or desktop computer and be up and running very quickly. We will also explain how to get started with the console and TinkerGraph a bit later in this section.

TinkerPop defines a number of capabilities that a graph store should support. Some are optional, others are not. If supported, you can query any TinkerPop enabled graph store to see which features are supported using a command such as 'graph.features()' once you have established the 'graph' object. We will look at how to do that soon. The following list shows the features supported by TinkerGraph. This is what you would get back should you call the 'features' method provided by TinkerGraph. We have arranged the list in two columns to aid readability. Don't worry if not all of these terms make sense right away – we'll get there soon!

Output from graph.features()

```
> GraphFeatures                       > VertexPropertyFeatures
>-- ConcurrentAccess: false           >-- UserSuppliedIds: true
>-- ThreadedTransactions: false       >-- StringIds: true
>-- Persistence: true                 >-- RemoveProperty: true
>-- Computer: true                    >-- AddProperty: true
>-- Transactions: false               >-- NumericIds: true
> VariableFeatures                    >-- CustomIds: false
>-- Variables: true                   >-- AnyIds: true
>-- LongValues: true                  >-- UuidIds: true
>-- SerializableValues: true          >-- Properties: true
>-- FloatArrayValues: true            >-- LongValues: true
>-- UniformListValues: true           >-- SerializableValues: true
>-- ByteArrayValues: true             >-- FloatArrayValues: true
>-- MapValues: true                   >-- UniformListValues: true
>-- BooleanArrayValues: true          >-- ByteArrayValues: true
>-- MixedListValues: true             >-- MapValues: true
>-- BooleanValues: true               >-- BooleanArrayValues: true
>-- DoubleValues: true                >-- MixedListValues: true
>-- IntegerArrayValues: true          >-- BooleanValues: true
>-- LongArrayValues: true             >-- DoubleValues: true
>-- StringArrayValues: true           >-- IntegerArrayValues: true
```

```
>-- StringValues: true
>-- DoubleArrayValues: true
>-- FloatValues: true
>-- IntegerValues: true
>-- ByteValues: true
> VertexFeatures
>-- AddVertices: true
>-- DuplicateMultiProperties: true
>-- MultiProperties: true
>-- RemoveVertices: true
>-- MetaProperties: true
>-- UserSuppliedIds: true
>-- StringIds: true
>-- RemoveProperty: true
>-- AddProperty: true
>-- NumericIds: true
>-- CustomIds: false
>-- AnyIds: true
>-- UuidIds: true
> EdgeFeatures
>-- RemoveEdges: true
>-- AddEdges: true
>-- UserSuppliedIds: true
>-- StringIds: true
>-- RemoveProperty: true
>-- AddProperty: true
>-- NumericIds: true
>-- CustomIds: false
>-- AnyIds: true
>-- UuidIds: true
>-- LongArrayValues: true
>-- StringArrayValues: true
>-- StringValues: true
>-- DoubleArrayValues: true
>-- FloatValues: true
>-- IntegerValues: true
>-- ByteValues: true
> EdgePropertyFeatures
>-- Properties: true
>-- LongValues: true
>-- SerializableValues: true
>-- FloatArrayValues: true
>-- UniformListValues: true
>-- ByteArrayValues: true
>-- MapValues: true
>-- BooleanArrayValues: true
>-- MixedListValues: true
>-- BooleanValues: true
>-- DoubleValues: true
>-- IntegerArrayValues: true
>-- LongArrayValues: true
>-- StringArrayValues: true
>-- StringValues: true
>-- DoubleArrayValues: true
>-- FloatValues: true
>-- IntegerValues: true
>-- ByteValues: true
```

TinkerGraph is really useful while learning to work with Gremlin and great for testing things out. One common use case where TinkerGraph can be invaluable is to create a sub-graph of a large graph and work with it locally. TinkerGraph can even be used in production deployments if an in-memory graph fits the bill. Typically, TinkerGraph is used to explore static (unchanging) graphs, but you can also use it from a programming language like Java and mutate its contents if you want to. However, TinkerGraph does not support some of the more advanced features you will find in implementations like JanusGraph such as an advanced transaction system, (though it does have basic transactions that are helpful to certain use cases as of 3.7.0) and external indexes. One other thing worth noting in the list above is that the 'UserSuppliedIds' option is set to true for vertex and edge ID values. This means that if you load a graph file, such as a GraphML format file, that specifies ID values for vertices and edges, then TinkerGraph will honor those IDs and use them. As we shall see later, this is not the case with some other graph database systems.

When running in the console, support for TinkerGraph should be on by default. If for any reason you find it to be off, you can enable it by issuing the following command.

```
:plugin use tinkerpop.tinkergraph
```

Once the TinkerGraph plugin is enabled, you will need to close and re-load the Gremlin Console. After doing that, you can create a new TinkerGraph instance from the console as follows:

```
g = TinkerGraph.open().traversal()
```

which is shorthand for

```
graph = TinkerGraph.open()
g = traversal().with(graph)
```

The shorthand is helpful to save a bit of typing, but you lose reference to the graph instance, which might be helpful when accessing 'graph.features()', creating indices, or initiating close operations on the graph itself. The longer form generally tends to be preferable for this reason. You will read more about this in the "Deep dive on traversal terminology" section.

In some cases you will want to pass parameters to the 'open' method providing more information on how the graph is to be configured. We will explore those options later on. The variable called 'g' created above is known as a 'graph traversal source' and will be used throughout the book at the start of each query we write.

Throughout the remainder of this book the variable name 'g' will be used for any object that represents an instance of a graph traversal source object.

2.4. Introducing the air-routes graph

The examples in this book use a sample dataset called the air-routes graph. It models airports as vertices, flight routes as edges, and includes properties such as airport codes, city and country names, geographic coordinates, and distances between airports. This graph is large enough to be interesting but still small enough to explore comfortably from the Gremlin Console.

The `air-routes.graphml` file can be downloaded from the `sample-data` folder located in the GitHub repository at the following URL: https://github.com/krlawrence/graph/tree/main/sample-data

While the air-routes graph was built from actual real-world data, routes are added and deleted by airlines all the time, so please don't use this graph to plan your next vacation or business trip! However, as a learning tool we hope you will find it useful and easy to relate to. If you feel so inclined, you can load the file into a text editor and examine how it is laid out. As you work with graphs, you will want to become familiar with popular graph serialization formats. Two common ones are GraphML and GraphSON. The latter is a JSON format that is defined by Apache TinkerPop and heavily used in that environment. GraphML is widely recognized by TinkerPop and many other tools as well, such as Gephi, a popular open source tool for visualizing graph data. A lot of graph ingestion tools also still use comma-separated values (CSV) format files.

We will briefly look at loading and saving graph data in Sections 2 and 4. We take a look at different ways to work with graph data stored in text format files including importing and exporting graph

data in the "COMMON GRAPH SERIALIZATION FORMATS" section towards the end of the book.

The 'air-routes' graph contains several vertex types that are specified using labels. The most common ones being 'airport' and 'country'. There are also vertices for each of the seven continents ('continent') and a single 'version' vertex that we provided as a way to test which version of the graph you are using.

Routes between airports are modeled as edges. These edges carry the 'route' label and include the distance between the two connected airport vertices as a property called 'dist'. Connections between countries and airports are modeled using an edge with a 'contains' label.

Each airport vertex has many properties associated with it, giving various details about that airport, including its IATA and ICAO codes, its description, the city it is in, and its geographic location.

Specifically, each airport vertex has a unique ID, a label of 'airport' and contains the following properties. The word in parentheses indicates the type of the property.

```
type    (string) : Vertex type. Will be 'airport' for airport vertices
code    (string) : The three letter IATA code like AUS or LHR
icao    (string) : The four letter ICAO code or none. Example KAUS or EGLL
desc    (string) : A text description of the airport
region  (string) : The geographical region like US-TX or GB-ENG
runways (int)    : The number of available runways
longest (int)    : Length of the longest runway in feet
elev    (int)    : Elevation in feet above sea level
country (string) : Two letter ISO country code such as US, FR or DE.
city    (string) : The name of the city the airport is in
lat     (double) : Latitude of the airport
lon     (double) : Longitude of the airport
```

We can use Gremlin once the air route graph is loaded to show us what properties an airport vertex has. As an example, here is what the Austin airport vertex looks like. We will explain the steps that make up the Gremlin query shortly. First, we need to dig a little bit into how to load the data and configure a few preferences.

```
// Query the properties of vertex 3
g.V().has('code','AUS').valueMap(true).unfold()

id=3
label=airport
type=[airport]
code=[AUS]
icao=[KAUS]
desc=[Austin Bergstrom International Airport]
region=[US-TX]
runways=[2]
longest=[12250]
elev=[542]
```

```
country=[US]
city=[Austin]
lat=[30.1944999694824]
lon=[-97.6698989868164]
```

Even though the airport vertex label is 'airport', we chose to also have a property called 'type' that also contains the string 'airport'. This was done to aid with indexing when working with other graph database systems and is explained in more detail later in this book.

You may have noticed that the values for each property are represented as lists (or arrays if you prefer), even though each list only contains one element. The reasons for this will be explored later in this book, but the quick explanation is that this is because TinkerPop allows us to associate a list of values with any vertex property. We will explore ways that you can take advantage of this capability in the "Attaching multiple values (lists or sets) to a single property" section.

The full details of all the features contained in the 'air-routes' graph can be learned by reading the comments at the start of the `air-routes.graphml` file or reading the `README.txt` file.

The graph currently contains a total of 3,619 vertices and 50,148 edges. Of these 3,374 vertices are airports, and 43,400 of the edges represent routes. While in big data terms this is really a tiny graph, it is plenty big enough for us to build up and experiment with some interesting Gremlin queries.

Lastly, here are some statistics and facts about the 'air-routes' graph. If you want to see a lot more statistics check the `README.txt` file that is included with the 'air-routes' graph.

```
Air Routes Graph (v1.0, 2025-Oct-22) contains:
  3,504 airports
  50,637 routes
  237 countries (and dependent areas)
  7 continents
  3,749 total vertices
  57,645 total edges

Additional observations:
  Longest route is between SIN and JFK (9,526 miles)
  Shortest route is between WRY and PPW (2 miles)
  Average route distance is 1,212.918 miles.
  Longest runway is 18,045ft (BPX)
  Shortest runway is 1,300ft (SAB)
  Average number of runways is 1.42123
  Furthest North is LYR (latitude: 78.2461013793945)
  Furthest South is USH (latitude: -54.8433)
  Furthest East is SVU (longitude: 179.341003418)
  Furthest West is TVU (longitude: -179.876998901)
  Closest to the Equator is MDK (latitude: 0.0226000007242)
  Closest to the Greenwich meridian is LDE (longitude: -0.006438999902457)
  Highest elevation is DCY (14,472 feet)
  Lowest elevation is GUW (-72 feet)
  Maximum airport vertex degree (routes in and out) is 620 (FRA)
  Region with the most airports: US-AK (150)
```

```
Country with the most airports: United States (586)
Continent with the most airports: North America (989)
Average degree (airport vertices) is 28.902
Average degree (all vertices) is 28.891
```

Here are the Top 15 airports sorted by overall number of routes (in and out). In graph terminology this is often called the degree of the vertex or just 'vertex degree'.

```
 POS  ID   CODE  TOTAL     DETAILS

  1    52   FRA  (620)  out:310 in:310
  2   161   IST  (618)  out:309 in:309
  3    51   CDG  (587)  out:293 in:294
  4    70   AMS  (568)  out:283 in:285
  5    80   MUC  (541)  out:270 in:271
  6    18   ORD  (529)  out:265 in:264
  7     8   DFW  (506)  out:253 in:253
  8    64   PEK  (497)  out:248 in:249
  9    58   DXB  (496)  out:248 in:248
 10     1   ATL  (484)  out:242 in:242
 11   102   DME  (465)  out:232 in:233
 12    50   LGW  (464)  out:232 in:232
 13    49   LHR  (442)  out:221 in:221
 14    31   DEN  (434)  out:217 in:217
 15    84   MAN  (431)  out:216 in:215
```

Throughout this book you will find Gremlin queries that can be used to generate many of these statistics.

The source code in this section comes from the 'graph-stats.groovy' sample located in: https://github.com/krlawrence/graph/tree/main/sample-code/groovy.

2.4.1. Updated versions of the air route data

Over time the air-routes graph has evolved, and several versions of the data files are included with the book sources. The examples in this book assume that you are using the 1.0 version of the air-routes data. Unless a section explicitly asks you to load a different file, you should use that default version when following along.

Even though no further releases of the dataset are expected, the GitHub repository still maintains a "latest" version. It is also 1.0. You can download it from https://github.com/krlawrence/graph/blob/main/sample-data/air-routes-latest.graphml

2.5. Loading the air-routes graph using the Gremlin Console

The easiest way to load the air-routes graph is to use the prepackaged version of the dataset that comes with TinkerGraph. We can demonstrate this in the Gremlin Console as follows.

```
graph = TinkerFactory.createAirRoutes()
g = traversal().with(graph)
```

That's it! The graph is now loaded with the 1.0 version of the dataset. These two lines simplify a longer set of manual steps which we will go through as an explanatory next step so that you can understand the details.

The following code loads the air-routes graph using the console by putting it into a file and using ':load' to load and run it or by entering each line into the console manually. These commands will set up the console environment, create a TinkerGraph, and load the `air-routes.graphml` file into it. Some extra console features are also enabled.

These commands create an in-memory TinkerGraph which will use LONG values for the vertex, edge, and vertex property IDs. As part of loading a graph, we need to set up a 'graph traversal source' object called 'g' which we will then refer to in our subsequent queries of the graph. We discussed the ':set max-iteration' command in Setting up console preferences.

If you are using a different graph environment and GraphML import is supported, you can still load the `air-routes.graphml` file by following the instructions specific to that system. Once loaded, the queries below should still work either unchanged or with minor modifications.

There is a file called `load-air-routes.groovy`, that contains the commands shown below, available in the `/sample-data` directory. https://github.com/krlawrence/graph/tree/main/sample-data

load-air-routes.groovy

```
conf = new BaseConfiguration()
conf.setProperty("gremlin.tinkergraph.vertexIdManager","LONG")
conf.setProperty("gremlin.tinkergraph.edgeIdManager","LONG")
conf.setProperty("gremlin.tinkergraph.vertexPropertyIdManager","LONG")
graph = TinkerGraph.open(conf)
g = traversal().with(graph)

// Change the path below to point to wherever you put the graphml file
g.io('/mydata/air-routes.graphml').read()
:set max-iteration 1000
```

Setting the ID manager as shown above is important. If you do not do this, by default, when using TinkerGraph, ID values will have to be specified as strings

such as '"3"' rather than just the numeral '3'.

```
:load load-air-routes.groovy
```

As a best practice, you should use the full path to the location where the GraphML file resides if at all possible to make sure that the GraphML reading code can find it.

Once you have the console up and running and have the graph loaded, if you feel like it, you can cut-and-paste queries from this book directly into it to see them run.

Once the 'air-routes' graph is loaded, you can enter the following command, and you will get back information about the graph. In the case of a TinkerGraph you will get back a useful message telling you how many vertices and edges the graph contains. Note that the contents of this message will vary from one graph system to another and should not be relied upon as a way to keep track of vertex and edge counts. We will look at some other ways of counting things a bit later.

```
// Tell me something about my graph
graph.toString()
```

When using TinkerGraph, the message you get back will look something like this.

```
tinkergraph[vertices:3749 edges:57645]
```

2.6. Turning off some of the Gremlin Console's output

Sometimes, especially when assigning a result to a variable and you are not interested in seeing all the steps that Gremlin took to get there, the Gremlin console displays more output than is desirable. An easy way to prevent this is to just add an empty list ";[]" to the end of your query as follows.

```
a=g.V().has('code','AUS').out().toList();[]
```

2.7. A word about indexes and schemas

Before going much further, it is worth briefly mentioning indexes and schemas. For now, you will be working with TinkerGraph, which does not enforce a formal schema and offers only simple in-memory indexing. That is sufficient for the examples in this chapter, but later chapters will return to these topics in more detail and show how indexes and schemas can have a big impact on query performance and data quality.

As most of the examples in this book are intended to work just fine with only a basic TinkerGraph, the subject of indexes is not covered in detail until Chapter 6 "MOVING BEYOND THE GREMLIN CONSOLE" . However, as TinkerGraph does have some indexing capability, we have also included

some discussion of it in the "Introducing TinkerGraph indexes" section. You should always refer to the specific documentation for the graph system you are using to decide what you need to do about creating an index and schema for your graph.

In general, for any graph database, regardless of whether it is optional or not, use of an index should be considered the best practice.

When working with TinkerGraph, there is no need to define a schema ahead of time. The types of each property are derived at creation time. This is a really convenient feature and allows us to get productive and do some experimenting really quickly.

In production systems, especially those where the graphs are large, the task of creating and managing indexes may include the use of additional software components, such as Apache Solr or Elasticsearch.

Chapter 3. WRITING GREMLIN QUERIES

Now that you hopefully have the 'air-routes' graph loaded, it's time to start writing some queries!

Chapter 3 is focused on queries that are simply read from an existing graph. If you are more interested in adding new vertices, edges and properties or modifying existing properties, you may want to jump to Chapter 4 and in particular the "Adding vertices, edges, and properties" section.

In this chapter we will begin to look at the Gremlin query language. We will start off with a quick look at how Gremlin and SQL differ and are yet in some ways similar, then present some fairly basic queries and finally get into some more advanced concepts. Hopefully, each set of examples presented, building upon things previously discussed, will be easy to understand.

3.1. Introducing Gremlin

Gremlin is the name of the graph traversal and query language that TinkerPop provides for working with property graphs. Gremlin can be used with any graph store that is Apache TinkerPop enabled. Gremlin is a fairly imperative language but has some more declarative constructs as well. Using Gremlin, we can traverse a graph looking for values, patterns and relationships, we can add or delete vertices and edges, we can create sub-graphs, and lots more.

3.1.1. A quick look at Gremlin and SQL

While it is not required to know SQL to be productive with Gremlin, if you do have some experience with SQL, you will notice many of the same keywords and phrases being used in Gremlin. As a simple example, the SQL and Gremlin examples below both show how we might count the number of airports there are in each country using firstly a relational database and secondly a property graph.

When working with a relational database, we might decide to store all the airport data in a single table called 'airports'. In a basic case (the air-routes graph actually stores a lot more data than this about each airport) we could set up our airports table so that it had entries for each airport as follows.

```
ID   CODE  ICAO  CITY            COUNTRY
---  ----  ----  --------------  ----------
1    ATL   KATL  Atlanta         US
3    AUS   KAUS  Austin          US
8    DFW   KDFW  Dallas          US
47   YYZ   CYYZ  Toronto         CA
49   LHR   EGLL  London          UK
51   CDG   LFPG  Paris           FR
52   FRA   EDDF  Frankfurt       DE
55   SYD   YSSY  Sydney          AU
```

We could then use a SQL query to count the distribution of airports in each country as follows.

```
SELECT country,count(country) FROM airports GROUP BY country;
```

We can do this in Gremlin using the 'air-routes' graph with a query like the one below (We will explain what all of this means later on in the book).

```
g.V().hasLabel('airport').groupCount().by('country')
```

You will discover that Gremlin provides its own flavor of several constructs that you will be familiar with if you have used SQL before, but again, prior knowledge of SQL is in no way required to learn Gremlin.

One thing you will not find when working with a graph using Gremlin is the concept of a SQL 'join'. Graph databases by their very nature avoid the need to join things together (as things that need to be connected already are connected), and this is a core reason why, for many use cases, graph databases are a great choice and can be more performant than relational databases.

Graph databases are usually a good choice for storing and modeling networks. The 'air-routes' graph is an example of a network graph. A social network is, of course, another good example. Networks can be modeled using relational databases too, but as you explore the network and ask questions like "who are my friends' friends?" in a social network or "where can I fly to from here with a maximum of two stops?" things rapidly get complicated and result in the need for multiple 'joins'.

As an example, imagine adding a second table to our relational database called routes. It will contain three columns representing the source airport, the destination airport and the distance between them in miles (SRC, DEST and DIST). It would contain entries that looked like this (the real table would, of course, have thousands of rows, but this gives a good idea of what the table would look like).

```
SRC  DEST  DIST
---  ----  ----
ATL  DFW   729
ATL  FRA   4600
AUS  DFW   190
AUS  LHR   4901
BOM  AGR   644
BOM  LHR   4479
CDG  DFW   4933
CDG  FRA   278
CDG  LHR   216
DFW  FRA   5127
DFW  LHR   4736
LHR  BOM   4479
LHR  FRA   406
YYZ  FRA   3938
YYZ  LHR   3544
```

If we wanted to write a SQL query to calculate the ways of traveling from Austin (AUS) to Agra (AGR) with two stops, we would end up writing a query that looked something like this:

```
SELECT a1.code,r1.dest,r2.dest,r3.dest FROM airports a1
  JOIN routes r1 ON a1.code=r1.src
  JOIN routes r2 ON r1.dest=r2.src
  JOIN routes r3 ON r2.dest=r3.src
  WHERE a1.code='AUS' AND r3.dest='AGR';
```

Using our 'air-routes' graph database, the query can be expressed quite simply as follows:

```
g.V().has('code','AUS').out().out().out().has('code','AGR').path().by('code')
```

Adding or removing hops is as simple as adding or removing one or more of the 'out' steps, which is a lot simpler than having to add additional 'join' clauses to our SQL query. This is a simple example, but as queries get more and more complicated in heavily connected data sets like networks, the SQL queries get harder and harder to write whereas, because Gremlin is designed for working with this type of data, expressing a traversal remains fairly straightforward.

We can go one step further with Gremlin and use 'repeat' to express the concept of 'three times' as follows.

```
g.V().has('code','AUS').repeat(out()).times(3).has('code','AGR').path().by('code')
```

Gremlin also has a 'repeat ... until' construct that we will see used later in this book. When combined with the 'emit' step, 'repeat' provides a nice way of getting back any routes between a source and destination no matter how many hops it might take to get there.

Again, don't worry if some of the Gremlin steps shown here are confusing, we will cover them all in detail a bit later. The key point to take away from this discussion of SQL and Gremlin is that for data that is very connected, graph databases provide a great way to store that data, and Gremlin provides a nice and fairly intuitive way to traverse that data efficiently.

One other point worthy of note is that every vertex and every edge in a graph has a unique ID. Unlike in the relational world where you may or may not decide to give a table an ID column, this is not optional with graph databases. In some cases the ID can be a user-provided ID, but more commonly it will be generated by the graph system when a vertex or edge is first created. If you are familiar with SQL, you can think of the ID as a primary key of sorts if you want to. Every vertex and edge can be accessed using its ID. Just as with relational databases, graph databases can be indexed, and any of the properties contained in a vertex or an edge can be added to the index and can be used to find things efficiently. In large graph deployments this greatly speeds up the process of finding things as you would expect. We look more closely at IDs in the "Working with IDs" section.

3.2. Some fairly basic Gremlin queries

A graph 'query' is often referred to as a 'traversal' as that is what we are in fact doing. We are

traversing the graph from a starting point to an ending point. Traversals consist of one or more 'steps' (essentially methods) that are chained together.

As we start to look at some simple traversals, here are a few 'steps' that you will see used a lot. Firstly, you will notice that almost all traversals start with either a 'g.V()' or a 'g.E()'. Sometimes there will be parameters specified along with those steps, but we will get into that a little later. You may remember from when we looked at how to load the 'air-routes' graph in Section 2, we used the following instruction to create a graph traversal source object for our loaded 'graph'.

```
g = traversal().with(graph)
```

Once we have a graph traversal source object, we can use it to start exploring the graph. The 'V' step returns vertices and the 'E' step returns edges. You can also use a 'V' step in the middle of a traversal as well as at the start, but we will examine those uses a little later. The 'V' and 'E' steps can also take parameters indicating which set of vertices or edges we are interested in. That usage is explained in the "Working with IDs" section.

If it helps with remembering, you can think of 'g.V()' as meaning "looking at all the vertices in the graph" and 'g.E()' as meaning "looking at all the edges in the graph". We then add additional steps to narrow down our search criteria.

The other steps we need to introduce are the 'has' and 'hasLabel' steps. They can be used to test for a certain label or property having a certain value. We will encounter a lot of different Gremlin steps as we explore various Gremlin queries throughout the book, including many other forms of the 'has' step, but these few examples are enough to get us started.

You can refer to the official Apache TinkerPop documentation for full details on all the graph traversal steps that are used in this tutorial. With this tutorial we have not tried to teach every possible usage of every Gremlin step and method, rather, We have tried to provide a good and approachable foundation in writing many different types of Gremlin queries using an interesting and real-world graph.

The latest TinkerPop documentation is always available at this URL: https://tinkerpop.apache.org/docs/current/reference/

Below are some simple queries against the 'air-routes' graph to get us started. It is assumed that the 'air-routes' graph has been loaded already per the instructions above. The query below will return any vertices that have the 'airport' label.

```
// Find vertices that are airports
g.V().hasLabel('airport')
```

This query will return the vertex that represents the Dallas Fort Worth (DFW) airport.

```
// Find the DFW vertex
```

```
g.V().has('code','DFW')
```

The next two queries combine the previous two into a single query. The first one just chains the queries together. The second shows a form of the 'has' step that we have not looked at before that takes an additional label value as its first parameter.

```
// Combining those two previous queries (two ways that are equivalent)
g.V().hasLabel('airport').has('code','DFW')

g.V().has('airport','code','DFW')
```

Here is what we get back from the query. Notice that this is the Gremlin Console's way of telling us we got back the 'Vertex' with an ID of 8.

```
v[8]
```

So, what we actually got back from these queries was a TinkerPop 'Vertex' data structure. Later in this book we will look at ways to store that value into a variable for additional processing. Remember that even though we are working with a Groovy environment while inside the console, everything we are working with here, at its core, is Java code. So we can use the 'getClass' method from Java to introspect the object. Note the call to 'next' which turns the result of the traversal into an object we can work with further.

```
g.V().has('airport','code','DFW').next().getClass()

class org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerVertex
```

The 'next' step that we used above is one of a series of steps that the TinkerPop documentation describes as 'terminal steps'. We will see more of these 'terminal steps' in use throughout this book. As mentioned above, a terminal step essentially ends the graph traversal and returns a concrete object that you can work with further in your application. We will see 'next' and other related steps used in this way when we start to look at using Gremlin from a standalone program a bit later on. We could even add a call to 'getMethods()' at the end of the query above to get back a list of all the methods and their types supported by the 'TinkerVertex' class. If you'd like to jump ahead to learn more about terminal steps, you can see more examples in the "Assigning query results to a variable with a terminal step" section.

3.2.1. Retrieving property values from a vertex

There are several different ways of working with vertex properties. We can add, delete and query properties for any vertex or edge in the graph. We will explore each of these topics in detail over the course of this book. Initially, let's look at a couple of simple ways that we can look up the property values of a given vertex.

```
// What property values are stored in the DFW vertex?
```

```
g.V().has('airport','code','DFW').values()
```

Here is the output that the query returns. Note that we just get back the values of the properties when using the 'values' step, we do not get back the associated keys. We will see how to do that later in the book.

```
US
DFW
13401
Dallas
607
KDFW
-97.0380020141602
airport
US-TX
7
32.896800994873
Dallas/Fort Worth International Airport
```

The 'values' step can take parameters that tell it to only return the values for the provided key names. The queries below return the values of some specific properties.

```
// Return just the city name property
g.V().has('airport','code','DFW').values('city')

Dallas

// Return the 'runways' and 'icao' property values.
g.V().has('airport','code','DFW').values('runways','icao')

KDFW
7
```

3.2.2. Does a specific property exist on a given vertex or edge?

You can test to see if a property exists as well as testing for it containing a specific value. To do this, we can just provide 'has' with the name of the property we are interested in. This works equally well for both vertex and edge properties.

```
// Find all edges that have a 'dist' property
g.E().has('dist')

// Find all vertices that have a 'region' property
g.V().has('region')

// Find all the vertices that do not have a 'region' property
g.V().hasNot('region')
```

```
// The above is shorthand for
g.V().not(has('region'))
```

3.2.3. Counting things

A common need when working with graphs is to be able to count how "many of something" there are in the graph. We will look in the next section at other ways to count groups of things, but first, let's look at some examples of using the 'count' step to count various things in our 'air-routes' graph. First, let's find out how many vertices in the graph represent airports.

```
// How many airports are there in the graph?
g.V().hasLabel('airport').count()

3504
```

Now, looking at edges that have a 'route' label, let's find out how many flight routes are stored in the graph. Note that the 'outE' step looks at outgoing edges. In this case we could also have used the 'out' step instead. The various ways that you can look at outgoing and incoming edges are discussed in the "Starting to walk the graph" section that is coming up soon.

```
// How many routes are there?
g.V().hasLabel('airport').outE('route').count()

50637
```

You could shorten the above a little as follows but this would cause more edges to get looked at as we do not first filter out all vertices that are not airports.

```
// How many routes are there?
g.V().outE('route').count()

50637
```

You could also do it this way but generally starting by looking at all the Edges in the graph is considered bad form as property graphs tend to have a lot more edges than vertices.

```
// How many routes are there?
g.E().hasLabel('route').count()

50637
```

We have not yet looked at the 'outE' step used above. We will look at it very soon however in the "Starting to walk the graph" section.

3.2.4. Counting groups of things

Sometimes it is useful to count how many of each type (or group) of things there are in the graph. This can be done using the 'group' and 'groupCount' steps. While for a very large graph it is not recommended to run queries that look at all vertices or all of the edges in a graph, for smaller graphs this can be quite useful. For the air-routes graph we could easily count the number of different vertex and edge types in the graph as follows.

```
// How many of each type of vertex are there?
g.V().groupCount().by(label)
```

If we were to run the query, we would get back a map where the keys are label names and the values are the counts for the occurrence of each label in the graph.

```
[continent:7,country:237,version:1,airport:3504]
```

There are other ways we could write the query above that will yield the same result. One such example is shown below.

```
// How many of each type of vertex are there?
g.V().label().groupCount()

[continent:7,country:237,version:1,airport:3504]
```

We can also run a similar query to find out the distribution of edge labels in the graph. An example of the type of result we would get back is also shown.

```
// How many of each type of edge are there?
g.E().groupCount().by(label)

[contains:7008,route:50637]
```

As before, we could rewrite the query as follows.

```
// How many of each type of edge are there?
g.E().label().groupCount()

[contains:7008,route:50637]
```

By way of a side note, the examples above are shorthand ways of writing something like this example which also counts vertices by label.

```
// As above but using group()
g.V().group().by(label).by(count())
```

```
[continent:7,country:237,version:1,airport:3504]
```

There are often a number of different ways to write Gremlin that will obtain the same result. See the A word about idiomatic Gremlin Section for more details.

We can be more selective in how we specify the groups of things that we want to count. In the examples below we first count how many airports there are in each country. This will return a map of key:value pairs where the key is the country code and the value is the number of airports in that country. As the fourth and fifth examples show, we can use 'select' to pick just a few values from the whole group that got counted. Of course, if we only wanted a single value, we could just count the airports connected to that country directly, but the last two examples are intended to show that you can count a group of things and still selectively only look at part of that group.

```
// How many airports are there in each country?
g.V().hasLabel('airport').groupCount().by('country')

// How many airports are there in each country? (look at country first)
g.V().hasLabel('country').group().by('code').by(out().count())
```

We can easily find out how many airports there are in each continent using 'group' to build a map of continent codes and the number of airports in that continent. The output from running the query is shown below also.

```
// How many airports are there in each continent?
g.V().hasLabel('continent').group().by('code').by(out().count())

[EU:605,AS:971,NA:989,OC:305,AF:321,AN:0,SA:313]
```

These queries show how 'select' can be used to extract specific values from the map that we have created. Again, you can see the results we get from running the query.

```
// How many airports are there in France (having first counted all countries)
g.V().hasLabel('airport').groupCount().by('country').select('FR')

59

// How many airports are there in France, Greece and Belgium respectively?
g.V().hasLabel('airport').groupCount().by('country').select('FR','GR','BE')

[FR:59,GR:39,BE:5]
```

The 'group' and 'groupCount' steps are invaluable when you want to count groups of things or collect things into a group using a selection criteria. You will find a lot more examples of grouping and counting things in the section called "Counting more things".

3.3. Starting to walk the graph

So far we have mostly just explored queries that look at properties on a vertex or count how many things we can find of a certain type. Where the power of a graph really comes into play is when we start to 'walk' or 'traverse' the graph by looking at the connections (edges) between vertices. The term 'walking the graph' is used to describe moving from one vertex to another vertex via an edge. Typically, when using the phrase 'walking a graph', the intent is to describe starting at a vertex traversing one or more vertices and edges and ending up at a different vertex or sometimes, back where you started in the case of a 'circular walk'. It is straightforward to traverse a graph in this way using Gremlin. The journey we took while on our 'walk' is often referred to as our 'path'. There are also cases when all you want to do is return edges or some combination of vertices and edges as the result of a query, and Gremlin allows this as well. We will explore a lot of ways to modify the way a graph is traversed in the upcoming sections.

The table below gives a brief summary of all the steps that can be used to 'walk' or 'traverse' a graph using Gremlin. You will find all of these steps used in various ways throughout the book. Think of a graph traversal as moving through the graph from one place to one or more other places. These steps tell Gremlin which places to move to next as it traverses a graph for you.

To better understand these steps, it is worth defining some terminology. One vertex is considered to be 'adjacent' to another vertex if there is an edge connecting them. A vertex and an edge are considered 'incident' if they are connected to each other.

Table 1. Where to move next while traversing a graph

out *	Outgoing adjacent vertices.
in *	Incoming adjacent vertices.
both *	Both incoming and outgoing adjacent vertices.
outE *	Outgoing incident edges.
inE *	Incoming incident edges.
bothE *	Both outgoing and incoming incident edges.
outV	Outgoing vertex.
inV	Incoming vertex.
otherV	The vertex that was not the vertex we came from.

Note that the steps labeled with a '*' can optionally take the name of one or more edge labels as a parameter. If omitted, all relevant edges will be traversed.

3.3.1. Some simple graph traversal examples

To get us started, in this section we will look at some simple graph traversal examples that use some of the steps that were just introduced. The 'out' step is used to find vertices connected by an outgoing edge to that vertex, and the 'outE' 'step' is used when you want to examine the outgoing edges from a given vertex. Conversely, the 'in' and 'inE' steps can be used to look for incoming vertices and edges. The 'outE' and 'inE' steps are especially useful when you want to look at the properties of an edge as we shall see in the "Examining the edge between two vertices" section.

There are several other steps that we can use when traversing a graph to move between vertices and edges. These include 'bothE', 'bothV' and 'otherV'. We will encounter those in the "Other ways to explore vertices and edges using 'both', 'bothE', 'bothV' and 'otherV'" section.

So let's use a few examples to help better understand these graph traversal steps. The first query below does a few interesting things. Firstly, we find the vertex representing the Austin airport (the airport with a property of 'code' containing the value 'AUS'). Having found that vertex, we then go 'out' from there. This will find all the vertices connected to Austin by an outgoing edge. Having found those airports, we then ask for the values of their 'code' properties using the 'values' step. Finally, the 'fold' step puts all the results into a list for us. This just makes it easier for us to inspect the results in Gremlin Console.

```
// Where can I fly to from Austin?
g.V().has('airport','code','AUS').out().values('code').fold()
```

Here is what you might get back if you were to run this query in your console.

```
[PHL,PDX,DTW,OKC,ONT,CLT,CUN,MEM,CVG,IND,MCI,DAL,STL,ABQ,MKE,MDW,OMA,TUL,PVR,NAS,LIT,
 JAX,PVD,SMF,BHM,SDF,BUF,BOI,LBB,ECP,HRL,RNO,CMH,DSM,CZM,AMA,BTR,CHS,GDL,GRR,LIR,PNS,
 SJD,TYS,VPS,XNA,HDN,SFB,BUR,ASE,BZN,BKG,PIE,ATL,BNA,BOS,BWI,DCA,DFW,FLL,IAD,IAH,JFK,
 LAX,MCO,MIA,MSP,ORD,PHX,RDU,SEA,SFO,SJC,TPA,SAN,LGB,SNA,SLC,LAS,DEN,SAT,YYZ,HNL,YVR,
 MSY,LHR,EWR,LGW,HOU,FRA,ELP,AMS,CLE,YYC,OAK,MEX,TUS,PIT]
```

All edges in a graph have a label. However, one thing we did not do in the previous query was specify a label for the 'out' step. If you do not specify a label, you will get back any connected vertex regardless of its edge label. In this case it does not cause us a problem as airports only have one type of outgoing edge, labeled 'route'. However, in many cases, in graphs you create or are working with, your vertices may be connected to other vertices by edges with differing labels, so it is good practice to get into the habit of specifying edge labels as part of your Gremlin queries. So we could change our query just a bit by adding a label reference on the 'out' step as follows.

```
// Where can I fly to from Austin?
g.V().has('airport','code','AUS').out('route').values('code').fold()
```

Despite having just stated that consistently using edge labels in queries is a good idea, unless you truly do want to get back all edges or all connected vertices, We will break our own rule quite a bit in this book. The reason for this is purely to save space and shorten the queries we present.

Here are a few more simple queries similar to the previous one. The first example can be used to answer the question, "Where can I fly to from Austin, with one stop on the way?". Note that, as written, coming back to Austin will be included in the results as this query does not rule it out!

```
// Where can I fly to from Austin, with one stop on the way?
g.V().has('airport','code','AUS').out('route').out('route').values('code')
```

This query uses an 'in' step to find all the routes that come into the London City Airport (LCY) and returns their IATA codes.

```
// What routes come in to LCY?
g.V().has('airport','code','LCY').in('route').values('code')
```

This query is perhaps a bit more interesting. It finds all the routes from London Heathrow airport in England that go to an airport in the United States and returns their IATA codes.

```
// Flights from London Heathrow (LHR) to airports in the USA
g.V().has('code','LHR').out('route').has('country','US').values('code')
```

3.3.2. What vertices and edges did I visit? — Introducing 'path'

A Gremlin method (often called a step) that you will see used a lot in this book is 'path'. After you have done some graph walking using a query, you can use 'path' to get a summary back of where you went. A simple example of a 'path' step being used is shown below. Throughout the book you will see numerous examples of 'path' being used including in conjunction with one or more 'by' steps to specify how the path result should be formatted.

This particular query will return the vertices and outgoing edges starting at the London City (LCY) airport vertex. You can read this query like this: "Start at the LCY vertex, find all outgoing edges and also find all the vertices that are on the other ends of those edges". The 'inV' step gives us the vertex at the other end of the outgoing edge.

```
// This time, for each route, return both vertices and the edge that connects them.
g.V().has('airport','code','LCY').outE().inV().path()
```

If you run that query as-is, you will get back a series of results that look like this. This shows that there is a route from vertex 88 to vertex 77 via an edge with an ID of 15142.

```
[v[88],e[15142][88-route->77],v[77]]
```

While this result is useful, we might want to return something more human-readable such as the IATA codes for each airport and perhaps the distance property from the edge that tells us how far apart the airports are. We could add some 'by' modulators to our query to do this. The Apache TinkerPop documentation uses the phrase 'modulator' to describe steps that are not really independent steps but instead alter the behavior of the steps that they are associated with.

A 'modulator' is a step that influences the behavior of the step that it is associated with. Examples of such modulator steps are 'by' and 'as'.

Take a look at the modified form of the query shown below and an example of the results that it will now return. If this is not fully clear yet, don't panic. Both 'path' and 'by' are used a lot

throughout this book.

```
g.V().has('airport','code','LCY').outE().inV().
      path().by('code').by('dist')
```

When you run this modified version of the query, you will receive a set of results that look like the following line.

```
[LCY,456,GVA]
```

The 'by' modulator steps are processed in a round-robin fashion. If there are not enough modulators specified for the total number of elements in the path, Gremlin just loops back around to the first 'by' step and so on. So even though there were three elements in the path that we wanted to have formatted, we only needed to specify two 'by' modulators. This is because the first and third elements in the path are of the same type, namely airport vertices, and we wanted to use the same property name, 'code', in each of those cases. If we instead wanted to reference a different property name for each element of the path result, we would need to specify three explicit 'by' modulator steps. This would be required if, for example, we wanted to reference the 'city' property of the third element in the path rather than its 'code'.

The 'by' modulator steps are processed in a round-robin fashion in cases where there are more results to apply them to than the number of 'by' modulators specified.

The example above is equivalent to this longer form of the same query.

```
g.V().has('airport','code','LCY').outE().inV().
      path().by('code').by('dist').by('code')
```

The example below shows a case where three different 'by' modulators are used. This time the third 'by' modulator step references the 'city' property rather than the airport 'code'. As you can see from the sample output, this time the city name 'Geneva' appears rather than the airport code 'GVA'.

```
g.V().has('airport','code','LCY').outE().inV().
      path().by('code').by('dist').by('city')

[LCY,456,Geneva]
```

Sometimes it is necessary to use a 'by' modulator that has no parameter as shown below. This is because the element in the path is not a vertex or edge containing multiple properties but rather a single value, in this case, an integer.

```
g.V().has('airport','code','LCY').out().limit(5).
      values('runways').
```

```
      path().by('code').by('code').by()
```

The results show the codes for the airports we visited along with a number representing the number of runways the second airport has.

```
[LCY,CDG,4]
[LCY,FRA,4]
[LCY,DUB,2]
[LCY,FCO,3]
[LCY,AMS,6]
```

It is also possible to use a traversal inside a 'by' modulator. Such traversals are known as '"anonymous traversals"', and they are discussed in greater details in the "Deep dive on traversal terminology" section.

For now, it is enough to know that they allow us to do things like combine multiple values as part of a path result. The example below finds five routes that start in Austin and creates a path result containing the airport code and city name for both the source and destination airports. In this case, the anonymous traversal contained within the 'by' modulator is applied to each element in the path.

```
g.V(3).out().limit(5).path().by(values('code','city').fold())

[[AUS,Austin],[PHL,Philadelphia]]
[[AUS,Austin],[PDX,Portland]]
[[AUS,Austin],[DTW,Detroit]]
[[AUS,Austin],[OKC,Oaklahoma City]]
[[AUS,Austin],[ONT,Ontario]]
```

To demonstrate that just about any arbitrary traversal can be placed inside the 'by' modulator, here is one more example that counts the number of outgoing routes for the source and destination airports as part of generating the 'path' result.

```
g.V(3).out().limit(5).path().by(outE().count())

[98,147]
[98,80]
[98,145]
[98,33]
[98,23]
```

3.3.3. Modifying a 'path' using 'from' and 'to' modulators

The 'from' and 'to' modulators for the 'path' step enables us to not return the entire path of a traversal but instead to be more selective.

First, look at the example below. In this case we have just used the same 'path' constructs used in the prior examples. The query returns the first 10 routes found starting at Austin (AUS) with one stop on the way.

```
g.V().has('airport','code','AUS').out().out().path().by('code').limit(10)
```

As expected, the results show each airport that was visited.

```
[AUS,PHL,SXM]
[AUS,PHL,RIC]
[AUS,PHL,LEX]
[AUS,PHL,ISP]
[AUS,PHL,SWF]
[AUS,PHL,DSM]
[AUS,PHL,MYR]
[AUS,PHL,CAE]
[AUS,PHL,CHA]
[AUS,PHL,CHS]
```

Given that every journey starts in Austin, we might not want the AUS airport code to be part of the returned results. We might just want to capture the places that we ended up visiting after leaving Austin. This can be achieved by labeling the parts of the traversal that we care about using 'as' steps and then using 'from' and 'to' modulators to tell the 'path' step what we are interested in. Take a look at the modified version of the query below.

```
g.V().has('airport','code','AUS').out().as('a').out().as('b').
      path().by('code').from('a').to('b').limit(10)
```

This time AUS is not included in the 'path' results.

```
[PHL,SXM]
[PHL,RIC]
[PHL,LEX]
[PHL,ISP]
[PHL,SWF]
[PHL,DSM]
[PHL,MYR]
[PHL,CAE]
[PHL,CHA]
[PHL,CHS]
```

Because after skipping the AUS part of the path, we did in fact want the rest of the results, we could have left off the 'to' modulator and written the query as follows.

```
g.V().has('airport','code','AUS').out().as('a').out().
```

```
      path().by('code').from('a').limit(10)
```

As you can see, the results are the same as before.

```
[PHL,SXM]
[PHL,RIC]
[PHL,LEX]
[PHL,ISP]
[PHL,SWF]
[PHL,DSM]
[PHL,MYR]
[PHL,CAE]
[PHL,CHA]
[PHL,CHS]
```

There are a lot of ways that 'from' and 'to' can be used. By way of one final example, let's create a version of the query with three 'out' steps. Note that a bit later we will see how 'repeat' can be used when the same steps need to be used repeatedly like this, but that is not important to this specific example.

```
g.V().has('airport','code','AUS').out().out().out().
      path().by('code').limit(10)
```

As expected, we now have an additional stop added to each of the journeys.

```
[AUS,PHL,SXM,NEV]
[AUS,PHL,SXM,AXA]
[AUS,PHL,SXM,EIS]
[AUS,PHL,SXM,EUX]
[AUS,PHL,SXM,SAB]
[AUS,PHL,SXM,ATL]
[AUS,PHL,SXM,BOS]
[AUS,PHL,SXM,FLL]
[AUS,PHL,SXM,IAD]
[AUS,PHL,SXM,JFK]
```

Let's now modify the query to limit which parts of the path are returned.

```
g.V().has('airport','code','AUS').out().as('a').out().as('b').out().
      path().by('code').from('a').to('b').limit(10)
```

As you can see, only the parts of the journey that we selected have been returned.

```
[PHL,SXM]
[PHL,SXM]
```

```
[PHL,SXM]
[PHL,SXM]
[PHL,SXM]
[PHL,SXM]
[PHL,SXM]
[PHL,SXM]
[PHL,SXM]
[PHL,SXM]
```

We could also have written the query as shown below to only show the results of each path up to a certain point.

```
g.V().has('airport','code','AUS').out().out().as('b').out().
      path().by('code').to('b').limit(10)
```

This time only the first three airports visited are included in each result.

```
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
[AUS,PHL,SXM]
```

By way of a side note, in cases like this where more than one of the results is identical, you may want to remove the duplicates. That is where the 'dedup' step is useful. You will find coverage of 'dedup' in the "Removing duplicates - introducing 'dedup'" section. However, as a little taste test, let's add a 'dedup' step to the end of our previous query and see what happens.

```
g.V().has('airport','code','AUS').out().out().as('b').out().
      path().by('code').to('b').limit(10).dedup()

[AUS,PHL,SXM]
```

As you can see, all the duplicate results have now been removed. Hopefully, this gives you a good basic understanding of the 'path' step. You will see it used a lot throughout the remainder of this book. However, there are a few things to be aware of when using 'path'. Those concerns are explained in the "A warning that path finding can be memory and CPU intensive" section a bit later.

Does an edge exist between two vertices? You can use the 'hasNext' step to check if an edge exists between two vertices and get a Boolean (true or false) value back. The first query below will return **true** because there is an edge (a route) between AUS and DFW. The second query will return **false**

because there is no route between AUS and SYD.

```
g.V().has('code','AUS').out('route').has('code','DFW').hasNext()

true

g.V().has('code','AUS').out('route').has('code','SYD').hasNext()

false
```

3.3.4. Using 'as', 'select' and 'project' to refer to traversal steps

Sometimes it is useful to be able to remember a point of a traversal by giving it a name (label) and refer to it later on in the same query. The query below uses an 'as' step to attach a label at two different parts of the traversal, each representing different vertices that were found. A 'select' step is later used to refer back to them.

```
g.V().has('code','DFW').as('from').out().
      has('region','US-CA').as('to').
      select('from','to')
```

This query, while a bit contrived, and in this case probably a poor substitute for using 'path', returns the following results.

```
[from:v[8],to:v[151]]
[from:v[8],to:v[181]]
[from:v[8],to:v[244]]
[from:v[8],to:v[384]]
[from:v[8],to:v[605]]
[from:v[8],to:v[865]]
[from:v[8],to:v[872]]
[from:v[8],to:v[877]]
[from:v[8],to:v[13]]
[from:v[8],to:v[23]]
[from:v[8],to:v[24]]
[from:v[8],to:v[26]]
[from:v[8],to:v[28]]
[from:v[8],to:v[42]]
```

In the example above, only the vertices themselves were selected. We can also use a 'by' modulator to specify which property to retrieve from the selected vertices.

```
g.V().has('code','DFW').as('from').out().
      has('region','US-CA').as('to').
      select('from','to').by('code')
```

This time the results contain the airport codes.

```
[from:DFW,to:ONT]
[from:DFW,to:PSP]
[from:DFW,to:SMF]
[from:DFW,to:FAT]
[from:DFW,to:BUR]
[from:DFW,to:BFL]
[from:DFW,to:MRY]
[from:DFW,to:SBA]
[from:DFW,to:LAX]
[from:DFW,to:SFO]
[from:DFW,to:SJC]
[from:DFW,to:SAN]
[from:DFW,to:SNA]
[from:DFW,to:OAK]
```

While the prior example was perhaps not ideal, it does show how 'as' and 'select' work. For completeness, here is the same query but using 'path'. You will see both the 'select' and 'path' steps used a lot throughout this book.

```
g.V().has('code','DFW').out().
      has('region','US-CA').
      path().by('code')
```

Which would produce the following results. Notice that this time the results do not have labels associated with them but are otherwise the same.

```
[DFW,ONT]
[DFW,PSP]
[DFW,SMF]
[DFW,FAT]
[DFW,BUR]
[DFW,BFL]
[DFW,MRY]
[DFW,SBA]
[DFW,LAX]
[DFW,SFO]
[DFW,SJC]
[DFW,SAN]
[DFW,SNA]
[DFW,OAK]
```

While the 'path' step is a lot more convenient, in some cases it can be costly in terms of memory and CPU usage, so it is worth remembering these alternative techniques using 'as' and 'select'. That topic is discussed in more detail in the "A warning that path finding can be memory and CPU intensive" section.

You can also give a point of a traversal multiple names and refer to each later on in the traversal/query as shown below.

```
g.V().has('type','airport').limit(10).as('a','b','c').
      select('a','b','c').
        by('code').by('region').by(out().count())
```

The 'project' step can achieve the same results as obtained from the combination of 'as' and 'select' steps. The example below shows the previous query, rewritten to use 'project' instead of 'as' and 'select'.

```
g.V().has('type','airport').limit(10).
      project('a','b','c').
        by('code').by('region').by(out().count())
```

This query, and the prior query, would return the following results.

```
[a:ATL,b:US-GA,c:242]
[a:ANC,b:US-AK,c:41]
[a:AUS,b:US-TX,c:98]
[a:BNA,b:US-TN,c:75]
[a:BOS,b:US-MA,c:143]
[a:BWI,b:US-MD,c:91]
[a:DCA,b:US-DC,c:96]
[a:DFW,b:US-TX,c:253]
[a:FLL,b:US-FL,c:158]
[a:IAD,b:US-VA,c:158]
```

In the prior example we gave our variables simple names like 'a' and 'b'. However, it is sometimes useful to give our traversal variables and named steps more meaningful names, and it is perfectly OK to do that. Let's rewrite the query to use some more descriptive variable names.

```
g.V().has('type','airport').limit(10).
      project('IATA','Region','Routes').
        by('code').by('region').by(out().count())
```

When we run the modified query, here is the output we get.

```
[IATA:ATL,Region:US-GA,Routes:242]
[IATA:ANC,Region:US-AK,Routes:41]
[IATA:AUS,Region:US-TX,Routes:98]
[IATA:BNA,Region:US-TN,Routes:75]
[IATA:BOS,Region:US-MA,Routes:143]
[IATA:BWI,Region:US-MD,Routes:91]
[IATA:DCA,Region:US-DC,Routes:96]
```

```
[IATA:DFW,Region:US-TX,Routes:253]
[IATA:FLL,Region:US-FL,Routes:158]
[IATA:IAD,Region:US-VA,Routes:158]
```

The 'project' step can be applied to inputs other than graph elements. It can actually operate on any incoming traverser. For example, when 'project' follows 'valueMap' the incoming traverser is a 'Map' object.

```
g.V().has('code','IAD').valueMap().
  project('r','ct').by('runways').by(count(local))

[r:[4],ct:12]
```

The prior example shows how 'project' can access the "runways" key and count the number of entries in the 'Map'.

3.3.5. Traits of 'by' modulators

We've seen enough use of the 'by' modulator now to dive deeper on their general behavior. As we learned earlier, a 'by' modulator is a form of step that influences the behavior of the step that it is associated with. Moreover, The 'by' modulators are processed in a round-robin fashion in cases where there are more results to apply them to than the number of 'by' modulators specified.

In addition to those basic definitions, there are some other points worth exploring. First, let's take a look at the list of steps that support 'by' modulation, as 'by' cannot be used on all steps:

'aggregate'	'cyclicPath'	'dedup'	'group'
'groupCount'	'math'	'order'	'path'
'project'	'propertyMap'	'sack'	'sample'
'select'	'simplePath'	'store'	'tree'
'valueMap'	'where'		

Next, let's revisit some usage with 'by' where it is commonly used with 'path' and 'project'.

```
g.V().has('airport','code','AUS').
  out().out().
  path().by('code').
  limit(10)

[AUS,PHL,SXM]
[AUS,PHL,RIC]
[AUS,PHL,LEX]
[AUS,PHL,ISP]
[AUS,PHL,SWF]
[AUS,PHL,DSM]
[AUS,PHL,MYR]
```

```
[AUS,PHL,CAE]
[AUS,PHL,CHA]
[AUS,PHL,CHS]

g.V().has('type','airport').limit(10).
      project('a','b','c').
        by('code').
        by('region').
        by(outE().count())

[a:ATL,b:US-GA,c:242]
[a:ANC,b:US-AK,c:41]
[a:AUS,b:US-TX,c:98]
[a:BNA,b:US-TN,c:75]
[a:BOS,b:US-MA,c:143]
[a:BWI,b:US-MD,c:91]
[a:DCA,b:US-DC,c:96]
[a:DFW,b:US-TX,c:253]
[a:FLL,b:US-FL,c:158]
[a:IAD,b:US-VA,c:158]
```

In both of the prior examples, the 'by' modulators were 'productive', which means that when the 'by' is used by the modulated step, it generates a result. When the 'by' does not emit a result, it is said to be 'unproductive' and introduces an important aspect of Gremlin semantics. Let's modify the first example with 'path' to make it unproductive by introducing an invalid property key and referring to it as '"cde"' instead of '"code"':

```
g.V().has('airport','code','AUS').
  out().out().
  path().by('cde').
  limit(10)

// no results
```

The prior example returns no results. The unproductive 'by' to 'path' forces it to behave a bit like a filter. Since 'path' can't use '"cde"' to access a valid property key, it simply drops that traverser to prevent an error. Now, let's modify the second example to make it unproductive for some cases, particularly those where the number of outgoing edges is less than 100:

```
g.V().has('type','airport').limit(10).
      project('a','b','c').
        by('code').
        by('region').
        by(outE().count().is(gt(100)))

[a:ATL,b:US-GA,c:101]
[a:ANC,b:US-AK]
[a:AUS,b:US-TX]
```

```
[a:BNA,b:US-TN]
[a:BOS,b:US-MA,c:101]
[a:BWI,b:US-MD]
[a:DCA,b:US-DC]
[a:DFW,b:US-TX,c:101]
[a:FLL,b:US-FL,c:101]
[a:IAD,b:US-VA,c:101]
```

As you can see, the 'project' step will not emit a key for '"c"' when a modulator is unproductive. Each step will have its own semantics for what it does with an unproductive 'by', but typically there is a form of filtering operation that occurs akin to what we've seen in the prior example. We will see more examples of 'by' introducing these kinds of behaviors as we learn new steps in future sections.

Let's look at another example with 'path' where the 'by' modulator is not productive.

```
g.V().has('airport','code','AUS').
  outE().inV().outE().inV().
  path().by('dist').
  limit(10).
  sum(local)

// no results
```

The above example is similar to the earlier one but expands the path Gremlin travels to include 'Edge' objects. In this case, the query seeks to sum the value of dist properties on edges in the path. The 'Vertex' objects will not have a dist property key and therefore be unproductive. One way to change that is to ensure that vertices produce a value with their 'by' modulator, by supplying a 'constant(0)' for those cases.

```
g.V().has('airport','code','AUS').
  outE().inV().outE().inV().
  path().
    by(constant(0)).
    by('dist').
  limit(10).
  sum(local)

3102
1628
1948
1559
1557
2399
1904
1952
2069
```

```
1980
```

With that small adjustment, vertices in the path can produce a '0' integer value which can be summed with the dist property value from the edges.

3.3.6. Using multiple 'as' steps with the same label

It is actually possible using 'as' step to give more than one part of a traversal the same label (name). In the example below, the label "a" is used twice, but you will notice that when the label is selected, only the last item added is returned.

```
g.V(1).as('a').V(2).as('a').select('a')

v[2]
```

There are some special keywords that can be used in conjunction with the 'select' step in cases like this one. These keywords are 'first', 'last' and 'all', and their usage is shown below.

```
g.V(1).as('a').V(2).as('a').select(first,'a')

v[1]

g.V(1).as('a').V(2).as('a').select(last,'a')

v[2]

g.V(1).as('a').V(2).as('a').select(all,'a')

[v[1],v[2]]
```

Here is another example of a query that labels two different parts of a traversal with the same "a" label. As you can see from the results, only the second one is used because of the 'last' keyword provided on the 'select' step.

```
g.V().has('code','AUS').as('a').
      out().as('a').limit(10).
      select(last,'a').by('code').fold()

[PHL,PDX,DTW,OKC,ONT,CLT,CUN,MEM,CVG,IND]
```

Here is the same query but using the 'first' keyword this time as part of the 'select' step.

```
g.V().has('code','AUS').as('a').
      out().as('a').limit(10).
      select(first,'a').by('code').fold()
```

```
[AUS,AUS,AUS,AUS,AUS,AUS,AUS,AUS,AUS,AUS]
```

Note that when the same name is used to label a step, the data structure created by Gremlin is essentially a List. As such, the 'by' modulator cannot be used when the 'all' keyword is used on the 'select' step. To get the values of each element in the list, we can use an 'unfold' step as shown below.

```
g.V().has('code','AUS').as('a').
      out().as('a').limit(10).
      select(all,'a').unfold().values('code').fold()

[AUS,AUS,AUS,AUS,AUS,AUS,AUS,AUS,AUS,AUS,
 PHL,PDX,DTW,OKC,ONT,CLT,CUN,MEM,CVG,IND]
```

Keywords such as 'all', 'first' and 'last' are discussed further in the "Important Classes and Enums to be aware of" section later on in the book.

3.3.7. Returning selected parts of a path

Sometimes, even using the 'from' and 'to' modulating steps along with a 'path' step will not give you the results you are interested in. Using a 'select' step and some 'as' steps in a similar way to the example in the previous section, we can select specific parts of a traversal's "path". Consider the query below that finds a route from Los Angeles (LAX) and returns the path.

```
g.V().has('code','LAX').
      out().
      out().
      out().
      out().
      out().
      limit(1).
      path().by('code')

[LAX,BER,SVG,KSU,OSL,BOS]
```

Now, imagine we want to just return every other stop as the result from our query. The example below shows how to do just that.

```
g.V().has('code','LAX').
      out().as('stop').
      out().
      out().as('stop').
      out().
      out().as('stop').
      limit(1).
      select(all,'stop').
      unfold().
```

```
      values('code').fold()

[BER,KSU,BOS]
```

3.3.8. Examining the edge between two vertices

Sometimes, it is the edge between two vertices that we are interested in examining and not the vertices themselves. Typically, this is because we want to look at one or more properties associated with that edge. By way of an example, let's imagine we wanted to know how many miles the flight is between Miami (MIA) and Dallas Fort Worth (DFW). In our air-routes graph, the distances between vertices are stored using a property called 'dist' on any edge that has a 'route' label. We can use the 'outE' and 'inV' steps to find the edge connecting Miami and Dallas. We can also use the 'select' and 'as' steps that we just learned about to help with this task. Take a look at the query below. This will find the outgoing 'route' edge from MIA to DFW, store it in the traversal variable 'e' and at the end of the query use 'select' to return it as the result of the query.

```
g.V().has('code','MIA').outE().as('e').inV().has('code','DFW').select('e')
```

If we were to run the query, we would get back something similar to this

```
e[4266][16-route->8]
```

So we found the 'route' edge that connects the vertex with an ID of 16 (MIA) with the airport that has an ID of 8 (DFW). While interesting, this is not exactly what we set out to achieve. What we actually are interested in is the distance property of that edge, so we can see how far it is from Miami to Dallas Fort Worth. We need to add one additional step to our query that will look at the 'dist' property of the edge. Let's modify our query to do that.

```
g.V().has('code','MIA').outE().as('e').
      inV().has('code','DFW').select('e').values('dist')
```

If we run the query again, we get back what we were looking for. We can see that it is 1,120 miles from Miami to Dallas Fort Worth.

```
1120
```

As a side note, we could have written the query using 'inE' and 'outV' and achieved the same result by looking at the edge from Dallas to Miami.

```
g.V().has('code','MIA').inE().as('e').
      outV().has('code','DFW').select('e').values('dist')

1120
```

Throughout the remainder of the book you will find lots of examples that use steps such as 'outE', 'inE', 'outV' and 'inV'.

3.4. Limiting the amount of data returned

It is sometimes useful, especially when dealing with large graphs, to limit the amount of data that is returned from a query. As shown in the examples below, this can be done using the 'limit' and 'tail' steps. A little later in this book we also introduce the 'coin' step that allows a pseudo random sample of the data to be returned.

```
// Only return the FIRST 20 results
g.V().hasLabel('airport').values('code').limit(20)

// Only return the LAST 20 results
g.V().hasLabel('airport').values('code').tail(20)
```

Depending upon the implementation, it is probably more efficient to write the query like this, with 'limit' coming before 'values' to guarantee fewer airports are initially returned, but it is also possible that an implementation would optimize both the same way.

```
// Only return the FIRST 20 results
g.V().hasLabel('airport').limit(20).values('code')
```

Note that 'limit' provides a shorthand alternative to 'range'. The first of the two examples above could have been written as follows.

```
// Only return the FIRST 20 results
g.V().hasLabel('airport').range(0,20).values('code')
```

We can also limit a traversal by specifying a maximum amount of time that it is allowed to run for. The following query is restricted to a maximum limit of ten milliseconds. The query looks for routes from Austin (AUS) to London Heathrow (LHR). All the parts of this query are explained in detail later on in this book, but we think what they do is fairly clear. The 'repeat' step is explained in detail in the "Shortest paths (between airports) - introducing 'repeat'" section.

```
// Limit the query to however much can be processed within 10 milliseconds
g.V().has('airport','code','AUS').
      repeat(timeLimit(10).out()).until(has('code','LHR')).path().by('code')
```

Here is what the query above returned when run on our laptop.

```
[AUS,LHR]
[AUS,PHL,LHR]
[AUS,PDX,LHR]
```

```
[AUS,DTW,LHR]
[AUS,CLT,LHR]
[AUS,NAS,LHR]
```

If we give the query another 10 milliseconds to run, so 20 in total, you can see that a few more routes were found.

```
// Limit the query to 20 milliseconds
g.V().has('airport','code','AUS').
      repeat(timeLimit(20).out()).until(has('code','LHR')).path().by('code')

[AUS,LHR]
[AUS,PHL,LHR]
[AUS,PDX,LHR]
[AUS,DTW,LHR]
[AUS,CLT,LHR]
[AUS,NAS,LHR]
[AUS,CHS,LHR]
[AUS,ATL,LHR]
[AUS,BNA,LHR]
[AUS,BOS,LHR]
[AUS,BWI,LHR]
[AUS,DFW,LHR]
```

3.4.1. Retrieving a range of vertices

Gremlin provides various ways to return a sequence of vertices. We have already seen the 'limit' and 'range' steps used in the previous section to return the first 20 elements of a query result. We can also use the 'range' step to select different ranges of vertices by giving a non-zero starting offset and an ending offset. The 'range' offsets are zero-based and are inclusive/exclusive.

```
// Return the first two airport vertices found
g.V().hasLabel('airport').range(0,2)

v[1]
v[2]
```

The starting value given to a 'range' step does not have to be '0'. In the example below we ask for the 3rd, 4th and 5th results found by specifying a range of '"(3,6)"'.

```
// Return the fourth, fifth and sixth airport vertices found (zero based)
g.V().hasLabel('airport').range(3,6)

v[4]
v[5]
v[6]
```

Here is an example of how we can use the index '-1' to mean '"until the end of the list"'. This is similar to the convention used in many programming languages when working with arrays and lists.

```
// Return all the remaining vertices starting at the 3500th one
g.V().range(3500,-1)
```

Here is another example that uses the 'range' step, this time looking only at vertices with a label of 'country'. Notice how this time we found vertices with much higher ID values.

```
g.V().hasLabel('country').range(0,2)

v[3505]
v[3506]
```

There is no guarantee as to which airport vertices will be selected as this depends upon how they are stored by the back end graph. Using TinkerGraph, the airports will most likely come back in the order they are put into the graph. This is not likely to be the case with other graph stores such as JanusGraph. So do not rely on any ordering expectations when using 'range' to process sets of vertices.

You can use 'a skip' step as an alternative to 'range' in some cases. The 'skip' step can be used whenever you would otherwise use 'range' where the second parameter would be '-1' meaning "all remaining".

The two examples below will produce the same results.

```
g.V().has('region','US-TX').skip(5).fold()

g.V().has('region','US-TX').range(5,-1).fold()
```

Here is the output you might get from running either query.

```
[v[39],v[186],v[273],v[278],v[289],v[314],v[356],v[357],v[358],v[361],v[368],
 v[370],v[390],v[394],v[404],v[405],v[423],v[426],v[428],v[1118],v[3313],v[3416]]
```

To prove that the 'skip' and 'range' steps used above worked again, we can run the query again with 'skip' removed and look at the results. You will notice, the first five vertices listed were not included as part of the results from the prior queries.

```
g.V().has('region','US-TX').fold()

[v[3],v[8],v[11],v[33],v[38],v[39],v[186],v[273],v[278],v[289],v[314],v[356],v[357],
 v[358],v[361],v[368],v[370],v[390],v[394],v[404],v[405],v[423],v[426],v[428],
```

```
 v[1118],v[3313],v[3416]]
```

You can also use the 'local' keyword to have 'skip' work on an incoming collection within a traversal. The example below, while contrived, applies skip to the list generated by the 'fold' step.

```
g.V().has('region','US-TX').fold().skip(local,3)

[v[33],v[38],v[39],v[186],v[273],v[278],v[289],v[314],v[356],v[357],v[358],v[361],
 v[368],v[370],v[390],v[394],v[404],v[405],v[423],v[426],v[428],v[1118],v[3313],
 v[3416]]
```

There are many other ways to specify a range of values using Gremlin. You will find several additional examples in the "Testing values and ranges of values with P" section.

3.4.2. Working with the end of a stream

When we want to take items from the end of the stream, we typically use the 'tail' step. With 'tail', you specify the number of elements to take from the end of the stream.

```
g.V().has('airport','code', within('IAD','DAL','JFK','DCA','DFW')).
  order().by('code').
  tail(1).
  values('code').fold()

[JFK]

g.V().has('airport','code', within('IAD','DAL','JFK','DCA','DFW')).
  order().by('code').
  tail(2).
  values('code').fold()

[IAD,JFK]
```

In the prior examples, we use 'order' to ensure consistent results for demonstration, and you often must use 'order' in your own query writing to achieve the same, but note that if you are using 'order', it might be better to rewrite the first as follows:

```
g.V().has('airport','code', within('IAD','DAL','JFK','DCA','DFW')).
  order().by('code',desc).
  limit(1).
  values('code').fold()

[JFK]
```

By using a 'desc' order, we can move JFK to the front of our stream and simply take the first item it encounters, which likely speeds up the query considerably as most graph databases should

optimize the 'order,' and we likely spare Gremlin from having to iterate the entire stream to get to the last item with 'tail'. Depending upon your requirements, you might find that you could use the same tactic for the second query but note that the results have a slight difference.

```
g.V().has('airport','code', within('IAD','DAL','JFK','DCA','DFW')).
  order().by('code',desc).
  limit(2).
  values('code').fold()

[JFK,IAD]
```

In the prior example you can see that we get the same two results, but they are in reversed order. You would have to reverse the 'order' to get the same result as the first query. In these examples, we are dealing with a small dataset with a small set of five results, so the performance implications are not big for either approach, but for larger scales you may find yourself making choices with these patterns that can have significant impact.

Sometimes you may find that you want to take the second-to-last item in the stream. Finding the penultimate element of the stream just means taking the first item found in the list of the last two:

```
g.V().has('airport','code', within('IAD','DAL','JFK','DCA','DFW')).
  order().by('code').
  tail(2).limit(1).
  values('code').fold()

[IAD]
```

Gremlin does not have a step for getting all elements of the stream except for the last one. The following pattern allows you to do this by using 'store' to hold the 'tail' of the stream, which you can then use later to filter it away.

```
g.V().has('airport','code', within('IAD','DAL','JFK','DCA','DFW')).
  order().by('code').
  fold().
  sideEffect(tail(local).unfold().
             local(aggregate('t'))).
  unfold().where(neq('t')).by().by(unfold()).
  values('code').fold()

[DAL,DCA,DFW,IAD]
```

The pattern demonstrated in the prior examples shows how flexible Gremlin can be. With a solid command of the steps Gremlin offers and a clever application of them, you can accomplish a great many things.

3.5. Removing duplicates - introducing 'dedup'

It is often desirable to remove duplicate values from query results. The 'dedup' step allows us to do this. If you are already familiar with Groovy collections, the 'dedup' step is similar to the 'unique' method that Groovy provides. In the example below, the number of runways for every airport in England is queried. Note that in the returned results, there are many duplicate values.

```
g.V().has('region','GB-ENG').values('runways').fold()

[2,2,2,1,1,1,1,1,1,1,1,1,1,1,2,1,2,2,1,3,1,3,3,4,1,1]
```

If we only wanted a set of unique values in the result, we could rewrite the query to include a 'dedup' step. This time the query results only include one of each value.

```
g.V().has('region','GB-ENG').values('runways').dedup().fold()

[2,1,3,4]
```

It is also possible to use a 'by' modulator to specify how 'dedup' should be applied. In the example below we only return one airport for each unique number of runways.

```
g.V().has('region','GB-ENG').dedup().by('runways').
      values('code','runways').fold()

[LHR,2,LCY,1,BLK,3,LEQ,4]
```

There is one more form of the 'dedup' step. In this form, one or more strings representing labeled steps are provided as parameters. First, take a look at the query below. It finds vertex 'V(3)' and labels it "a". It then finds vertex 'V(4)' and labels it "c". Next it finds all the vertices connected to V(4) and labels those "b". Only the first 10 are retrieved. Lastly, a 'select' step is used to return the results. As expected, vertices 3 and 4 are present in all the results.

```
g.V(3).as('a').V(4).as('c').both().as('b').limit(10).
  select('a','b','c')

[a:v[3],b:v[1],c:v[4]]
[a:v[3],b:v[3],c:v[4]]
[a:v[3],b:v[5],c:v[4]]
[a:v[3],b:v[6],c:v[4]]
[a:v[3],b:v[7],c:v[4]]
[a:v[3],b:v[8],c:v[4]]
[a:v[3],b:v[9],c:v[4]]
[a:v[3],b:v[10],c:v[4]]
[a:v[3],b:v[11],c:v[4]]
[a:v[3],b:v[12],c:v[4]]
```

Taking the same query but adding a 'dedup' step that references the "a" and "c" labels, removes all duplicate references that include those vertices from the results, so this time even though a 'limit' of 10 is used, we only actually get one result back.

```
g.V(3).as('a').V(4).as('c').both().as('b').limit(10).
  dedup('a','c').select('a','b','c')

[a:v[3],b:v[1],c:v[4]]
```

A bit later we will take a look at the concept of 'local' scope when working with traversals. There are some examples of 'local' scope being used in conjunction with 'dedup' in the "Using 'local' scope with collections" section.

It is also possible to use 'sets' to achieve similar results as we shall see in some of the following sections such as the "Introducing 'toList', 'toSet', 'bulkSet' and 'fill'" section that is coming up soon.

3.6. Using 'valueMap' to explore the properties of a vertex or edge

A call to 'valueMap' will return all the properties of a vertex or edge as an array of key:value pairs. In Java terms this is called a 'HashMap'. You can also select which properties you want 'valueMap' to return if you do not want them all. Each element in the map can be addressed using the name of the key. By default, the ID and label are not included in the map unless a parameter of 'true' is provided.

The query below will return the keys and values for all properties associated with the Austin airport vertex.

```
// Return all the properties and values the AUS vertex has
g.V().has('code','AUS').valueMap().unfold()
```

If you are using the Gremlin Console, the output from running the previous command should look something like this. The 'unfold' step at the end of the query is used to make the results easier to read.

```
country=[US]
code=[AUS]
longest=[12250]
city=[Austin]
elev=[542]
icao=[KAUS]
lon=[-97.6698989868164]
type=[airport]
region=[US-TX]
runways=[2]
lat=[30.1944999694824]
```

```
desc=[Austin Bergstrom International Airport]
```

Notice how each key, like 'country', is followed by a value returned as an element of a list. This is because it is possible (for vertices but not for edges) to provide more than one property value for a given key by encoding them as a list or as a set. We discuss how to better control this output in the paragraphs that follow.

Here are some more examples of how 'valueMap' can be used. If a parameter of 'true' is provided, then the results returned will include the ID and label of the element being examined.

```
// If you also want the ID and label, add a parameter of true
g.V().has('code','AUS').valueMap(true).unfold()

id=3
label=airport
country=[US]
code=[AUS]
longest=[12250]
city=[Austin]
elev=[542]
icao=[KAUS]
lon=[-97.6698989868164]
type=[airport]
region=[US-TX]
runways=[2]
lat=[30.1944999694824]
desc=[Austin Bergstrom International Airport]
```

You can also include 'true' along with requesting the map for specific properties. The next example will just return the ID, label and 'region' property.

```
// If you want the ID, label and a specific field like the region, you can do this
g.V().has('code','AUS').valueMap(true,'region')

[id:3,region:[US-TX],label:airport]
```

If you only need the keys and values for specific properties to be returned, it is recommended to pass the names of those properties as parameters to the 'valueMap' step so it does not return a lot more data than you need. Think of this as the difference, in the SQL world, between selecting just the columns you are interested in from a table rather than doing a 'SELECT *'.

As shown above, you can specify which properties you want returned by supplying their names as parameters to the 'valueMap' step. For completeness, it is worth noting that you can also use a 'select' step to refine the results of a 'valueMap'.

```
// You can 'select' specific fields from a value map
g.V().has('code','AUS').valueMap().select('code','icao','desc')

[code:[AUS],icao:[KAUS],desc:[Austin Bergstrom International Airport]]
```

As an additional example of Gremlin's flexibility, note that you can also restrict the selected keys to all but those you specify.

```
 g.V('3').valueMap().as('vm').unfold().
   filter(select(keys).is(without('city','desc')))

country=[US]
code=[AUS]
longest=[12250]
elev=[542]
icao=[KAUS]
lon=[-97.6698989868164]
type=[airport]
region=[US-TX]
runways=[2]
lat=[30.1944999694824]
```

If you are reading the output of queries that use 'valueMap' from the console, it is sometimes easier to read the output if you add an 'unfold' step to the end of the query as follows. The 'unfold' step will unbundle a collection for us. You will see it used in many parts of this book.

```
g.V().has('code','AUS').valueMap(true,'code','icao','desc','city').unfold()

code=[AUS]
city=[Austin]
icao=[KAUS]
id=3
label=airport
desc=[Austin Bergstrom International Airport]
```

You can also use 'valueMap' to inspect the properties associated with an edge. In this example, the edge with an ID of 5161 is examined. As you can see, the edge represents a route and has a distance ('dist') property with a value of 1357 miles.

```
g.E(5161).valueMap(true)

[id:5161,label:route,dist:4663]
```

There are other ways to control the results that a 'valueMap' step returns using the 'with' modulator.

The valueMap configuration options are described in the official documentation at the following link https://tinkerpop.apache.org/docs/current/reference/#valuemap-step.

Instead of using 'valueMap(true)' to include the ID and label of an element (a vertex or an edge) in the results, the new 'with(WithOptions.tokens)' construct can now be used as shown below.

```
g.V().has('code','SFO').valueMap().with(WithOptions.tokens).unfold()

id=23
label=airport
country=[US]
longest=[11870]
code=[SFO]
city=[San Francisco]
lon=[-122.375]
type=[airport]
elev=[13]
icao=[KSFO]
region=[US-CA]
runways=[4]
lat=[37.6189994812012]
desc=[San Francisco International Airport]
```

All the possible values that can be specified using WithOptions can be found in the official Apache TinkerPop Javadoc documentation https://tinkerpop.apache.org/javadocs/current/full/org/apache/tinkerpop/gremlin/process/traversal/step/util/WithOptions.html.

You can still include the ID and label in the results, along with a subset of the properties, by explicitly naming the property keys you are interested in. In the example below, only the 'code' property is requested.

```
g.V().has('code','SFO').valueMap('code').with(WithOptions.tokens).unfold()

id=23
label=airport
code=[SFO]
```

You can use additional 'WithOptions' qualifiers to select just the labels.

```
g.V().has('code','SFO').
      valueMap('code').with(WithOptions.tokens,WithOptions.labels).
      unfold()

label=airport
```

```
code=[SFO]
```

In the same way you can choose to just have the ID value returned without the label.

```
g.V().has('code','SFO').
      valueMap('code').with(WithOptions.tokens,WithOptions.ids).
      unfold()

id=23
code=[SFO]
```

As discussed in the previous section, the property values returned by 'valueMap' are by default represented as lists even if there is only a single property value present.

You can very easily request that these values be returned as single values, not wrapped in lists. This can be done using a 'by' step modulator as shown below.

```
g.V().has('code','SFO').valueMap().by(unfold()).unfold()
```

Notice how all the values, such as the city name, "San Francisco", are now just simple strings or numeric values and not a single value wrapped in a list of length one.

```
country=US
code=SFO
longest=11870
city=San Francisco
elev=13
icao=KSFO
lon=-122.375
type=airport
region=US-CA
runways=4
lat=37.6189994812012
desc=San Francisco International Airport
```

There are additional 'WithOptions' settings we can use to change how properties with meta-properties are returned by 'valueMap' This is covered later as part of the "Using 'unfold' and 'WithOptions' with Meta-Properties" section.

3.7. An alternative to 'valueMap' - introducing 'elementMap'

The 'elementMap' step is similar in many ways to the 'valueMap' step but makes some things a little easier. When using 'valueMap', you need to explicitly request that the ID and label of a vertex or an edge are included in query results. This is not necessary when using 'elementMap'.

```
g.V().has('code','AUS').elementMap().unfold()

id=3
label=airport
country=US
code=AUS
longest=12250
city=Austin
elev=542
icao=KAUS
lon=-97.6698989868164
type=airport
region=US-TX
runways=2
lat=30.1944999694824
desc=Austin Bergstrom International Airport
```

As with 'valueMap', you can request only certain property values be included in the resulting map. Note, however, that the property values are not returned as list members. This is a key difference from 'valueMap'. In fact, if the value for a given property is a list or set containing multiple values, 'elementMap' will only return the first member of that list or set. If you need to return 'set' or 'list' cardinality values, you should use 'valueMap' instead.

```
g.V().has('code','AUS').elementMap('city')

[id:3,label:airport,city:Austin]
```

The biggest difference between 'elementMap' and 'valueMap' becomes apparent when looking at edges. For a given edge, as well as the ID and label and properties, information about the incoming and outgoing vertices is also returned.

```
g.V(3).outE().limit(1).elementMap()

[id:3840,label:route,IN:[id:45,label:airport],OUT:[id:3,label:airport],dist:1430]
```

A similar result could be generated using 'valueMap' as shown below, but it is definitely a bit more work.

```
g.E(5161).project('v','IN','OUT').
            by(valueMap(true)).
            by(inV().union(id(),label()).fold()).
            by(outV().union(id(),label()).fold())

[v:[id:5161,label:route,dist:4663],IN:[132,airport],OUT:[1,airport]]
```

To make the output look even closer to the results returned by 'elementMap', we could decide to

add some additional 'project' steps.

```
g.E(5161).project('v','IN','OUT').
            by(valueMap(true)).
            by(project('id','label').
              by(inV().id()).
              by(inV().label())).
            by(project('id','label').
              by(outV().id()).
              by(outV().label())).
            unfold()
```

The results of running the query are shown below. We added an 'unfold' step to the query just to make the results a little easier to read.

```
v={id=5161, label=route, dist=4663}
IN={id=132, label=airport}
OUT={id=1, label=airport}
```

3.8. Assigning query results to a variable with a terminal step

It is extremely useful to be able to assign the results of a query to a variable. The example below stores the results of the 'valueMap' call shown above into a variable called 'aus'.

```
// Store the properties for the AUS airport in the variable aus.
aus=g.V().has('code','AUS').valueMap().next()
```

It is necessary to add a call to 'next' to the end of the query in order for this to work. Forgetting to add the call to 'next' is a very commonly made mistake by people getting used to the Gremlin query language. The call to 'next' terminates the traversal part of the query and generates a concrete result that can be stored in a variable. We refer to 'next' as a "terminal step". There are other terminal steps such as 'toList' and 'toSet' that also perform this traversal termination action. We will see those steps used later on.

Once you have some results in a variable, you can refer to it as you would in any other programming language. We will explore mixing Java and Groovy code with your Gremlin queries later in this book. For now let's just use the Groovy 'println' to display the results of the query that we stored in 'aus'. We will take a deeper look at the use of variables with Gremlin later in the book when we look at mixing Gremlin and Groovy in the "Making Gremlin even Groovier" section.

```
// We can now refer to aus using key:value syntax
println "The AUS airport is located in " + aus['city'][0]
```

```
The AUS airport is located in Austin
```

Properties are stored as arrays of values. Even if there is only one property value for the given key, we still have to add the '[0]' when referencing it; otherwise the whole array will be returned if we just used 'aus['city']'. We will explore why property values are stored in this way in the "Attaching multiple values (lists or sets) to a single property" section.

As a side note, the 'next' step can take a parameter value that tells it how much data to return. For example, if you want the next three vertices from a query like the one below, you can add a call to 'next(3)' at the end of the query. Note that doing this turns the result into an ArrayList. Each element in the list will contain a vertex.

```
verts=g.V().hasLabel('airport').next(3)

v[1]
v[2]
v[3]
```

We can call the Java 'getClass' method to verify the type of the values returned.

```
verts.getClass()

class java.util.ArrayList

verts.get(1).getClass()

class org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerVertex
```

When using the Gremlin Console, you can check to see what variables you have defined using the command ':show variables'.

3.8.1. Introducing 'toList', 'toSet', 'bulkSet' and 'fill'

It is often useful to return the results of a query as a list or as a set. One way to do this is to use 'toList' or 'toSet' methods. Below you will find an example of each. The call to 'join' is used just to make the results easier to read on a single line.

```
// Create a list of runway counts in Texas
listr = g.V().has('airport','region','US-TX').
              values('runways').toList().join(',')

2,7,5,3,4,3,3,3,3,4,2,3,2,3,2,2,3,2,1,3,2,3,4,3,4,2,1
```

Now let's create a set and observe the different results we get back.

```
// Create a set of runway counts in Texas (no duplicates)
setr = g.V().has('airport','region','US-TX').
             values('runways').toSet().join(',')

1,2,3,4,5,7
```

As a side note, in many cases we can use the 'dedup' step to remove duplicates from a result. However, it is worth knowing that a set can be created as a result type as in some cases this can be highly useful. The example below performs the same 'runways' query using a 'dedup' step. We added an 'order' step so that it is easier to compare the results with the previous query.

```
// Create a list of runway counts in Texas (no duplicates)
g.V().has('airport','region','US-TX').
      values('runways').dedup().order().fold()

[1,2,3,4,5,7]
```

Finally, let's create the list again, but without the call to 'join', as that creates a single string result, which is not what we want in this case.

```
listr = g.V().has('airport','region','US-TX').
        values('runways').toList()
```

The variable can now be used as you would expect.

```
listr[1]
7

listr.size()
26

listr[1,3]
7
3
```

TinkerPop also provides a third method called 'bulkSet' that can be used to create a collection at the end of a traversal. The difference between a 'bulkSet' and a 'set' is that 'bulkSet' is a so-called 'weighted set'. A 'bulkSet' stores every value but includes a count of how many of each type are present. Let's look at a few examples. First, we can check that the 'bulkSet' does indeed contain all the values.

```
setb= g.V().has('airport','region','US-TX').values('runways').toBulkSet().join(',')
```

```
2,2,2,2,2,2,2,2,7,5,3,3,3,3,3,3,3,3,3,3,3,4,4,4,4,1,1
```

A 'bulkSet' offers some additional methods that we can call. One of these is 'uniqueSize' that will tell us how many unique values are present.

```
setb= g.V().has('airport','region','US-TX').values('runways').toBulkSet()

// How many unique values are in the set?
setb.uniqueSize()
6

// How many total values are present?
setb.size()
27
```

The 'asBulk' method returns a map of key/value pairs where the key is the number and the value is the number of times that number appears in the set.

```
setb.asBulk()

2=8
7=1
5=1
3=11
4=4
1=2
```

There is another way to store the results of a query into a collection. This is achieved using the 'fill' method. Unlike 'toList' and the other methods that we just looked at, 'fill' will store the results into a pre-existing variable. The query below defines a list called 'a' and stores the results of the query into it. This will produce the same result as using 'toList'.

```
a = []
g.V().has('airport','region','US-TX').values('runways').fill(a)

a.size()
27

a[1,3]
7
3
```

We can define a variable that is a set and use 'fill' to achieve the same result as using 'toSet'.

```
s = [] as Set
g.V().has('airport','region','US-TX').values('runways').fill(s)
```

```
println s

[2, 7, 5, 3, 4, 1]
```

3.8.2. Ignoring query results with 'iterate'

There are a number of cases where the results of a query are not of interest to you. A common situation where this happens is when you have a graph mutation query and are only interested in persisting those changes to the database but are not interested in doing anything with the output that the traversal produces.

```
g.addV('airport').property('code','AUS').as('aus').
  addV('airport').property('code','DFW').as('dfw').
  addE('route').from('aus').to('dfw').iterate()
```

In the example above, you can see that we add two vertices and an edge between them with the return value being the edge. As we used 'next' as the terminal step, the query would have returned the newly created edge. With the use of 'iterate' as the terminal step, the query has no value returned. The vertices are added to the graph along with the edge, and the edge itself is discarded. Additional calls to 'next' after 'iterate' will result in a 'NoSuchElementException'.

The use of 'iterate' can bring some performance improvements because it signals to the graph that you are not interested in the results, which could save some processing costs (e.g., serialization, network).

3.9. Deep dive on traversal terminology

In Gremlin, we use the word "traversal" often, and there is a fair bit of terminology associated with it. Understanding that terminology will greatly help with your understanding of Gremlin. To get started in this learning, let's return to how we create "g", known as the 'GraphTraversalSource', which is the connection to the graph from which we can write Gremlin to query our data.

```
// create the Graph object which in this case is a TinkerGraph
graph = TinkerGraph.open()

// create the GraphTraversalSource object
g = traversal().with(graph)
```

In looking at the above code, you might wonder where the 'traversal()' method comes from. This function is a member of the 'AnonymousTraversalSource' class, and it is considered good form to statically import it so that it can be called without the class. Calling 'traversal()' constructs an anonymous traversal source, which is like having a graph traversal source without a connection to a graph. Calling 'with' on this object binds it to a particular graph data source, and then "g" is ready for you to write your queries.

You may see examples in this book, blog posts or other documentation that shorthand the creation of 'g' like

```
g = TinkerGraph.open().traversal()
```

While you can take this approach, it does preclude you from having reference to the 'Graph' object. While this is typically not a problem for TinkerGraph, it can be an issue for other TinkerPop-enabled graphs that might require that reference to release resources or access provider-specific APIs.

Another tempting shorthand is to skip the creation of "g" like

```
graph = TinkerGraph.open()
graph.traversal().V()
```

If you take this approach, you unnecessarily create 'GraphTraversalSource' objects which will just end up in garbage collection. Typically, you create 'g' once and then reuse it. One of the other benefits to creating 'g' once is that it allows you to set configurations on the graph traversal source once rather than for each traversal you execute.

Configuration steps are used to set up options on the graph traversal source. These steps are prefixed with 'with' and work in a builder pattern style as follows

```
graph = TinkerGraph.open()
g = traversal().with(graph)

// construct a GraphTraversalSource configured with ReadOnlyStrategy and a
// List side effect named "d"
g = g.withStrategies(ReadOnlyStrategy.instance()).
      withSideEffect('d', [1,2,3])
```

Once 'g' is configured, you can then use start steps to spawn a 'Traversal'. In TinkerPop, a "traversal" is a term we consider synonymous with "query". In the context of querying graphs, the word "traversal" implies movement which aligns with the visual image of traversing about the vertices and edges of a graph to get the data that you need.

While Gremlin has dozens of steps, not all of them can be used to spawn a 'Traversal'. Only those found on the 'GraphTraversalSource' can be used to do that. The most commonly used start step is 'V', but there are a number of others, like 'E' to start by traversing edges or 'addV' to add a new vertex to the graph. It is important to realize that a 'Traversal' object spawned from 'g' does little on its own and requires that a terminal step be added to get the query results. This point was introduced in the "Assigning query results to a variable with a terminal step" section, but it is worth revisiting again as it is a common mistake made by new Gremlin users. It is easy to illustrate this point clearly with an example in Java.

```
// create the Traversal object - "t" is just a reference to the traversal and is
```

```
// not the result of "g.V()" being executed
Traversal t = g.V();

// to get the result we need a terminal step, in this case toList()
List<Vertex> l1 = t.toList();

// more commonly the traversal has the terminal step added right when it is spawned
List<Vertex> l2 = g.V().toList();
```

There is one more piece of terminology to consider, and we've touched on them earlier. Anonymous traversals were first introduced in the "What vertices and edges did I visit? — Introducing 'path'" section, where they were used to modify 'Path' objects returned from the 'path' step. One of those examples demonstrated how a 'by' modulator could be given an anonymous traversal to convert vertices in the returned path into a count of the number of outgoing routes from airports.

```
g.V(3).out().limit(5).path().by(outE().count())

[98,147]
[98,80]
[98,145]
[98,33]
[98,23]
```

We often see anonymous traversals used with 'by,' but they can actually be used as an argument for a great many Gremlin steps, making them an integral part of the language. Let's take a step back from usage and examine anonymous traversals as a concept. Like the anonymous traversal source, an anonymous traversal is one that is not bound to a graph. Instead of spawning from 'g', an anonymous traversal spawns from a class named with a double-underscore.

```
// spawns a traversal bound to a graph
g.V().out()

// spawns an anonymous traversal without binding to a graph
__.V().out()

// the preferred approach is to statically import V() so that you can omit "__."
// the following anonymous traversal is the same as the one above
V().out()
```

As shown earlier with 'by', an anonymous traversal can be used as an argument for many different traversal steps.

```
g.V().has('airport','code','AUS').
      repeat(timeLimit(10).out()).until(has('code','LHR')).path().by('code')
```

Without driving too deeply into the details of the example above, note that we've used an anonymous traversal inside both 'repeat' and 'until'.

In this section, we learned about Gremlin terminology. In summary, here are the key points to remember:

- A 'Traversal' is a graph query.
- A 'GraphTraversalSource' is usually named "g" by convention and spawns 'Traversal' objects.
- When either of these objects is referred to as "anonymous", they are not connected to any graph.

Now that you've learned all of this important Gremlin terminology, you are well-prepared for the more advanced topics to come.

3.10. Working with IDs

Every vertex, every edge and even every property in a graph has a unique ID that can be used to reference it individually or as part of a group. These IDs are immutable and, therefore, cannot be changed once assigned. Beware that the IDs you provide when loading a graph from a GraphML or GraphSON file may not in many cases end up being the IDs that the back-end graph store actually uses as it builds up your graph. TinkerGraph, for example, will preserve user-provided IDs, but many graph databases such as JanusGraph generate their own IDs. The same is true when you add vertices and edges using a graph traversal or using the TinkerPop API. This is a long-winded way of saying that you should not depend on the IDs in your GraphML or GraphSON file that you just loaded remaining unchanged once the data has been loaded into the graph store. When you add a new vertex or edge to your graph using a traversal, the graph system will automatically generate a new, unique ID for it. If you need to figure out the ID for a vertex or an edge, you can always get it from a query of the graph itself.

Don't rely on the graph preserving the ID values you provide. Write code that can query the graph itself for ID values. How IDs are managed will be graph database implementation-dependent.

Especially when dealing with large graphs, because using IDs is typically very efficient, you will find that many of your queries will involve collecting one or more IDs and then passing those on to other queries or parts of the same query. In most if not all cases, the underlying graph system will have set up its data structures, whether on disk or in memory, to be very rapidly accessed by ID value.

Let's demonstrate the use of ID values using a few simple examples. The query below finds the ID, which is 8, for the vertex that represents the DFW airport.

```
// What is the ID of the "DFW" vertex?
g.V().has('code','DFW').id()

8
```

Let's reverse the query and find the code for the vertex with an ID of 8.

```
// Simple lookup by ID
g.V().hasId(8).values('code')

DFW
```

We could also have written the above query as follows.

```
// which is the same as this
g.V().has(id,8).values('code')
```

Here are some more examples that make use of the ID value.

```
// vertices with an ID between 1 and 5 (note this is inclusive/exclusive)
g.V().hasId(between(1,6))

// Which is an alternate form of this
g.V().has(id,between(1,6))

// Find routes from the vertex with an ID of 6 to any vertex with an ID less than 46
g.V().hasId(6).out().has(id,lt(46)).path().by('code')

// Which is the same as
g.V().hasId(6).out().hasId(lt(46)).path().by('code')
```

You can also pass a single ID or multiple IDs directly into the 'V' step. Take a look at the two examples below.

```
// What is the code property for the vertex with an ID of 3?
g.V(3).values('code')

AUS

// As above but for all of the specified vertex IDs
g.V(3,6,8,15).values('code')

AUS
BWI
DFW
MCO
```

You can also pass a list of ID values into the 'V' step. We take a closer look at using variables in this way in the "Using a variable to feed a traversal" section.

```
a=[3,6,8,15]

g.V(a).values('code')
```

If the graph database that you are using supports it, you can set the ID of a vertex at the time you create it. How that can be done is explained in the "Using 'inject' to specify new vertex ID values" section.

Every property in the graph also has an ID as we shall explore in the "Properties have IDs too" section a bit later on.

3.11. Working with labels

It's a good idea when designing a graph to give the vertices and edges meaningful labels. As with IDs, labels are immutable and therefore cannot be changed once assigned. You can use these to help refine searches. For example, in the 'air-routes' graph, every airport vertex is labeled 'airport' and every country vertex, not surprisingly, is labeled 'country'. Similarly, edges that represent a flight route are labeled 'route'. You can use labels in many ways. We already saw the 'hasLabel' step being used in the basic queries section to test for a particular label. Here are a few more examples.

```
// What label does the LBB vertex have?
g.V().has('code','LBB').label()

// What airports are located in Australia? Note that 'contains' is an
// edge label and 'country' is a vertex label.
g.V().hasLabel('country').has('code','AU').out('contains').values('code')

// We could also write this query as follows
g.V().has('country','code','AU').out().values('code')
```

By using labels in this way we can effectively group vertices and edges into classes or types. Imagine if we wanted to build a graph containing different types of vehicles. We might decide to label all the vertices just 'vehicle', but we could decide to use labels such as 'car', 'truck' and 'bus'. Ultimately, the overall design of your graph's data model will dictate the way you use labels, but it is good to be aware of their value.

As useful as labels are, not all graph database engines provide support for indexing them. You should check to see if the graph database technology you are using allows for labels to be indexed. If that is not the case, it is recommended to use a vertex or edge property that can be indexed, as a surrogate for the label. This can then be used in graph queries rather than relying on the vertex label. This is especially important when working with large graphs where performance can become an issue if the items you are looking for are not backed by an index.

Here are a few more examples of ways we can work with labels.

```
// You can explicitly reference a vertex label using the label() method
g.V().where(label().is(eq('airport'))).count()

// Or using the label key word
g.V().has(label,'airport').count()

// But you would perhaps use the hasLabel() method in this case instead
g.V().hasLabel('airport').count()

// How many non airport vertices are there?
g.V().has(label,neq('airport')).count()
g.V().where(label().is(neq('airport'))).count()

// Again, it might be more natural to actually write this query like this:
g.V().not(hasLabel('airport')).count()
```

The same concepts apply equally well when looking at edge labels as shown below.

```
// The same basic concepts apply equally to edges
g.E().has(label,'route').count()

g.E().where(label().is(eq('route'))).count()

g.E().hasLabel('route').count()
```

Of course, we have already seen another common place where labels might get used. Namely, in the three-parameter form of 'has' as in the example below. The first parameter is the label value. The next two parameters test the properties of all vertices that have the 'airport' label for a code of "'SYD'".

```
g.V().has('airport','code','SYD')
```

It is also possible to specify more than one label in the same step as shown below. In general, whenever a step can be provided a label, more than one may also be provided.

```
g.E().hasLabel('route','contains')
```

3.12. Using the 'local' step to make sure we get the result we intended

Sometimes it is important to be able to do calculations based on the current state of a traversal rather than waiting until near the end. A common place where this is necessary is when calculating the average value of a collection. In the next section we are going to look at a selection of numerical and statistical operations that Gremlin allows us to perform. However, for now let's use the 'mean'

step to calculate the average of something and look at the effect the 'local' step has on the calculation. The 'mean' step works just like you would expect, it returns the mean, or average, value for a set of numbers.

If we wanted to calculate the average number of routes from an airport, the first query that we would write might look like the one below.

```
g.V().hasLabel('airport').out('route').count().mean()

50637.0
```

As you can see the answer we got back, '43400.0' looks wrong, and indeed it is. That number is, in fact, the total number of outgoing routes in the entire graph. This is because as written, the query counts all the routes, adds them all up, but does not keep track of how many airports it visited. This means that calling the 'mean' step is essentially the same as dividing the count by one.

So how do we fix this? The answer is to use the 'local' step. What we really want to do is to create, in essence, a collection of values, where each value is the route count for just one airport. Having done that, we want to divide the sum of these numbers by the number of members, airports in this case, into the collection.

Take a look at the modified query below.

```
// Average number of outgoing routes from an airport.
g.V().hasLabel('airport').local(out('route').count()).mean()

14.451198630136986
```

The result this time is a much more believable answer. Notice how this time we placed the *out("route").count()* steps inside a 'local' step. The query below, with the mean step removed, shows what is happening during the traversal as this query runs. We truncated the output to just show a few lines.

```
g.V().hasLabel('airport').local(out('route').count()).limit(10)

242
41
98
75
143
91
96
253
158
158
```

What this shows is that for the first ten airports, the collection that we are building up contains one

entry for each airport that represents the number of outgoing routes that airport has. Then, when we eventually apply the 'mean' step, it will calculate the average value of our entire collection and give us back the result that we were looking for.

Let's look at another example where we can use the 'local' step to change the results of a query usefully. First, take a look at the query below and the results that it generates. The query first finds all the airports located in Scotland using the region code of 'GB-SCT'. It then creates an ordered list of airport codes and city names into a list.

```
g.V().has('region','GB-SCT').order().by('code').
      values('code','city').fold()
```

Here are the results from running the query.

```
[ABZ,Aberdeen,BEB,Balivanich,BRR,Eoligarry,CAL,Campbeltown,DND,Dundee,EDI,Edinburgh,EO
I,Eday,FIE,Fair Isle,FOA,Foula,
 GLA,Glasgow,ILY,Port Ellen,INV,Inverness,KOI,Orkney Islands,LSI,Lerwick,LWK,Lerwick
,NDY,Sanday,NRL,North Ronaldsay,
 PIK,Glasgow,PPW,Papa Westray,PSV,Papa Stour Island,SOY,Stronsay,SYY,Stornoway,TRE
,Balemartine,WIC,Wick,WRY,Westray]
```

However, it would be more convenient perhaps to have the results be returned as a list of lists where each small list contains the airport code and city name with all the small lists wrapped inside a big list. We can achieve this by wrapping the second half of the query inside a 'local' step as shown below.

```
g.V().has('region','GB-SCT').order().by('code').
      local(values('code','city').fold())
```

Here are the results of running the modified query. We have arranged the results in two columns to aid readability.

```
[ABZ,Aberdeen]
[BEB,Balivanich]
[BRR,Eoligarry]
[CAL,Campbeltown]
[DND,Dundee]
[EDI,Edinburgh]
[EOI,Eday]
[FIE,Fair Isle]
[FOA,Foula]
[GLA,Glasgow]
[ILY,Port Ellen]
[INV,Inverness]
[KOI,Orkney Islands]
[LSI,Lerwick]
```

```
[LWK,Lerwick]
[NDY,Sanday]
[NRL,North Ronaldsay]
[PIK,Glasgow]
[PPW,Papa Westray]
[PSV,Papa Stour Island]
[SOY,Stronsay]
[SYY,Stornoway]
[TRE,Balemartine]
[WIC,Wick]
[WRY,Westray]
```

There are many other ways that 'local' can be used. You will find examples of those throughout the book. You will see some that show how local can be used as a parameter to the 'order' step when we dig deeper into route analysis in the "Distribution of routes in the graph (mode and mean)" section.

3.13. Basic statistical and numerical operations

The following queries demonstrate concepts such as calculating the amount of a particular item that is present in the graph, calculating the average (mean) of a set of values and calculating a maximum or minimum value. The table below summarizes the available steps.

Table 2. Basic statistical steps

count	Count how many of something exists.
sum	Sum (add up) a collection of values.
max	Find the maximum value in a collection of values.
min	Find the minimum value in a collection of values.
mean	Find the mean (average) value in a collection.

We will dig a bit deeper into some of these capabilities and explain in more detail in the "Distribution of routes in the graph (mode and mean)" section of the book. Some of these examples also take advantage of the 'local' step that was introduced in the previous section. A way of calculating the standard deviation within a data set is presented later in the "Gremlin's scientific calculator – introducing 'math'" section. The results of running each query are also shown in the examples below.

```
// How many routes are there from Austin?
g.V().has("airport","code","AUS").out().count()

98

// Sum of values - total runways of all airports
g.V().hasLabel('airport').values('runways').sum()

4980
```

The 'mean' step allows us to find the mean (average) value in a data set.

```
// Statistical mean (average) value - average number of runways per airport
g.V().hasLabel('airport').values('runways').mean()

1.4212328767123288

// Average length of the longest runway across all airports
g.V().hasLabel('airport').values('longest').mean()

7544.5559360730595

// Average number of routes to and from an airport
g.V().hasLabel('airport').local(both('route').count()).mean()

28.902397260273972
```

The following queries find maximum and minimum values using the 'max' and 'min' steps.

```
//maximum value - longest runway
g.V().hasLabel('airport').values('longest').max()

18045

// What is the biggest number of outgoing routes any airport has?
g.V().hasLabel('airport').local(out('route').count()).max()

310

//minimum value - shortest runway
g.V().hasLabel('airport').values('longest').min()

1300
```

It is also possible in more recent versions of Apache TinkerPop to use the 'min' and 'max' steps with more than just numeric values. These steps apply to any values that are considered *"comparable"*. So, for example, we can now compare strings as well as numbers. The examples below look for the minimum and maximum value in the descriptive names of the continents.

```
g.V().hasLabel('continent').values('desc').min()

Africa

g.V().hasLabel('continent').values('desc').max()

South America
```

3.14. Working with text

Gremlin has a wide variety of steps for manipulating text. String values in Gremlin often derive directly from property values, but you may encounter them by other means as well. One of these ways is by means of the 'asString' step, which takes any type and converts it to a string representation.

```
// 18045 is a Long type
g.V().hasLabel('airport').values('longest').max()

18045

// 18045 is a String type
g.V().hasLabel('airport').values('longest').max().asString()

18045
```

Many of the operations you are used to having for working with strings are available in Gremlin.

concat	Concatenates one or more string value together with the incoming traverser
format	Formats a string using a user supplied template
length	Counts the number of characters in the string
lTrim	Removes whitespace from the left of the string
toLower	Converts the string to all lower case
toUpper	Converts the string to all upper case
replace	Replaces occurrences of a specified string with another string value
reverse	Reverses the characters in a string
rTrim	Removes whitespace from the right of the string
split	Splits a string into multiple strings based on a specified delimiter
substring	Exacts a substring from the incoming string based on specified indices
trim	Removes whitespace from both ends of the string

The above steps should be generally recognizable to most developers and are for the most part self-explanatory. Let's look at a few examples of each to get a feel for how they work. We'll start with ones that require the least explanation first.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city')

Washington D.C.
Los Angeles
Miami
```

```
// convert each city to all lower case with toLower()
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city').toLower()

washington d.c.
los angeles
miami

// convert each city to all upper case with toUpper()
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city').toUpper()

WASHINGTON D.C.
LOS ANGELES
MIAMI

// get the number of characters for each city with length()
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city').length()

15
11
5

// reverse() the characters for each city
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city').reverse()

.C.D notgnihsaW
selegnA soL
imaiM
```

The 'substring' step is 0-based (inclusive) and optionally accepts an end index (exclusive).

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('code').
  substring(1)

AD
AX
IA

g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('code').
  substring(0, 1)

I
L
M
```

The 'split' step is a bit different than most of the string manipulation steps in that all the other steps produce one string as an out, whereas 'split' produces a list of strings.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city').
  split(' ')

[Washington,D.C.]
[Los,Angeles]
[Miami]
```

Sometimes you will want to find occurrences of a particular character or string and replace that with another.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('code').replace(' ', '0')

I0D
MI0
L0X
```

We can use the 'replace' step to help demonstrate the steps that remove whitespace. The following examples use 'inject' to start the traversal with some strings that include whitespace before and after a character. We will use the various 'trim' steps to remove the whitespace and then 'replace' any that is left over to see which bit of whitespace has been removed.

```
g.inject(' x', 'x ', ' x ').lTrim().replace(' ', 'Z')

x
xZ
xZ

g.inject(' x', 'x ', ' x ').rTrim().replace(' ', 'Z')

Zx
x
Zx

g.inject(' x', 'x ', ' x ').trim().replace(' ', 'Z')

x
x
x
```

When we want to put strings together, we can use 'concat'. This most simple form of this step might be best demonstrated by adding a suffix to the end of a string or prefix to the start of one.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('code').concat('_suffix')

IAD_suffix
LAX_suffix
MIA_suffix

g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('code').as('c').
  constant('prefix_').
  concat(select('c'))

prefix_IAD
prefix_LAX
prefix_MIA
```

The 'concat' step has another form to consider where it can take a 'Traversal' as an argument for more dynamic use cases.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).as('v').
  values('code').concat(constant(' is in '), select('v').values('city'))

IAD is in Washington D.C.
LAX is in Los Angeles
MIA is in Miami
```

There is an easier way to produce the output shown above. We can use the 'format' step to produce that in a more readable way.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  format("%{code} is in %{city}")

IAD is in Washington D.C.
LAX is in Los Angeles
MIA is in Miami
```

Note how 'format' takes the incoming traverser, in this case an airport vertex, and resolve the variable expressions designated by the '%{ }' forms. Here the value in the curly braces is resolved to a property value. They can also be used to resolve to map keys and step names.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  elementMap().
  format("%{code} is in %{city}")

IAD is in Washington D.C.
LAX is in Los Angeles
```

```
MIA is in Miami
```

Another important feature of 'format' is the ability to add positional variables which can be transformed using a 'by' modulator. This feature makes 'format' an extremely flexible step.

```
g.V().has('airport','code',within('IAD','MIA','LAX')).
  format("%{_} is in %{_} and has %{_} outgoing routes").
    by('code').
    by('city').
    by(outE('route').count())

IAD is in Washington D.C. and has 136 outgoing routes
LAX is in Los Angeles and has 195 outgoing routes
MIA is in Miami and has 171 outgoing routes
```

It's worth noting that many of these steps can take a 'Scope' argument. By default, the scope is 'global', which means that the step operates on the value of the current traverser in the stream. In contrast, a 'local' scope implies that the incoming traverser in the stream contains a list of values on which the step will be applied. So far, we've only looked at 'global' scope usage of these steps, but 'local' scope allows for some interesting use cases.

```
// grab the first letter of each word in each city and lower case it
g.V().has('airport','code',within('IAD','MIA','LAX')).
  values('city').
  split(' ').
  substring(local,0,1).
  toLower(local)

[w,d]
[l,a]
[m]
```

3.15. Working with collections

Collections are containers for other values and refer to things like 'List', 'Set' and 'Map'. Gremlin offers a variety of steps that help construct and manipulate these objects. We've already seen how we can use a step like 'fold' to create a 'List' of the objects in the traversal stream:

```
g.V().has('region','GB-ENG').values('runways').dedup().fold()

[2,1,3,4]
```

As for maps, Gremlin can produce them whenever you use a step like 'group' or 'valueMap':

```
g.V().has('code','AUS').valueMap(true,'region')
```

```
[id:3,region:[US-TX],label:airport]
```

If you look at the result in the example above, collections might contain other collections, as shown with the region key where "US-TX" is in a 'List'. Since collections are core to the Gremlin language, the following steps tend to be quite helpful in many situations:

any	Determines if any object in a 'List' matches a specified predicate
all	Determines if all objects in a 'List' match a specified predicate
combine	Combines an incoming 'List' with the list provided as an argument
conjoin	Takes objects in the incoming 'List' and converts them to a string joined by the specified delimiter
difference	Calculates the difference between the incoming 'List' and the one provided as an argument
disjunct	Calculates the disjunct set between the incoming 'List' and the provided 'List' argument
intersect	Calculates the intersection between the incoming 'List' and the provided 'List' argument
merge	Merges the incoming collection with the one provided as an argument
none	Determines if no objects in a 'List' matches a specific predicate
product	Calculates the cartesian product of the incoming 'List' and the provided 'List'
reverse	Reverses the order of the incoming 'List'

The steps noted in the prior table are meant specifically for working with various collection types, but you should also note that many Gremlin steps inherently work with collections. For example, the 'local' versions of 'range', 'limit or 'tail' are good at taking parts of a collection or 'unfold' that can be used to deconstruct collections. These are some obvious examples, but as you learn more about Gremlin, you will find many more.

Let's take a deeper look at these steps and how you might use them. We will start with the three filtering steps of 'any' 'all' and 'none'. Oftentimes you will find yourself in a situation where you have written some Gremlin that collects your results into a 'List' like the following example where we have the number of runways for all airports in the "US-TX" region collected:

```
g.V().has('airport','region','US-TX').values('runways').fold()

[2,7,5,3,4,3,3,3,3,4,2,3,2,3,2,2,3,2,1,3,2,3,4,3,4,2,1]
```

If you were only interested in this result, if the 'List' had a value of "4" in it, you could use 'any' and the 'P.eq' predicate:

```
g.V().has('airport','region','US-TX').values('runways').fold().any(eq(4))

[2,7,5,3,4,3,3,3,3,4,2,3,2,3,2,2,3,2,1,3,2,3,4,3,4,2,1]
```

Similarly, if you only wanted the 'List' if all the values were greater than 7, then you could use 'all':

```
g.V().has('airport','region','US-TX').values('runways').fold().any(gt(7))

// no results
```

As the final complement to the 'all' and 'any' steps, 'none' can be used in cases where you only wanted the 'List' if none of its values were greater than 6:

```
g.V().has('airport','region','US-TX').values('runways').fold().none(gt(6))

// no results
```

If you need to push two 'List' objects together, you can use 'combine'. You can see a simple example of this step demonstrated as follows where we combine the incoming 'List' from 'fold' with a constant 'List' containing the value of 100:

```
g.V().has('airport','region','US-TX').values('runways').fold().combine([100])

[2,7,5,3,4,3,3,3,3,4,2,3,2,3,2,2,3,2,1,3,2,3,4,3,4,2,1,100]
```

A more advanced and likely useful form of 'combine', though, is to supply a traversal as the argument to dynamically choose the 'List' to combine. In the following example, we aggregate the runway values for all the airports in the "US-VA" region into "v", which is a 'List' and has the value '[1,1,2,2,2,3,4]'. We then traverse 'out' along the route edges and to three adjacent airport vertices. For each of those vertices, the 'map' step gets the runway value and does a 'fold' to produce a 'List' with just that value in it. As a final step, it uses 'combine' to push that single value 'List' together with the one stored in the aggregate "v" which we gather dynamically with 'select'.

```
g.V().has('airport','region',within('US-VA')).
  aggregate('v').by('runways').
  out('route').limit(3).
  map(values('runways').fold().combine(select('v')))

[1,4,2,2,2,2,3,1,1]
[3,4,2,2,2,2,3,1,1]
[2,4,2,2,2,2,3,1,1]
```

While we won't visit this form for all the 'List' steps we look at, you will notice that all of these sorts of 'List' transformation steps have both a literal constant form and this more dynamic form that

uses a traversal as the argument. We saw a similar pattern in <<textsteps>>.

The 'conjoin' step is interesting in that it is the only 'map'-like step that transforms the incoming 'List' to a different type. When you use 'conjoin' on a 'List', it will take the values within it, transform each to a string and then join them all together to a single string using the value that you specify as a delimiter. A simple use case for this feature might be to produce a delimited list of values, as that could be helpful for doing a fast export of data in a structured form.

```
g.V().has('airport','region','US-VA').
  valueMap('code','longest','city').by(unfold()).
  map(select(values).conjoin('\t'))

11500   IAD     Washington D.C.
9001    ORF     Norfolk
9003    RIC     Richmond
6800    ROA     Roanoke
6001    CHO     Charlottesville
8003    PHF     Newport News
6002    SHD     Staunton/Waynesboro/Harrisonburg
5799    LYH     Lynchburg
```

Gremlin also allows you to perform basic operations on sets with 'difference', 'disjunct', 'intersect' and 'product'. When you use these steps, it is important to recognize that they each accept an incoming 'List' or 'Set', but if it is a 'List', it will be treated as a 'Set'. The same can be said for the argument given to these steps. The output will be a 'Set'.

```
g.V().has('airport','region','US-VA').values('code').fold()

[IAD,ORF,RIC,ROA,CHO,PHF,SHD,LYH]

g.V().has('airport','region','US-VA').values('code').fold().
  difference(['SHD', 'MIA'])

[ORF,ROA,LYH,CHO,IAD,RIC,PHF]

g.V().has('airport','region','US-VA').values('code').fold().
  disjunct(['SHD', 'MIA'])

[ORF,MIA,ROA,LYH,CHO,IAD,RIC,PHF]

g.V().has('airport','region','US-VA').values('code').fold().
  intersect(['SHD', 'MIA'])

[SHD]

g.V().has('airport','region','US-VA').values('code').fold().
  product(['SHD', 'MIA'])

[[IAD,SHD],[IAD,MIA],[ORF,SHD],[ORF,MIA],[RIC,SHD],[RIC,MIA],[ROA,SHD],[ROA,MIA],
```

```
  [CHO,SHD],[CHO,MIA],[PHF,SHD],[PHF,MIA],[SHD,SHD],[SHD,MIA],[LYH,SHD],[LYH,MIA]]
```

We should not overlook the fact that these steps can also take a traversal as an argument as shown in the following example that uses 'aggregate' to collect all the airport codes in "US-VA" into a list side effect of "x" and then does an 'intersect' on that value against a list of codes for outgoing routes:

```
g.V().has('airport','region','US-VA').aggregate('x').by('code').
  out('route').values('code').fold().
  intersect(select('x'))

[ORF,ROA,CHO,IAD,RIC,SHD]
```

The 'reverse' step reverses the order of an incoming object. While its typical use would be for a list, it could also be used on a string.

```
g.V().has('airport','region','US-VA').values('code').fold()

[IAD,ORF,RIC,ROA,CHO,PHF,SHD,LYH]

g.V().has('airport','region','US-VA').values('code').fold().reverse()

[LYH,SHD,PHF,CHO,ROA,RIC,ORF,IAD]
```

When you need to merge two collections, you can use 'merge' step, which works for lists, sets and maps. This step is similar to 'combine' but does not allow duplicates.

```
g.V().has('airport','region','US-VA').values('code').fold().merge(['MIA','IAD'])

[ORF,MIA,ROA,LYH,CHO,IAD,RIC,SHD,PHF]

g.V().has('airport','region','US-VA').valueMap('code').merge([x: 10])

[x:10,code:[IAD]]
[x:10,code:[ORF]]
[x:10,code:[RIC]]
[x:10,code:[ROA]]
[x:10,code:[CHO]]
[x:10,code:[PHF]]
[x:10,code:[SHD]]
[x:10,code:[LYH]]
```

Once again, you may also provide a traversal argument to 'merge' to dynamically provide the object to merge to the incoming one.

```
g.V().has('airport','region','US-VA').aggregate('x').by('code').
  out('route').limit(5).values('code').fold().
```

```
  merge(select('x'))

[AOO,ORF,ROA,BGM,CHO,IAD,RIC,SHD,BKW,PHF,PBG,LYH,YTZ]
```

In the prior example, we gather the airport codes in "x" and then traverse 'out' along route edges and fold 5 of them to a list which we then 'merge' to "x" providing a single list output of both lists without duplicates.

In many Gremlin examples we've seen so far, the 'select' step has been used to reference traverser data at a particular step label or, in the example just prior, to get data designated as a side effect. The 'select' step is also capable of accessing 'Map' data and can do so in multiple ways. You can use it to get all the 'keys' or 'values' of a 'Map' with the 'Columns' enum:

```
g.V().has('airport','region','US-VA').valueMap('code','country').select(keys)

[country,code]
[country,code]
[country,code]
[country,code]
[country,code]
[country,code]
[country,code]
[country,code]

g.V().has('airport','region','US-VA').valueMap('code','country').select(values)

[[US],[IAD]]
[[US],[ORF]]
[[US],[RIC]]
[[US],[ROA]]
[[US],[CHO]]
[[US],[PHF]]
[[US],[SHD]]
[[US],[LYH]]
```

You can also use it to select individual values by their key:

```
g.V().has('airport','region','US-VA').valueMap('code','country').select('code')

[IAD]
[ORF]
[RIC]
[ROA]
[CHO]
[PHF]
[SHD]
[LYH]
```

The 'select' step supports more complex use cases as well. Let's envision a case where you had some external data that you wanted to integrate into your traversal results. For example, here's a 'Map' of average temperature data keyed by a city name.

```
temps = [Atlanta:66.2,Austin:70.9,Anchorage:42.6,Seattle:54.1]

Atlanta=66.2
Austin=70.9
Anchorage=42.6
Seattle=54.1
```

We can include that 'Map' as a side effect to the traversal and access it via 'select'. In the following example, the first select gets the 'Map' from the side effect and injects it into the traversal stream. The second 'select' gets the value in the Atlanta key of that 'Map'.

```
g.withSideEffect('t',temps).V().limit(1).
  select('t').select('Atlanta')

66.2
```

The ability to use 'select' in this fashion sets up the case for the data integration case we initially wanted to showcase. The following query uses 'project' to transform each city to a 'Map' with the city name going to the Location key and the AvgTemp key getting set with a combined group of 'select' steps.

```
g.withSideEffect('t',temps).V().
  has('city',within('Austin','Atlanta','Anchorage','Seattle')).
  values('city').as('c').
  project('Location','AvgTemp').
    by().
    by(select('t').select(select('c')))

[Location:Atlanta,AvgTemp:66.2]
[Location:Anchorage,AvgTemp:42.6]
[Location:Austin,AvgTemp:70.9]
[Location:Seattle,AvgTemp:54.1]
```

Let's break down that group of 'select' steps in the second 'by' modulator. We've already seen what 'select("t").select("Atlanta")' does in getting the temp map and grabbing the Atlanta key. When we replace Atlanta with 'select("c")', we're referencing the 'values("city")' step labeled with 'as("c")' and dynamically inserting its value as the key to extract from the 'Map', which ultimately allows us to supply that value to 'project'. In short, chaining and nesting 'select' steps in this way provides a powerful pattern for dynamically pulling values from side effect maps, enabling flexible data retrieval and transformation directly within the traversal.

The collection steps provide important utility functions for Gremlin. They allow you to do helpful

transformations to your data that can get your final result into the form your application ultimately needs. They also help make Gremlin queries read more concisely where they often replace patterns of other Gremlin steps that do the same job making queries much more readable.

3.16. Working with dates

From birthdays to auditing values like the "createdDate", applications often call you to store dates and times. There are several ways you might store dates in your graph:

1. A long value representing milliseconds since the epoch
2. An ISO-8601 formatted date stored as a string
3. A native 'DateTime' value (if your graph database supports this type)

All three work well in their own way, but if you want to use Gremlin's date steps, you will need an actual date type to use with them, which means that for the first two options you would need to use the 'asDate' step to convert them to date types.

```
g.inject(1690934400000).asDate()

2023-08-02T00:00Z

g.inject("2023-08-02T00:00:00Z").asDate()

2023-08-02T00:00Z
```

To construct a date in Gremlin, you can use the 'datetime' function, which takes an IOS-8601 formatted string as an argument:

```
g.inject(datetime("2023-08-02T00:00:00Z"))

2023-08-02T00:00Z
```

There are two date manipulation steps in Gremlin, 'dateAdd' and 'dateDiff'. The 'dateAdd' step adds some number of specified time units to a date value. In the following example we add 1 week to the given date.

```
g.inject("2023-08-02T00:00:00Z").asDate().dateAdd(DT.day, 7)

2023-08-09T00:00Z
```

By supplying a negative number of units, we can produce a subtraction operation as shown in the next example.

```
g.inject("2023-08-02T00:00:00Z").asDate().dateAdd(DT.day, -7)
```

```
2023-07-26T00:00Z
```

The 'DT' enum allows units of time for 'second', 'minute', 'hour and 'day'.

Date conversion in the above examples assume a time zone offset of UTC(+00:00).

The 'dateDiff' step is helpful when you want to find the difference between two dates in epoch time.

```
g.inject("2023-08-02T00:00:00Z").asDate().
  dateDiff(constant("2023-08-03T00:00:00Z").asDate())

-86400000
```

3.17. Testing values and ranges of values with P

We have already seen some ways of testing whether a value is within a certain range. Gremlin provides a number of different predicates that we can use to do range testing. The list below provides a summary of the available predicates. We will see each of these in use throughout this book.

Table 3. Predicates that test values or ranges of values

eq	Equal to
neq	Not equal to
gt	Greater than
gte	Greater than or equal to
lt	Less than
lte	Less than or equal to
inside	Inside a lower and upper bound, neither bound is included.
outside	Outside a lower and upper bound, neither bound is included.
between	Between two values inclusive/exclusive (upper bound is excluded)
within	Must match at least one of the values provided. Can be a range or a list
without	Must not match any of the values provided. Can be a range or a list
typeOf	Must match the specified data type

Table 4. Other predicates

and	Logical and conjunction
or	Logical or conjunction
not	Negation of a predicate

There are 'and', 'or' and 'not' predicates, but there are also 'and', 'or', and 'not'

> steps. It is often helpful to use the appropriate 'P' or '__' prefix respectively for clarity when using this syntax, but you may also find it required if your compiler cannot figure out which you intend to call.

The following queries demonstrate these capabilities being used in different ways. First, here are some examples of some of the direct compare steps such as 'gt' and 'gte' being used. The 'fold' step conveniently folds all of the results into a list for us.

```
// Airports with at least 5 runways
g.V().has('runways',gte(5)).values('code','runways').fold()
```

Here is the output that we might get from running the query.

```
[ATL,5,BOS,6,DFW,7,IAH,5,ORD,7,DEN,6,DTW,6,YYZ,5,
 AMS,6,SNN,5,MKE,5,MDW,5,GIS,5,HLZ,5,NPE,5,NSN,5,
 PPQ,5,TRG,5,UFA,5,KRP,5]
```

The next three queries show examples of 'lt', 'eq' and 'neq' being used.

```
// Airports with fewer than 3 runways
g.V().has('runways',lt(3)).values('code','runways').fold()

// How many airports have 3 runways?
g.V().has('runways',eq(3)).count()

// How many airports have anything but just 1 runway?
g.V().has('runways',neq(1)).count()
```

Note that in some cases, such as when using a simple 'has' step, the 'eq' is not required. For example, the query used above could be written as follows instead.

```
g.V().has('runways',3).count()
```

You could also write this query using an 'is' step. You will find the 'is' step used a lot in this book but mostly in conjunction with 'where' steps. To me the usage below does not feel as elegant as the 'has' step alternative used above.

```
// How many airports have 3 runways?
g.V().values('runways').is(3).count()
```

Here are examples of 'inside' and 'outside' being used.

```
// Airports with greater than 3 but fewer than 6 runways.
g.V().has('runways',inside(3,6)).values('code','runways')
```

```
// Airports with fewer than 3 or more than 6 runways.
g.V().has('runways',outside(3,6)).values('code','runways')
```

Below are some examples showing 'within' and 'without' being used.

```
// Airports with at least 3 but not more than 6 runways
g.V().has('runways',within(3..6)).values('code','runways').limit(15)

// Airports with 1,2 or 3 runways.
g.V().has('runways',within(1,2,3)).values('code','runways').limit(15)

// Airports with fewer than 3 or more than 6 runways.
g.V().has('runways',without(3..6)).values('code','runways').limit(15)
```

The 'between' step lets us test the provided value for being greater than or equal to a lower bound but less than an upper bound. The query below will find any airport that has 5,6 or 7 runways. In other words, any airport that has at least 5 but fewer than 8 runways.

```
// Airports with at least 5 runways but fewer than 8
g.V().has('runways',between(5,8)).values('code','runways').fold()
```

Here is the result of running the query.

```
[ATL,5,BOS,6,DFW,7,IAH,5,ORD,7,DEN,6,DTW,6,YYZ,5,
 AMS,6,SNN,5,MKE,5,MDW,5,GIS,5,HLZ,5,NPE,5,NSN,5,
 PPQ,5,TRG,5,UFA,5,KRP,5]
```

As with many queries we may build, there are several ways to get the same answer. Each of the following queries will return the same result. To an extent, which one you use comes down to personal preference, although in some cases one form of a query may be better than another for reasons of performance.

```
g.V().hasId(gt(0)).hasId(lte(46)).out().hasId(lte(46)).count()

g.V().hasId(within(1..46)).out().hasId(lte(46)).count()

g.V().hasId(within(1..46)).out().hasId(within(1L..46L)).count()

g.V().hasId(between(1,47)).out().hasId(lte(46)).count()

g.V().hasId(within(1..46)).out().hasId(between(1,47)).count()

g.V().hasId(inside(0,47)).out().hasId(lte(46)).count()
```

The values do not have to be numbers. We could also compare strings, for example.

Let's now look at a query that compares strings rather than numbers. The following query finds all airports located in the state of Texas in the United States but only returns their code if the name of the city the airport is located in is not 'Houston'.

```
g.V().has('airport','region','US-TX').
      has('city',neq('Houston')).
      values('code')
```

This next query can be used to find routes between Austin and Las Vegas. We use a 'within' step to limit the results we get back to just routes that have a plane change in Dallas, San Antonio or Houston airports.

```
g.V().has('airport','code','AUS').
      out().has('code',within('DFW','DAL','IAH','HOU','SAT')).
      out().has('code','LAS').path().by('code')
```

Here is what the query returns. It looks like we can change planes in Dallas or Houston, but nothing goes via San Antonio.

```
[AUS,DAL,LAS]
[AUS,DFW,LAS]
[AUS,IAH,LAS]
[AUS,SAT,LAS]
[AUS,HOU,LAS]
```

Conversely, if we wanted to avoid certain airports, we could use 'without' instead. This query again finds routes from Austin to Las Vegas but avoids any routes that go via Phoenix (PHX) or Los Angeles (LAX).

```
g.V().has('airport','code','AUS').
      out().has('code',without('PHX','LAX')).
      out().has('code','LAS').path().by('code')
```

Lastly, this query uses both within and without to modify the previous query to just airports within the United States or Canada as Austin now has a direct flight to London in England. We probably don't want to go that way if we are headed to Vegas!

```
g.V().has('airport','code','AUS').out().
      has('country',within('US','CA')).
      has('code',without('PHX','LAX')).out().
      has('code','LAS').path().by('code')
```

The 'within' and 'without' steps can take a variety of input types. For example, each of these queries will yield the same results.

```
// Range of values (inclusive, inclusive)
g.V().hasId(within(1..3))

// Explicit set of values
g.V().hasId(within(1,2,3))

// List of values
g.V().hasId(within([1,2,3]))
```

3.17.1. Refining flight routes analysis using 'not', 'neq', 'within' and 'without'

As we saw in the previous section, it is often useful to be able to specifically include or exclude values from a query. We have already seen a few examples of 'within' and 'without' being used in the section above. The following examples show additional queries that use 'within' and 'without' as well as some examples that use the 'neq' (not equal) and 'not' steps to exclude certain airports from query results.

The following query finds routes from AUS to SYD with only one stop but ignores any routes that stop in DFW.

```
g.V().has('airport','code','AUS').
      out().has('code',neq('DFW')).
      out().has('code','SYD').path().by('code')
```

We could also have written the query using 'not'

```
g.V().has('airport','code','AUS').
      out().not(values('code').is('DFW')).
      out().has('code','SYD').path().by('code')
```

Similar to the above but adding an 'and' clause to also avoid LAX.

```
g.V().has('airport','code','AUS').
      out().and(has('code',neq('DFW')),has('code',neq('LAX'))).
      out().has('code','SYD').path().by('code')
```

The above syntax uses the '__.and()' step as opposed to the 'P.and()' predicate. We can rewrite this query using the latter form, which is a bit more compact and readable.

```
g.V().has('airport','code','AUS').
      out().has('code',neq('DFW').and(neq('LAX'))).
```

```
      out().has('code','SYD').path().by('code')
```

We could also have written the prior query this way replacing 'and' with 'without'. This approach feels a lot cleaner, and it is straightforward to add more airports to the 'without' test as needed. We will look more at steps like 'and' in the "Boolean operations" section that is coming up soon.

```
// Flights to Sydney avoiding DFW and LAX
g.V().has('airport','code','AUS').
      out().has('code',without('DFW','LAX')).
      out().has('code','SYD').path().by('code')
```

Using 'without' is especially useful when you want to exclude a list of items from a query. This next query finds routes from San Antonio to Salt Lake City with one stop but avoids any routes that pass through a list of specified airports.

```
// How can I get from SAT to SLC but avoiding DFW,LAX,PHX and JFK ?
g.V().has('airport','code','SAT').
      out().has('code',without('DFW','LAX','PHX','JFK')).
      out().has('code','SLC').path().by('code')
```

In a similar way, 'within' allows us to specifically give a list of things that we **are** interested in. The query below again looks at routes from SAT to SLC with one stop, but this time only returns routes that stop in one of the designated airports.

```
// From AUS to SLC with a stop in any one of DFW,LAX,PHX or TUS
g.V().has('airport','code','SAT').
       out().has('code',within('DFW','LAX','PHX','TUS')).
       out().has('code','SLC').path().by('code')
```

Here is what the query returns.

```
[SAT,DFW,SLC]
[SAT,LAX,SLC]
[SAT,PHX,SLC]
```

Here are two more examples that use 'without' and 'within' to find routes based on countries.

```
// Flights from Austin to countries outside (without) the US and Canada
g.V().has('code','AUS').out().has('country',without('US','CA')).values('city')
```

Here is the output from running the query.

```
Cancun
```

```
Puerto Vallarta
Nassau
Cozumel
Guadalajara
Liberia
San José del Cabo
London
London
Frankfurt
Amsterdam
Mexico City
```

Here is a twist on the previous query that looks for destinations in Mexico or Canada that you can fly to non-stop from Austin.

```
// Flights from Austin to airports in (within) Mexico or Canada
g.V().has('code','AUS').out().has('country',within('MX','CA')).values('city')
```

This is what we get back from our new query.

```
Cancun
Puerto Vallarta
Cozumel
Guadalajara
San José del Cabo
Toronto
Vancouver
Calgary
Mexico City
```

Returning to the earlier use of 'neq', it is worth considering one subtlety of its behavior as compared to various forms of 'not' used in conjunction with 'eq'. Let's examine the differences with the following example where we start by getting a vertex count and a count of vertices that have a city property.

```
g.V().count()

3749

g.V().has('city').count()

3504
```

Recall that 'not' is both a step and a predicate. When used as a predicate from 'P', you essentially negate the state of the predicate given to it directly, so 'eq' simply becomes 'neq' as shown below.

```
g.V().has('city',neq('I-do-not-exist')).count()

3504

g.V().has('city',P.not(neq('I-do-not-exist'))).count()

0
```

The city property must exist for equality to be tested. We'll next see that this is not the case if we try to negate with the 'not' step.

```
g.V().not(has('city',eq('I-do-not-exist'))).count()

3749
```

Using 'not' step in the fashion demonstrates all 3749 vertices do not have a city property with the name "i-do-not-exist". The difference here is that with 'not' step negates the entirety of what 'has' is doing, accounting for a false if the city is not present at all or a false where the 'eq' is not satisfied. The predicate for of 'not', on the other hand, does not bother evaluating the predicate at all if the city property isn't present.

3.17.2. Using 'coin' and 'sample' to sample a dataset

In any sort of analysis work it is often useful to be able to take a sample, perhaps a pseudo random sample, of the data set contained within your graph. The 'coin' step allows you to do just that. It simulates a biased coin toss. You give 'coin' a value indicating how biased the toss should be. The value should be between 0 and 1, where 0 means there is no chance something will get picked (not that useful!), 1 means everything will get picked (also not that useful!) and 0.5 means there is an even 50/50 chance that an item will get selected.

The following query simply picks airports with a 50/50-coin toss and returns the airport code for the first 20 found.

```
// Pick 20 airports at random with an evenly biased coin (50% chance).
g.V().hasLabel('airport').coin(0.5).limit(20).values('code')
```

This next query is similar to the first but takes a subtly different approach. It will select a pseudo random sample of vertices from the graph and for each one picked, return its code and its elevation. Note that a tiny value of 0.05 (or 5% chance of success) is used for the 'coin' bias parameter this time. This has the effect that only a small number of vertices are likely to get selected, but there is a better chance they will come from all parts of the graph and avoids needing a 'limit' step. Of course, there is no guarantee how many airports this query will pick!

```
// Select some vertices at random and return them with their elevation.
g.V().hasLabel('airport').coin(0.05).values('code','elev').fold()
```

We can see how fairly the 'coin' step is working by counting the number of vertices returned. The following query should always return a count representing approximately half of the airports in the graph.

```
g.V().hasLabel('airport').coin(0.5).count()
```

If all you want is, say 20 randomly selected vertices, without worrying about setting the value of the coin yourself, you can use the 'sample' step instead.

```
g.V().hasLabel('airport').sample(20).values('code')
```

3.17.3. Using 'Math.random' to more randomly select a single vertex

While the 'sample' step allows you to select one or more vertices at random, in our testing, at least when using a TinkerGraph, it tends to favor vertices with lower index values. So, for example, in a test we ran this query 1000 times.

```
g.V().hasLabel('airport').sample(1).id()
```

What we found was that we always got back an ID of less than 200. This leads me to believe that the 'sample(1)' call is doing something similar to this

```
g.V().hasLabel('airport').coin(0.01).limit(1)
```

Look at the code below. Even if we run that simple experiment many times, it always gives results similar to these.

```
(1..10).each { println g.V().hasLabel('airport').sample(1).id().next()}

69
143
94
115
36
47
23
22
129
67
```

Given the air-routes graph has over 3,500 airport vertices, we wanted to come up with a query that gave a more likely result of picking any one airport from across all the possible airports in the graph. By taking advantage of the Java Math class we can do something that does seem to much more 'randomly' pick one airport from across all the possible airports. Take a look at the snippets of

Groovy/Gremlin code below.

More examples of using variables to store values and other ways to use additional Groovy classes and methods with Gremlin are provided in the "Making Gremlin even Groovier" and "Using a variable to feed a traversal" sections.

```
// How many airports are there?
numAirports = g.V().hasLabel('airport').count().next()

3504

// Pick a random airport ID
x=Math.round(numAirports*Math.random()) as Integer

1968

// Get the code for our randomly selected airport
g.V(x).values('code')

BSG
```

This experiment shows that the numbers being generated using the 'Math.random' approach appear to be a lot more evenly distributed across all the possible airports.

```
(1..10).each { println Math.round(numAirports*Math.random()) as Integer}

435
285
2189
3459
1355
624
3437
1656
1620
1896
```

Note that this approach only works unmodified with the 'air-routes' graph loaded into a TinkerGraph. This is because we know that the TinkerGraph implementation honors user-provided IDs and that in the 'air-routes' graph, airport IDs begin at one and are sequential with no gaps. However, you could easily modify this approach to work with other graphs without relying on knowing that the index values are sequential. For example, you could extract all the IDs into a list and then select one randomly from that list.

It is likely that this apparent lack of randomness is more specific to TinkerGraph and the fact that it will respect user-provided ID values, whereas other graph systems will probably store vertices in a more random order to begin with and therefore yield a better selection of airports from across the

graph.

If the airport IDs were not all known to be in a sequential order one after the other, we could create a list of all the airport IDs and then select one at random by doing something like this if we wanted to use our 'Math.random' technique.

```
airports = g.V().hasLabel('airport').id().toList()

numAirports = airports.size()

3504

x=Math.round(numAirports*Math.random()) as Integer

1754

g.V(airports[x-1])

v[1757]

g.V(airports[x-1]).values('code')

JTY
```

3.18. Checking data types with 'typeOf'

A quietly helpful predicate is 'typeOf' which can be used to help identify if an object is of a particular type or not. There are a number of cases where you might want to have this capability. For example, let's take a basic output of 'path' for a simple traversal:

```
g.V().has('airport','code','IAD').outE().inV().
  path().
  limit(3)

[v[10],e[6144][10-route->1053],v[1053]]
[v[10],e[6145][10-route->1059],v[1059]]
[v[10],e[6146][10-route->1060],v[1060]]
```

Each 'Path' in the above results is a mix of two vertices and a single edge. We could envision a long extension this query with many more 'outE().inV()' chains which would make each 'Path' result have an even longer vertex to edge pattern. There are limited ways to control what appears as a result of calling 'path()' and there is no way to omit certain objects from the middle of a path if the traversal had passed through them. If you only wanted to have the vertices in your 'Path', you would be forced to apply a filter. With the help of 'typeOf', the task becomes straightforward:

```
g.V().has('airport','code','IAD').outE().inV().
  path().
```

```
  limit(3).
  map(unfold().is(typeOf(VERTEX)).fold())

[v[10],v[1053]]
[v[10],v[1059]]
[v[10],v[1060]]
```

The 'typeOf' predicate accepts several values as an argument. In the above case, we used the 'GType' enum, which contains a listing of all the common types that Gremlin supports. That full listing is as follows:

Table 5. GType listing

BIGDECIMAL	BIGINT	BINARY
BOOLEAN	BYTE	CHAR
DATETIME	DOUBLE	DURATION
EDGE	FLOAT	GRAPH
INT	LIST	LONG
MAP	NULL	NUMBER
PATH	PROPERTY	SET
SHORT	STRING	TREE
UUID	VERTEX	VPROPERTY

The 'typeOf' predicate could also be used for graph administration functions, like for identifying data of the wrong type stored among properties.

```
g.V().has('airport','code','IAD').values('lat','lon')

-77.45580292
38.94449997
```

Given the schema for air-routes, we would expect our "lat" and "lon" values to be stored as 'double'. We could verify this with Gremlin with the help of 'typeOf':

```
g.V().has('airport','code','IAD').
  filter(values('lat','lon').is(P.not(typeOf(DOUBLE))))
```

3.19. Text search predicates

Among the '"predicates"' Gremlin has available are the ones used in performing more focused text searches. There are three predicates that search for the existence of one or more characters within a string of text and three that search for the non-existence of one or more characters.

Additional information on the text predicates can be found in the official Apache TinkerPop documentation here: https://tinkerpop.apache.org/docs/current/reference/#a-note-on-predicates

Table 6. Text searching predicates

startingWith	Match text that starts with the given character(s)
endingWith	Match text that ends with the given character(s)
containing	Match text that contains the given character(s)
regex	Match text using a regular expression
notStartingWith	Match text that does not start with the given character(s)
notEndingWith	Match text that does not end with the given character(s)
notContaining	Match text that does not contain the given character(s)
notRegex	Match text that does not match a regular expression

In the sections below you will find examples of each predicate being used. Each predicate is case-sensitive, so bear that in mind as you use them. To do a case-insensitive search, you can chain multiple steps together combined by 'or' step.

All of these predicates are ***case-sensitive***.

Most TinkerPop enabled graph stores that you are likely to use for any sort of serious deployment will also be backed by an indexing technology like Solr or Elasticsearch. In those cases, some number of more sophisticated search methods will likely be made available to you. You should always check the documentation for the system you are using to see what is recommended.

These predicates add to the existing Gremlin predicates that we looked at in the "Testing values and ranges of values with P" section.

3.19.1. startingWith

The text that you search for can be one or more characters. Here is a simple example that looks for unique city names that begin with an uppercase "X".

```
g.V().hasLabel('airport').
      has('city',startingWith('X')).
      values('city')
```

As expected, when run we get back a set of names all beginning with an "X".

```
Xiamen
Xianyang
Xuzhou
```

```
Xilinhot
Xiangfan
Xining
Xalapa
Xieng Khouang
Xiahe
Xiaguan
Xichang
Xingyi
Xinyuan
Xigaze
Xinyang
```

The example below looks for any cities with names starting with "Dal". A 'dedup' step is used to get rid of any duplicate names in the results.

```
g.V().hasLabel('airport').
      has('city',startingWith('Dal')).
      values('city').
      dedup().
      fold()
```

When run, the query finds all the city names in the graph that begin with the characters "Dal" as expected.

```
[Dallas,Dalaman,Dalian,Dalcahue,Dalat,Dalanzadgad]
```

As we mentioned, all the text predicates are case-sensitive. If we were to search for city names starting with the characters "dal" we would not find any matches. The query below demonstrates this.

```
g.V().hasLabel('airport').
      has('city',startingWith('dal')).
      count()

0
```

Given the predicates are case-sensitive, if, for example, you need to find matches for both 'Dal' or 'dal', you can do that as shown below using an 'or' step and two 'has' steps.

```
g.V().hasLabel('airport').
      or(has('city',startingWith('dal')),
         has('city',startingWith('Dal'))).
      dedup().by('city').
      count()
```

```
6
```

3.19.2. endingWith

The example below looks for any city names ending with the characters "zhi".

```
g.V().hasLabel('airport').
      has('city',endingWith('zhi')).
      values('city')

Changzhi
```

3.19.3. containing

We can also look for cities whose names contain a certain string of one or more characters. The example below looks for any cities with the string "gzh" in their name.

```
g.V().hasLabel('airport').
      has('city',containing('gzh')).
      values('city')
```

When run, the query produces the following results.

```
Guangzhou
Hangzhou
Zhengzhou
Changzhi
Changzhou
Yongzhou
Yangzho
```

3.19.4. regex

For more advanced text matching scenarios you can use regular expressions as part of a query. Note that the first example below could also be achieved using a Gremlin 'within' step as it is still really doing exact string comparisons, but it gives us a template for how to write any query containing a regular expression. The example that follows finds all airports in cities with names that begin with 'Dal', so it will find Dallas, Dalaman, Dalian, Dalcahue, Dalat and Dalanzadgad!.

```
// Using a filter to search using a regular expression
g.V().has('airport','type','airport').
      has('city', TextP.regex('Dallas|Austin')).values('code')


// A regular expression to find any airport with a city name that begins with "Dal"
```

```
g.V().has('airport','type','airport').
      has('city', TextP.regex('^Dal\\w*')).values('city')
```

Gremlin adheres to the regex syntax prescribed by the Java `Pattern` class documented at https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/regex/Pattern.html The Java regular expression syntax may be different than the one you are used to so it is worth taking a few minutes to study the documentation at that URL.

3.19.5. notStartingWith

Each of the text predicates has an inverse step. We can use the 'notStartingWith' predicate to look for city names that do not start with "Dal".

```
g.V().hasLabel('airport').
      has('city',notStartingWith('Dal')).
      count()

3497
```

The example above returns the same results we would get if we were to negate a 'startingWith' predicate as shown below.

```
g.V().hasLabel('airport').
      not(has('city',startingWith('Dal'))).
      count()

3497
```

3.19.6. notEndingWith

Using 'notEndingWith', we can easily find cities whose names do not end with "zhi".

```
g.V().hasLabel('airport').
      has('city',notEndingWith('zhi')).
      count()

3503
```

3.19.7. notContaining

The query below counts the number of cities that do not contain the string "berg" in their name.

```
g.V().hasLabel('airport').
      has('city',notContaining('berg')).
      count()
```

```
3500
```

Let's now do something a little more interesting. The query below chains together a number of 'has' steps using 'notContaining' and 'containing' predicates to find cities with names containing no basic, lowercase, vowels commonly used in the English language but containing either of the secondary vowels.

```
g.V().hasLabel('airport').
      has('city',notContaining('e')).
      has('city',notContaining('a')).
      has('city',notContaining('i')).
      has('city',notContaining('u')).
      has('city',notContaining('o')).
      or(has('city',containing('y')),
         has('city',containing('h'))).
      values('city').
      dedup()
```

Only two results are found. Note that one of the results does contain a vowel, but it is an uppercase "O" and as such is allowed by the constraints that we specified.

```
Brønnøy
Osh
Kyzyl
```

3.19.8. notRegex

The 'notRegex' predicate is mostly present for Gremlin language symmetry as regular expressions can naturally express negations itself.

```
// A regular expression to find airports not in Dallas or Austin
g.V().has('airport','type','airport').
      has('city', TextP.notRegex('Dallas|Austin')).values('code')
```

3.20. Sorting things – introducing 'order'

You can use 'order' to sort things in either ascending (the default) or descending order. Note that the sort does not have to be the last step of a query. It is perfectly OK to sort things in the middle of a query before moving on to a further step. We can see examples of that in the first two queries below. Note that the first query will return different results than the second one due to the placement of the 'limit' step. We used 'fold' at the end of the query to collect all the results into a list. The 'fold' step can also do more than this. It provides a way of doing the 'reduce' part of map-reduce operations. We will see some other examples of its use elsewhere in this book, such as in the "Using 'fold' to do simple Map-Reduce computations" section.

```
// Sort the first 20 airports returned in ascending order
g.V().hasLabel('airport').limit(20).values('code').order().fold()

[ANC,ATL,AUS,BNA,BOS,BWI,DCA,DFW,FLL,IAD,IAH,JFK,LAX,LGA,MCO,MIA,MSP,ORD,PBI,PHX]
```

As above, but this time perform the 'limit' step after the 'order' step.

```
// Sort all of the airports in the graph by their code and then return the first 20
g.V().hasLabel('airport').order().by('code').limit(20).values('code').fold()

[AAA,AAE,AAL,AAN,AAQ,AAR,AAT,AAX,AAY,ABA,ABB,ABD,ABE,ABI,ABJ,ABL,ABM,ABQ,ABR,ABS]
```

Here is a similar example to the previous two. We find all the places you can fly to from Austin (AUS) and sort the results as before, using the airport's IATA code, but this time we also include the ICAO code for each airport in the result set.

```
g.V().has('code','AUS').out().order().by('code').
                        values('code','icao').fold()
```

Here are the results from running the query.

```
[ABQ,KABQ,AMA,KAMA,AMS,EHAM,ASE,KASE,ATL,KATL,BHM,KBHM,BKG,KBBG,BNA,KBNA,BOI,KBOI,
 BOS,KBOS,BTR,KBTR,BUF,KBUF,BUR,KBUR,BWI,KBWI,BZN,KBZN,CHS,KCHS,CLE,KCLE,CLT,KCLT,
 CMH,KCMH,CUN,MMUN,CVG,KCVG,CZM,MMCZ,DAL,KDAL,DCA,KDCA,DEN,KDEN,DFW,KDFW,DSM,KDSM,
 DTW,KDTW,ECP,KECP,ELP,KELP,EWR,KEWR,FLL,KFLL,FRA,EDDF,GDL,MMGL,GRR,KGRR,HDN,KHDN,
 HNL,PHNL,HOU,KHOU,HRL,KHRL,IAD,KIAD,IAH,KIAH,IND,KIND,JAX,KJAX,JFK,KJFK,LAS,KLAS,
 LAX,KLAX,LBB,KLBB,LGB,KLGB,LGW,EGKK,LHR,EGLL,LIR,MRLB,LIT,KLIT,MCI,KMCI,MCO,KMCO,
 MDW,KMDW,MEM,KMEM,MEX,MMMX,MIA,KMIA,MKE,KMKE,MSP,KMSP,MSY,KMSY,NAS,MYNN,OAK,KOAK,
 OKC,KOKC,OMA,KOMA,ONT,KONT,ORD,KORD,PDX,KPDX,PHL,KPHL,PHX,KPHX,PIE,KPIE,PIT,KPIT,
 PNS,KPNS,PVD,KPVD,PVR,MMPR,RDU,KRDU,RNO,KRNO,SAN,KSAN,SAT,KSAT,SDF,KSDF,SEA,KSEA,
 SFB,KSFB,SFO,KSFO,SJC,KSJC,SJD,MMSD,SLC,KSLC,SMF,KSMF,SNA,KSNA,STL,KSTL,TPA,KTPA,
 TUL,KTUL,TUS,KTUS,TYS,KTYS,VPS,KVPS,XNA,KXNA,YVR,CYVR,YYC,CYYC,YYZ,CYYZ]
```

By default, a sort performed using 'order' returns results in ascending order. To obtain results in descending order instead, 'desc' can be specified using a `by` modulator. Likewise, 'asc' can be used to make it clear that sorting in ascending order is required.

```
// Sort the first 20 airports returned in descending order
g.V().hasLabel('airport').limit(20).values('code').order().by(desc).fold()

[PHX,PBI,ORD,MSP,MIA,MCO,LGA,LAX,JFK,IAH,IAD,FLL,DFW,DCA,BWI,BOS,BNA,AUS,ATL,ANC]
```

You can also sort things into a random order using 'shuffle'. Take a look at the example below and the output it produces.

```
g.V().hasLabel('airport').limit(20).values('code').order().by(shuffle).fold()

[BWI,MSP,PHX,DCA,IAH,LAX,DFW,BNA,BOS,MCO,ORD,LGA,AUS,PBI,MIA,FLL,ATL,JFK,IAD,ANC]
```

It is important to call attention to the use of 'by' modulators with 'order' as they introduce some special semantics when they are unproductive, a topic we learned about in the "Traits of 'by' modulators" section. Let's modify an earlier example to make the 'by' unproductive by introducing a spelling mistake to the property key we want to order on:

```
g.V().has('code','AUS').out().
  order().by('cde').
  values('code','icao').fold()

[]
```

As you can see, an unproductive 'by' makes 'order' behave like a filter for those traversers that can't make use of '"cde"' (which in this case is all of them). This filtering semantic is often quite helpful when working with heterogeneous elements by providing some flexibility in the face of what could otherwise just be an exception. These semantics could lead to a temptation to use them to explicitly filter in the 'by' as follows:

```
g.V().has('code','AUS').out().
  order().by(__.as('v').values('runways').is(gt(4)).select('v').values('code')).
  values('code','icao').fold()

[ATL,KATL,BOS,KBOS,DEN,KDEN,DFW,KDFW,DTW,KDTW,IAH,KIAH,MDW,KMDW,ORD,KORD,YYZ,CYYZ]
```

While the prior example works, it is not an advisable approach because it reduces the readability of the query. It would be far more idiomatic to write the above query with an explicit 'has' filter as shown next:

```
g.V().has('code','AUS').
  out().has('runways',gt(4)).
  order().by('code').
  values('code','icao').fold()

[AMS,EHAM,ATL,KATL,BOS,KBOS,DEN,KDEN,DFW,KDFW,DTW,
 KDTW,IAH,KIAH,MDW,KMDW,MKE,KMKE,ORD,KORD,YYZ,CYYZ]
```

Below is an example where we combine the field we want to sort by 'longest' and the direction we want the sort to take, 'desc' into a single 'by' instruction.

```
// List the 10 airports with the longest runways in decreasing order.
g.V().hasLabel('airport').order().by('longest',desc).valueMap().
```

```
      select('code','longest').limit(10)
```

Here is the output from running the query. To save space, we have split the results into two columns.

```
[code:[BPX],longest:[18045]]
[code:[RKZ],longest:[16404]]
[code:[ULY],longest:[16404]]
[code:[UTN],longest:[16076]]
[code:[DEN],longest:[16000]]
[code:[DOH],longest:[15912]]
[code:[GOQ],longest:[15748]]
[code:[HRE],longest:[15502]]
[code:[FIH],longest:[15420]]
[code:[ZIA],longest:[15092]]
```

Let's look at another way we could have coded the query we used earlier to find the longest runway in the graph. As you may recall, we used the following query. While the query does indeed find the longest runway in the graph, if we wanted to know which airport or airports had runways of that length, we would have to run a second query to find them.

```
g.V().hasLabel('airport').values('longest').max()
```

Now that we know how to sort things, we could write a slightly more complex query that sorts all the airports by the longest runway in descending order and returns the 'valueMap' for the first of those. While this query could probably be written more efficiently and also improved to handle cases where more than one airport has the longest runway, it provides a nice example of using 'order' to find an airport that we are interested in.

```
g.V().hasLabel('airport').order().by(values('longest'),desc).limit(1).valueMap()
```

In the case of the 'air-routes' graph, there is only one airport with the longest runway. The runway at the Chinese city of Bangda is 18,045 feet long. The reason the runway is so long is due to the altitude of the airport, which is located 14,219 feet above sea level. Aircraft need a lot more runway to operate safely at that altitude!

```
[country:[CN],longest:[18045],code:[BPX],city:[Bangda],lon:[97.1082992553711],type:[ai
rport],elev:[14219],icao:[ZUBD],region:[CN-54],runways:[1],lat:[30.5536003112793
],desc:[Qamdo Bangda Airport]]
```

3.20.1. Sorting by key or value

Sometimes, when the results of a query are a set of one or more key:value pairs, we need to sort by either the key or the value in either ascending or descending order. Gremlin offers us ways that we

can control the sort in these cases. Examples of how these work are shown below.

The following example shows the difference between running a query with and without the use of 'order' to sort using the keys of the map created by the 'group' step.

```
// Query but do not order
g.V().hasLabel('airport').limit(5).group().by('code').by('runways')

[BNA:[4],ANC:[3],BOS:[6],ATL:[5],AUS:[2]]
```

Notice also how 'local' is used as a parameter to 'order'. This is required so that the ordering is done while the final list is being constructed. If you do not specify 'local', then 'order' will have no effect as it will be applied to the entire result which is treated as a single entity at that point.

```
// Query and order by airport code (the key)
g.V().hasLabel('airport').limit(5).
      group().by('code').by('runways').
      order(local).by(keys,asc)

[ANC:[3],ATL:[5],AUS:[2],BNA:[4],BOS:[6]]
```

In this example we make the numbers of runways the key field and sort on it in descending order.

```
g.V().hasLabel('airport').limit(10).
      group().by('runways').by('code').
      order(local).by(keys,desc)

[7:[DFW],6:[BOS],5:[ATL],4:[BNA,IAD],3:[ANC,BWI,DCA],2:[AUS,FLL]]
```

3.21. Boolean operations

Gremlin provides a set of logical operators such as 'and', 'or' and 'not' that can be used to form Boolean (true/false) type queries. In a lot of cases we find that 'or' can be avoided by using 'within', for example, and that 'not' can sometimes be avoided by using 'without', but it is still good to know that these operators exist. The 'and' operator can sometimes be avoided by chaining 'has' steps together. That said, there are always cases where having these boolean steps available is extremely useful.

```
// Simple example of doing a Boolean AND operation
g.V().and(has('code','AUS'),has('icao','KAUS'))

// Simple example of doing a Boolean OR operation
g.V().or(has('code','AUS'),has('icao','KDFW'))
```

You can also use 'and' in an infix way as follows so long as you only want to 'and' two traversals

together.

```
g.V().has('code','AUS').and().has('icao','KAUS')
```

As you would probably expect, an 'or' step can have more than two choices. This one below has four. Also, note that in practice, for this query, using 'within' would be a better approach, but this suffices as an example of a bigger 'or' expression.

```
g.V().hasLabel('airport').or(has('region','US-TX'),
                             has('region','US-LA'),
                             has('region','US-AZ'),
                             has('region','US-OK')).
                           order().by('region',asc).
                           valueMap().select('code','region')
```

Using 'within' the example above could be written like this, so always keep in mind that using 'or' may not always be the best approach to use for a given query. We will look more closely at the 'within' and 'without' steps in the following section.

```
g.V().hasLabel('airport').has('region',within('US-TX','US-LA','US-AZ','US-OK')).
                          order().by('region',asc).
                          valueMap().select('code','region')
```

This next example uses an 'and' step to find airports in Texas with a runway at least 12,000 feet long.

```
g.V().hasLabel('airport').and(has('region','US-TX'),
                              has('longest',gte(12000))).values('code')
```

As with the 'or' step, using 'and' is not always necessary. We could rewrite the previous query as follows.

```
g.V().has('region','US-TX').has('longest',gte(12000))
```

Gremlin also provides a 'not' step which works as you would expect. This query finds vertices that are not airports.

```
g.V().not(hasLabel('airport')).count()
```

This previous query could also be written as follows.

```
g.V().has(label,neq('airport')).count()
```

Depending on the model of your graph and the query itself, it may or may not make sense to use the boolean steps. Sometimes, as described above, chaining 'has' steps together may be more efficient or using a step like 'within' or 'without' may make more sense.

Boolean steps such as 'and' can also be dot combined as the example below shows. This query finds all the airports that have fewer than 100 outbound routes but more than 94 and returns them grouped by airport code and route count. Notice how in this case the 'and' step is added to the 'lt' step using a dot rather than having the 'and' be the containing step for the whole test. The results from running the query are shown below as well.

```
g.V().hasLabel('airport').
      where(out().count().is(lt(100).and(gt(94)))).
      group().by('code').by(out().count())

[RIX:96,DCA:96,VCE:99,GRU:97,SFB:95,AUS:98,OPO:95]
```

The query we just looked at could also be written as follows, but in this case using the 'and' step inline by dot combining it (as above) feels cleaner to me. As you can see, we get the same result as before.

```
g.V().hasLabel('airport').where(and(out().count().is(lt(100)),
                out().count().is(gt(94)))).
                group().by('code').by(out().count())

[RIX:96,DCA:96,VCE:99,GRU:97,SFB:95,AUS:98,OPO:95]
```

As we have pointed out several times already, there are often many ways to write a query that will produce the same result. Here is an example of the previous two queries rewritten to use a 'between' step instead of an 'and' step. Remember that 'between' is inclusive/exclusive, so we have to specify 101 as the upper bound and 95 as the lower bound.

```
g.V().hasLabel('airport').where(out().count().is(between(95,101))).
                          group().by('code').by(out().count())

[RIX:96,BRS:100,DCA:96,KBP:100,VCE:99,GRU:97,AYT:100,SFB:95,AUS:98,OPO:95]
```

Just for fun, here is the same query but rewritten to use an 'inside' step.

```
g.V().hasLabel('airport').where(out().count().is(inside(94,100))).
                          group().by('code').by(out().count())

[RIX:96,DCA:96,VCE:99,GRU:97,SFB:95,AUS:98,OPO:95]
```

As a side note, if we wanted to reverse the grouping so that the airports were grouped by the counts

rather than the codes, we could do that as follows.

```
g.V().hasLabel('airport').
      where(out().count().is(lt(100).and(gt(94)))).
      group().by(out().count()).by('code')

[96:[DCA,RIX],97:[GRU],98:[AUS],99:[VCE],95:[OPO,SFB]]
```

You can also add additional inline 'and' steps to a query as shown below. Notice that this time 'AUH' and 'NCE' are not part of the result set as they have 97 routes which our new 'and' test eliminates.

```
g.V().hasLabel('airport').
      where(out().count().is(lt(100).and(gt(94)).and(neq(97)))).
      group().by(out().count()).by('code')

[96:[DCA,RIX],98:[AUS],99:[VCE],95:[OPO,SFB]]
```

The 'where' step was used a lot in the examples above. Hopefully, the effect of using it was clear. Nonetheless, we will explain in more detail how the 'where' step works in the next section.

3.22. Using 'where' to filter things out of a result

We have already seen the 'where' step used in some of the prior examples. In this section we will take a slightly more focussed look at the 'where' step. The 'where' step is an example of a 'filter'. It takes the current state of a traversal and only allows anything that matches the specified constraint to pass on to any following steps. 'Where' can be used by itself, or as we shall see later, in conjunction with a 'by' modulator or following a 'match' or a 'choose' step.

It is worth noting that some queries that use 'where' can also be written using 'has' instead.

Let's start by looking at a simple example of how 'has' and 'where' can be used to achieve the same result.

```
// Find airports with more than five runways
g.V().has('runways',gt(5))

// Find airports with more than five runways
g.V().where(values('runways').is(gt(5)))
```

In examples like the one above, both queries will yield the exact same results, but the 'has' step feels simpler and cleaner for such cases. Notice how in the 'where' step version the 'gt' predicate has to be placed inside an 'is' step. The next example starts to show the real power of the 'where' step. We can include a traversal inside the 'where' step that does some filtering for us. In this case, we first find all vertices that are airports and then use a 'where' step to only keep airports that have

more than 60 outgoing routes. Finally, we count how many such airports we found.

```
// Airports with more than 60 unique routes from them
g.V().hasLabel('airport').where(out('route').count().is(gt(60))).count()

215
```

In our next example, we want to find routes between airports that have an ID of less than 47 but only return routes that are longer than 4,000 miles. Again, notice how we are able to look at the incoming vertex ID values by placing a reference to 'inV' at the start of the 'where' expression. The 'where' step is placed inside 'and' step so that we can also examine the 'dist' property of the edge. Finally, we return the path as the result of our query.

```
// Routes longer than 4,000 miles between airports with and ID less than 47
g.V().hasId(lt(47)).outE().
      and(where(inV().id().is(lt(47))),values('dist').is(gt(4000))).
      inV().path().by('code').by('dist')
```

Below is what we get back as a result of running the query.

```
[ATL,4502,HNL]
[BOS,5083,HNL]
[IAD,4806,HNL]
[JFK,4970,HNL]
[ORD,4230,HNL]
[EWR,4950,HNL]
[HNL,4502,ATL]
[HNL,5083,BOS]
[HNL,4806,IAD]
[HNL,4970,JFK]
[HNL,4230,ORD]
[HNL,4950,EWR]
```

Sometimes you will be looking for results that match the inverse condition of a 'where' step. One way this can be achieved is to wrap the 'where' step inside a 'not' step, as shown below.

The double underscore prefix '"__."' before the 'in' step is required as 'in' is a reserved word in Groovy. If you are using the Gremlin Console and do not include the prefix, you will get an error. This is explained in more detail in the "A warning about reserved word conflicts and collisions" section a bit later. The '"__."' notation is actually a reference to a special TinkerPop Java class that has the name '"__"' (double underscore) but don't worry about that for the time being. In fact, for now, think of '"__."' as meaning '"the result of the previous step"'. This will become important once we start looking at writing a Java program to issue Gremlin queries and is discussed in the "The Apache TinkerPop interfaces and classes" section quite a bit later on.

The query starts by finding the Austin airport (AUS) and then finds all outgoing routes. We then look at all incoming 'contains' edges to see which country each airport we can fly to is in. The 'not' step ensures that only airports that do not have the United States country code of 'US' are selected.

```
g.V().has('airport','code','AUS').
      out().not(where(__.in('contains').has('code','US'))).
      valueMap('code','city')
```

As you can see, when we run the query, only destinations outside the United States are returned.

```
[code:[CUN],city:[Cancun]]
[code:[PVR],city:[Puerto Vallarta]]
[code:[NAS],city:[Nassau]]
[code:[CZM],city:[Cozumel]]
[code:[GDL],city:[Guadalajara]]
[code:[LIR],city:[Liberia]]
[code:[SJD],city:[San José del Cabo]]
[code:[YYZ],city:[Toronto]]
[code:[YVR],city:[Vancouver]]
[code:[LHR],city:[London]]
[code:[LGW],city:[London]]
[code:[FRA],city:[Frankfurt]]
[code:[AMS],city:[Amsterdam]]
[code:[YYC],city:[Calgary]]
[code:[MEX],city:[Mexico City]]
```

This pattern comes in useful whenever you want to use a traversal inside 'where' step and negate the results. Of course, there are other ways we could write this particular query, but we wanted to show an example of this technique being used. For completeness, two simpler ways of writing this query that do not use 'where' at all are shown below, but there will be cases where combining 'not' and 'where' are your best option.

```
g.V().has('airport','code','AUS').
      out().has('country', neq('US')).valueMap('code','city')

g.V().has('airport','code','AUS').
      out().not(has('country', 'US')).valueMap('code','city')
```

It is also possible to use some special forms of the 'and' and 'or' steps when working with a 'where' step. Take a look at the query below. This will match airports that you can fly to from Austin (AUS) so long as they have more than four runways and do not have exactly six runways.

```
g.V().has('airport','code','AUS').out().
      where(values('runways').is(gt(4).and(neq(6)))).
      valueMap('code','runways')
```

Here is what the query returns. As you can see, we only found airports that you can fly to from AUS that have 5,7 or 8 runways.

```
[code:[MKE],runways:[5]]
[code:[MDW],runways:[5]]
[code:[ATL],runways:[5]]
[code:[DFW],runways:[7]]
[code:[IAH],runways:[5]]
[code:[ORD],runways:[7]]
[code:[YYZ],runways:[5]]
```

The same is true for the 'or' step. We could rewrite our query to find airports we can fly to from Austin that have more than six or exactly four runways.

```
g.V().has('airport','code','AUS').out().
      where(values('runways').is(gt(6).or(eq(4)))).
      valueMap('code','runways')
```

The above is a shorter form of the following query that demonstrates another way we could use the boolean operators within a 'where' step. In this case only two traversals are allowed to be compared using the boolean operator (in this case an 'or' step).

```
g.V().has('airport','code','AUS').out().
      where(values('runways').is(gt(6)).or().values('runways').is(4)).
      valueMap('code','runways')
```

Our last example in this section uses a 'where' step to make sure we don't end up back where we started when looking at airline routes with one stop. We find routes that start in Austin, with one intermediate stop, that then continue to another airport but never end up back in Austin. A 'limit' step is used to just return the first 10 results that match this criteria. Notice how the 'as' step is used to label the 'AUS' airport so that we can refer to it later in our 'where' step. The effect of this query is that routes such as AUS→DFW→AUS will not be returned, but AUS→DFW→LHR will be as it does not end up back in Austin.

```
// List 10 places you can fly to with one stop, starting at Austin but
// never ending up back in Austin
g.V().has('airport','code','AUS').as('a').
      out().out().where(neq('a')).
      path().by('code').limit(10)
```

As you work with Gremlin, you will find that the 'where' step is one that you use a lot. You will see many more examples of 'where' being used throughout the remainder of this book. In the next section we will look at some additional ways that 'where' can be used.

3.22.1. Using 'where' and 'by' to filter results

A 'where' step can be followed with a 'by' modulator. The query below starts at the Austin airport and finds all the airports that you can fly to from there. A 'where' 'by' step is then used to filter the results to just those airports that have the same number of runways that Austin has. What is really nice about this is that we do not have to know ahead of time how many runways Austin itself has since the query handles that for us.

```
g.V().has('code','AUS').as('a').out().
      where(eq('a')).by('runways').valueMap('code','runways')
```

Combining the 'where' and 'by' steps allows you to write powerful queries in a nice and simple way.

If you were to run the query in the Gremlin Console, these are the results that you should see. Note that all the airports returned have two runways. This is the same number of runways that the Austin airport has.

```
[code:[ONT],runways:[2]]
[code:[CUN],runways:[2]]
[code:[NAS],runways:[2]]
[code:[JAX],runways:[2]]
[code:[PVD],runways:[2]]
[code:[SMF],runways:[2]]
[code:[BHM],runways:[2]]
[code:[BUF],runways:[2]]
[code:[CMH],runways:[2]]
[code:[DSM],runways:[2]]
[code:[CZM],runways:[2]]
[code:[AMA],runways:[2]]
[code:[CHS],runways:[2]]
[code:[GDL],runways:[2]]
[code:[PNS],runways:[2]]
[code:[TYS],runways:[2]]
[code:[VPS],runways:[2]]
[code:[BUR],runways:[2]]
[code:[FLL],runways:[2]]
[code:[SNA],runways:[2]]
[code:[MSY],runways:[2]]
[code:[LHR],runways:[2]]
[code:[LGW],runways:[2]]
[code:[MEX],runways:[2]]
```

The ability to combine 'where' and 'by' steps allows us to avoid having to write the previous query in more complicated ways such as the one shown below.

```
g.V().has('code','AUS').as('a').out().as('b').
```

```
      filter(select('a','b').by('runways').where('a',eq('b'))).
      valueMap('code','runways')
```

There is also a two-parameter form of the 'where' step that you may have noticed used above. In this case the first parameter refers to a label defined earlier in the query. Take a look at the example below. We find the vertex for the Austin (AUS) airport and label it "a". We then look at all the airports you can fly to from there and label them "b". We then use a 'where' step to compare "a" and "b". Only airports with fewer runways than Austin should be returned.

```
g.V().has('airport','code','AUS').as('a').out().as('b').
      where('a',gt('b')).by('runways').valueMap('code','runways')
```

Austin has two runways, so only airports with one runway return for this query.

```
[code:[PVR],runways:[1]]
[code:[ECP],runways:[1]]
[code:[LIR],runways:[1]]
[code:[SJD],runways:[1]]
[code:[XNA],runways:[1]]
[code:[HDN],runways:[1]]
[code:[ASE],runways:[1]]
[code:[BKG],runways:[1]]
[code:[SAN],runways:[1]]
```

It is also possible to compare two different properties by adding a second 'by' modulator. This is useful when vertex properties have different key names but may contain the same values. The query below is definitely contrived, you could achieve the same thing in a more simple way, but it does demonstrate two 'by' modulators being used. The 'country' property of an airport vertex is compared with the 'code' property of a 'country' vertex. The query first finds any airport vertex with a 'city' property containing the string 'London'. Next, any connecting 'country' vertices (ones connected by contains edges) are found. The 'where' test compares the country code value of the two vertices. Lastly, a 'select' is used to pick the results that we want to return.

```
g.V().has('airport','city','London').as('a','r').
      in('contains').as('b').
      where('a',eq('b')).by('country').by('code').
      select('a','r','b').by('code').by('region')
```

When the query is run, we get back all the airports with a city name of 'London' along with their region code and country code.

```
[a:LHR,r:GB-ENG,b:UK]
[a:LGW,r:GB-ENG,b:UK]
[a:LCY,r:GB-ENG,b:UK]
[a:STN,r:GB-ENG,b:UK]
```

```
[a:LTN,r:GB-ENG,b:UK]
[a:YXU,r:CA-ON,b:CA]
```

As we mentioned, the above query was used just as an example. In reality, the following query that does not use any 'where' steps would have sufficed in this case.

```
g.V().has('airport','city','London').
      valueMap('code','region','country')

[country:[UK],code:[LHR],region:[GB-ENG]]
[country:[UK],code:[LGW],region:[GB-ENG]]
[country:[UK],code:[LCY],region:[GB-ENG]]
[country:[UK],code:[STN],region:[GB-ENG]]
[country:[UK],code:[LTN],region:[GB-ENG]]
[country:[CA],code:[YXU],region:[CA-ON]]
```

As the 'where' step takes a predicate for its matching conditions, it allows for a great deal of flexibility in the types of filtering that you can do. In the following example, we find '"IAD"' and use the first three letters of its city name to find other airports that start with those three letters:

```
g.V().has('airport','code','IAD').as('a').
  V().hasLabel('airport').
  where(startingWith('a')).
    by('city').
    by(values('city').substring(0,3)).
  values('city')
```

Note that the order of the 'by' modulators is such that the first 'by' refers to the incoming traverser to 'where' and the second refers to the "a" vertex referenced in the 'startingWith'.

We've seen how the number of 'by' modulators corresponds to the parameters in the 'where' and that they are applied in a round-robin fashion, starting with the value of the traverser being compared and then to the values it is to be compared with. We can further see this on display when we look at a query that uses 'between' where there are three arguments to consider. For example, let's imagine we wanted to find five airports that have a number of runways that are between one less and one more than 'IAD'.

```
g.V().has('airport','code','IAD').values('runways').as('iadl','iadh').
  V().hasLabel('airport').
  where(between('iadl','iadh')).
    by('runways').
    by(math('_ - 1')).
    by(math('_ + 1')).
  limit(5).
  values('code').fold()
```

```
[ANC,BNA,BWI,DCA,IAD]
```

In this prior example, we find IAD and then keep that value in two-step labels. Those labels are used later as arguments to 'between'. Note that there are three 'by' modulators that follow the 'where'. The first refers to the current airport vertex that we are considering for this filter, and that 'by' ensures we use the runways value in the comparison to 'between'. You can see that this initial 'by' relates to an implied reference to the current traverser as there is no label for it present in the 'where'. The second two 'by' modulators refer to the step labels in the 'between' and do a 'math' operation that subtracts and adds 1 to the number of runways for 'IAD'. In this way, you get the effect of dynamically calculating filtering arguments based on a prior portion of a query.

Let's imagine that we want to write a query to find all airports that have the same region code as the French airport of Nice. Let's assume for now that we do not know what the region code is, so we cannot just write a simple query to find all airports in that region. So, instead we need to write a query that will first find the region code for Nice and then use that region code to find any other airports in the region. Finally, we want to return the airport code along with the city name and the region code. One way of writing this query is to take advantage of the 'where...by' construct.

Take a look at the query below and the output it generates.

```
g.V().has('code','NCE').values('region').as('r').
  V().hasLabel('airport').as('a').values('region').
      where(eq('r')).by().
      local(select('a').values('city','code','region').fold())

[NCE,Nice,FR-U]
[MRS,Marseille,FR-U]
[TLN,Toulon/Hyères/Le Palyvestre,FR-U]
[AVN,Avignon/Caumont,FR-U]
```

In the sample programs folder you will find a program called GraphRegion.java that shows how to perform the query shown above in a Java program.

There are several things about this query that are interesting. Firstly, because we are comparing the results of two 'values' steps, we do not provide a parameter to the 'by' step as we do not need to provide a property key. Secondly, we use a second 'V' step in the query to find all the airports that have the same region code as Nice. Note that this also means that Nice is included in the results. Lastly, we wrap the part of the query that prepares the output in a form we want in a 'local' step so that a separate list is created for each airport.

3.23. Using 'choose' to write if...then...else type queries

The 'choose' step allows the creation of queries that are a lot like the "if then else" constructs found in most programming languages. If we were programming in Java, we might find ourselves writing something like the following.

```
if (longest > 12000) {
    return(code);
} else {
    return(desc);
}
```

Gremlin offers us a way to do the same thing. The query below finds all the airports in Texas and then will return the value of the 'code' property if an airport has a runway longer than 12,000 feet, otherwise it will return the value of the 'desc'.

```
// If an airport has a runway > 12,000 feet return its code else return its
description
g.V().has('region','US-TX').
      choose(values('longest').is(gt(12000)),
             values('code'),
             values('desc')).
      limit(5)
```

When run, the output returned should look like this.

```
AUS
DFW
IAH
San Antonio
Houston Hobby
```

If the "else" part of the 'choose' step is not provided, then it behaves as a simple "if".

```
g.V().has('region','US-TX').
      choose(values('longest').is(gt(12000)),
             values('code')).
             limit(5)
```

The "else" in this case is implied, and the incoming element that the 'choose' step received is passed on as shown below.

```
AUS
DFW
IAH
v[33]
v[38]
```

Here is another example that uses the same constructs.

```
// If an airport has a code of AUS or DFW report its region else report its country
g.V().hasLabel('airport').
      choose(values('code').is(within('AUS','DFW')),
             values('city'),
             values('region')).
     limit(5)
```

3.23.1. Including a constant value – introducing 'constant'

Sometimes it is useful, as the example below demonstrates, to return constant rather than derived values as part of a query. This query will return the string "some" if an airport has fewer than four runways or "lots" if it has more than four.

```
// You can also return constants using the constant() step
g.V().hasLabel('airport').limit(10).
      choose(values('runways').is(lt(4)),
             constant('some'),
             constant('lots'))
```

The 'constant' step can be used to return a constant value as part of a query.

Here is one more example that uses a 'sample' step to pick 10 airports and then return either "lots" or "not so many" depending on whether the airport has more than 50 routes or not. Note also how 'as' and 'select' are used to combine both the derived and constant parts of the query result that we will ultimately return.

```
g.V().hasLabel('airport').sample(10).as('a').
      choose(out('route').count().is(gt(50)),
             constant('lots'),
             constant('not so many')).as('b').
      select('a','b').by('code').by()
```

Here is an example of what the output from running this query might look like.

```
[a:PKA,b:not so many]
[a:ISL,b:not so many]
[a:NDR,b:not so many]
[a:PAZ,b:not so many]
[a:NER,b:not so many]
[a:KIX,b:lots]
[a:THD,b:not so many]
[a:YCD,b:not so many]
[a:BTJ,b:not so many]
[a:KBP,b:lots]
```

We could go one step further if you don't want the 'a:' and 'b:' keys returned as part of the result by adding a 'select(values)' to the end of the query as follows.

```
g.V().hasLabel('airport').sample(10).as('a').
      choose(out('route').count().is(gt(50)),
             constant('lots'),
             constant('not so many')).as('b').
      select('a','b').by('code').by().select(values)
```

Here is what the output from the modified form of the query.

```
[IBZ,lots]
[DOH,lots]
[LRE,not so many]
[TCT,not so many]
[AMQ,not so many]
[YYC,lots]
[GUC,not so many]
[CEZ,not so many]
[PSC,not so many]
[JCB,not so many]
```

The 'constant' step is not limited to use within 'choose' steps. It can be used wherever needed. You will find many examples of its use throughout this book including in the "Using 'union' to combine query results" section.

3.24. Using 'option' to write case/switch type queries

When 'option' is combined with 'choose' the result is similar to the 'case' or 'switch' style construct found in most programming languages. In Java, for example, we might code a 'switch' statement as follows.

```
switch(airport) {
    case "DFW": System.out.println(desc); break;
    case "AUS": System.out.println(region); break;
    case "LAX": System.out.println(runways);
}
```

We can write a Gremlin query that follows the same pattern as our Java 'switch' statement. As in the Java example, we decided to lay our query out across multiple lines to aid readability and clarity.

```
// You can combine choose and option to generate a more "case statement" like query
g.V().hasLabel('airport').
      choose(values('code')).
        option('DFW',values('desc')).
```

```
        option('AUS',values('region')).
        option('LAX',values('runways'))
```

The example below shows a 'choose' followed by four options. Note the default case of 'none' is used as the catchall. Notice how in this case, the values returned are constants.

```
// You can return constant values if you need to
g.V().hasLabel('airport').limit(10).
      choose(values('runways')).
        option(1,constant('just one')).
        option(2,constant('a couple')).
        option(7,constant('lots')).
        option(none,constant('quite a few'))
```

The 'option' step can also include a predicate, allowing more complex comparisons to be made as part of that 'choose'. This allows testing that a value is greater than or less than another, for example, as part of the 'option' step without needing to write a more complex query.

The query below creates a group containing the counts of airports that fall into one of the categories generated by the 'choose' and 'option' steps. Any airport situated above 5,000 feet of elevation will be categorized as "high", greater than 3,000 feet as "medium" and all others as "low".

```
g.V().hasLabel('airport').
      groupCount().
        by(choose(values('elev')).
             option(gt(5000),constant('high')).
             option(gt(3000),constant('medium')).
             option(none,constant('low')))

[high:163,low:3127,medium:214]
```

3.25. Using 'match' to do pattern matching

The 'match' step allows a more declarative style of pattern-based query to be expressed using Gremlin. 'match' can be a bit hard to master, but once you figure it out, it can be a very powerful way of traversing a graph looking for specific patterns. As we shall see however, sometimes a `where` step is more than adequate and can be used to express similar patterns to those supported by 'match'.

Below is an example that uses 'match' to look for airline route patterns where there is a flight from one airport to another but no return flight back to the original airport. The first query looks for such patterns involving the JFK airport as the starting point. You can see the output from running the query below it. This is the correct answer as, currently, the British Airways Airbus A318 flight from London City (LCY) airport stops in Dublin (DUB) to take on more fuel on the way to JFK but does not need to stop on the way back because of the trailing wind.

```
// Find any cases of where you can fly from JFK non stop
// to a place you cannot get back from non stop. This query
// should return LCY, as the return flight stops in Dublin
// to refuel.
g.V().has('code','JFK').
      match(__.as('s').out().as('d'),
            __.not(__.as('d').out().as('s'))).
      select('s','d').by('code')

[s:JFK,d:LCY]
```

We can expand the query by leaving off the specific starting point of JFK and look for this pattern anywhere in the graph. This really starts to show how important and useful the 'match' Gremlin step is. We don't have any idea what we might find, but by using 'match', we are able to describe the pattern of behavior that we are looking for and Gremlin does the rest.

```
// Same as above but from any airport in the graph.
g.V().hasLabel('airport').
      match(__.as('s').out().as('d'),
            __.not(__.as('d').out().as('s'))).
      select('s','d').by('code')
```

If you were to run the query, you would find that there are, in fact, over 200 places in the graph where this situation applies. We can add a 'count' to the end of the query to find out just how many there are.

```
// How many occurrences of the pattern in the graph are there?
g.V().hasLabel('airport').
      match(__.as('s').out().as('d'),
            __.not(__.as('d').out().as('s'))).
      count()

339
```

The next query looks for routes that follow the pattern A→B→C but where there is no direct flight of the form A→C. In other words, it looks for all routes between two airports with one intermediate stop where there is no direct flight alternative available. Note that the query also eliminates any routes that would end up back at the airport of origin. To achieve the requirement that we not end up back where we started, a 'where' step is included to make sure we do not match any routes of the form A→B→A.

```
g.V().hasLabel('airport').
      match(__.as('a').out().as('b')
           ,__.as('b').out().where(neq('a')).as('c')
           ,__.not(__.as('a').out().as('c'))).
```

```
      select('a','b','c').by('code').limit(10)
```

There are, of course, a lot of places in the 'air-routes' graph where this pattern can be found. Here are just a few examples of the results you might get from running the query.

```
[a:ATL,b:YYZ,c:YVR]
[a:ATL,b:YYZ,c:LGW]
[a:ATL,b:YYZ,c:DEL]
[a:ATL,b:YYZ,c:HKG]
[a:ATL,b:YYZ,c:PEK]
[a:ATL,b:YYZ,c:BOM]
[a:ATL,b:YYZ,c:PRG]
[a:ATL,b:YYZ,c:BCN]
[a:ATL,b:YYZ,c:VIE]
[a:ATL,b:YYZ,c:YOW]
```

Here is another example of using 'match' along with a 'where'. This is actually a different way of writing a query we saw earlier. This query starts out by looking at how many runways Austin has and then looks at every airport that you can fly to from Austin and then looks at how many runways those airports have. Only airports with the same number as Austin are returned. Using a 'match' step for this task is overkill. However, it does show the basic constructs used by the 'match' step and again illustrates using values calculated in one part of a query later on in that same query.

```
g.V().has('code','AUS').
      match(__.as('aus').values('runways').as('ausr'),
            __.as('aus').out('route').as('outa').values('runways').as('outr')
              .where('ausr',eq('outr'))).
      select('outa').valueMap().select('code','runways')
```

As we mentioned, the example above is not the best way to write this query, and it can be done without using a 'match' step at all and just using a 'where' step as shown in the three examples below. Each one is simpler than its predecessor.

One way we could choose to write this query is using multiple select steps. This is also not a very efficient solution but does work.

```
g.V().has('code','AUS').as('aus').values('runways').as('ausr').
      select('aus').out().as('outa').values('runways').as('outr').
      where('ausr',eq('outr')).
      select('outa').valueMap().select('code','runways')
```

A better way than either of the prior two combines a 'filter' step with the 'where' and 'select' steps.

```
g.V().has('code','AUS').as('a').out().as('b').
      filter(select('a','b').by('runways').where('a',eq('b'))).
```

```
      valueMap('code','runways')
```

As mentioned in the "Using 'where' and 'by' to filter results" section, this query can be simplified further using a 'where' step and a 'by' modulator.

```
g.V().has('code','AUS').as('a').out().
       where(eq('a')).by('runways').
       valueMap('code','runways')
```

So, while there is often a simpler way to write a query that avoids using the 'match' step, for some queries, especially in more complex cases, it provides a useful and powerful way to express in a more declarative way, a set of criteria that must be met. However, before we resort to using a 'match' step, we always think carefully about other ways that we could write the query that might be simpler. We do this because the syntax of the 'match' step can be tricky to get right without a fair bit of trial and error in our experience.

3.25.1. Pattern matching using a 'where' step

As the Gremlin language has evolved, many queries that might seem a perfect candidate for a 'match' step can actually also be written using a 'where' step but still in a more declarative style. We can rewrite the query from the previous section that looks at routes from JFK that do not have a return flight using a 'where' step as shown below.

The way to read this, in a similar way shown with a 'match' step is as follows. Starting at JFK, look at all the places we can fly to but only keep those airports that do not have a route back to JFK. Note that the 'as' step plays two roles in this query. Outside the 'where' step it is used to label steps in the query we want to refer back to later. Inside the 'where' step it provides a convenient shorthand way to refer back to the departure airport.

```
g.V().has('code','JFK').as('s').
      out().as('d').
      where(__.not(out().as('s'))).
      select('s','d').by('code')

[s:JFK,d:LCY]
```

As we have seen in some prior examples, if you only want the airport codes in the result and not the "s" and "d" keys, adding an additional 'select' step is all that it takes.

```
g.V().has('code','JFK').as('s').
      out().as('d').
      where(__.not(out().as('s'))).
      select('s','d').by('code').
      select(values)

[JFK,LCY]
```

The query can be expanded to search for this same pattern across the whole graph. The example below returns the first ten routes found where there is no return flight. The results are sorted in ascending order using the departure airport code.

```
g.V().hasLabel('airport').as('s').
      out().as('d').
      where(__.not(out().as('s'))).
      limit(10).
      select('s','d').by('code').
      order().by(select('s'))
```

When run, the query returns the following results.

```
[s:AMS,d:DSS]
[s:BNE,d:BCI]
[s:BNE,d:BKQ]
[s:BOM,d:DIU]
[s:CLE,d:SJC]
[s:HEL,d:IVL]
[s:IAD,d:AMM]
[s:JFK,d:LCY]
[s:ORD,d:ADD]
[s:SIN,d:PBH]
```

3.26. Using 'union' to combine query results

The 'union' step works just as you would expect from its name. It allows us to combine parts of a query into a single result. Just as with the boolean 'and' and 'or' steps, it is sometimes possible to find other ways to do the same thing without using 'union,' but it does offer some helpful capability.

Here is a simple example that uses a 'union' step to produce a list containing a vertex and the number of outgoing routes from that vertex. Note that in the next section we will see that there are simpler ways to write this query while still using a 'union' step. The main point to take away from this example is that you can use a 'union' step to combine the results of multiple traversals. This example combines the results of two traversals, but you can certainly combine more as needed. Note that the 'out' step starts from the vertex that was found immediately before the 'union' step, which in this case is the DFW vertex. So in other words, the output from the prior step is available to the steps within the 'union' step just as with other Gremlin steps we have already looked at.

```
g.V().has('airport','code','DFW').as('a').
      union(select('a'),out().count()).fold()

[v[8],253]
```

Not that this is recommended, but the previous query could also be written as follows using two 'has' steps both inside a single 'union' step. This does, however, demonstrate that you can use a

'union' step to combine the results of fairly arbitrary graph traversals.

```
g.V().union(has('airport','code','DFW'),
            has('airport','code','DFW').
               out().count()).fold()

[v[8],253]
```

As a side note, instead of using a 'union' step and producing a list, we might decide to use a 'group' step and produce a map. A map might be preferable if you want to access individual keys and values directly. It all depends, as always, on the results that best fit the problem you are solving.

```
g.V().has('airport','code','DFW').
      group().by().by(out().count())

[v[8]:253]]
```

Another aspect of 'union' to consider is that it can be used as a start step, which means that you can essentially start a traversal from multiple points.

```
g.union(V().has('airport','code','DFW'),
        V().has('airport','code','IAD')).
  values('code').fold()

[DFW,IAD]
```

The prior example is meant as a demonstration. Typically, it will make more sense to use a `union` this way when you have disparate sets of things you wish to combine to a single traversal stream. In this case, you can see that we're just filtering airports by their code. It's more likely that you would write that query as follows:

```
g.V().has('airport','code',within('DFW','IAD')).values('code').fold()

[DFW,IAD]
```

3.26.1. Introducing the 'identity' step

Gremlin has an 'identity' step that we have not seen used so far in this book. The 'identity' step simply returns the entity passed in to the current step of a traversal (in this case 'union') from the prior step. We can rewrite the query we used above to use an 'identity' step. This simplifies the query as it removes the need to use the 'as' and 'select' steps. As shown below, using 'identity' causes the vertex 'V[8]' representing the DFW airport from the prior 'has' step to be included in the result.

```
g.V().has('airport','code','DFW').
      union(identity(),out().count()).fold()
```

```
[v[8],253]
```

We could modify the query slightly to have the first part of the result returned be the airport's IATA code rather than just a vertex.

```
g.V().has('airport','code','DFW').
      union(identity().values('code'),out().count()).fold()

[DFW,253]
```

3.26.2. Using 'constant' values as part of a 'union'

We have already seen the 'constant' step used in the "Including a constant value – introducing 'constant'" section. As you might expect, you can also use 'constant' steps within a 'union' step, as the two examples below show.

```
g.V(3).union(constant("Hello"),
             constant("There")).fold()

[Hello,There]
```

The 'identity' step introduced above could be used to add the 'V[3]' vertex to the result. We are now combining three traversal steps inside the 'union' step.

```
g.V(3).union(constant("Hello"),
             constant("There"),
             identity()).fold()

[Hello,There,v[3]]
```

Finally, let's change the query again to include a city name in the result. Note that the 'values' step refers to the property of the vertex that was referenced immediately before the 'union' step, so it will return the 'city' property of vertex 'V[3]'.

```
g.V(3).union(constant("Hello"),
             constant("There"),
             values('city')).fold()

[Hello,There,Austin]
```

3.26.3. More examples of the 'union' step

The following query uses a 'sample' step to select 10 airports at random from the graph. For each selected airport, a 'union' step is then used to combine the 'id' of the vertex with a few properties.

Note that 'local' scope is used so that the results of each 'union' step are folded into a list.

```
g.V().hasLabel("airport").sample(10).
      local(union(id(),values("code","city")).fold())
```

Here is the output we got back from running the query.

```
[1534,CUL,Culiacán]
[2075,CKB,Clarksburg]
[1828,PZI,Panzhihua]
[2244,LOD,Longana]
[1348,GET,Geraldton]
[92,OSL,Oslo]
[2107,UIN,Quincy]
[2764,ODY,Oudomsay]
[3288,MCW,Mason City]
[3331,MPN,Mount Pleasant]
```

If 'local' scope had not been used, the result would have been a single list containing all the results as shown below.

```
g.V().hasLabel("airport").sample(10).
      union(id(),values("code","city")).fold()

[2410,FKS,Sukagawa,3159,YBX,Lourdes-De-Blanc-Sablon,918,HLN,Helena,
 465,DOL,Deauville,1258,YXY,Whitehorse,2000,VHC,Saurimo,2562,RCH,Riohacha,
 225,MBJ,Montego Bay,2535,LOH,La Toma (Catamayo),1385,REG,Reggio Calabria]
```

By way of another simple example, the following query returns flights that arrive in AUS from the UK or that leave AUS and arrive in Mexico.

```
// Flights to AUS from the UK or from AUS to Mexico
g.V().has('code','AUS').
      union(__.in().has('country','UK'),
            out().has('country','MX')).
            path().by('code')
```

When we run that query, we get the following results showing that there are routes from LHR and LGW in the UK and to the six airports PVR, CZM, SJD, MEX, CUN and GDL in Mexico.

```
[AUS,LGW]
[AUS,LHR]
[AUS,CUN]
[AUS,PVR]
[AUS,CZM]
```

```
[AUS,GDL]
[AUS,SJD]
[AUS,MEX]
```

This query solves the problem "Find all routes that start in London, England and end up in Paris or Berlin with no stops". Because city names are used and not airport codes, all airports in the respective cities are considered.

```
g.V().has('city','London').has('region','GB-ENG').
        union(out('route').has('city','Paris'),
              out('route').has('city','Berlin')).
              path().by('code')
```

Here are the results from running the query. Note that routes from five different London airports were found.

```
[LHR,ORY]
[LHR,CDG]
[LHR,BER]
[LGW,CDG]
[LGW,BER]
[LCY,CDG]
[LCY,ORY]
[STN,BER]
[LTN,CDG]
[LTN,ORY]
[LTN,BER]
```

As mentioned previously, sometimes, especially for fairly simple queries, there are alternatives to using 'union'. Indeed, it is actually not necessary to use a 'union' step to achieve the prior result. The re-written version of the query below will return the same results as the version that uses a 'union' step. This time a simple 'has' step featuring a 'within' predicate is used instead.

```
g.V().has('city','London').has('region','GB-ENG').
      out('route').has('city',within('Berlin','Paris')).
      path().by('code')
```

The previous two queries used a 'path' step to essentially show each route. We can adjust the version of the query that uses a 'union' step just a little bit and instead turn the result into a series of lists where the first item in the list is the origin airport and the remaining items in the list are the places you can fly to from there within our criteria. The double underscore '"__"' is used in the way that 'identity' could have been used to refer to the incoming vertex. The 'union' step is wrapped in a 'local' step so that each 'union' is individually folded. If the 'local' step was omitted, all the results would be folded into a single list. In this case, the 'union' step makes writing the query relatively easy, and this is probably a good example of where the 'union' step should be used. Namely, when you want to combine multiple traversal results.

```
g.V().has('city','London').has('region','GB-ENG').
      local(union(__.values('code'),
                  out('route').has('city','Paris').values('code'),
                  out('route').has('city','Berlin').values('code')).
      fold())
```

Running the amended query shows us the results in perhaps a more useful form.

```
[LHR,ORY,CDG,BER]
[LGW,CDG,BER]
[LCY,CDG,ORY]
[STN,BER]
[LTN,CDG,ORY,BER]
```

3.26.4. Using 'union' to combine more complex traversal results

So far the examples we have looked at mostly show fairly simple traversals being used inside of a 'union' step. This next query is a bit more interesting. We again start from any airport in London, but then we want routes that meet any of the criteria:

- Go to Berlin and then to Lisbon
- Go to Paris and then Barcelona
- Go to Edinburgh and then Rome

We also want to return the distances in each case. Note that you can union together as many items as you need to. In this example we combine the results of three sets of traversals to get the desired results.

```
// Returns any paths found along with the distances between airport pairs.
g.V().has('city','London').has('region','GB-ENG').
      union(outE().inV().has('city','Berlin').
                  outE('route').inV().has('city','Lisbon').
                  path().by('code').by('dist').by('code').by('dist'),
            outE().inV().has('city','Paris').
                  outE('route').inV().has('city','Barcelona').
                  path().by('code').by('dist').by('code').by('dist'),
            outE().inV().has('city','Edinburgh').
                  outE('route').inV().has('city','Rome').
                  path().by('code').by('dist').by('code').by('dist'))
```

Here is what we get back when we run our query

```
[LHR,598,BER,1432,LIS]
[LHR,227,ORY,513,BCN]
[LHR,216,CDG,533,BCN]
```

```
[LGW,592,BER,1432,LIS]
[LGW,191,CDG,533,BCN]
[LCY,204,CDG,533,BCN]
[LCY,217,ORY,513,BCN]
[STN,564,BER,1432,LIS]
[LTN,589,BER,1432,LIS]
[LTN,236,CDG,533,BCN]
[LTN,248,ORY,513,BCN]
```

The next query finds the total distance of all routes from any airport in Madrid to any airport anywhere and also does the same calculation but minus any routes that end up in any Paris airport. We have not yet seen the 'filter' step used below. It is one of the foundational Gremlin steps that many others such as 'where' build upon. A 'filter' step will only pass on to the next step in the query incoming elements that meet the criteria specified within the 'filter'.

```
g.V().has('city','Madrid').outE('route').
      union(values('dist').sum(),
            filter(inV().has('city',neq('Paris'))).values('dist').sum())
```

Here is the output from running the query. As you can see, the first number is slightly larger than the second as all routes involving Paris have been filtered out from the calculation.

```
435724
434426
```

It is worth noting that it is not required that every traversal inside a union step returns a result. The returned results will include any of the traversals that did return something. The example below demonstrates this. Of course, in practice you would not write this particular query this way. However, we think this example demonstrates a feature of the 'union' step that it is important to understand.

```
g.V().has('airport','code','AUS').
      union(out().has('code','LHR'),
            out().has('code','SYD'),
            out().has('code','DFW')).
      values('code')
```

If we run the query, you will see that SYD is not part of the results as there is no route between Austin and Sydney.

```
LHR
DFW
```

For completeness, this query would more likely be written as follows rather than using a 'union'.

```
g.V().has('airport','code','AUS').
      out().has('code',within('LHR','DFW','SYD')).
      values('code')

LHR
DFW
```

3.27. Using 'sideEffect' to do things on the side

The 'sideEffect' step allows you to do some additional processing as part of a query without changing what gets passed on to the next stage of the query. The example below finds the airport vertex V(3) and then uses a 'sideEffect' to count the number of places that you can fly to from there and aggregates it in a traversal variable named 'a' before counting how many places you can get to with one stop and storing that value in 'b'. Note that there are other ways we could write this query, but it demonstrates quite well how 'sideEffect' works.

```
g.V(3).sideEffect(out().count().aggregate('a')).
       out().out().count().as('b').select('a','b')

[a:[98],b:8354]
```

Later in the book we will discuss lambda functions, sometimes called closures, and how they can be used. The example below combines a closure with a 'sideEffect' to print a message before displaying information about the vertex that was found. Again, notice how the 'sideEffect' step has no effect on what is seen by the subsequent steps. You can see the output generated below the query.

```
g.V().has('code','SFO').sideEffect{println "I'm working on it"}.values('desc')

I'm working on it
San Francisco International Airport
```

Later in the book we will look at other ways that side effects can be used to solve more interesting problems.

3.28. Using 'aggregate' to create a temporary collection

At the time of writing this book, there were 59 places you could fly to directly (non-stop) from Austin. We can verify this fact using the following query.

```
g.V().has('code','AUS').out().count()

98
```

If we wanted to count how many places we could go to from Austin with one stop, we could use the

following query. The 'dedup' step is used as we only want to know how many unique places we can go to, not how many different ways of getting to all of those places there are.

```
g.V().has('code','AUS').out().out().dedup().count()

1044
```

There is however a problem with this query. The 871 places is going to include (some or possibly all of) the places we can also get to non-stop from Austin. What we really want to find are all the places that you can only get to from Austin with one stop. So what we need is a way to remember all of those places and remove them from the 871 somehow. This is where 'aggregate' is useful. Take a look at the modified query below

```
g.V().has('code','AUS').out().aggregate('nonstop').
     out().where(without('nonstop')).dedup().count()

946
```

After the first 'out' step all of the vertices that were found are stored in a collection we chose to call 'nonstop'. Then, after the second 'out' we can add a 'where' step that essentially says "only keep the vertices that are not part of the nonstop collection". We still do the 'dedup' step as otherwise we will still end up counting a lot of the remaining airports more than once.

Notice that 812 is precisely 59 less than the 871 number returned by the previous query, which shows the places you can get to non-stop were all correctly removed from the second query. This also tells us there are no flights from Austin to places that you cannot get to from anywhere else!

We will take a more in depth look at the various types of collections that you can use as part of a Gremlin query in the "Collections revisited" section a bit later.

3.29. Using 'inject' to insert values into a query

Sometimes you may want to add something additional to be returned along with the results of your query. This can be done using the 'inject' step. To start off, here is a simple example showing how 'inject' fundamentally works. We insert some numbers and ask Gremlin to give us the mean value.

```
g.inject(1,2,3,4,5).mean()

3.0
```

Of course, just using 'inject' so we can do a simple mathematical computation is of limited use. The next example shows how 'inject' can be used as part of a query. The string 'ABIA', another acronym commonly used when referring to the Austin Bergstrom International Airport, is injected into the query.

```
g.V().has('code','AUS').values().inject('ABIA')
```

If we were to run the query, here is what we would get back

```
ABIA
US
AUS
12250
Austin
542
KAUS
-97.6698989868164
airport
US-TX
2
30.1944999694824
Austin Bergstrom International Airport
```

3.29.1. A useful trick using 'inject'

There is also a useful trick that can be achieved using 'inject'. Take a look at the query below.

```
g.V().choose(V().hasLabel('XYZ').count().is(0),constant("None found"))
```

If we were to run it, we would get back multiple lines like those below. One for each vertex in the graph, in fact.

```
None found
None found
None found
None found
None found
...
```

To get just one result, we might be tempted to write the query as shown below.

```
g.choose(V().hasLabel('XYZ').count().is(0),constant("None found"))
```

However, this will cause an error to be returned as a choose step cannot come immediately after a traversal source object ('g'). To get around this, we can rewrite the query with an 'inject' step after the 'g'. We do not use the value of the 'inject' in the query, but its presence allows us to follow it with a 'choose' step.

```
g.inject(1).choose(V().hasLabel('XYZ').count().is(0),constant("None found"))
```

```
None found
```

This is a trick that can come in useful from time to time. The query used to demonstrate this point could indeed be rewritten to avoid any use of 'choose' but hopefully the usefulness of 'inject' as a way to avoid using a 'V' when not wanted is clear.

3.30. Using 'coalesce' to see which traversal returns a result

Sometimes, when you are uncertain as to which traversal of a set you are interested in will return a result, you can have them evaluated in order by the 'coalesce' step. The first of the traversals that you specify that returns a result will cause that result to be the one that is returned to your query.

Look at the example below. Starting from the vertex with an ID of 3, it uses 'coalesce' to first see if there are any outgoing edges with a label of 'fly'. If there are any, the vertices connected to those edges will be returned. If there are not any, any vertices on any incoming edges labeled 'contains' will be returned.

```
// Return the first step inside coalesce that returns a vertex
g.V(3).coalesce(out('fly'),__.in('contains')).valueMap()
```

As there are not any edges labeled 'fly' in the 'air-routes' graph, the second traversal will be the one whose results are returned.

If we were to run the above query using the 'air-routes' graph, this is what would be returned.

```
[code:[NA],type:[continent],desc:[North America]]
[code:[US],type:[country],desc:[United States]]
```

We can put more than two traversals inside of a 'coalesce' step. In the following example there are now three. Because some 'contains' edges do exist for this vertex, the 'route' edges will not be looked at as the traversals are evaluated in left to right order.

```
g.V(3).coalesce(out('fly'),
                __.in('contains'),
                  out('route')).valueMap()
```

As we can see, the results returned are still the same.

```
[code:[NA],type:[continent],desc:[North America]]
[code:[US],type:[country],desc:[United States]]
```

3.30.1. Combining 'coalesce' with a 'constant' value

The 'coalesce' step can also be useful when combined with a 'constant' value. In the example below, if the airport is in Texas, then its description is returned. If it is not in Texas, the string "Not in Texas" is returned instead.

```
g.V(1).coalesce(has('region','US-TX').values('desc'),constant("Not in Texas"))

Not in Texas

g.V(3).coalesce(has('region','US-TX').values('desc'),constant("Not in Texas"))

Austin Bergstrom International Airport
```

3.30.2. Combining 'coalesce' with 'fail' step

We've not looked at 'fail' step before, but it is a very straightforward step. If it is encountered in the traversal, it will throw an exception.

```
g.V().fail()

fail() Step Triggered
==============================
Message  > fail() step triggered
Traverser> v[0]
  Bulk   > 1
Traversal> V().fail()
Metadata > {}
==============================
```

The above example shows what the Gremlin Console formatting of the generated 'FailException'. You can also give it a custom error message.

```
g.V().fail("failed!!")

fail() Step Triggered
==============================
Message  > failed!!
Traverser> v[0]
  Bulk   > 1
Traversal> V().fail("failed!!")
Metadata > {}
==============================
```

When used with 'coalesce', you can trigger the failure condition if a particular traversal does not produce the result you expect. Using the example from the "Combining 'coalesce' with a 'constant' value" section, if the airport is in Texas, then its description is returned. If it is not in Texas, then the

exception is thrown.

```
g.V(1).coalesce(has('region','US-TX').values('desc'),fail())

fail() Step Triggered
================================================================================
==========
Message  > fail() step triggered
Traverser> v[1]
  Bulk   > 1
Traversal> fail()
Parent   > CoalesceStep [V((int) 1).coalesce(__.has("region","US-TX").values("desc"
),__.fail())]
Metadata > {}
================================================================================
==========

g.V(3).coalesce(has('region','US-TX').values('desc'),fail())

Austin Bergstrom International Airport
```

Using 'coalesce' inside of 'by'

In the "Combining 'coalesce' with a 'constant' value" and "Combining 'coalesce' with 'fail' step" sections we saw patterns that have clear applicability to 'by' modulators. Let's look at an example query that uses `groupCount'.

```
g.V().groupCount().by("city").
  unfold().
  filter(select(values).is(gt(3)))

San Jose=4
London=6
Melbourne=4
Santa Rosa=4
```

In the above example, we do a 'groupCount' by the city of the airport. We 'unfold' the 'Map' to entries and then find only those entries that have a count greater than 3. When we use 'by("city")' we tell Gremlin to ignore vertices that do not have a city as a property key. If we'd like to include those in the count, we could use 'coalesce' inside of the 'by' so that we could return a 'constant' value to represent that missing property.

```
g.V().groupCount().by(coalesce(values('city'), constant('No City'))).
  unfold().
  filter(select(values).is(gt(3)))

San Jose=4
No City=245
```

```
London=6
Melbourne=4
Santa Rosa=4
```

A bit later in the "When you need more flexibility than 'mergeV' and 'mergeE'" section we will again use 'coalesce' to check to see if a vertex already exists before we try to add it.

3.31. Returning one of two possible results - introducing 'optional'

Sometimes it may be useful to return one of two results depending upon the outcome of an attempted traversal. The 'optional' step will return either the results of the provided traversal if there is a result or the result of the prior step if there is no result.

In the example below, there is no direct route between Austin (AUS) and Sydney (SYD) so the Austin vertex is returned by the 'optional' step.

```
g.V().has('code','AUS').optional(out().has('code','SYD')).values('city')

Austin
```

However, there is a route between Austin and Dallas Fort Worth (DFW) so as the example below shows, this time the 'optional' step returns the DFW vertex.

```
g.V().has('code','AUS').optional(out().has('code','DFW')).values('city')

Dallas
```

Note that the previous queries behave in the same way that the 'coalesce' step would behave if used as shown below. In this case, an 'identity' step is used to return the prior vertex if the provided traversal does not return a result.

```
g.V().has('code','AUS').
      coalesce(out().has('code','SYD'),identity()).values('city')

Austin

g.V().has('code','AUS').
      coalesce(out().has('code','DFW'),identity()).values('city')

Dallas
```

3.32. Using 'V' and 'E' mid-traversal

All of our queries so far have used the 'V' and 'E' step to help start a traversal. We've seen them at the start of a traversal spawned from 'g' and as a spawn of anonymous child traversals with '__'. As start steps, they inject the objects that flow through the traversal stream. While they are commonly used at the start of a traversal, they can also be used in the middle of one.

```
g.V().has('code','AUS').V().has('code', within('DFW','IAD')).
  values('code').fold()

[DFW,IAD]
```

In the above example, we first find the '"AUS"' airport vertex. That start traverser triggers the mid-traversal 'V' that does a lookup for '"DFW"' and '"IAD"'. As a result, we get two vertices as the result. Each traverser that flows to the mid-traversal 'V' will issue that same filter, therefore, if we return two starting vertices, we will get some duplication and 4 results as follows:

```
g.V().has('code', within('SFO','AUS')).V().has('code', within('DFW','IAD')).
  values('code').fold()

[DFW,IAD,DFW,IAD]
```

Mid-traversal 'V' has a similar counterpart for edges with 'E' and behaves in the same fashion.

```
g.V().has('code',within('SFO','AUS')).E(7612)

e[7612][23-route->184]
e[7612][23-route->184]
```

You typically find use for these mid-traversal steps when you do a mutation or side effect first, but then want to traverse a wholly new path after that. Let's look at a more advanced example to demonstrate this. First, let's find out what the longest route is leaving '"IAD"':

```
g.V().has('code','IAD').outE('route').
  values('dist').max()

8135
```

Now, let's extend on that query with mid-traversal 'V' to use that result to find "all routes from DFW that are longer than the longest route from IAD":

```
g.V().has('code','IAD').outE('route').
  values('dist').max().as('iad').
  V().has('code','DFW').outE('route').as('r').
  where(values('dist').where(gt('iad'))).
```

```
  inV().values('code').fold()

[SYD]
```

3.33. Other ways to explore vertices and edges using 'both', 'bothE', 'bothV' and 'otherV'

We have already looked at examples of how you can walk a graph and examine vertices and edges using steps such as 'out', 'in', 'outE' and 'inE'. In this section we introduce some additional ways to explore vertices and edges.

As a quick recap, we have already seen examples of queries like the one below that simply count the number of outgoing edges from the vertex with an ID of 3.

```
g.V(3).outE().count()

98
```

Likewise, this query counts the number of incoming edges to that same vertex.

```
g.V(3).inE().count()

100
```

The following query introduces the 'bothE' step. What this step does is return all the edges connected to this vertex whether they are outgoing or incoming. As we can see, the count of 120 lines up with the values we got from counting the number of outgoing and incoming edges. We might want to retrieve the edges, as a simple example, to examine a property on each of them.

```
g.V(3).bothE().count()

198
```

If we wanted to return vertices instead of edges, we could use the 'both' step. This will return all of the vertices connected to the vertex with an ID of 3 regardless of whether they are connected by an outgoing or an incoming edge.

```
g.V(3).both().count()

198
```

This next query can be used to show us the 198 vertices that we just counted in the previous query. We sorted the results and used 'fold' to build them into a list to make the results easier to read. Note how vertex 3 is **not** returned as part of the results. This is important, for as we shall see in a few

examples time, this is not always the case.

```
g.V(3).both().order().by(id).fold()

[v[1],v[1],v[4],v[4],v[5],v[5],v[6],v[6],v[7],v[7],v[8],v[8],v[9],v[9],v[10],
 v[10],v[11],v[11],v[12],v[12],v[13],v[13],v[15],v[15],v[16],v[16],v[17],v[17],
 v[18],v[18],v[20],v[20],v[21],v[21],v[22],v[22],v[23],v[23],v[24],v[24],v[25],
 v[25],v[26],v[26],v[27],v[27],v[28],v[28],v[29],v[29],v[30],v[30],v[31],v[31],
 v[33],v[33],v[34],v[34],v[35],v[35],v[37],v[37],v[38],v[38],v[39],v[39],v[41],
 v[41],v[42],v[42],v[43],v[43],v[45],v[45],v[46],v[46],v[47],v[47],v[48],v[48],
 v[49],v[49],v[50],v[50],v[52],v[52],v[70],v[70],v[99],v[99],v[136],v[136],v[147],
 v[147],v[149],v[149],v[150],v[150],v[151],v[151],v[178],v[178],v[180],v[180],v[182],
 v[182],v[183],v[183],v[184],v[184],v[185],v[185],v[186],v[186],v[187],v[187],v[188],
 v[188],v[189],v[189],v[190],v[190],v[193],v[193],v[194],v[194],v[195],v[195],v[208],
 v[208],v[227],v[227],v[239],v[239],v[240],v[240],v[244],v[244],v[265],v[265],v[268],
 v[268],v[269],v[269],v[271],v[271],v[273],v[273],v[277],v[277],v[278],v[278],v[280],
 v[280],v[281],v[281],v[362],v[362],v[365],v[365],v[368],v[368],v[371],v[371],v[375],
 v[375],v[389],v[389],v[395],v[395],v[403],v[403],v[416],v[416],v[422],v[422],v[429],
 v[429],v[430],v[430],v[431],v[431],v[444],v[444],v[549],v[549],v[605],v[605],v[883],
 v[883],v[909],v[909],v[929],v[929],v[1274],v[1274],v[3730],v[3744]]
```

You probably also noticed that most of the vertices appear twice. This is because for most air-routes there is an outgoing and an incoming edge. If we want to eliminate any duplicate results, we can do that by adding a 'dedup' step to our query.

```
g.V(3).both().dedup().order().by(id).fold()

[v[1],v[4],v[5],v[6],v[7],v[8],v[9],v[10],v[11],v[12],v[13],v[15],v[16],v[17],
 v[18],v[20],v[21],v[22],v[23],v[24],v[25],v[26],v[27],v[28],v[29],v[30],v[31],
 v[33],v[34],v[35],v[37],v[38],v[39],v[41],v[42],v[43],v[45],v[46],v[47],v[48],
 v[49],v[50],v[52],v[70],v[99],v[136],v[147],v[149],v[150],v[151],v[178],v[180],
 v[182],v[183],v[184],v[185],v[186],v[187],v[188],v[189],v[190],v[193],v[194],
 v[195],v[208],v[227],v[239],v[240],v[244],v[265],v[268],v[269],v[271],v[273],
 v[277],v[278],v[280],v[281],v[362],v[365],v[368],v[371],v[375],v[389],v[395],
 v[403],v[416],v[422],v[429],v[430],v[431],v[444],v[549],v[605],v[883],v[909],
 v[929],v[1274],v[3730],v[3744]]
```

We can do another count using our modified query to check we got the expected number of results back.

```
g.V(3).both().dedup().count()

100
```

There is a similar set of things we can do when working with edges using the 'bothV' and 'otherV' steps. The 'bothV' step returns the vertices at both ends of an edge, and the 'otherV' step returns the vertex at the other end of the edge. This is relative to how we are looking at the edge.

The query below starts with our same vertex with the ID of 3 and then looks at all the edges no matter whether they are incoming or outgoing and retrieves all the vertices at each end of those edges using the 'bothV' step. Notice that this time our count is 396. This is because for every one of the 198 edges, we asked for the vertex at each end, so we ended up with 396 of them.

```
g.V(3).bothE().bothV().count()

396
```

We can again add a 'dedup' step to get rid of duplicate vertices as we did before and redo the count, but notice this time we get back 62 instead of the 61 we got before. So what is going on here?

```
g.V(3).bothE().bothV().dedup().count()

101
```

Let's run another query and take a look at all of the vertices that we got back this time.

```
g.V(3).bothE().bothV().dedup().order().by(id()).fold()

[v[1],v[3],v[4],v[5],v[6],v[7],v[8],v[9],v[10],v[11],v[12],v[13],v[15],v[16],
 v[17],v[18],v[20],v[21],v[22],v[23],v[24],v[25],v[26],v[27],v[28],v[29],v[30],
 v[31],v[33],v[34],v[35],v[37],v[38],v[39],v[41],v[42],v[43],v[45],v[46],v[47],
 v[48],v[49],v[50],v[52],v[70],v[99],v[136],v[147],v[149],v[150],v[151],v[178],
 v[180],v[182],v[183],v[184],v[185],v[186],v[187],v[188],v[189],v[190],v[193],
 v[194],v[195],v[208],v[227],v[239],v[240],v[244],v[265],v[268],v[269],v[271],
 v[273],v[277],v[278],v[280],v[281],v[362],v[365],v[368],v[371],v[375],v[389],
 v[395],v[403],v[416],v[422],v[429],v[430],v[431],v[444],v[549],v[605],v[883],
 v[909],v[929],v[1274],v[3730],v[3744]]
```

Can you spot the difference? This time, vertex 3 (v[3]) **is** included in our results. This is because we started out by looking at all of the edges and then asked for all the vertices connected to those edges. Vertex 3 gets included as part of that computation. So beware of this subtle difference between using 'both' and the 'bothE().bothV()' pattern.

Let's rewrite the queries we just used again but replace 'bothV' with 'otherV'. Notice that when we count the number of results, we are back to 100 again.

```
g.V(3).bothE().otherV().dedup().count()

100
```

So let's again look at the returned vertices and see what the difference is.

```
g.V(3).bothE().otherV().dedup().order().by(id()).fold()
```

```
[v[1],v[4],v[5],v[6],v[7],v[8],v[9],v[10],v[11],v[12],v[13],v[15],v[16],v[17],
 v[18],v[20],v[21],v[22],v[23],v[24],v[25],v[26],v[27],v[28],v[29],v[30],v[31],
 v[33],v[34],v[35],v[37],v[38],v[39],v[41],v[42],v[43],v[45],v[46],v[47],v[48],
 v[49],v[50],v[52],v[70],v[99],v[136],v[147],v[149],v[150],v[151],v[178],v[180],
 v[182],v[183],v[184],v[185],v[186],v[187],v[188],v[189],v[190],v[193],v[194],
 v[195],v[208],v[227],v[239],v[240],v[244],v[265],v[268],v[269],v[271],v[273],
 v[277],v[278],v[280],v[281],v[362],v[365],v[368],v[371],v[375],v[389],v[395],
 v[403],v[416],v[422],v[429],v[430],v[431],v[444],v[549],v[605],v[883],v[909],
 v[929],v[1274],v[3730],v[3744]]
```

As you can see, when we use 'otherV' we do not get 'v[3]' returned as we are only looking at the other vertices relative to where we started from, which was 'v[3]'.

3.34. Shortest paths (between airports) - introducing 'repeat'

Gremlin provides a 'repeat...until' looping construct similar to those found in many programming languages. This gives us a nice way to perform simple shortest path type queries. We can use a 'repeat...until' loop to look for paths between two airports without having to specify an explicit number of 'out' steps to try.

While performing such computations, we may not want paths we have already traveled to be traveled again. We can ask for this behavior using the 'simplePath' step. Doing so will speed up queries that do not need to travel the same paths through a graph multiple times. Without the 'simplePath' step being used, the query we are about to look at could take a lot longer. The addition of a 'limit' step is also important as without it this query will run for a LONG time looking for every possible path!!

The query below looks for routes between Austin (AUS) and Agra (AGR). An important query for those Austinites wanting to visit the Taj Mahal!

```
// What are some of the ways to travel from AUS to AGR?
g.V().has('code','AUS').
      repeat(out().simplePath()).
        until(has('code','AGR')).
        path().by('code').limit(10)
```

Here are the results from running the query. Notice how, using the 'repeat...until' construct, we did not have to specify how many steps to try.

```
[AUS,JFK,BOM,AGR]
[AUS,YYZ,BOM,AGR]
[AUS,LHR,BOM,AGR]
[AUS,EWR,BOM,AGR]
[AUS,FRA,BOM,AGR]
[AUS,AMS,BOM,AGR]
```

```
[AUS,PHL,YYZ,BOM,AGR]
[AUS,PHL,LHR,BOM,AGR]
[AUS,PHL,CDG,BOM,AGR]
[AUS,PHL,FRA,BOM,AGR]
```

You can also place the 'until' before the 'repeat' as shown below.

```
// Another shortest path example using until...repeat instead
g.V().has('code','AUS').
      until(has('code','SYD')).
        repeat(out().simplePath()).limit(10).
      path().by('code')
```

Here are the results from running the query.

```
[AUS,DFW,SYD]
[AUS,IAH,SYD]
[AUS,LAX,SYD]
[AUS,SFO,SYD]
[AUS,HNL,SYD]
[AUS,YVR,SYD]
[AUS,PHL,DFW,SYD]
[AUS,PHL,IAH,SYD]
[AUS,PHL,DOH,SYD]
[AUS,PHL,LAX,SYD]
```

We can also specify an explicit number of out steps to try using a 'repeat...times' loop but this of course assumes that we know ahead of time how many stops we want to look for between airports. In the next section we will introduce the 'emit' step that gives you more control over the behavior of what is returned from 'repeat' loops.

```
g.V().has('code','AUS').repeat(out()).times(2).has('code','SYD').path().by('code')

[AUS,DFW,SYD]
[AUS,IAH,SYD]
[AUS,LAX,SYD]
[AUS,SFO,SYD]
[AUS,HNL,SYD]
[AUS,YVR,SYD]
```

The previous query is equivalent to this next one, but doing it this way is less flexible in that we cannot as easily vary the number of 'out' steps, should, for example, we want to next try five hops instead of the current two.

```
g.V().has('code','AUS').out().out().has('code','SYD').path().by('code')
```

As is often the case when working with Gremlin, there is more than one way to achieve the same result. The 'loops' step, that can be used to control how long a repeat loop runs, is essentially equivalent to the 'times' step. Take a look at the two queries below. Both achieve the same result. The first uses 'loops' while the second uses 'times'. We prefer the readability offered by the use of 'times'.

```
g.V(3).repeat(out()).until(loops().is(2)).count()

8354

g.V(3).repeat(out()).times(2).count()

8354
```

If you simply want to find out if any route exists between two airports, there is a nice optimization that can be used. This is discussed in the "Does any route exist between two airports?" section later in the book.

In the next section, we will look at how 'emit' can be used to adjust the behavior of a 'repeat...times' loop.

3.34.1. Using 'emit' to return results during a 'repeat' loop

Sometimes it is useful to be able to return the results of a traversal as it executes. The example below starts at the Santa Fe airport (SAF) and uses a 'repeat' to keep going out from there. By placing an 'emit' right after the 'repeat', we will be able to see the paths that are taken by the traversal. If we did not put the 'emit' here, this query would run for a very long time as the 'repeat' has no other ending condition!

```
g.V().has('code','SAF').repeat(out()).emit().path().by('code').limit(10)

[SAF,DFW]
[SAF,LAX]
[SAF,PHX]
[SAF,DEN]
[SAF,DFW,YYZ]
[SAF,DFW,YVR]
[SAF,DFW,LHR]
[SAF,DFW,CDG]
[SAF,DFW,FRA]
[SAF,DFW,HEL]
```

Another place where 'emit' can be useful is when 'repeat' and 'times' are used together to find paths between vertices. Ordinarily, if you use a step such as 'times(3)' then the query will only return results that are three hops out. However, if we use an 'emit' we can also see results that take fewer hops. First, take a look at the query below that does not use an 'emit' and the results that it generates.

```
g.V(3).repeat(out()).times(3).has('code','MIA').
       limit(5).path().by('code')
```

The paths returned show a selection of ways to get to Miami from Austin with two stops, but none of the results show fewer than two stops. Is this what we really wanted?

```
[AUS,PHL,SXM,MIA]
[AUS,PHL,RIC,MIA]
[AUS,PHL,ISP,MIA]
[AUS,PHL,CHS,MIA]
[AUS,PHL,GRR,MIA]
```

Now let's change the query to use an 'emit'. This time you can think of the query as saying "at most three hops" or in airline terms "at most two stops".

```
g.V(3).repeat(out().simplePath()).emit().times(3).has('code','MIA').
       limit(5).path().by('code')
```

As you can see, by adding an 'emit' we got back a quite different set of results. This is a really useful and powerful capability. Being able to express ideas such as "at most three" provides us a way to write very clean queries in cases like this.

```
[AUS,MIA]
[AUS,PHL,MIA]
[AUS,DTW,MIA]
[AUS,CLT,MIA]
[AUS,CUN,MIA]
```

Note that using the 'emit' step in the previous query, we were able to write a more compact form of this query which essentially does the same thing. Note the use of the inline 'or' step in this example.

```
g.V(3).repeat(out().simplePath()).
         until(has('code','MIA').or().loops().is(3)).
       has('code','MIA').
       path().by('code').limit(5)
```

When run, as you can see, we get the same results back.

```
[AUS,MIA]
[AUS,PHL,MIA]
[AUS,DTW,MIA]
[AUS,CLT,MIA]
[AUS,CUN,MIA]
```

The 'emit' step can also take a parameter such as a 'has' step to filter out intermediate results that we are not interested in. The query below will only show intermediate results as the 'repeat' operates if they meet a given condition. In this case the condition is that the path must have passed through the Prague (PRG) airport's vertex. A 'limit' step is used to only show the first 10 results.

```
g.V(3).repeat(out().simplePath()).emit(has('code','PRG')).
       path().by('code').limit(10)
```

Here are the results from running the query.

```
[AUS,PHL,PRG]
[AUS,YYZ,PRG]
[AUS,LHR,PRG]
[AUS,EWR,PRG]
[AUS,LGW,PRG]
[AUS,FRA,PRG]
[AUS,AMS,PRG]
[AUS,PHL,BER,PRG]
[AUS,PHL,YYZ,PRG]
[AUS,PHL,LHR,PRG]
```

Without the condition as part of the 'emit' step, we get different results as we are shown every path the graph traverser is taking.

```
g.V(3).repeat(out().simplePath()).emit().path().by('code').limit(10)

[AUS,PHL]
[AUS,PDX]
[AUS,DTW]
[AUS,OKC]
[AUS,ONT]
[AUS,CLT]
[AUS,CUN]
[AUS,MEM]
[AUS,CVG]
[AUS,IND]
```

So far, while interesting, many of the results shown look at first glance as if they could have been generated without using an 'emit'. However, the query below is more interesting in that we use an 'until' step to specify a target airport of Austin (AUS) that we are interested in getting to from Lerwick (LSI) in the Shetland Islands. We also specify, as part of the 'emit' step, that we are interested in seeing any routes found that involve any airports in New York State regardless of whether they end up in Austin.

```
g.V().has('code','LSI').
     repeat(out().simplePath()).
```

```
        emit(has('region','US-NY')).
        until(has('code','AUS')).
      path().by('code').limit(10)
```

Here are the results the query generates, Notice how we got a mixture of New York airports as well as Austin as our final destinations.

```
[LSI,MAN,JFK]
[LSI,MAN,EWR]
[LSI,EDI,JFK]
[LSI,EDI,EWR]
[LSI,EDI,SWF]
[LSI,GLA,JFK]
[LSI,GLA,EWR]
[LSI,MAN,ATL,ROC]
[LSI,MAN,ATL,BUF]
[LSI,MAN,ATL,ALB]
```

The 'emit' can also be placed before the 'repeat' step. This will cause the result of the previous step in the query to be emitted before the results that follow. In the example below, we start in Austin and go out two hops using a 'repeat' loop. The first ten airport codes of the places we found are returned. Notice how AUS is returned as the first value even though that is where we started from due to our use of 'emit'.

```
g.V().has('airport','code','AUS').
      emit().repeat(out().simplePath()).times(2).limit(10).
      values('code').fold()

[AUS,PHL,SXM,RIC,LEX,ISP,SWF,DSM,MYR,CAE]
```

In some cases, an 'emit' placed after a 'repeat' step has the same effect as an 'until' step. Both queries below look for routes between Johannesburg (JNB) and Sydney (SYD).

```
g.V().has('code','JNB').repeat(out()).until(has('code','SYD')).
      path().by('code').limit(3)

g.V().has('code','JNB').repeat(out()).emit(has('code','SYD')).
      path().by('code').limit(3)
```

When either query is run, the following results are returned.

```
[JNB,SYD]
[JNB,SIN,SYD]
[JNB,DXB,SYD]
```

You will see more examples of 'emit' being used in the "Modeling an ordered binary tree as a graph" section a bit later.

3.34.2. Nested and named 'repeat' steps

The 'repeat' step can be nested inside another 'repeat' step as well as inside 'emit' and 'until' steps.

The official documentation for these new capabilities can be located here: https://tinkerpop.apache.org/docs/current/reference/#repeat-step

It is also possible to label a repeat step with a name so that it can be referenced later in a traversal. Nested 'repeat' steps allow for some interesting new graph traversal patterns. For example, you might be traversing along a set of outgoing edges, and for each vertex along the way want to traverse a set of incoming edges. The `air-routes` graph does not have any relationships that demonstrate an ideal use case for nested 'repeat' steps but the query below shows a simple example.

```
g.V().has('code','SAF').
      repeat(out('route').simplePath().
        repeat(__.in('route')).times(3)).
      times(2).
      path().by('code').
      limit(3).
      toList()
```

Running the query will generate results similar to those shown below. We start at Santa Fe (SAF) and take one outbound route and arrive at Dallas Fort Worth (DFW). We then look at three incoming routes which yield Lake Charles (LCH), Houston (IAH) and Lomé (LFW). We then take another outbound hop from DFW and find ourselves in Los Angeles(LAX) we then look at three incoming routes from Los Angeles and find San Diego (SAN), Albuquerque (ABQ) and one of San Francisco (SFO), Portland (PDX) or Atlanta (ATL).

```
[SAF,DFW,LCH,IAH,LFW,LAX,SAN,ABQ,SFO]
[SAF,DFW,LCH,IAH,LFW,LAX,SAN,ABQ,PDX]
[SAF,DFW,LCH,IAH,LFW,LAX,SAN,ABQ,ATL]
```

As we mentioned, working with the air-routes data set does not perhaps present an ideal use case for using nested repeat steps. Most of the edges are routes and most of the vertices are airports. However, if your data had a broader variety of vertex and edge types, this capability may come in quite handy.

There is a stand-alone example in the `sample-code/groovy` folder that creates a small social graph and performs various nested 'repeat' step operations. That sample is located here: https://github.com/krlawrence/graph/blob/main/sample-code/groovy/nested-repeat.groovy

When using nested 'repeat' steps, in order for a 'loops' step to know which repeat step it is attached to it is necessary to give each 'repeat' step its own label name. The example below gives the 'repeat' step a label of '"r1"' and refers to that label in the subsequent 'loops' step. This example does not contain any nested repeats but hopefully shows how this new labeling capability can be used.

```
g.V().has('code','SAF').
      repeat('r1',out().simplePath()).
        until(loops('r1').is(3).or().has('code','MAN')).
      path().by('city').
      limit(3).
      toList()
```

The results below show that we found Manchester once and reached our 'loops' limit the other two times.

```
[Santa Fe,Los Angeles,Manchester]
[Santa Fe,Dallas,Toronto,Atlanta]
[Santa Fe,Dallas,Toronto,Austin]
```

3.34.3. Limiting the results at each depth

Sometimes, while exploring a graph using 'repeat' steps, it is desirable to limit the amount of results returned at any given depth of a traversal. Achieving this result is straightforward using 'limit' inside 'repeat'.

```
g.V().has('code','SFO').
      repeat(out().simplePath().limit(5)).
        times(3).
      path().by('code')

[SFO,FCA,LAS,CPH]
[SFO,FCA,LAS,CLT]
[SFO,FCA,LAS,CUN]
[SFO,FCA,LAS,PSP]
[SFO,FCA,LAS,MEM]
```

We get five results here, but we aren't seeing the results at each depth. We are only seeing the output from the last iteration. We need to add 'emit' to 'repeat' so that we instead see the output of each iteration.

```
g.V().has('code','SFO').
      repeat(out().simplePath().limit(5)).
        times(3).emit().
      path().by('code')

[SFO,FCA]
```

```
[SFO,JAC]
[SFO,MSO]
[SFO,MTJ]
[SFO,SUN]
[SFO,FCA,LAS]
[SFO,FCA,DEN]
[SFO,FCA,AZA]
[SFO,FCA,LAX]
[SFO,FCA,MSP]
[SFO,FCA,LAS,CPH]
[SFO,FCA,LAS,CLT]
[SFO,FCA,LAS,CUN]
[SFO,FCA,LAS,PSP]
[SFO,FCA,LAS,MEM]
```

Now the results clearly show the iteration-based output we hoped to see with 5 paths taken for each of the three steps away from SFO.

3.34.4. Haven't I been here before? - Introducing 'cyclicPath'

You can use the 'cyclicPath' step to find paths through the graph that revisit a vertex seen earlier in the traversal. This does not necessarily mean revisiting the starting vertex, it can be any vertex already seen while traversing a graph. An example of some cyclic paths is shown below. The rather contrived query below finds ten routes that both start and end in Austin (AUS) with a stop along the way.

```
// From Austin and back again with one stop.
g.V().has('code','AUS').
      out().out().cyclicPath().
      limit(10).path().by('code')
```

Here are the results from running the query.

```
[AUS,PHL,AUS]
[AUS,PDX,AUS]
[AUS,DTW,AUS]
[AUS,OKC,AUS]
[AUS,ONT,AUS]
[AUS,CLT,AUS]
[AUS,CUN,AUS]
[AUS,MEM,AUS]
[AUS,CVG,AUS]
[AUS,IND,AUS]
```

You can also use 'cyclicPath' as a termination condition for a 'repeat' loop. The query below keeps following outbound 'route' edges until it ends up back where it started. Once again, only the first ten results are selected.

```
g.V().has('code','AUS').
      repeat(out('route')).until(cyclicPath()).
      limit(10).path().by('code')
```

Here are the results from running the query. As you can see, it generated the same results in this instance, but if we let it run for longer than just ten results, it would ultimately find a lot more cyclic paths as it is not restricted to just going out one hop from its starting point. Indeed, if we did not restrict the number of results we wanted, the query would ultimately find every cyclic route in the graph. That query could run for quite a while, so we would not recommend trying it!

```
[AUS,PHL,AUS]
[AUS,PDX,AUS]
[AUS,DTW,AUS]
[AUS,OKC,AUS]
[AUS,ONT,AUS]
[AUS,CLT,AUS]
[AUS,CUN,AUS]
[AUS,MEM,AUS]
[AUS,CVG,AUS]
[AUS,IND,AUS]
```

If we let the previous query run a few hundred times you would start to see examples of where the cycle is not back to the starting vertex. The two results below were generated by letting the query find 500 cyclic paths. Note that in the second case the cycle is back to Philadelphia (PHL) and not the starting vertex (AUS).

```
[AUS,PHL,SXM,PHL]
[AUS,PHL,RIC,PHL]
```

You can also use 'cyclicPath' combined with a 'not' predicate to avoid returning cyclic results from a query.

```
g.V().has('code','AUS').
      out().
      out().not(cyclicPath()).limit(10).
      path().by('code')
```

Note that for the 'air-routes' graph this has the same effect as if we had written the query as shown below.

```
g.V().has('code','AUS').as('a').
      out().
      out().where(neq('a')).limit(10).
      path().by('code')
```

Some graphs contain vertices that have an edge that loops immediately back to the same vertex. Take a look at the code below that creates an edge with a label of 'loop' from the vertex with an ID of 3 back to the same vertex.

```
g.V(3).as('a').addE('loop').to('a')

e[61394][3-loop->3]
```

We can then use the 'cyclicPath' step to find such loops in the graph.

```
g.V().out().cyclicPath().path()

[v[3],v[3]]
```

To include the edge in the result, you just need to modify the query a little as shown below.

```
g.V().outE().inV().cyclicPath().path()

[v[3],e[61394][3-loop->3],v[3]]
```

3.34.5. A warning that path finding can be memory and CPU intensive

Take a look at the query below. It returns the first 10 routes found that will get you from Papa Stour (PSV), a small airport in the Shetland Islands, to Austin (AUS). The 'simplePath' step is used to make sure the same exact path is never looked at twice. This query runs quickly and returns some useful results.

```
g.V().has('code','PSV').
      repeat(out().simplePath()).
      until(has('code','AUS')).
      limit(10).
      path().
        by('code')
```

Here are some of the routes returned.

```
[PSV,LWK,FIE,KOI,MAN,ATL,AUS]
[PSV,LWK,FIE,KOI,MAN,BOS,AUS]
[PSV,LWK,FIE,KOI,MAN,IAD,AUS]
[PSV,LWK,FIE,KOI,MAN,IAH,AUS]
[PSV,LWK,FIE,KOI,MAN,JFK,AUS]
[PSV,LWK,FIE,KOI,MAN,LAX,AUS]
[PSV,LWK,FIE,KOI,MAN,MCO,AUS]
[PSV,LWK,FIE,KOI,MAN,MIA,AUS]
[PSV,LWK,FIE,KOI,MAN,ORD,AUS]
```

```
[PSV,LWK,FIE,KOI,MAN,SEA,AUS]
```

If we reverse the query as shown below, we can run into trouble. In fact, if you run this query on your laptop, after a few minutes of high CPU usage and increased fan noise, it is likely you will get an error that the query ran out of available memory.

```
g.V().has('code','AUS').
      repeat(out().simplePath()).
      until(has('code','PSV')).
      limit(10).
      path().
        by('code')
```

The reason this happens is as follows. There are very few routes from PSV and not many more from the airports it is closely connected to. Therefore, if you start the query from PSV, you will fairly quickly find some paths that end up in AUS. However, if you start from AUS there are a lot of possible routes that Gremlin has to explore before it gets close to finding PSV. If it helps, think of a funnel where PSV is at the narrow end and AUS is at the other. This is also sometimes referred to as having a high or low "fan out" of possible routes depending on the query direction.

3.34.6. A warning that the 'path' and 'as' steps can also be memory intensive

The Gremlin 'path' step is incredibly useful, and we find ourselves using it a lot. However, there are some downsides to the use of 'path', especially when searching entire graphs for non-trivial results. Even if the paths that you are looking for are less challenging to find than in the prior section, you should be aware that keeping track of all the paths that you find and retrieving them using a 'path' step can require a lot of memory to be used.

The 'path' and 'as' steps can use a lot of memory and in some cases cause issues.

The reason that so much memory can be consumed is that the 'path' step, even if 'simplePath' is used to avoid traveling the same exact path more than once, requires the query processor to potentially store up a huge number of results before finding the ones we are actually interested in. So while the 'path' step is incredibly useful, be aware that in cases where a lot of paths need to be tracked, it can get you into trouble if not used with care. A Gremlin query processor will have a limited amount of memory available in which to execute a given query. Storing a large amount of path information may exceed that limit, causing query execution to fail.

Something that may not be immediately obvious is that using an 'as' step can also require the query processor to, at least in part, store path information requiring significant amounts of memory. The 'as' step allows us to refer back to the previous state of a traversal but potentially requires a lot of memory to hold this state during complex queries.

If you find yourself running into memory limitation issues, there are often ways to circumvent the problem using different approaches to writing the query. You will find an example of such a case in the "Comparing properties and constants to the value of a sack" section.

3.35. Calculating vertex degree

While working with graphs, the word 'degree' is used when discussing the number of edges coming into a vertex (in degree), going out from a vertex (out degree) or potentially both coming in and going out (degree). It's quite simple with Gremlin to calculate various measures of degree as the following examples demonstrate.

The simplest way to calculate vertex degree is simply to count edges as shown below.

```
// Outgoing degree
g.V().has('airport','code','LHR').out().count()

221

// Incoming degree
g.V().has('airport','code','LHR').in().count()

223

// Overall degree
g.V().has('airport','code','LHR').both().count()

444
```

If you want to calculate the degree values for more than a single vertex, it can be done more easily using the 'group' step. The query below will calculate the number of outgoing routes for every airport in the graph. If you run this query, you will get quite a lot of result data back as there are over 3,300 airports in the graph.

```
// Out degree (number of routes) from each vertex (airport)
g.V().hasLabel('airport').group().by('code').by(out('route').count())
```

The query below builds upon the prior one but just selects a few of the results.

```
// Outbound routes (degree) from LHR, JFK and DFW
g.V().hasLabel('airport').group().by('code').by(out('route').count()).
      select('LHR','JFK','DFW')
```

If we were to run the query, the output should look like this.

```
[LHR:221,JFK:204,DFW:253]
```

We could change the query a little to calculate the in degree values. Note that we can see that JFK has one fewer incoming route than it has outgoing.

```
g.V().hasLabel('airport').
      group().by('code').by(__.in('route').count()).
      select('LHR','JFK','DFW')

[LHR:221,JFK:203,DFW:253]
```

The query below is a little more complex but can be used to find the 10 airports with the highest number of outgoing routes. Some of the concepts used such as 'local' scope are covered in more detail a bit later on in the "Using 'local' scope with collections" section.

```
g.V().hasLabel('airport').
      group().by('code').by(out().count()).
      order(local).by(values).unfold().tail(10).fold()
```

Here are the results from running the query. As you can see, Frankfurt Airport (FRA) has the highest number of outgoing routes. The topic of analyzing routes is revisited in detail in the "Airports with the most routes" section.

```
[ATL=242,DXB=248,PEK=248,DFW=253,ORD=265,MUC=270,AMS=283,CDG=293,IST=309,FRA=310]
```

The next query will calculate the route degree based on all, incoming and outgoing, routes for ten airports. The query takes advantage of the 'project' step.

```
// Calculate degree (in and out) for each vertex.
g.V().hasLabel('airport').limit(10).
      project("v","degree").by('code').by(bothE('route').count())
```

Here is the output that this query generates.

```
[v:ATL,degree:484]
[v:ANC,degree:82]
[v:AUS,degree:196]
[v:BNA,degree:150]
[v:BOS,degree:286]
[v:BWI,degree:182]
[v:DCA,degree:192]
[v:DFW,degree:506]
[v:FLL,degree:316]
[v:IAD,degree:315]
```

We could, of course, also write the same query using a 'group' step as shown below.

```
g.V().hasLabel('airport').limit(10).
```

```
      group().by('code').by(bothE('route').count())
```

The results below were generated by running the query.

```
[DCA:192,BNA:150,DFW:506,BWI:182,ANC:82,BOS:286,FLL:316,ATL:484,IAD:315,AUS:196]
```

3.36. Gremlin's scientific calculator – introducing 'math'

As we have seen in some of the prior sections, there are some Gremlin steps such as 'sum', 'count' and 'mean' that can be used to perform some fairly basic mathematical operations. In addition to those steps, 'math' step provides us to perform scientific calculator style mathematical operations as part of a Gremlin graph traversal. As these operators build upon the Java Math class, it is worth becoming familiar with that class if you are not already.

The table below provides a summary of the available operators sorted alphabetically.

Table 7. Scientific calculator operators

+	Arithmetic plus.
-	Arithmetic minus.
*	Arithmetic multiply.
/	Arithmetic divide.
%	Arithmetic modulo (remainder).
^	Raise to the power. (n^x).
abs	Absolute value
acos	Arc (inverse) cosine in radians.
asin	Arc (inverse) sine in radians.
atan	Arc (inverse) tangent in radians.
cbrt	Cube root
ceil	Returns the smallest (closest to negative infinity) double value that is greater than or equal to the argument and is equal to a mathematical integer.
cos	Cosine of angle given in radians.
cosh	Hyperbolic cosine.
exp	Returns Euler's number '"e"' raised to the given power '(e^x)'
floor	Returns the largest (closest to positive infinity) double value that is less than or equal to the argument and is equal to a mathematical integer.
log	Natural logarithm (base 'e')
log10	Logarithm (base 10)

log2	Logarithm (base 2)
signum	Returns the 'signum' function of the argument; zero if the argument is zero, 1.0 if the argument is greater than zero, -1.0 if the argument is less than zero.
sin	Sine of angle given in radians.
sinh	Hyperbolic sine
sqrt	Square root
tan	Tangent of angle given in radians.
tanh	Hyperbolic tangent.

The 'math' step behaves differently from other steps that we have looked at so far in as much as the entire expression is passed in as a single string. This means that you can use labels you have assigned as part of a traversal, but you cannot use external variables or static constant references such as 'Math.PI' inside the expression itself. There are ways to easily work around this, however, as we shall see below. We have not attempted to give an example of every single operator being used, but the examples provided should provide all the basic building blocks you will need to incorporate mathematical operators into your own Gremlin queries.

3.36.1. Performing simple arithmetic

Let's start by looking at a few basic examples. First, the query below shows that we can perform mathematical operations on literal values as part of a traversal. The 'math' step does not use the vertex we found at the start of the traversal in this case.

```
g.V().limit(1).math('100/2')

50.0
```

If you want to use the result of the prior step of a traversal as part of a 'math' step, the special symbol '"_"' (underscore) can be used as shown below. Note that the 'inject' step provides us a nice way to feed in values while experimenting with the 'math' step

```
g.inject(100).math('_ /2')

50.0
```

Now let's look at how vertex properties can be used by a 'math' step. To do this, we can also use named traversal steps as part of a 'math' operation. The examples below start by checking how many runways the DFW and SFO airports have. Then a 'math' step is used to show how we can add those values together as part of a traversal.

```
// How many runways does DFW have?
g.V().has('airport','code','DFW').values('runways')

7
```

```
// How many runways does SFO have?
g.V().has('airport','code','SFO').values('runways')

4
```

Now let's use 'math' to add the values together as part of a single traversal. Note that even though we are adding two integers together, the result comes back as a double precision value. Also, note that the named steps "a" and "b" are specified inside the single string that is passed to the math step. This is a key difference from all other steps where we refer to one or more traversal labels inside a step. Lastly, notice that a 'by' modulator is used to tell the 'math' step which properties we want to add together.

```
// Use named steps to add some results together.
g.V().has('airport','code','DFW').as('a').
      out().has('code','SFO').as('b').
      math('a + b').by('runways')

11.0
```

The 'math' step returns a double. If you prefer an integer value, you can always convert it with 'asNumber'.

```
// Use named steps to add some results together.
g.V().has('airport','code','DFW').as('a').
      out().has('code','SFO').as('b').
      math('a + b').by('runways').
      asNumber(GType.INT)

11
```

The examples below show division and modulo operators being used on the results of a 'count' step.

```
g.V(3).out().count()

98

g.V(3).out().count().math('_ / 2')

49.0

g.V(3).out().count().math('_ % 5')

3.0
```

Note that the underscore character allowed us to avoid having to write the previous queries using a

pattern like the one used below where the count step is labeled as "a".

```
g.V(3).out().count().as('a').math('a / 2')

49.0
```

3.36.2. Using a 'by' modulator with a 'math' step

As with many Gremlin steps, a 'math' step can also be combined with one or more 'by' modulators. First, let's write a simple query to inspect the number of runways and Santa Fe (SAF) and all the places that you can fly to from there.

```
g.V().has('code','SAF').as('a').
      out().as('b').
      select('a','b').
        by(values('code','runways').fold())

[a:[SAF,3],b:[DFW,7]]
[a:[SAF,3],b:[LAX,4]]
[a:[SAF,3],b:[PHX,3]]
[a:[SAF,3],b:[DEN,6]]
```

Now let's modify the query to use a 'math' step to add the number of runways each pair of airports has.

```
g.V().has('code','SAF').as('a').
      out().as('b').
      math('a + b').by('runways').
      asNumber(INT)

10
7
6
9
```

Of course, this is a simple example where 'math' is not really needed, but hopefully it shows how a 'by' modulator can be used with a 'math' step. For completeness, here is the query rewritten to use a 'sum' step.

```
g.V().has('code','SAF').as('a').
      out().as('b').
      select('a','b').by('runways').select(values).sum(local)

10
7
6
```

```
9
```

3.36.3. Converting feet to meters

The length of the longest runway at each airport in the graph is stored as units of feet. We can use a simple 'math' step to convert that value to meters.

```
g.V().has('code','DFW').values('longest').math('_ * 0.3048')

4084.6248
```

We could make the result a bit more interesting by adding a 'project' step to the query.

```
g.V().has('code','DFW').
      project('Longest runway at','feet','meters').
        by('code').
        by('longest').
        by(values('longest').math('_ * 0.3048'))

[Longest runway at:DFW,feet:13401,meters:4084.6248]
```

3.36.4. Using the trigonometric functions

The trigonometric operators work as you would expect. All angles need to be specified as radians and not degrees. You can do the conversion to radians yourself or use the Java 'Math.toRadians' helper method if you prefer. The query below uses the 'math' step to calculate the sine of 60 degrees and stores the result in a variable called '"x"'.

```
// Calculate the sine of 60 degrees
x=g.inject(60*(Math.PI/180)).math('sin(_)').next()

0.8660254037844386
```

We can use our variable '"x"' to calculate the arcsine.

```
// Calculate the arcsine
g.inject(x).math('asin(_)')

1.0471975511965976
```

We can use the Gremlin console as a calculator to prove that we got the correct answer back.

```
// Prove this is the right answer
Math.toRadians(60)
```

```
1.0471975511965976
```

Note that just as when using the Java Math library, you have to be aware of possible rounding errors. You would expect the calculation below to return 1.0, but it does not as the conversion of 45 degrees to radians is not precise enough for the Math library. Note that using the Java 'Math.toRadians' method does not achieve the desired result either.

```
// Manual conversion
g.inject(45*(Math.PI/180)).math('tan(_)')

0.9999999999999999
```

Same experiment but using the helper method.

```
// Library conversion
g.inject(Math.toRadians(45)).math('tan(_)')

0.9999999999999999
```

This presents us with a chance to experiment with another of the operators that the 'math' step provides. We can use the 'ceil' operator to round our result up to the nearest integer.

```
g.inject(Math.toRadians(45)).math('tan(_)').math('ceil(_)')

1.0
```

3.36.5. Using signum to make a choice

The 'signum' operator allows us to make a decision depending upon whether a numeric value is positive, negative or zero as shown below.

```
g.inject(-10).math('signum(_)')
-1.0

g.inject(10).math('signum(_)')
1.0

g.inject(0).math('signum(_)')
0.0
```

We can use this capability to build a query that reports which side of the Greenwich meridian an airport is located. Note that we had to use the '"D"' suffix on the numbers in the 'option' steps as 'math' returns a double precision result.

```
g.V().hasLabel('airport').sample(12).
      project('IATA','city','position').
        by('code').
        by('city').
        by(choose(values('lon').math('signum(_)')).
          option(0D,constant('on the meridian')).
          option(1D,constant('East')).
          option(-1D,constant('West')))
```

Here is an example of the output from running the query.

```
[IATA:MJZ,city:Mirny,position:East]
[IATA:ULN,city:Ulaanbaatar,position:East]
[IATA:AHU,city:Al Hoceima,position:West]
[IATA:AMA,city:Amarillo,position:West]
[IATA:JTC,city:Bauru,position:West]
[IATA:PIZ,city:Point Lay,position:West]
[IATA:UZR,city:Urdzhar,position:East]
[IATA:MNT,city:Minto,position:West]
[IATA:MKE,city:Milwaukee,position:West]
[IATA:PSA,city:Pisa,position:East]
[IATA:FWA,city:Fort Wayne,position:West]
[IATA:DBQ,city:Dubuque,position:West]
```

3.36.6. Calculating a standard deviation

We could use the new 'math' step to implement a query that calculates the standard deviation for the number of runways each airport in the graph has. This allows us to see use of the 'sqrt' and power (^) operators. We broke the solution into three queries rather than try to force it all into one. Even now the final query of the three is complicated enough! Notice how multiple 'math' steps are used in the same query with the results from one being used as input to the next.

First, let's calculate the mean (or average) number of runways in the graph. Not surprisingly, this number is close to 1.5 as the majority of the airports only have one or two runways.

```
// Average number of runways
mean=g.V().hasLabel('airport').values('runways').mean().next()

1.4212328767123288
```

We also need to know how many airports there are in the graph so that we can calculate the variance as part of the standard deviation calculation.

```
// Total number of airports
count = g.V().hasLabel('airport').count().next()
```

```
3504
```

Now we are ready to make use of the square root and power operators and calculate the standard deviation. As a reminder, the standard deviation is found by taking the square root of the variance in a data set. The variance itself is calculated by for each airport subtracting the mean from the number of runways it has and squaring it and then taking the sum of those values and finally dividing that sum by the number of airports. Let's write a query that can do all of that for us.

```
// Calculate the standard deviation
g.withSideEffect("m",mean).
  withSideEffect("c",count).
  V().hasLabel('airport').values('runways').
  math('(_ - m)^2').sum().math('_ / c').math('sqrt(_)')

0.7429770644987477
```

We could use another query to check on the distribution of runways in the graph to see if we believe our standard deviation result.

```
 g.V().hasLabel('airport').groupCount().by('runways')

[1:2429,2:775,3:227,4:53,5:14,6:4,7:2]
```

Looking at the distribution, where a large majority of the airports have either one or two runways, our result looks pretty reasonable. The few airports with six, seven or eight runways are the outliers in this sample and would fall well outside the standard deviation from the mean that we calculated.

Just for fun, let's use the same basic set of steps once again, but this time to find the standard deviation for the number of outgoing routes in the graph.

As before, we need to find the mean value for the data set. This time we need to find the average number of outgoing routes in the graph. The airport count remains the same, of course.

```
mean=g.V().hasLabel('airport').local(out().count()).mean().next()

14.451198630136986

count = g.V().hasLabel('airport').count().next()

3504
```

Now we are ready to again calculate the standard deviation for the data set representing all outgoing routes per airport.

```
g.withSideEffect("m",mean).
```

```
  withSideEffect("c",count).
  V().hasLabel('airport').local(out().count()).
  math('(_ - m)^2').sum().math('_ / c').math('sqrt(_)')

32.00790808644156
```

This time we got a much bigger number back as a result compared to when we looked at runways. This reflects the differing distribution of routes between major and more minor airports.

3.36.7. Calculating a standard deviation in one query

In the previous section the steps to calculate the standard deviation were broken up into three graph queries. It is possible to perform the entire task in a single query as shown below.

```
g.V().hasLabel('airport').
      values('runways').fold().as('runways').
      mean(local).as('mean').
      select('runways').unfold().
      math('(_-mean)^2').mean().math('sqrt(_)')

0.7429770644987477
```

Using this approach means that we can avoid making multiple round trips to the graph to generate the values we need. The query is not really a lot more complex either. Which technique you find more convenient may come down to personal preference. Making multiple queries in general is not always a bad thing, but in this case using a single query we think makes sense as it adds little additional complexity to the steps involved.

3.36.8. Using 'project' to feed values to 'math'

The 'project' step can be used to create a map of key/value pairs that can in turn be passed to a 'math' step. First, let's create a query that generates a simple projection containing the number of incoming and outgoing routes at the Austin airport.

```
g.V().has('code','AUS').
      project('in','out').
        by(__.in('route').count()).
        by(out('route').count())

[in:98,out:98]
```

We can now add a 'math' step that uses the key names from the map created by the 'project' step and adds their values together.

```
g.V().has('code','AUS').
      project('in','out').
        by(__.in('route').count()).
```

```
        by(out('route').count()).
      math('in + out')

196.0
```

There are obviously many other operators that we have not provided examples for, but hopefully the ones we have provided give you a feel for ways that the 'math' step can be used to create interesting queries.

3.37. Including an index with results – introducing 'withIndex' and 'indexed'

If for any reason you want an index value included as part of the results from a query, you can use the Groovy 'withIndex' or 'indexed' methods as shown below.

The 'withIndex' method adds the index value at the end of a list, whereas the 'indexed' method adds it at the start. You can provide a starting index value as a parameter. If no value is provided, the default value for the first index will be zero.

```
g.V().has('region','US-OK').values('code').withIndex()

[OKC,0]
[TUL,1]
[LAW,2]
[SWO,3]
```

Note that 'indexed' and 'withIndex' are Groovy methods and not Gremlin traversal steps. They will only work using graph databases that allow Groovy code to be included as part of a query.

Here is the same query as before but using 1 as the starting index.

```
g.V().has('region','US-OK').values('code').withIndex(1)

[OKC,1]
[TUL,2]
[LAW,3]
[SWO,4]
```

Below is the query used again but this time with the 'indexed' method being used to generate the index value.

```
g.V().has('region','US-OK').values('code').indexed(1)

[1,OKC]
```

```
[2,TUL]
[3,LAW]
[4,SWO]
```

3.37.1. The 'index' step

The 'index' step adds a counter value to a 'list' or 'map' object, and usage only really makes sense in that context. As shown below, without a 'fold' step in the query, the result set will consist of a number of individual lists all with an index of zero.

The official Apache TinkerPop documentation for the 'index' step can be found at the following link https://tinkerpop.apache.org/docs/current/reference/#index-step.

```
g.V().has('code','LHR').
      out().limit(5).
      values('code').index()

[[YUL,0]]
[[YEG,0]]
[[CGN,0]]
[[GOT,0]]
[[VCE,0]]
```

If we had a 'fold' step before calling 'index', however, the results will be indexed in increments of one, starting at zero.

```
g.V().has('code','LHR').
      out().limit(5).
      values('code').fold().index()

[[YUL,0],[YEG,1],[CGN,2],[GOT,3],[VCE,4]]
```

Adding an 'unfold' step will yield a set of individual lists with each containing an airport code and its index.

```
g.V().has('code','LHR').
      out().limit(5).
      values('code').fold().index().
      unfold()

[YUL,0]
[YEG,1]
[CGN,2]
[GOT,3]
[VCE,4]
```

The new 'with' modulator can be used to control the type of collection that 'index' produces. To have the result returned as a map where the key is the index value, the query can be modified as follows.

```
g.V().has('code','LHR').
      out().limit(5).
      values('code').fold().
      index().with(WithOptions.indexer,WithOptions.map)

[0:YUL,1:YEG,2:CGN,3:GOT,4:VCE]
```

Finally, the example below shows a 'with' step being used to ask for results as a list which is the default. The results are then sorted in reverse order.

All the possible values that can be specified using 'WithOptions' can be found in the official Apache TinkerPop Javadoc documentation at this location.

Note that this query uses 'desc' rather than the now deprecated 'decr' to ask for descending order results.

```
g.V().has('code','LHR').
      out().limit(5).
      values('code').fold().
      index().with(WithOptions.indexer,WithOptions.list).
      unfold().
      order().by(tail(local,1),desc)

[VCE,4]
[GOT,3]
[CGN,2]
[YEG,1]
[YUL,0]
```

3.37.2. Using 'index' to reverse a list

The Gremlin query language does not have a built-in step or function that can be used to reverse the contents of a list or other collection. However, this can be achieved using the 'index' step. The example below takes advantage of the index step to give each element of a list an index number.

```
g.inject(['A','B','C','D']).index()

[[A,0],[B,1],[C,2],[D,3]]
```

Given that building block, we can use those index values to order the list.

```
g.inject(['A','B','C','D']).index().
```

```
  unfold().
  order().
    by(tail(local,1),desc)

[D,3]
[C,2]
[B,1]
[A,0]
```

The query can be further refined so that the index values are not part of the result.

```
g.inject(['A','B','C','D']).index().
  unfold().
  order().
    by(tail(local,1),desc).
  limit(local,1).
  fold()

[D,C,B,A]
```

The same technique can be used with any collection generated as part of a query.

```
g.V().has('code','SAF').out().values('code').fold().index()

[[DFW,0],[LAX,1],[PHX,2],[DEN,3]]
```

Once again we can order the results using the index values.

```
g.V().has('code','SAF').
  out().values('code').fold().
  index().
  unfold().
  order().
    by(tail(local,1),desc).
  limit(local,1).
  fold()

[[DEN],[PHX],[LAX],[DFW]]
```

3.38. More examples using concepts we have covered so far

The examples in this section build upon the topics that we have covered so far. The query below finds cities that can be flown to from any airport in the Hawaiian islands.

```
// Which cities can I fly to from any airport in the Hawaiian islands?
g.V().has('airport','region','US-HI').out().path().by('city')
```

If we run the query, we would get back results similar to those below. Only a few of the full result set are shown.

```
[Honolulu,Brisbane]
[Honolulu,Shanghai]
[Honolulu,Kuala Lumpur]
[Honolulu,Tokyo]
[Honolulu,Seoul]
[Honolulu,Portland]
[Honolulu,Kahului]
[Honolulu,Hagta]
[Honolulu,Taipei]
...
```

The query below looks for airports in Europe using the continent vertex with a code of 'EU' as the starting point. The results are sorted in ascending order and folded into a list.

```
// Find all the airports that are in Europe (The graph stores continent information
// as "contains" edges connected from a "continent" vertex to each airport vertex.
g.V().has('continent','code','EU').out('contains').values('code').order()
```

Here are some of the results that you get back from running the query.

```
AAL
AAQ
AAR
ABZ
ACE
ACH
ACI
AER
AES
AEY
AGB
AGF
AGH
AGP
AHO
AJA
AJR
ALC
...
```

The next queries show two ways of finding airports with 6 or more runways.

```
g.V().where(values('runways').is(gte(6))).values('code')

g.V().has('runways',gte(6)).values('code')
```

Next, let's look at two ways of finding flights from airports in South America to Miami. The first query uses 'select', which is what we would have had to do in the TinkerPop 2 days. The second query, which feels cleaner to me, uses the 'path' and 'by' step combination introduced in TinkerPop.

```
g.V().has('continent','code','SA').out().as('x').out().as('y').
      has('code','MIA').select('x','y').by('code')

g.V().has('continent','code','SA').out().out().
      has('code','MIA').path().by('code')
```

This query finds the edge that connects Austin (AUS) with Dallas Ft. Worth (DFW) and returns the 'dist' property of that edge so we know how far that journey is.

```
// How far is it from DFW to AUS?
g.V().has('code','DFW').outE().as('a').
  inV().has('code','AUS').select('a').values('dist')

190
```

As an alternative approach, we could return a path that would include both airport codes and the distance. Notice how we need to use 'outE' and 'inV' rather than just 'out' as we still need the edge to be part of our path so that we can get its 'dist' property using a 'by' step.

```
g.V().has('code','DFW').outE().inV().has('code','AUS').path().by('code').by('dist')

[DFW,190,AUS]
```

If we wanted to find out if there are any ways to get from Brisbane to Austin with only one stop, this query will do nicely!

```
// Routes from BNE to AUS with only one stop
g.V().has('code','BNE').out().out().has('code','AUS').path().by('code')

[BNE,LAX,AUS]
[BNE,SFO,AUS]
[BNE,HNL,AUS]
[BNE,YVR,AUS]
```

This is another way of doing the same thing but, once again, to me using 'path' feels more concise.

The only advantage of this query is that if all you want is the name of any intermediate airports, then that's all you get!

```
g.V().has('code','BNE').out().as('stop').
                        out().has('code','AUS').select('stop').values('code')

LAX
SFO
HNL
YVR
```

A common thing you will find yourself doing when working with a graph is counting things. This next query looks for all the airports that have fewer than five outgoing routes and counts them. It does this by counting the number of airports that have fewer than five outgoing edges with a 'route' label. There are a surprisingly high number of airports that offer this small number of destinations.

```
// Airports with fewer than 5 outgoing edges
g.V().hasLabel('airport').where(out('route').count().is(lt(5))).count()

2046
```

In a similar vein, this query finds the airports with more than 200 outgoing routes. The second query shows that 'where' is a synonym for 'filter' in many cases.

```
g.V().hasLabel("airport").where(outE('route').count().is(gt(200))).values('code')

g.V().hasLabel("airport").filter(outE("route").count().is(gt(200))).values('code')
```

Here are two more queries that look for things that meet specific criteria. The first finds routes where the distance is exactly 100 miles and returns the source and destination airport codes. The first query uses 'as' and 'select' while the second one uses 'path' and includes the distance in the result.

```
// List ten (or less) routes where the distance is exactly 100 miles
g.V().as('a').outE().has('dist',eq(100)).limit(10).inV().as('b').
      select('a','b').by('code')

[a:IAD,b:RIC]
[a:HNL,b:OGG]
[a:OGG,b:HNL]
[a:SJO,b:LIR]
[a:SVG,b:KRS]
[a:RIC,b:IAD]
[a:LIR,b:SJO]
[a:KRS,b:SVG]
[a:CYB,b:GCM]
```

```
[a:GCM,b:CYB]
```

Similar to the prior query but using 'path' and displaying the distance as well as the airport codes.

```
g.V().outE().has('dist',eq(100)).limit(10).inV().path().by('code').by('dist')

[IAD,100,RIC]
[HNL,100,OGG]
[OGG,100,HNL]
[SJO,100,LIR]
[SVG,100,KRS]
[RIC,100,IAD]
[LIR,100,SJO]
[KRS,100,SVG]
[CYB,100,GCM]
[GCM,100,CYB]
```

This query looks for any airports that have an elevation above 10,000 feet. Two ways of achieving the more or less the same result are shown. The first uses 'valueMap' and the second uses a 'project' step instead.

```
// Airports above 10,000ft sorted by ascending elevation
g.V().has('airport','elev', gt(10000)).
      order().by('elev',asc).valueMap('city','elev')


g.V().has('airport','elev', gt(10000)).order().by('elev',asc).
      project('city','elevation').by('city').by('elev')
```

If we ran the query that uses the 'project' step, here is what we should get back.

```
[city:Xiahe,elevation:10510]
[city:Leh,elevation:10682]
[city:Shangri-La,elevation:10761]
[city:Cusco,elevation:10860]
[city:Jauja,elevation:11034]
[city:Andahuaylas,elevation:11300]
[city:Jiuzhaigou,elevation:11327]
[city:Navoi,elevation:11420]
[city:Hongyuan,elevation:11598]
[city:Lhasa,elevation:11713]
[city:Oruro,elevation:12152]
[city:Xigaze,elevation:12408]
[city:Golog,elevation:12427]
[city:Juliaca,elevation:12552]
[city:Yushu,elevation:12816]
[city:Potosí,elevation:12913]
```

```
[city:Quijarro,elevation:12972]
[city:La Paz / El Alto,elevation:13355]
[city:Shiquanhe,elevation:14022]
[city:Kangding,elevation:14042]
[city:Bangda,elevation:14219]
[city:Daocheng,elevation:14472]
```

The next query finds any routes between Austin and Sydney that only require one stop.The 'by' step offers a clean way of doing this query by combining it with 'path'.

```
g.V().has('code','AUS').out().out().has('code','SYD').path().by('code')
```

The following three queries all achieve the same result. They find flights from any airport in Africa to any airport in the United States. These queries are interesting as the continent information is represented in the graph as edges connecting an airport vertex with a continent vertex. This is about as close as Gremlin gets to a SQL join statement!

The first query starts by looking at airports the second starts from the vertex that represents Africa. The third query uses 'where' to show an alternate way of achieving the same result.

```
g.V().hasLabel('airport').as('a').in('contains').has('code','AF').
      select('a').out().has('country','US').as('b').select('a','b').by('code')

g.V().hasLabel('continent').has('code','AF').out().as('a').
      out().has('country','US').as('b').
      select('a','b').by('code')

g.V().hasLabel('airport').where(__.in('contains').has('code','AF')).as('a').
      out().has('country','US').as('b').select('a','b').by('code')
```

If we run the query, we should get results that look like this.

```
[a:CPT,b:EWR]
[a:JNB,b:ATL]
[a:JNB,b:JFK]
[a:JNB,b:EWR]
[a:NBO,b:JFK]
[a:CAI,b:JFK]
[a:ADD,b:IAD]
[a:ADD,b:EWR]
[a:LOS,b:ATL]
[a:LOS,b:IAD]
[a:LOS,b:IAH]
[a:LOS,b:JFK]
[a:ACC,b:IAD]
[a:ACC,b:JFK]
[a:CMN,b:IAD]
```

```
[a:CMN,b:JFK]
[a:CMN,b:MIA]
[a:GCK,b:DFW]
[a:SID,b:IAD]
[a:DKR,b:IAD]
[a:DKR,b:JFK]
[a:ABJ,b:EWR]
[a:LFW,b:IAH]
[a:LFW,b:LAX]
[a:LFW,b:EWR]
[a:RAI,b:BOS]
[a:DSS,b:IAD]
```

The query below shows how to use the 'project' step, along with 'order' and 'select' to produce a sorted table of airports you can fly to from AUSTIN along with their runway counts. The 'limit' step is used to only return the top ten results. You will find several examples elsewhere in this book that use variations of this collection of steps.

```
g.V().has('code','AUS').out().project('ap','rw').by('code').by('runways').
      order().by(select('rw'),desc).limit(10)
```

Here are the results we get from running the query.

```
[ap:DFW,rw:7]
[ap:ORD,rw:7]
[ap:DTW,rw:6]
[ap:BOS,rw:6]
[ap:DEN,rw:6]
[ap:AMS,rw:6]
[ap:MKE,rw:5]
[ap:MDW,rw:5]
[ap:ATL,rw:5]
[ap:IAH,rw:5]
```

Chapter 4. BEYOND BASIC QUERIES

So far we have looked mostly at querying an existing graph. In the following sections we will look at many other topics that it is also important to be familiar with when working with Gremlin. These topics include mixing in some Groovy or Java code with your queries, as well as adding vertices, edges, and properties to a graph and also deleting them. We will also look at how to create a sub-graph and how to save a graph to an XML or JSON file and a lot more. Let's start off with a short discussion of query layout, reserved words, and data modeling.

4.1. A word about layout and indentation

As you begin to write more complex Gremlin queries, they can get quite lengthy. To make them easier for others to read, it is recommended to spread them over multiple lines and indent them in a way that makes sense. We are not going to propose an indentation standard, we believe this should be left to personal preference, however, there are a few things we want to mention in passing. When working with the Gremlin Console, if you want to spread a query over multiple lines, then you will need to end each line with a backslash character or with a character such as a period or a comma that tells the Gremlin parser that there is more to come.

TinkerPop offers a Gremlin formatting tool called Gremlint - https://tinkerpop.apache.org/gremlint/ - which will apply some basic spacing and indenting to make Gremlin more readable.

The following example shows the query we already looked at in the Boolean operations section of this book but this time edited so that it could be copied and pasted directly into the Gremlin console.

```
g.V().hasLabel('airport') \
     .has('region',within('US-TX','US-LA','US-AZ','US-OK')) \
     .order().by('region',asc)   \
     .valueMap().select('code','region')
```

We can avoid the use of backslash characters if we lay the query out as follows. Each line ends with a period which tells the parser that there are more steps coming.

```
g.V().hasLabel('airport').
  has('region',within('US-TX','US-LA','US-AZ','US-OK')).
  order().by('region',asc).
  valueMap().select('code','region')
```

If we do not give the parser one of these clues that there is more to come, the Gremlin console will try and execute each line without waiting for the next line.

Some people find it easier to read queries when each step or modulator is given its own line and indented appropriately. So we could lay out the query as shown below, and it will still work just fine.

```
g.V().hasLabel('airport').
      has('region',within('US-TX','US-LA','US-AZ','US-OK')).
      order().
        by('region',asc).
      valueMap().
      select('code','region')
```

Whether you decide to use the backslash as a continuation character or leave the period on the previous line is really a matter of personal preference. Just be sure to do one or the other if you want to use multiple line queries within the Gremlin console. There is no golden rule as to how many lines and how much indenting you should use when laying out your more complex queries. However, whatever you decide to do, it is worth remembering that others reading your work may find well-laid-out and appropriately indented steps easier to read and understand.

4.2. A word about idiomatic Gremlin

As you may have seen in earlier sections of this book, there are often several ways to write a Gremlin traversal that will obtain the same result. A great example was shown in the Counting groups of things Section where we saw these two queries that returned an identical result:

```
// How many of each type of vertex are there?
g.V().groupCount().by(label)

g.V().label().groupCount()

g.V().group().by(label).by(count())
```

We could expand that list further with other forms as well, so the list does not really even end there. Much of Gremlin's power lies in its flexibility as a language, but with that flexibility comes a wider number of options that you get to choose from when you write your traversals.

When we think about the idiomatic way to write a particular Gremlin traversal, it is often best to think about the readability aspect first. What Gremlin forms will be most widely recognized and best convey my intent with the least introspection? There aren't wide, hardened rules for this particularly, but you will notice them as you continue to read this book and the official TinkerPop Documentation.

Let's consider the example above which determines an answer to the question: "How many of each type of vertex are there?" In this example, most experienced Gremlin developers would reach for the first traversal of 'g.V().groupCount().by(label)' as the most idiomatic, because, unlike the second, it simply reads left-to-right like SQL and, unlike the third, prefers 'groupCount', which is a step intentionally meant for this task rather than pairing 'group' with a separate 'count'.

Intention is perhaps a second part to consider when thinking about idiomatic Gremlin. When we prefer a step intentionally meant to serve a specific task, we not only provide better readability, but we also increase the chance that the underlying graph database will recognize that intention and optimize the traversal's performance. When a graph database gets this query, most of them will

analyze its structure to see if there is anything it can do to make the query run faster. Depending upon how advanced that analysis is, it may or may not recognize that all three of those queries mean the same thing. You increase the likelihood of recognition if you prefer encoding intent as a single step, rather than a pattern of several steps.

We've used 'groupCount' here as an example, but these concepts apply to many other steps. For example, you could use a complex pattern to do an 'upsert' of a vertex using 'coalesce' and 'fold', but it is more likely your upsert would be optimized with a single 'mergeV' which is meant to explicitly handle that functionality.

Idiomatic Gremlin will continue to emerge and evolve as the community and the language grow, so it is not something you learn once and are finished with. As you read this book and explore other Gremlin sources, pay attention to the patterns that feel natural, expressive, and intentional, and favor those when you write your own traversals. Over time, that habit will make your Gremlin easier for others to understand and maintain, and will often give your graph database the best opportunity to run your traversals efficiently.

4.3. A warning about reserved word conflicts and collisions

Most of the time the issue we are about to describe will not be a problem. However, there are cases where names of Gremlin steps conflict with reserved words and method names in Groovy. Remember that Gremlin is coded in Groovy and Java. If you hit one of these cases, often the error message that you will get presented with does not make it at all clear that you have run into this particular issue. Let's look at some examples. A step name in Gremlin that can sometimes run into this naming conflict is the 'in' step. However, you do not have to worry about this in all cases. First, take a look at the following query.

```
g.V().has('code','AUS').in()
```

That query does not cause an error and correctly returns all the vertices that are connected by an incoming edge, to the 'AUS' vertex. There is no conflict of names here because it is clear that the 'in' reference applies to the result of the has step. However, now take a look at this query.

```
g.V().has('code','AUS').union(in(),out())
```

In this case the 'in' is on its own and not 'dot connected' to a previous step. The Gremlin runtime (which remember is written in Groovy) will try to interpret this and will throw an error because it thinks this is a reference to its own 'in' method. To make this query work, we have to adjust the syntax slightly as follows.

```
g.V().has('code','AUS').union(__.in(),out())
```

Notice that we added the '"__." (underscore underscore period) in front of the 'in' step. This is shorthand for '"the thing we are currently looking at"', so in this case, the result of the 'has' step.

There are currently not too many Groovy reserved words to worry about. The three that you have to watch out for are 'in', 'not' and 'as' which have special meanings in both Gremlin and Groovy. Remember, though, you will only need to use the '"__."' notation when it is not clear what the reserved word, like 'in', applies to.

You will find an example of 'not' being used with the '"__."' prefix in the "Modeling an ordered binary tree as a graph" section a bit later on.

4.4. Thinking about your data model

As important as it is to become good at writing effective Gremlin queries, it is equally important, if not more so, to put careful consideration into how you model your data as a graph. Ideally, you want to arrange your graph so that it can efficiently support the most common queries that you foresee it needing to handle.

Consider this query description. "Find all flight routes that exist between airports anywhere in the continent of Africa and the United States". When putting the 'air-routes' graph together, we decided to model continents as their own vertices. So each of the seven continents has a vertex. Each vertex is connected to airports within that continent by an edge labeled "contains".

We could have chosen to just make the continent a property of each airport vertex, but had we done that, to answer the question about "routes starting in Africa", we would have to look at every single airport vertex in the graph just to figure out which continent contained it. By giving each continent its own vertex, we are able to greatly simplify the query we need to write.

Take a look at the query below. We first look just for vertices that are continents. We then only look at the Africa vertex and the connections it has (each will be to a different airport). By starting the query in this way, we have very efficiently avoided looking at a large number of the airports in the graph altogether. Finally, we look at any routes from airports in Africa that end up in the United States. This turns out to yield a nice and simple query in no small part because our data model in the graph made it so easy to do.

```
// Flights from any Airport in Africa to any airport in the United States
g.V().hasLabel('continent').has('code','AF').out().as('a').
  out().has('country','US').as('b').
  select('a','b').by('code')
```

We could also have started our query by looking at each airport and looking to see if it is in Africa, but that would involve looking at a lot more vertices. The point to be made here is that even if our data model is good, we still need to always be thinking about the most efficient way to write our queries.

```
// Gives same results but not as efficient
g.V().hasLabel('airport').as('a').in('contains').has('code','AF').
      select('a').out().has('country','US').as('b').select('a','b').by('code')
```

Now for a fairly simple graph, like 'air-routes', this discussion of efficiency is perhaps not such a big

deal, but as you start to work with large graphs, getting the data model right can be the difference between good and bad query response times. If the data model is bad, you won't always be able to work around that deficiency simply by writing clever queries!

4.4.1. Keeping information in two places within the same graph

Sometimes, to improve query efficiency, we find it is actually worth having the data available more than one place within the same graph. An example of this in the air-routes graph would be the way we decided to model countries. We have a unique vertex for each country, but we also store the country code as a property of each airport vertex. In a small graph this perhaps is overkill, but we did it to make a point. Look at the following two queries that return the same results – the cities in Portugal that have airports in the graph.

```
g.V().has('country','code','PT').out("contains").values('city')

g.V().has('airport','country','PT').values('city')
```

The first query finds the country vertex for Portugal and then finds all the countries connected to it. The second query looks at all airport vertices and looks to see if they contain 'PT' as the country property.

In the first example it is likely that a lot fewer vertices will get looked at than the first even though a few edges will also get walked as there are over 3,000 airport vertices but fewer than 300 country vertices. Also, in a production system with an index in place, finding the 'Portugal' vertex should be extremely fast.

Conversely, if we were already looking at an airport vertex for some other reason and just wanted to see what country it is in, it is more convenient to just look at the 'country' property of that vertex.

So there is no golden rule here, but it is something to think about while designing your data model.

4.4.2. Using a graph as an index into other data sources

While on the topic of what to keep in the graph, something to resist being drawn into in many cases is the desire to keep absolutely everything in the graph. For example, in the air-routes graph we do not keep every single detail about an airport (radio frequencies, runway names, weather information, etc.) in the airport vertices. That information is available in other places and easy to find. In a production system you should consider carefully what needs to be in your graph and what more naturally belongs elsewhere. One thing we could do is add a URL as a property of each airport vertex that points to the airports home page or some other resource that has all the information. In this way the graph becomes a high-quality index into other data sources. This is a common and useful pattern when working with graphs. This model of having multiple data sources working together is sometimes referred to as 'Polyglot storage'.

4.4.3. A few words about 'supernodes'

When a vertex in a graph has a large number of edges and is disproportionately connected to many of the other vertices in the graph, it is likely that many, if not all, graph traversals of any

consequence will include that vertex. Such vertices are often referred to as 'supernodes'. In some cases the presence of 'supernodes' may be unavoidable, but with careful planning as you design your graph model, you can reduce the likelihood that vertices become 'supernodes'. The reason we worry about 'supernodes' is that they can significantly impact the performance of graph traversals. This is because it is likely that any graph traversal that goes via such a vertex will have to look at most if not all the edges connected to that vertex as part of a traversal.

The 'air-routes' graph does not really have anything that could be classed as a 'supernode'. The vertex with the most edges is the continent vertex for North America that has approximately 980 edges. The busiest airports are IST and AMS, and they both have just over 530 total edges. So in the case of the 'air-routes' graph, we do not have to worry too much.

If we were building a graph of a social network that included famous people, we might have to worry. Consider that some people on Twitter with millions of followers. Without taking precautions, such a social network, modeled as a graph, could face issues.

As you design your graph model, it is worth considering that some things are perhaps better modeled as a vertex property than as a vertex with lots of edges needing to connect to it. For example, in the air-routes graph there are country vertices, and each airport is connected to one of the country vertices. In the air-routes graph this is not a problem as even if all the airports in the graph were in the same country, that would still give us fewer than 3,500 edges connected to that vertex. However, imagine if we were building a graph of containing a very large number of people. If we had several million people in the graph all living in the same country, that would be a guaranteed way to get a 'supernode' if we modeled that relationship by connecting every person vertex to a country vertex using a 'lives in' edge. In such situations, it would be far more sensible to make the country where a person lives a property of their own vertex.

A detailed discussion of 'supernode' mitigation is beyond the scope of this book, but we encourage you to always be thinking about their possibility as you design your graph and also be thinking about how you can prevent them becoming a big issue for you.

4.5. Making Gremlin even Groovier

As we have already discussed, the Gremlin console builds upon the Groovy console, and Groovy itself is coded in Java. This means that all the classes and methods that you would expect to have available while writing Groovy or Java programs are also available to you as you work with the Gremlin Console. You can intermix additional features from Groovy and Java classes along with the features provided by the TinkerPop classes as needed. This capability makes Gremlin additionally powerful. You can also take advantage of these features when working with Gremlin Server and with other TinkerPop enabled graph services with the caveat that some features may be blocked if viewed as a potential security risk to the server or simply because they are not supported.

Every Gremlin query we have demonstrated so far is also, in reality, valid Groovy. We have already shown examples of storing values into variables and looping using Groovy constructs as part of a single or multipart Gremlin query.

In this section we are going to go one step further and actually define some methods, using Groovy syntax, that can be run while still inside the Gremlin Console. By way of a simple example, let’s define a method that will tell us how far apart two airports are and then invoke it.

```
// A simple function to return the distance between two airports
def dist(g,from,to) {
  d=g.V().has('code',from).outE().as('a').inV().has('code',to)
         .select('a').values('dist').next()
  return d }

// Can be called like this
dist(g,'AUS','MEX')
```

This next example shows how to define a slightly longer method that prints out information about the degree of a vertex in a nice, human-readable, form.

```
// Groovy function to display vertex degree
def degree(g,s) {
  v = g.V().has('code',s).next();
  o=g.V(v).out().count().next();
  i=g.V(v).in().count().next() ;
  println "Edges in  : " + i;
  println "Edges out : " + o;
  println "Total     : " +(i+o);
}

// Can be called like this
degree(g,'LHR')
```

Here is an example that shows how we can query the graph, get back a list of values, and then use a 'for' loop to display them. Notice this time how we initially store the results of the query into the variable 'x'. The call to 'toList' ensures that 'x' will contain a list (array) of the returned values.

```
// Using a Groovy for() loop to iterate over a list returned by Gremlin
x=g.V().hasLabel('airport').limit(10).toList()
for (a in x) {
  println(a.values('code').next() + " " + a.values('icao').next() + " " + a.values
('desc').next())
}

// We can also do this just using a 'for' loop and not storing anything into a
variable.
for (a in g.V().hasLabel('airport').limit(10).toList()) {
  println(a.values('code').next()+""+a.values('icao').next())
}
```

Sometimes, as you have seen above, it is necessary to make a call to 'next' to get the result you expect returned to your variable.

```
number = g.V().hasLabel('airport').count().next()
```

```
println "The number of airports in the graph is " + number
```

Here is another example that makes a Gremlin query inside a 'for' loop.

```
for (a in 1..10) print g.V().has(id,a).values('code').next()+" "
```

This example returns a hash of vertices, with vertex labels as the keys and the code property as the values. It then uses the label names to access the returned hash.

```
a=g.V().group().by(label).by('code').next()
println(a["country"].size())
println(a["country"][5])
println(a["airport"][2])
```

Here is another example. This time we define a method that takes as input a traversal object and the code for an airport. It then uses those parameters to run a simple Gremlin query to retrieve all the places that you can fly to from that airport. It then uses a simple 'for' loop to print the results in a table. Note the use of 'next' as part of the 'println'. This is needed to get the actual values that we are looking for. If we did not include the calls to 'next', we would actually get back the iterator object itself and not the actual values.

```
// Given a traversal and an airport code print a list of all the places you can
// fly to from there including the IATA code and airport description.
def from(g,a) {
  places=g.V().has('code',a).out().toList();
  for (x in places) {println x.values('code').next()+" "+x.values('desc').next()}
}

// Call like this
from(g,'AUS')
```

This example creates a hash map of all the airports, using their IATA code as the key. We can then access the map using the IATA code to query information about those airports. Remember that the ';[]' at the end of the query just stops the console from displaying unwanted output.

```
// Create a map (a) of all vertices with the code property as the key
a=g.V().group().by('code').next();[]

// Show the description stored in the JFK vertex
a['JFK'][0].values('desc')
```

Another useful way to work with variables is to establish the variable and then use the 'fill' step to place the results of a query into it. The example below creates an empty list called 'german'. The query then finds all the vertices for airports located in Germany and uses the 'fill' step to place them into the variable.

```
german = []
g.V().has('airport','country','DE').fill(german)
```

We can then use our list as you would expect. Remember that as we are running inside the Gremlin Console, we do not have to explicitly iterate through the list as you would if you were writing a standalone Groovy application.

```
// How many results did we get back?
german.size

34

// Query some values from one of the airports in the list
german[0].values('city','code')

FRA
Frankfurt

// Feed an entry from our list back into a traversal
g.V(german[1]).values('city')

Munich

g.V(german[1]).out().count()

270
```

Towards the end of the book, in the "Working with TinkerGraph from a Groovy application" section, we will explore writing some standalone Groovy code that can use the TinkerPop API and issue Gremlin queries while running outside the Gremlin Console as a standalone application.

4.5.1. Using a variable to feed a traversal

Sometimes it is very useful to store the result of a query in a variable and then, later on, use that variable to start a new traversal. You may have noticed we did that in the very last example of the prior section where we fed the 'german' variable back in to a traversal. By way of another simple example, the code below stores the result of the first query in the variable 'austin' and then uses it to look for routes from Austin in the second query. Notice how we do this by passing the variable containing the Austin vertex into the 'V' step.

```
austin=g.V().has('code','AUS').next()
g.V(austin).out()
```

You can take this technique one step further and pass an entire saved list of vertices to 'V'. In the next example we first generate a list of all airports that are in Scotland and then pass that entire list into 'V' to first all count how many routes there are from those airports, and then we start another

query that looks for any route from those airports to airports in Germany.

```
// Find all airports in Scotland
a=g.V().hasLabel('airport').has('region','GB-SCT').toList()

// How many routes from these airports?
g.V(a).out().count()

// How many of those routes end up in Germany?
g.V(a).out().has('country','DE').values('code')
```

In this example of using with variables to drive traversals, we again create a list of airports. This time we find all the airports in Texas. We then use a Groovy 'each' loop to iterate through the list. For each airport in the list we print the code of the starting airport and then the codes of every airport that you can fly to from there.

```
// Find all of the airports in Texas
texas=g.V().has('region','US-TX').toList()

// For each airport, print a list of all the airports that you can fly to from there.
texas.each {println it.values('code').next() + "===>" +
                    g.V(it).out().values('code').toList()}
```

This example, which is admittedly a bit contrived, we use a variable inside a 'has' step. We initially create a list containing all the IATA codes for each airport in the graph. We then iterate through that list and calculate how many outgoing routes there are from each place and print out a string containing the airport IATA code and the count for that airport. Note that this could easily be done just using a Gremlin query with no additional Groovy code. The point of this example is more to show another example of mixing Gremlin, Groovy, and variables. Knowing that you can do this kind of thing may come in useful as you start to write more complicated graph database applications that use Gremlin. You will see this type of query done using just Gremlin in the section called "Finding unwanted parallel edges" later in this book.

```
m=g.V().hasLabel('airport').values('code').toList()
for (a in m) println a + " : " + g.V().has('code',a).out().count().next()
```

Lastly, here is an example that uses an array of values to seed a query.

```
['AUS','RDU','MCO','LHR','DFW'].
     each {println g.V().has('code','JFK').outE().inV().
                         has('code',it).path().by('code').by('dist').next()}
```

Here is the output from running the code.

```
[JFK, 1520, AUS]
```

```
[JFK, 427, RDU]
[JFK, 945, MCO]
[JFK, 3440, LHR]
[JFK, 1390, DFW]
```

4.6. Adding vertices, edges, and properties

So far in this book we have largely focussed on loading a graph from a file and running queries against it. As you start to build your own graphs, you will not always start with a graph saved as a text file in GraphML, CSV, GraphSON, or some other format. You may start with an empty graph and incrementally add vertices and edges. Just as likely you may start with a graph like the air-routes graph, read from a file, but want to add vertices, edges, and properties to it over time. In this section we will explore various ways of doing just that.

4.6.1. Adding an airport (vertex) and a route (edge)

The following code creates a new airport vertex and then adds a route as an edge from it to the existing DFW vertex. We can specify the label name ('airport') and as many properties as we wish to while creating the vertex. In this case we just provide three. We can additionally add and delete vertex properties after a vertex has been created. Examples of how to do that are coming up next.

```
// Add an imaginary airport with a code of 'XYZ' and connect it to DFW
xyz = g.addV('airport').property('code','XYZ').
                        property('icao','KXYZ').
                        property('desc','This is not a real airport').next()
```

Notice, in the code above where we add the vertex, how each 'property' step can be chained to the previous one when adding multiple properties. Whether you need to do it while creating a vertex or to add and edit properties on a vertex at a later date, you can use the same 'property' step.

The 'property' step also has a second form that takes a 'Map' of properties as an argument. This form can be useful as shorthand compared to chained 'property' calls, but also because it is common to have data in the form of a 'Map' in the normal course of development.

```
// Add an imaginary airport with a code of 'XYZ' and connect it to DFW
xyz = g.addV('airport').
        property([code: 'XYZ', icao: 'KXYZ', desc: 'This is not a real airport']).
next()
```

We can now add a route from DFW to XYZ. We are able to use our 'xyz' variable to specify the destination of the new route using 'to' step.

```
// Add a route from DFW to XYZ
g.V().has('code','DFW').addE('route').to(xyz)
```

We could have written the previous line to use a second 'V' step if we had not previously saved anything in a variable. Note that while this use of a second 'V' step will work locally, if you are sending queries to a Gremlin Server, a topic we will discuss later in this book, this syntax is not supported and will not work.

```
g.V().has('code','DFW').addE('route').to(V().has('code','XYZ'))
```

We might also want to add a returning route from XYZ back to DFW. We can do this using the 'from' step in a similar way as we used the 'to' step above.

```
// Add the return route back to DFW
g.V().has('code','DFW').addE('route').from(xyz)
```

Another way that we could have chosen to create our edge involves labeling the "XYZ" vertex using an `as` step. The example below demonstrates this. Notice also how a 'V' step is used to start a new traversal midway through the current one. The label created using the `as` step is used to instruct the `to` step about the target vertex for the new edge.

```
g.V().has('code','XYZ').as('a').V().has('code','DFW').addE('route').to('a')
```

You will see a bigger example that uses 'as' steps while creating vertices and edges in the "Quickly building a graph for testing" section that is coming up soon.

4.6.2. Using a traversal to determine a new label name

When using the 'addV' and 'addE' steps, we can use a traversal to determine what the label used by a new vertex or edge should be. Take a look at the query below. We have seen this type of query used earlier in the book. It simply tells us what label the vertex representing the Austin (AUS) airport has.

```
g.V().has('code','AUS').label()

airport
```

The traversal above can be used inside an 'addV' step as shown below. The first string result returned by the provided traversal will be used as the label name.

```
g.addV(V().has('code','AUS').label()).property('code','XYZ')

v[61394]
```

We can inspect the new vertex using 'valueMap' to make sure that our label was correctly assigned.

```
g.V(61394).valueMap(true)

[id:61394,code:[XYZ],label:airport]
```

We can now do something similar to dynamically work out what the label should be for an edge between our new airport and Austin.

```
g.V(61394).addE(V().has('code','AUS').outE().limit(1).label()).
           to(V().has('code','AUS'))

e[61396][61394-route->3]
```

Later in the book we will build upon these concepts to show how the property keys and values from one vertex, as well as the label, can be copied into a new vertex using a single query.

Once again, we can use a 'valueMap' step to make sure our new edge label looks OK.

```
g.E(61396).valueMap(true)

[id:61396,label:route]
```

4.6.3. Using a traversal to seed a property with a list

You can use the results of a traversal to create or update properties. The example below creates a new property called 'places' for the Austin airport vertex. The values of the property are the results of finding all the places that you can travel to from that airport and folding their 'code' values into a list.

```
// Add a list as a property value
g.V().has('code','AUS').property('places',out().values('code').fold())
```

We can use a 'valueMap' step to make sure the property was created as we expected it to be. As you can see, a new property called 'places' has been created containing as its value a list of codes.

```
g.V().has('code','AUS').valueMap('places')

[places:[[PHL,PDX,DTW,OKC,ONT,CLT,CUN,MEM,CVG,IND,MCI,DAL,STL,ABQ,MKE,MDW,OMA,TUL,PVR,
NAS,LIT,JAX,PVD,SMF,BHM,SDF,BUF,BOI,LBB,ECP,HRL,RNO,CMH,DSM,CZM,AMA,BTR,CHS,GDL,GRR,LI
R,PNS,SJD,TYS,VPS,XNA,HDN,SFB,BUR,ASE,BZN,BKG,PIE,ATL,BNA,BOS,BWI,DCA,DFW,FLL,IAD,IAH,
JFK,LAX,MCO,MIA,MSP,ORD,PHX,RDU,SEA,SFO,SJC,TPA,SAN,LGB,SNA,SLC,LAS,DEN,SAT,YYZ,HNL,YV
R,MSY,LHR,EWR,LGW,HOU,FRA,ELP,AMS,CLE,YYC,OAK,MEX,TUS,PIT]]]
```

To gain access to these values from your code or Gremlin console queries, we can use the 'next'

step. A simple example is given below where 'values' is used to retrieve the values of the 'places' property, and then we use 'size' to see how many entries there are in the list.

```
g.V().has('code','AUS').values('places').next().size()

98
```

Once we have access to the list of values, we can access them using the normal Groovy array syntax. The example below returns the three values with an index between 2 and 4.

```
g.V().has('code','AUS').values('places').next()[2..4]

DTW
OKC
ONT
```

4.6.4. Using 'inject' to specify new vertex ID values

If the graph database you are using supports user-provided ID values, you can use an 'inject' step as one way to specify what you want the ID value of a new vertex to be. For example, consider the example below.

```
g.inject(99999L).addV().property(id,identity())

v[99999]
```

You can also specify more than one ID value if you want to create multiple vertices.

```
g.inject(99997L,99998L).addV().property(id,identity())

v[99997]
v[99998]
```

We chose to show the use of 'inject' as it provides an interesting example. However, it is not required to create new IDs in this way. Both of the examples below are also valid ways to do the same thing. The first example just uses a literal value.

```
g.addV().property(id,99999L)

v[99999]
```

Alternatively, we could pass in a variable.

```
n=99999L;
```

```
g.addV().property(id,n)

v[99999]
```

Remember that these methods of specifying an ID value will only work if the graph database that you are using allows you to specify your own ID values. This varies by graph database implementation, and you should check the documentation for the system you are using before assuming that you can create your own custom ID values.

Even if the graph database that you are using does support user-provided ID values, you should check to see what data types can be used for them. All the examples above used LONG values. However, as one example, some graph databases that do allow you to specify custom IDs only support String values. So the key thing is to check the documentation before you start building your graph.

Even if a graph database does support custom ID values, if you try to create a vertex using an ID that already exists, the operation will fail. The example below shows what happens when we try to add a vertex using an ID that already exists to a TinkerGraph.

```
g.inject(99999L).addV().property(id,identity())

Vertex with id already exists: 99999
```

4.6.5. Quickly building a graph for testing

Sometimes for testing and for when you want to report a problem or ask for help on a mailing list, it is handy to have a small standalone graph that you can use. The code below will create a mini version of the air-routes graph in the Gremlin Console. Note how all the vertices and edges are created in a single query with each step joined together.

```
graph = TinkerGraph.open()
g = traversal().with(graph)
g.addV('airport').property('code','AUS').as('aus').
  addV('airport').property('code','DFW').as('dfw').
  addV('airport').property('code','LAX').as('lax').
  addV('airport').property('code','JFK').as('jfk').
  addV('airport').property('code','ATL').as('atl').
  addE('route').from('aus').to('dfw').
  addE('route').from('aus').to('atl').
  addE('route').from('atl').to('dfw').
  addE('route').from('atl').to('jfk').
  addE('route').from('dfw').to('jfk').
  addE('route').from('dfw').to('lax').
  addE('route').from('lax').to('jfk').
  addE('route').from('lax').to('aus').
```

```
  addE('route').from('lax').to('dfw')
```

The form of 'addV' that used to allow creation of a vertex and a property using something like, 'g.addV(label,"airport","code","AUS")', is now deprecated and should not be used.

4.6.6. Adding vertices and edges using a loop

Sometimes it is more efficient to define the details of the vertices or edges that you plan to add to the graph in an array and then add each vertex or edge using a simple 'for' loop that iterates over it. The following example adds our imaginary airports directly to the graph using such a loop. Notice that we do not have to specify the ID that we want each vertex to have. The graph will assign a unique ID to each new vertex for us.

```
vertices = [["WYZ","KWYZ"],["XYZ","KXYZ"]]
for (a in vertices) {g.addV("airport").property("code",a[0],"iata",a[1]).iterate()}
```

Note the call to 'iterate' at the end, for without that terminal step, the query would not execute. If you don't recall why 'iterate' is important, you can read about it in the "Ignoring query results with 'iterate'" section.

This technique of creating vertices and/or edges using a 'for' loop can also be useful when working with graphs remotely over HTTP connections. It is a very convenient way to combine a set of creation steps into a single REST API call.

4.6.7. Adding vertices and edges using 'inject'

The previous section on "Adding vertices and edges using a loop" used the loop semantics of programming languages to add multiple vertices by iterating over an array of data. In this section we will learn how you can do this entirely with Gremlin steps. As before, we construct an array representing our graph data and give that to 'inject'. The "looping" we saw previously is similarly produced here by the standard Gremlin semantics, which takes the array and uses 'unfold' to flatten the data such that each internal array triggers a call to the 'addV' step.

```
vertices = [["WYZ","KWYZ"],["XYZ","KXYZ"]]
g.inject(vertices).unfold().as("v").
  addV("airport").property("code", select("v").limit(local,1)).
                  property("iata", select("v").tail(local,1))
```

Each internal array is then given a label of "v" so that it can be referenced later in the traversal. We reference it in the calls to 'property' where we 'select' the array and grab either the first item in the array (i.e., the "code") or the last (i.e.m the "iata") to serve as the property value.

Unlike the approach that uses a more standard Java loop, this approach executes the graph mutation in a single query, where as the former approach will execute one query per item in the "vertices" array.

4.6.8. Adding vertices using 'repeat'

The previous section on "Adding vertices and edges using 'inject'" demonstrated how to use add multiple vertices given an array of data using only Gremlin steps. Another approach worth knowing about is the use of 'repeat' step to achieve a similar end:

```
g.inject(0).repeat(addV('test').property(T.id, loops())).times(5)
```

Note the call to 'loops' when setting the identifier for the vertex. The 'loops' step tracks the number of times that the 'repeat' has been called, essentially incrementing by 1 for each call to 'addV' creating an auto-incrementing identifier. This technique for creating vertices is often helpful when there is a need to generate some vertices for testing.

4.6.9. Using 'mergeV' and 'mergeE' for upserting

Checking to see if a vertex or edge already exists and then either inserting that element if it does not or updating the element that is found is a common database access pattern often referred to as an 'upsert'. There are a variety of ways to implement an upsert in Gremlin, but the 'mergeV' and 'mergeE' steps are designed specifically for this purpose and offer a wide degree of flexibility in this task.

Let's assume we wanted to add a new airport, with the code "'XYZ,"' but we are not sure if the airport might have already been added. We can do all of those things using the most basic form of 'mergeV':

```
g.mergeV([code:'XYZ'])

v[61394]
```

As you can see, 'mergeV' takes a 'Map' as an argument where the keys and values become the search criteria for the vertex. In the above case, it is similar to writing `has("code","XYZ")' to find a vertex. If 'mergeV' finds a vertex with that code and value, it returns it. However, 'mergeV' has a dual purpose. Should the vertex not be found, then 'mergeV' will automatically create a new vertex with a "code" of "XYZ" and return that. Evaluating that same query again will result in returning the existing 'v[61394]'.

You may match on as many properties as necessary, including 'T.id' and 'T.label':

```
g.mergeV([(T.label): 'airport', (T.id): 999999, code:'XYZ'])
```

Note that the match must be complete in that all keys designated in the 'Map' must match or else a new vertex will be created. In the example above, there is a vertex present that has a code of "'XYZ"', but the one we created initially lacks a 'T.label' of "airport" and has a different 'T.id' all together.

We do not always want to utilize the same criteria for searching as we do creating. Most commonly,

we tend to search by the element identifier, which is the fastest way to check for existence. For these cases you will only want to provide that identifier to the 'Map' given to 'mergeV' and separately specify the properties you want to use to create the vertex if it is not found. You can do this with the 'option' modulator and the 'Merge' enum.

```
g.mergeV([(T.id): 999999]).
    option(Merge.onCreate, [(T.label): 'airport', code:'XYZ'])
```

In this prior example, 'mergeV' will match on the vertex identifier of '999999' and will return it if found. Otherwise, it will defer to the 'onCreate' action which specifies a 'Map' of keys and values to use to create the new vertex. The 'onCreate' 'Map' inherits keys and values from the search map and would therefore include the '999999' in the creation of the vertex.

As we have a 'onCreate' to deal with the "does not exist" path for the search, we also have a 'onMatch' to allow more control over the "does exist" path to update an existing vertex. For example, if we wanted to change the code to update the code property to a lower-case '"xyz"' in the event the vertex exists, we could write the following:

```
g.mergeV([(T.id): 999999]).
    option(Merge.onCreate, [(T.label): 'airport', code:'XYZ']).
    option(Merge.onMatch, [code:'xyz'])
```

At this point we've demonstrated the general form of 'mergeV' where you give it a 'Map' of key/value pairs to use for a search and then use 'option' modulators to provide specific data to use to either create or update based on the initial match. For more advanced use cases, we have some additional flexibility in how we supply the 'Map' arguments as they may all be supplied by way of a 'Traversal':

```
g.withSideEffect('search', [(T.id): 999998]).
  withSideEffect('create', [code: 'ZYX', (T.label): 'airport']).
  withSideEffect('match', [code: 'zyx']).
  mergeV(select('search')).
    option(Merge.onCreate, select('create')).
    option(Merge.onMatch, select('match'))
```

The prior example shows how you can use a step like 'select' to dynamically provide a 'Map' argument to the various 'mergeV' parameters.

The counterpart to 'mergeV' is 'mergeE' where the same patterns can be used to upsert edges. Edges have a bit more complexity in that addition to the property values they may have, they also have an 'IN' and an 'OUT' (or 'from' and 'to') vertex that applies to them. Let's assume we want to look for an existing route edge between the '"XYZ"' airport that we just added and '"DFW"'. If we find that edge with that search criteria, we simply return it, but if it is not found, then we will create it with a dist property of 0. To do this, we need the vertex ids that this edge relates to. We can then use those ids to reference them in the search criteria:

```
g.V().has('code','DFW').id()

8

g.V().has('code','xyz').id()

999999

g.mergeE([(T.label): 'route', (Direction.from): 8, (Direction.to): 999999]).
    option(Merge.onCreate, [dist: 0])

e[61394][8-route->999999]

g.E(61394L).elementMap()

[id:61394,label:route,IN:[id:999999,label:airport],OUT:[id:8,label:airport],dist:0]
```

It is not possible to create new vertices from 'mergeE' automatically. They must already exist for the edge to be created. In addition to 'onMatch' and 'onCreate', 'mergeE' also allows for two other 'option' modulators: 'outV' and 'inV', which lets you specify search criteria as a 'Map' for the 'from' and the 'to' of the edge. Using that capability, we could rewrite the previous example as follows:

```
g.mergeE([(T.label): 'route', (Direction.from): Merge.outV, (Direction.to): Merge.
inV]).
    option(Merge.outV, [code: 'DFW']).
    option(Merge.inV, [code: 'xyz']).
    option(Merge.onCreate, [dist: 0])

e[61394][8-route->999999]
```

This section introduced the basics of the 'mergeV' and 'mergeE' steps, which provide a powerful way in which to encapsulate upsert logics for vertices and edges. As you continue through the following sections, you will find more advanced features with these steps.

4.6.10. Using 'inject' with 'mergeV'

In the "Adding vertices and edges using 'inject'" section, we saw how data could be injected to a traversal stream to create multiple vertices at a time with 'addV'. A similar tactic could be used with 'addE' to create multiple edges at a time. The pattern with 'inject' also extends to 'mergeV' and 'mergeE'. In the most simple form, 'mergeV' will simply accept an incoming 'Map' traverser as its argument.

```
m = [[code: 'IAD',country: 'US',desc: 'Washington Dulles International Airport', (T
.label): 'airport'],
     [code: 'LTA',country: 'US',desc: 'Little Tiny Airport', (T.label): 'airport'],
     [code: 'DCA',country: 'US',desc: 'Ronald Reagan Washington National Airport', (T
.label): 'airport']]
```

```
g.inject(m).unfold().mergeV().elementMap('code','country','desc')

[id:10,label:airport,country:US,code:IAD,desc:Washington Dulles International Airport]
[id:61395,label:airport,country:US,code:LTA,desc:Little Tiny Airport]
[id:7,label:airport,country:US,code:DCA,desc:Ronald Reagan Washington National
Airport]
```

The prior example flattens the list of maps in "m", feeding them one at a time to 'mergeV' where it does a match on each of the keys returning those vertices it finds and creating those it doesn't. You could take a more advanced approach and perform different operations for the 'onCreate" and 'onMatch' options driven fully on the data supplied to 'inject'. In the next example, we form a list of maps, where each key refers to the search, create, or match 'Map' options that 'mergeV' expects. We can then use 'select' to grab the appropriate one and supply it as an argument to the step.

```
m = [[search: [code: 'IAD', (T.label): 'airport'], create: [desc: 'Washington Dulles
International Airport', country: 'US'], match: [elev: 313]],
     [search: [code: 'LTA',(T.label): 'airport'], create: [country: 'US',desc: 'Little
Tiny Airport'], match: [elev: 555]],
     [search: [code: 'DCA', (T.label): 'airport'], create: [country: 'US',desc:
'Ronald Reagan Washington National Airport'], match: [elev: 14]]]

g.inject(m).unfold().as('m').
  mergeV(select('m').select('search')).
    option(onCreate, select('m').select('create')).
    option(onMatch, select('m').select('match')).
  elementMap('code','country','desc','elev')

[id:10,label:airport,country:US,code:IAD,elev:313,desc:Washington Dulles International
Airport]
[id:61394,label:airport,country:US,code:LTA,desc:Little Tiny Airport]
[id:7,label:airport,country:US,code:DCA,elev:14,desc:Ronald Reagan Washington National
Airport]
```

4.6.11. Incorporating 'fail' with 'mergeV' or 'mergeE'

We learned about 'fail' step in "Combining 'coalesce' with 'fail' step" where it will throw an exception when it is encountered. You can incorporate this step into 'mergeV' and 'mergeE' to stop a traversal in cases where you don't want the traversal to continue in 'onCreate' or 'onMatch'.

Let's envision a case where you do not expect the '"XYZ"' airport to be present in the graph and that if it is, you do not want the query to proceed any further in its processing. Since the 'option' modulator takes a traversal as an argument, you can give it a fail step for its 'onMatch'.

```
g.mergeV([code: 'xyz']).
    option(Merge.onCreate, [(T.label): 'airport']).
    option(Merge.onMatch, fail('xyz airport already exists'))
```

```
fail() Step Triggered
================================================================================
================================================================================
Message  > xyz airport already exists
Traverser> v[999999]
  Bulk   > 1
Traversal> fail("xyz airport already exists")
Parent   > MergeVertexStep [mergeV(["code":"xyz"]).option(Merge.onCreate,[(T.label
):"airport","code":"xyz"]).option(Merge.onMatch,__.fail("xyz airport already
exists"))]
Metadata > {}
================================================================================
================================================================================
```

Note that the output above is the string representation of a 'FailException' as printed in Gremlin Console.

4.6.12. Specifying cardinality with 'mergeV'

Most of the time, vertex properties tend to be modeled with 'single' cardinality. However, you may have situations where other cardinalities are used or are using a graph that defaults to a cardinality other than 'single'. The 'mergeV' step provides two ways to explicitly set the cardinality for the properties given to it, and both make use of the 'Cardinality' enum. The first way to do this is to explicitly set the cardinality per property value. For example, let's imagine that you want to use 'set' cardinality for the code property.

```
g.mergeV([(T.id): 999999]).
    option(Merge.onCreate, [(T.label): 'airport', code: set('XYZ')])
```

By wrapping the '"XYZ"' in the 'set', which is statically imported from 'Cardinality.set', you mark that value in a way that 'mergeV' knows to tell the graph to use that specific cardinality for that property. You could also do set this value globally for the step by providing it as an argument after the 'Map':

```
g.mergeV([(T.id): 999999]).
    option(Merge.onCreate, [(T.label): 'airport', code: 'XYZ'], VertexProperty
.Cardinality.set)
```

When you offer 'set' this way, 'mergeV' will assume all property keys to use this cardinality. If there is an explicit cardinality specified, it will override the global setting.

4.6.13. When you need more flexibility than 'mergeV' and 'mergeE'

The 'mergeV' and 'mergeE' steps offer a wide breath of options to encapsulate upsert-like logic with varying mechanisms for matching, creating, and updating. While these steps are extremely flexible, you may yet find a scenario where they do not do everything you require. In these cases, you may fall back to a lower order of Gremlin steps that can offer more options, but be a bit more complex

to implement and possibly lack the same performance capability as different graph providers may not be able to optimize these queries as well as the more directly purposed 'mergeV' and 'mergeE' steps.

In the "Combining 'coalesce' with a 'constant' value" section we looked at how coalesce could be used to return a constant value if the other entities that we were looking for did not exist. We can reuse that pattern to produce a traversal that will only add a vertex to the graph if that vertex has not already been created.

Let's assume we wanted to add a new airport, with the code '"XYZ,"' but we are not sure if the airport might have already been added.

We can check to see if the airport exists, using a basic 'has' step.

```
g.V().has('code','XYZ')
```

If it does not exist yet, which in this case it does not, nothing will be returned. We could go one step further and change the query to return an empty list '[]' if the airport does not exist by adding a 'fold' step to the query.

```
g.V().has('code','XYZ').fold()

[]
```

Now that we have a query that can return an empty list if a vertex does not exist, we can take advantage of this in a 'coalesce' step. The query below looks to see if the airport already exists and passes the result of that into a 'coalesce' step. Remember, 'coalesce' will return the result of the first traversal it looks at that returns a good result. We can make the first parameter passed to 'coalesce' and 'unfold' step. This way in the case where the airport does not exist, 'unfold' will return nothing and so 'coalesce' will attempt the second step. In this case our second step creates a vertex for the airport '"XYZ"'.

```
g.V().has('code','XYZ').fold().coalesce(unfold(),addV().property('code','XYZ'))

v[61394]
```

As you can see, the query above created a new vertex with an ID of '61394' as the 'XYZ' airport did not already exist. However, if we run the same query again, notice that we get the same vertex back that we just created and not a new one. This is because this time, the 'coalesce' step **does** find a result from the 'unfold' step and so completed before attempting the 'addV' step.

```
g.V().has('code','XYZ').fold().coalesce(unfold(),addV().property('code','XYZ'))

v[61394]
```

Using 'coalesce' in this way provides us with a nice pattern for a commonly performed task of checking to see if something already exists before we try to update it and otherwise create it. This is often called an '"upsert"' pattern as the operation potentially updates or inserts a vertex based on its existence or not.

The query below is perhaps a better example of an '"upsert"'. The query looks to see if the vertex with an ID of 3 already exists. If it does, it updates the 'runways' property of that vertex be the value 3. If it does not exist, it creates a new vertex and the property. As vertex v[3] already exists, that is what the query returns, having first updated the 'runways' property.

```
g.V(3).fold().
       coalesce(unfold().property('runways',3),
       addV('airport').property('runways',3))

v[3]
```

If we now examine the properties of the vertex 'v[3]', we can see that the 'runways' value has been set to 3.

```
g.V(3).valueMap().unfold()

country=[US]
code=[AUS]
longest=[12250]
city=[Austin]
elev=[542]
icao=[KAUS]
lon=[-97.6698989868164]
type=[airport]
region=[US-TX]
runways=[3]
lat=[30.1944999694824]
desc=[Austin Bergstrom International Airport]
```

In the air-routes graph there is no vertex with an ID of 9999999. So if we rerun the previous '"upsert"' query, this time a new vertex will be created.

```
g.V(9999999).fold().
             coalesce(unfold().property('runways','3'),
             addV('airport').property('runways',3))

v[61397]
```

If we look at the 'valueMap' for the new vertex, we can see that it was created as we would have expected.

```
g.V(61397).valueMap().unfold()

runways=[3]
```

At the start of this section, we mentioned that this pattern with the 'coalesce' step was useful when you needed more flexibility that what 'mergeV' and 'mergeE' provided. The power in this pattern lies in the fact that the logic executed by `coalesce` is only limited by what you can do in Gremlin. For example, the following example shows how you might try to find a vertex, but if not found, you could not only add that missing vertex, but another structure as well, in this case another vertex and an edge between them.

```
g.V(9999999).fold().
  coalesce(unfold(), addV().property(id, 9999999).as('a').
                     addV().property(id, 9999998).as('b').
                     addE('knows').from('a'))
```

This is a powerful aspect of the pattern but can complicate performance for different graph databases. Always be sure to understand the capabilities of your graph so that you understand the implications of the Gremlin that you are writing.

4.6.14. Creating one vertex based on another

It is sometimes useful to be able to create a new vertex using the label and properties from an existing vertex. We have already looked, in the "Using a traversal to determine a new label name" section, at some ways to create a new label using the value of other labels, but we have not yet looked at how to clone the properties from one vertex onto another. A technique for doing that is discussed in the "Making a copy of the DFW vertex" section. Feel free to skip ahead but be aware that the techniques used in that section have not yet been fully covered, so you may want to also take a look at some other sections along the way.

4.7. Deleting vertices, edges, and properties

So far in this book we have looked at several examples where we created new vertices, edges, and properties but we have not yet looked at how we can delete them. Gremlin provides the 'drop' step that we can use to remove things from a graph.

4.7.1. Deleting a vertex

In some of our earlier examples we created a fictitious airport vertex with a code of 'XYZ' and added it to the air-routes graph. If we now wanted to delete it, we could use the following Gremlin code. Note that removing the vertex will also remove any edges we created connected to that vertex.

```
// Remove the XYZ vertex
g.V().has('code','XYZ').drop()
```

4.7.2. Deleting an edge

We can also use 'drop' to remove specific edges. The following code will remove the flights, in both directions between AUS and LHR.

```
// Remove the flight from AUS to LHR (both directions).
g.V().has('code','AUS').outE().as('e').inV().has('code','LHR').select('e').drop()
g.V().has('code','LHR').outE().as('e').inV().has('code','AUS').select('e').drop()
```

4.7.3. Deleting a property

Lastly, we can use 'drop' to delete a specific property value from a specific vertex. Let's start by querying the properties defined by the 'air-routes' graph for the San Francisco airport.

```
g.V().has('code','SFO').valueMap()

[country:[US],code:[SFO],longest:[11870],city:[San Francisco],elev:[13],icao:[KSFO
],lon:[-122.375],type:[airport],region:[US-CA],runways:[4],lat:[37.6189994812012
],desc:[San Francisco International Airport]]
```

Let's now drop the 'desc' property and re-query the property values to prove that it has been deleted.

```
g.V().has('code','SFO').properties('desc').drop()

g.V().has('code','SFO').valueMap()

[country:[US],code:[SFO],longest:[11870],city:[San Francisco],elev:[13],icao:[KSFO
],lon:[-122.375],type:[airport],region:[US-CA],runways:[4],lat:[37.6189994812012]]
```

If we wanted to delete all the properties currently associated with the SFO airport vertex, we could do that as follows.

```
g.V().has('code','SFO').properties().drop()
```

Another common pattern we might consider is to remove a property while simultaneously adding a new one. Given what we've learned of Gremlin so far, we might be tempted to try to follow 'drop' with another call to 'property' as follows.

```
g.V().has('code','SFO').properties('desc').drop().property('newProperty','test')

g.V().has('code','SFO').valueMap()

[country:[US],code:[SFO],longest:[11870],city:[San Francisco],lon:[-122.375],type:
```

```
[airport],elev:[13],icao:[KSFO],region:[US-CA],runways:[4],lat:[37.6189994812012]]
```

We can see that we did indeed drop the 'desc' property, but the 'newProperty' key we'd intended to insert is not present. This example presents an important aspect of 'drop'. The 'drop' step deletes the objects that flow into it and, as that deleted object no longer exists, it also removes the object from the traversal stream. As a result, steps following the 'drop' will not have an object to trigger their execution. In our example above, 'property' never executes.

Since 'drop' does not allow steps placed after it to execute, it is often found at the end of a traversal. It is worth noting however that its typical position in a traversal does not make it a "Assigning query results to a variable with a terminal step", i.e., it does not iterate the traversal. Since 'drop' returns no results, the usual terminating step to use in conjunction with it is 'iterate'.

Despite the 'drop' filter behavior, we'd still like to be able to remove a property and then add a new one in the same traversal. In the "Using 'sideEffect' to do things on the side" section, we introduced how 'sideEffect' can be used to include additional processing without changing what gets passed on to the next stage of the query. We can wrap 'drop' in this step to handle the property removal as a side effect of the traversal execution.

```
g.V().has('code','SFO').sideEffect(properties('desc').drop()).
                        property('newProperty','test')

g.V().has('code','SFO').valueMap()

[country:[US],code:[SFO],longest:[11870],newProperty:[test],city:[San Francisco],
lon:[-122.375],type:[airport],elev:[13],icao:[KSFO],region:[US-CA],runways:[4],lat:
[37.6189994812012]]
```

4.7.4. Removing all the edges or vertices in the graph

This may not be something you want to do very often, but should you wish to remove every edge in the graph, you could do it using the traversal object, 'g', as follows. Note that for very large graphs this may not be the most efficient way of doing it depending upon how the graph store handles this request.

```
// Remove all the edges from the graph
g.E().drop()
```

You could also delete the whole graph, vertices, and edges by deleting all vertices!

```
// Delete the entire graph!
g.V().drop()
```

4.8. Property keys and values revisited

We have already looked, earlier in the book, at numerous queries that retrieve, create, or manipulate in some way the value of a given property. There are still, however, a few things that we have not covered in any detail concerning properties. Most of the property values we have looked at so far have been simple types such as a String or an Integer. In this section we shall look more closely at properties and explain how they can in fact be used to store lists and sets of values. We will also introduce in this section the concept of a property ID.

4.8.1. The 'Property' and 'VertexProperty' interfaces

In a TinkerPop enabled graph, all properties are implementations of the 'Property' interface. Vertex properties implement the 'VertexProperty' interface, which itself extends the 'Property' interface. These interfaces are documented as part of the Apache TinkerPop Javadoc. The interface defines the methods that you can use when working with a vertex property object in your code. One important thing to note about vertex properties is that they are immutable. You can create them, but once created, they cannot be updated.

We will look more closely at the Java interfaces that TinkerPop defines in the "Working with TinkerGraph from a Java Application" section a bit later in this book.

The VertexProperty interface does not define any "setter" methods beyond the basic constructor itself. Your immediate reaction to this is likely to be, "but I know you can change a property's value using the 'property' step". Indeed, we have already discussed doing just that in this book. However, behind the scenes, what actually happens when you change a property is that a new property object is created and used to replace the prior one. We will examine this more in a minute, but first let's revisit a few of the basic concepts of properties.

In a 'property graph' both vertices and edges can contain one or more properties. We have already seen a query like the one below that retrieves the values from each of the property keys associated with the DFW airport vertex.

```
g.V().has('airport','code','DFW').values()

US
DFW
13401
Dallas
607
KDFW
-97.0380020141602
airport
US-TX
7
32.896800994873
Dallas/Fort Worth International Airport
```

What we have not mentioned so far, however, is that the previous query is a shortened form of this

one.

```
g.V().has('airport','code','DFW').properties().value()

US
DFW
13401
Dallas
607
KDFW
-97.0380020141602
airport
US-TX
7
32.896800994873
Dallas/Fort Worth International Airport
```

If we wanted to retrieve the VertexProperty ('vp') objects for each of the properties associated with the DFW vertex, we could do that too. In a lot of cases it will be sufficient just to use 'values' or 'valueMap' to access the values of one or more properties, but there are some cases, as we shall see when we look at property IDs, where having access to the vertex property object itself is useful.

```
g.V().has('airport','code','DFW').properties()

vp[country->US]
vp[code->DFW]
vp[longest->13401]
vp[city->Dallas]
vp[elev->607]
vp[icao->KDFW]
vp[lon->-97.0380020141602]
vp[type->airport]
vp[region->US-TX]
vp[runways->7]
vp[lat->32.896800994873]
vp[desc->Dallas/Fort Worth In]
```

We have already seen how each property on a vertex or edge is represented as a key and value pair. If we wanted to retrieve a list of all the property keys associated with a given vertex, we could write a query like the one below that finds all the property keys associated with the DFW vertex in the 'air-routes' graph.

```
g.V().has('airport','code','DFW').properties().key()

country
code
longest
city
```

```
elev
icao
lon
type
region
runways
lat
desc
```

We could likewise find the names, with duplicates removed, of any property keys associated with any outgoing edges from the DFW vertex using this query. Note that edge properties are implementations of 'Property' and not 'VertexProperty'.

```
g.V().has('code','DFW').outE().properties().key().dedup()

dist
```

We can use the fact that we now know how to specifically reference both the key and value parts of any property to construct a query like the one below that adds up the total length of all the longest runway values and number of runways in the graph and groups them by property key first and sum of the values second.

```
g.V().hasLabel("airport").
      properties("runways","longest").
      group().by(key).by(value().sum())

[longest:26436124,runways:4980]
```

4.8.2. The 'element' traversal step

When you use the 'properties' step to navigate to a property object, you may find that you want to return to the parent element that it is related to. The 'element' step is used for this purpose and can be used to navigate from a property back to a vertex, edge, or vertex property. That final case of a "vertex property" refers to situations where you have used 'properties' to access a meta-property as described a bit later in the "Adding properties to other properties (meta-properties)" section.

```
g.V().has('airport','code','IAD').
  properties('code','city').element()

v[10]
v[10]
```

In the example shown above, we can see that "v[10]" is returned twice because 'element' was called twice, once for the "code" property and once for the "city" property, and both belong to the "v[10]" vertex.

4.8.3. The 'propertyMap' traversal step

We have previously used the 'valueMap' step to produce a map of key/value pairs for all the properties associated with a vertex or edge. There is also a 'propertyMap' step that can be used that yields a similar result, but the map includes the vertex property objects for each property.

```
g.V().has('code','AUS').propertyMap()
```

Here are the properties returned.

```
[country:[vp[country->US]],longest:[vp[longest->12250]],code:[vp[code->AUS]],city:[vp
[city->Austin]],lon:[vp[lon->-97.6698989868164]],type:[vp[type->airport]],elev:[vp
[elev->542]],icao:[vp[icao->KAUS]],region:[vp[region->US-TX]],runways:[vp[runways->
2]],lat:[vp[lat->30.1944999694824]],desc:[vp[desc->Austin Bergstrom ...]]]
```

4.8.4. Properties have IDs too

We have seen many examples already that show how both vertices and edges have a unique ID. What may not have been obvious, however, is that properties also have an ID. Unlike vertex and edge IDs, property IDs are not guaranteed to be unique across the graph. Certainly, with TinkerGraph we have encountered cases where a vertex and a property share the same ID. This is not really an issue because they are used in different ways to access their associated graph element.

The query below returns the vertex property object (vp) for any property in the graph that has a value of 'London'.

```
g.V().properties().hasValue('London')
```

The query finds several London values.

```
vp[city->London]
vp[city->London]
vp[city->London]
vp[city->London]
vp[city->London]
vp[city->London]
```

At first glance, each of the values returned above looks identical. However, let's now query their ID values.

```
g.V().properties().hasValue('London').id()
```

As you can see, each property has a different, and unique, ID.

```
584
596
1052
1124
2468
7784
```

We can use these ID values in other queries in the same way as we have for vertices and edges in some of our earlier examples.

```
g.V().properties().hasId(584)

vp[city->London]
```

We can query the value of this property as you would expect.

```
g.V().properties().hasId(584).value()

London
```

We can retrieve the name of the property key as follows.

```
g.V().properties().hasId(584).key()

city
```

We could also have used 'label' instead of 'key':

```
g.V().properties().hasId(584).label()

city
```

We can also find out which element (vertex or edge) that this property belongs to.

```
g.V().properties().hasId(584).element()

v[49]
```

We can also look at other property values of the element containing our property with an ID of 584.

```
g.V().properties().hasId(584).next().element().values('desc')
```

```
London Heathrow
```

Should you need to, you can also find out which graph this property is part of. In this case it is part of a TinkerGraph.

```
g.V().has('airport','code','DFW').properties('city').next().graph()

tinkergraph[vertices:3749 edges:57645]
```

To further show that each property has an ID, the following code retrieves a list of all the vertex properties associated with vertex 'V(3)' and prints out the property key along with its corresponding ID.

```
p = g.V(3).properties().toList()
p.each {println it.key + "\t:" + it.id}

country :29
longest :31
code    :30
city    :32
lon     :35
type    :36
elev    :33
icao    :34
region  :37
runways :38
lat     :39
desc    :40
```

If you update a property, its ID value will also be changed as you have in reality replaced the property with a new one which is allocated a new ID.

Take a look at the example below. First, we query the ID of the 'city' property from vertex 'V(4)'. Next we change its value to be 'newname' and then query the property ID again. Note that the ID has changed. As mentioned above, vertex properties are immutable. When you update a property value using the 'property' step, a new property object is created that replaces the prior one.

```
g.V(4).properties('city').id()

44

g.V(4).property('city','newname')

g.V(4).properties('city').id()

42784
```

The fact that every property in a graph has an ID can improve performance of accessing properties, especially in large graphs.

4.8.5. Attaching multiple values (lists or sets) to a single property

A vertex property value can be a basic type such as a String or an Integer, but it can also be more sophisticated such as a Set or a List containing multiple values. You can think of these values as being an array, but depending on how you create them, you have to work with them differently. In this section we will look at how we can create multiple values for a single property key. Such values can be set up when the vertex is first created or added afterward. These more complex types of property values are not supported on edges.

If we wanted to store the IATA and ICAO codes for the Austin airport in a list associated with a single property rather than as separate properties, we could have created them when we created the Austin vertex follows. You can also add properties to an existing vertex that have lists of values. We will look at how to do that later in this section.

```
g.addV().property('code','AUS').property('code','KAUS')
```

The version of 'addV' that allowed you to specify something like 'g.addV('code','AUS','code','KAUS')' is now deprecated and should not be used.

By creating the 'code' property in this way, its cardinality type is now effectively 'LIST' rather than 'SINGLE'. While working with TinkerGraph, we do not need to set up explicit schemas for our property types. However, more sophisticated graph systems, such as JanusGraph, will require you to make some decisions in this area. Those discussions are beyond the scope of this book, however.

Now that we have created the 'code' property to have a list of values, we can query either one of the values in the list. If we look at the value map we get back from the following example queries, you can see both values in the list we associated with the property 'code'.

```
g.V().has('code','AUS').valueMap()

[code:[AUS,KAUS]]

g.V().has('code','KAUS').valueMap()

[code:[AUS,KAUS]]
```

we can also query the values as normal.

```
g.V().has('code','AUS').values()

AUS
KAUS
```

We can also use the 'properties' step to get the result back as vertex properties (vp). We discuss vertex properties in detail in the "The 'Property' and 'VertexProperty' interfaces" section.

```
g.V().has('code','AUS').properties()

vp[code->AUS]
vp[code->KAUS]
```

For completeness, we could also do this.

```
g.V().properties().hasValue('AUS')

vp[code->AUS]
```

A word of caution – behavior differences with 'property'

Be aware!

There is a subtlety to be aware of when using 'property'. What happens can vary depending on the context in which it is used. Only when done as part of an 'addV' step immediately followed by multiple 'property' steps using the same key value will a list be created. Look at the two examples below. They do not produce the same results.

```
g.addV().property('one','hi').
         property('one','hello').
         property('two','goodbye').
         property('one','hello again').
         valueMap()

[one:[hi,hello,hello again],two:[goodbye]]
```

So our first query creates a property with a key called 'one' followed by a list containing '[hi,hello,hello again]'. Let's do the same test again, but this time create the vertex first and use an already created vertex to add properties to.

```
v = g.addV().next()

g.V(v).property('one','hi').
       property('one','hello').
       property('two','goodbye').
       property('one','hello again').
       valueMap()

[one:[hello again],two:[goodbye]]
```

This time, as we were not creating the vertex as part of the same set of steps, the behavior changes.

Each time the property key of 'one' is used, the existing value is replaced rather than being added as part of a list. We have seen this behavior cause confusion more than once, and it is something to be aware of! In the next section we will ask Gremlin to explain this behavior to us!

4.8.6. What did Gremlin do? - introducing 'explain'

If you ever want to know how Gremlin compiles your query into a form that it is able to execute, you can ask it to tell you by adding an 'explain' step to the end of your query. The query will not execute, instead you will be shown how Gremlin decided to optimize your query. It actually shows you all the choices it considered, but in the examples below we are just going to show the one it picked in each case.

So, thinking about our previous discussion of how 'property' works differently depending upon the context, if you were to use the 'explain' step to have Gremlin show us the way it is going to execute our query, you can clearly see the difference between the two forms. We have truncated the output to keep things simple.

Here is what Gremlin shows us for the first query when we use an 'explain' step. As you can see, our query has been compiled into an 'AddVertexStep' with two properties, one of which is a list.

```
g.addV().property('one','hi').
         property('one','hello').
         property('two','goodbye').
         property('one','hello again').
         explain()

Final Traversal    [AddVertexStartStep({one=[hi, hello, hello
                      again], two=[goodbye]})]
```

Now if we look at the case where we have already created a vertex, let's see what 'explain' returns. What we find is that this time Gremlin has compiled our query to a 'TinkerGraphStep' and is handling each property one by one. This has the result that each time the same key is reused, the previous value is replaced.

```
g.V(v).property('one','hi').
       property('one','hello').
       property('two','goodbye').
       property('one','hello again').
       explain()

Final Traversal  [TinkerGraphStep(vertex,[v[61394]]),
                    AddPropertyStep({value=[hi], key=[one]}),
                    AddPropertyStep({value=[hello], key=[one]}),
                    AddPropertyStep({value=[goodbye], key=[two]}
                    ), AddPropertyStep({value=[hello again], key=[one]})]
```

If you need to work with properties and treat them as lists once the vertex has been created, you need to explicitly add the 'list' keyword as part of the 'property' step, as we shall see in the next

section.

4.8.7. Updating properties stored in a list

So now we know how to create a vertex with property values in a list we need a way to update those properties. We can do this using a special form of the 'property' step where the first parameter is 'list' to show that what follows are updates to the existing list and not replacements for the whole list.

The example below adds another code sometimes used when talking about the Austin airport to our list of codes. If we left off the 'list' parameter, the whole property would be overwritten with a value of 'ABIA'.

```
g.V().has('code','AUS').property(list,'code','ABIA')
```

If we query the properties again, we can see that there are now three values for the 'code' property.

```
g.V().has('code','AUS').properties()
vp[code->AUS]
vp[code->KAUS]
vp[code->ABIA]
```

We can observe the same thing by looking at our 'valueMap' results again.

```
g.V().has('code','AUS').valueMap()
[code:[AUS,KAUS,ABIA]]
```

If we want to delete one of the properties from the list, we can do it using 'drop'. If we look at the value map after dropping 'ABIA', we can indeed see that it is gone from the list.

```
g.V().has('code','AUS').properties().hasValue('ABIA').drop()

g.V().has('code','AUS').valueMap()
[code:[AUS,KAUS]]
```

If we want to drop an entire property containing one or more values, we can do it as follows.

```
g.V().has('code','AUS').properties('code').drop()
```

To add multiple values to the same property key in the same query, we just need to chain the 'property' steps together as shown below.

```
g.V().has('code','AUS').
      property(list,'desc','Austin Airport').
```

```
      property(list,'desc','Bergstrom')
```

This technique can be used to update an existing property that already has a list of values or to add a new property with a list of values.

The same value can appear more than once with a property that has LIST cardinality. The code fragment below creates a new vertex, with some duplicate values associated with the 'dups' property.

```
g.addV('test').property('dups','one').property('dups','two').property('dups','one')

g.V().hasLabel('test').valueMap()

[dups:[one,two,one]]
```

We can add additional duplicate values after the vertex has been created.

```
g.V().hasLabel('test').property(list,'dups','two')

g.V().hasLabel('test').valueMap()

[dups:[one,two,one,two]]
```

4.8.8. Creating properties that store sets

So far we have just created values in a list that have 'LIST' cardinality which means that duplicate values are allowed. If we want to prevent that from happening, we can use the 'set' keyword when adding properties to force a cardinality of 'SET'.

In the example below we create a new property called 'hw' for vertex 'V(3)' with multiple values but using the 'set' keyword rather than the 'list' keyword that we have used previously. We then look at the valueMap for the 'hw' property to check that indeed our set was created.

```
g.V(3).property(set,'hw','hello').property(set,'hw','world')

g.V(3).valueMap('hw')

[hw:[hello,world]]
```

Let's now test that our set is really working as a set by adding a couple of additional values. Note that we have already added the value 'hello' in the prior steps, so with the cardinality being 'SET', we expect that value to be ignored as there is already a value of 'hello' in the set. We again display the valueMap to prove that the set only has unique values in it.

```
g.V(3).property(set,'hw','hello').property(set,'hw','apple')
```

```
g.V(3).valueMap('hw')

[hw:[hello,world,apple]]
```

4.8.9. One more note about sets and lists

Note that the other examples we have shown in this section are different from just adding a list directly as a property value. In the example below, the entire list is treated as a single value.

```
g.V().has('code','AUS').property('x',['AAAA','BBBB'])

g.V().has('code','AUS').valueMap('x')

[x:[[AAAA,BBBB]]]
```

4.8.10. Adding properties to other properties (meta-properties)

TinkerPop can add a property to another property. Think of this in a way as being able to add a bit of metadata about a property to the property itself. This capability, not surprisingly, is often referred to as '"adding a meta-property to a property"'. There are a number of use cases where this capability can be extremely useful. Common ones might be adding a date that a property was last updated or perhaps adding access control information to a property.

The example below adds a meta-property with a key of 'date' and a date value of '2025-06-21T00:00:00Z' to the property with a key of 'code' and a value of 'AUS'.

```
g.V().has('code','AUS').properties().hasValue('AUS').
  property('date', datetime('2025-06-21T00:00:00Z'))
```

If you wanted to add the date to the 'code' property regardless of its current value, then you could just do this.

```
g.V().has('code','AUS').properties('code').
  property('date', datetime('2025-06-21T00:00:00Z'))
```

We can retrieve all the meta-properties on a specific property, such as the 'code' property as follows.

```
g.V().has('code','AUS').properties('code').properties()

p[date->2025-06-21T00:00Z]
```

If you want to find all the properties associated with the AUS vertex that have a meta-property with a date of '2025-06-21T00:00:00Z', you can do that as follows.

```
g.V().has('code','AUS').properties().has('date',datetime('2025-06-21T00:00:00Z'))

vp[code->AUS]
```

We can query for a specific meta-property as follows, which will return any meta-properties that have a key of 'date'.

```
g.V().has('code','AUS').properties().hasValue('AUS').properties('date')

p[date->2023-06-21T00:00Z]
```

You can add multiple meta-properties to a property while creating it. The following will add a property called 'comment' to vertex 'V(3)' and also add two meta properties to it representing the date the comment was made and who made it.

```
g.V(3).property('comment','I like this airport','date',datetime('2025-06-
21T00:00:00Z'),'user','Kelvin')
```

We can query the graph to make sure everything worked as expected.

```
g.V(3).properties('comment')

vp[Comment->I like this airport]

g.V(3).properties('comment').properties()

==>p[date->2025-06-21T00:00Z]
==>p[user->Kelvin]
```

You can use 'drop' to remove meta-properties but take care when doing so. Take a look at the query below, which looks like it might drop the meta-property 'date', but will in fact drop the whole vertex.

```
g.V().has('code','AUS').
  properties().hasValue('AUS').property('date',datetime('2025-06-21T00:00:00Z')).
drop()
```

To remove a single meta-property, we need to use drop in this way

```
 g.V().has('code','AUS').properties('code').properties('date').drop()
```

Note that you cannot chain meta-properties together endlessly. The main properties on a vertex are 'VertexProperty' types. The meta-property is a 'Property' type, and you cannot add another property

to those. You can, however, add more than one meta-property to the same vertex property.

So as mentioned above, the meta-property provides a way to attach metadata to another property. This enables a number of important use cases, including being able to attach a date or ACL information to individual properties.

4.8.11. Using 'unfold' and 'WithOptions' with Meta-Properties

There is an easy way to include both properties and their meta-properties in the result from a 'valueMap' step that follows a 'properties' step. Using this capability requires that the graph database you are using has support for meta-properties. The examples below build upon the examples shown in the previous section.

All the possible values that can be specified using 'WithOptions' can be found in the official Apache TinkerPop Javadoc documentation https://tinkerpop.apache.org/javadocs/current/full/org/apache/tinkerpop/gremlin/process/traversal/step/util/WithOptions.html.

First, we know how to inspect the properties present for any vertex using the 'properties' step.

```
g.V().has('code','AUS').properties()

vp[country->US]
vp[code->AUS]
vp[longest->12250]
vp[city->Austin]
vp[elev->542]
vp[icao->KAUS]
vp[lon->-97.6698989868164]
vp[type->airport]
vp[region->US-TX]
vp[runways->2]
vp[lat->30.1944999694824]
vp[desc->Austin Bergstrom ...]
```

We also know how to look at the meta-properties by following the 'properties' step with a 'valueMap' step or a second 'properties' step. However, a value is returned only when a property has a meta-property. For all other properties the result is simply an empty list.

```
g.V().has('code','AUS').properties().valueMap()

[]
[date:2025-06-21T00:00:00Z]
[]
[]
[]
[]
[]
```

```
[]
[]
[]
[]
[]
```

Using the new 'with' step and specifying options using 'WithOptions', we can generate a result from 'valueMap' that includes the property values and their respective meta-properties. The example below generates a map containing the values for all properties plus the key and value for any meta-properties present.

```
g.V().has('code','AUS').
      properties().
      valueMap().with(WithOptions.tokens,WithOptions.values)
```

The values are shown for all properties, but for the case where we have a 'date' meta property, that is also shown.

```
[value:US]
[value:AUS,date:2025-06-21T00:00Z]
[value:12250]
[value:Austin]
[value:542]
[value:KAUS]
[value:-97.6698989868164]
[value:airport]
[value:US-TX]
[value:2]
[value:30.1944999694824]
[value:Austin Bergstrom International Airport]
```

Similarly, we could just decide to include the key names in the results.

```
g.V().has('code','AUS').
      properties().
      valueMap().with(WithOptions.tokens,WithOptions.keys)
```

Once again, the key and value are shown for the 'date' meta-property.

```
[key:country]
[key:code,date:2025-06-21T00:00Z]
[key:longest]
[key:city]
[key:elev]
[key:icao]
[key:lon]
```

```
[key:type]
[key:region]
[key:runways]
[key:lat]
[key:desc]
```

To include both the keys and the values in the result along with the meta-properties, 'WithOptions.all' can be used.

```
g.V().has('code','AUS').
      properties().
      valueMap().with(WithOptions.tokens,WithOptions.all)
```

Note that in this case, the ID for each property is also shown.

```
[id:28,key:country,value:US]
[id:29,key:code,value:AUS,date:2025-06-21T00:00Z]
[id:30,key:longest,value:12250]
[id:31,key:city,value:Austin]
[id:32,key:elev,value:542]
[id:33,key:icao,value:KAUS]
[id:34,key:lon,value:-97.6698989868164]
[id:35,key:type,value:airport]
[id:36,key:region,value:US-TX]
[id:37,key:runways,value:2]
[id:38,key:lat,value:30.1944999694824]
[id:39,key:desc,value:Austin Bergstrom International Airport]
```

4.9. Deducing the schema of a graph using queries

Sometimes, you may find yourself working with a graph while being unsure of its data model or schema. Using some simple Gremlin queries, we can quite easily figure out the major elements that a graph contains. This technique should only be used if the graph database you are using does not provide an explicit API for working with the schema of a graph. First, we can figure out the vertex labels that are in use as shown below. The 'dedup' step insures that we get a list of unique label names back.

```
g.V().label().dedup()

version
airport
country
continent
```

Similarly, we can find out the names of the edge labels in the graph.

```
g.E().label().dedup()

route
contains
```

You might also find it helpful to gather some statistical information about these labels. You can collect a label distribution using 'groupCount' for vertices or edges.

```
g.V().groupCount().by(label)

[continent:7,country:237,version:1,airport:3504]

g.E().groupCount().by(label)

[contains:7008,route:50637]
```

Now that we know the label names, it is very easy to get the names of the property keys for a vertex with a given label. The query below will display the names of the property keys found in an 'airport' vertex.

```
g.V().hasLabel('airport').limit(1).next().keys()

country
code
longest
city
elev
icao
lon
type
region
runways
lat
desc
```

You could easily put the previous query inside a simple loop if you wanted to iterate through each of the vertex labels that were discovered. As with vertices, we can also query edges to discover their key names. The query below finds the property key names for 'route' edges.

```
g.E().hasLabel('route').limit(1).next().keys()

dist
```

Lastly, now that we know the label names, and we know how to find out the property key names, we can also figure out the types associated with each key. The code below creates an array called 'pkeys' containing all the property key names for an 'airport' vertex. Having done that, the code

iterates through the list in a simple loop to find the type for each key. The code as shown is intended to be run inside the Gremlin Console.

```
pkeys=g.V().hasLabel('airport').limit(1).next().keys()

pkeys.each {
  printf("%10s : %s\n" , it,
          g.V().hasLabel('airport').limit(1).
            values(it).next().class)};[]
```

When run, we get back a nicely formatted table showing the key names and their types.

```
   country : class java.lang.String
      code : class java.lang.String
   longest : class java.lang.Integer
      city : class java.lang.String
      elev : class java.lang.Integer
      icao : class java.lang.String
       lon : class java.lang.Double
      type : class java.lang.String
    region : class java.lang.String
   runways : class java.lang.Integer
       lat : class java.lang.Double
      desc : class java.lang.String
```

We mentioned the use of 'groupCount' for doing label distributions to get some basic statistics on the graph schema. You can also use 'groupCount' to try to understand the graph structure by examining its paths and counting unique ones:

```
g.V().outE().otherV().groupCount().by(path().by(label)).unfold()

==>path[country, contains, airport]=3504
==>path[continent, contains, airport]=3504
==>path[airport, route, airport]=50637
```

Various graph databases may provide their own APIs for accessing the schema of a graph that don't involve surveying the graph. For example, JanusGraph allows us to define an explicit schema and also to query the schema using its Graph Management API, and Amazon Neptune provides a Graph Summary API that can yield these details.

4.10. Collections revisited

As we have seen in many of the prior examples, very often, either in the middle or at the end of a traversal, or both, we generate some kind of collection. In this section we are going to take a more focused look at these collections and how to work with them. In the following section we will look at collections and how they can be used effectively in conjunction with so-called 'reducing barrier'

traversal steps.

4.10.1. Steps that generate collections

Let's start this discussion by first reviewing a few ways that a collection can be generated. A simple example of a collection is the map generated by the 'group' step as shown in the example below.

```
g.V(1..5).group().by('code').by('runways')

[BNA:[4],ANC:[3],BOS:[6],ATL:[5],AUS:[2]]
```

Similarly, a map is created when the 'groupCount' step is used.

```
g.V().hasLabel('airport').limit(40).groupCount().by('region')

[US-FL:5,PR-U-A:1,US-NV:1,US-MN:1,US-HI:1,US-IL:1,US-TX:6,US-AK:1,US-WA:1,US-VA:1,US-
NY:4,US-CO:1,US-NC:1,US-LA:1,US-MD:1,US-IA:1,US-MA:1,US-CA:6,US-DC:1,US-UT:1,US-AZ:1
,US-GA:1,US-TN:1]
```

Likewise, when we use the fold step, a list is generated. We can use the 'order' step with a 'local' scope to order the contents of the list.

```
g.V().hasLabel('airport').limit(20).values('runways').
      fold().order(local)

[2,2,2,3,3,3,3,3,4,4,4,4,4,4,4,5,5,6,7,7]
```

Here is an example of a 'union' step followed by a 'fold' step that generates a list of two values. The 'union' step contains an 'identity' step to indicate that we want the value of the incoming vertex as the first item of the union. We 'union' that vertex with a count of all routes from that vertex (DFW) and finally use a 'fold' step to generate a list.

```
g.V().has('airport','code','DFW').union(identity(),out().count()).fold()

[v[8],253]
```

Note that the above syntax is a shorthand form of the following.

```
g.V().has('airport','code','DFW').as('a').union(select('a'),out().count()).fold()

[v[8],253]
```

If we wanted to generate a map with keys and values rather than a list, we could use a 'group' step. In this case the vertex is the keys, and the number of outgoing routes is the value.

```
g.V().has('airport','code','DFW').group().by().by(out().count())

[v[8]:253]
```

Another way that a map can be created is when the 'project' step is used.

```
g.V().has('airport','country','IE').project('loc','iata').by('city').by('code')

[loc:Dublin,iata:DUB]
[loc:Shannon,iata:SNN]
[loc:Cork,iata:ORK]
[loc:Charleston,iata:NOC]
[loc:Killarney,iata:KIR]
[loc:Waterford,iata:WAT]
[loc:Donegal,iata:CFN]
```

Also using a 'project' step, but a little more complex, this example creates a map with two keys. The first, called 'dfw', will contain the vertex for the DFW airport and the second, called 'route_count', will contain the number of outgoing routes from DFW. Notice how the first 'by' step has no parameters, so it returns the actual vertex (rather than say a property from the vertex that we could select).

```
 g.V().has('code','DFW').project('dfw','route_count').
       by().by(outE().count())

[dfw:v[8],route_count:253]
```

As well as generating maps, we can also generate a set using an 'aggregate' step so that duplicate values are not stored. The 'withSideEffect' step can be used to initialize the set. The 'cap' step emits the collection resulting from the side effect that we created using the 'store' step. Among other things, this allows us to return this collection as the final result of a query.

```
g.withSideEffect('s', [] as Set).
  V().hasLabel('airport').limit(20).values('runways').
      aggregate('s').cap('s').order(local)

[2,3,4,5,6,7]
```

The 'aggregate' step can also be used in conjunction with a 'by' modulator to specify exactly what is aggregated. The query below uses an 'aggregate' step to create a collection of runways but avoids the need to use a 'values' step.

```
g.V().has('region','US-TX').aggregate('r').by('runways').cap('r')
```

```
[2,2,2,2,2,2,2,2,7,5,3,3,3,3,3,3,3,3,3,3,4,4,4,4,1,1]
```

As another example, if we wanted to store the number of runways that each airport in Ireland has, we could do so as follows.

```
g.V().has('airport','country','IE').
      aggregate('ireland').by('runways').cap('ireland')

[2,2,2,5,1,1,1]
```

In the "Traits of 'by' modulators" section we learned about the concept of an unproductive 'by'. As a reminder, an unproductive 'by' is one that does not produce a result for the step that it is tied to. When 'aggregate' encounters an unproductive 'by', it has a filtering effect for that current traverser. For example, if in the prior example we made a mistake and mistyped '"runways"' as '"rnways"' we would have no results:

```
g.V().has('airport','country','IE').
      aggregate('ireland').by('rnways').cap('ireland')

[]
```

This behavior is meant as a convenience when using 'aggregate' with heterogeneous or incomplete elements where a value may not exist, thereby sparing an error. It may be tempting to exploit this mechanic as a direct means of filtering, as demonstrated in the following example:

```
g.V().has('airport','country','IE').
      aggregate('ireland').by(values('runways').is(gt(3))).
      cap('ireland')

[5]
```

While the previous example is syntactically correct, it is a bit of an indirection from a readaiblity perspective. It would be more idiomatic to write an explicit filter instead:

```
g.V().has('airport','country','IE').
      has('runways', gt(3)).
      aggregate('ireland').by('runways').
      cap('ireland')

[5]
```

If we are ever unsure what type of object has been created, a call to 'getClass' can be used to find out.

```
g.V(1..5).group().by('code').by('runways').next().getClass()

class java.util.HashMap

g.V(1..5).aggregate('a').cap('a').next().getClass()

class org.apache.tinkerpop.gremlin.process.traversal.step.util.BulkSet
```

Now that we have examined the various ways in which collections may get generated during a traversal, it is important to understand how the contents of a collection can be accessed and manipulated.

4.10.2. Accessing the contents of a collection

The keywords 'keys' and 'values' can be used to access the respective parts of a collection that is a map. Take a look at the query below which returns a map where airport codes are the keys and their city names are the values.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city')

[BNA:[Nashville],ANC:[Anchorage],BOS:[Boston],ATL:[Atlanta],AUS:[Austin]]
```

We can use a 'count' step with 'local' scope to find out how big the collection is.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').count(local)

5
```

The queries below extract the keys and values from the map that the 'group' step creates.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').select(keys)

[BNA,ANC,BOS,ATL,AUS]

g.V().hasLabel('airport').limit(5).group().by('code').by('city').select(values)

[[Nashville],[Anchorage],[Boston],[Atlanta],[Austin]]
```

We can also extract the keys and values from the results of the 'project' step we used earlier. Note that the values comeback as a list containing the DFW vertex and the number of routes from DFW.

```
g.V().has('code','DFW').project('dfw','route_count').
      by().by(outE().count()).select(values)
```

```
[v[8],253]
```

Likewise, the keys come back in a list.

```
g.V().has('code','DFW').project('dfw','route_count').
      by().by(outE().count()).select(keys)

[dfw,route_count]
```

We could also be even more specific and select which values we are interested in.

```
g.V().has('code','DFW').project('dfw','route_count').
       by().by(outE().count()).select('route_count')

253
```

We can also access the DFW vertex directly from the map.

```
g.V().has('code','DFW').project('dfw','route_count').
      by().by(outE().count()).select('dfw')

v[8]
```

Having extracted the vertex, we can retrieve values from it. A bit later we will look at ways we could continue our traversal from this point if we needed to, perhaps looking at outgoing routes from DFW or adding a new route. You will find that discussion in the "Collections and reducing barrier steps" section.

```
g.V().has('code','DFW').project('dfw','route_count').
      by().by(outE().count()).select('dfw').values('desc')

Dallas/Fort Worth International Airport
```

As we shall see in the next two sections, sometimes it is necessary to use the 'unfold' step to access the contents of a collection, and it is also sometimes necessary to use 'local' scope.

4.10.3. Using 'unfold' to unbundle a collection

Sometimes it is desirable to unbundle a collection so that we can work on it further. This is what the 'unfold' step does. If we apply 'unfold' to the previous query, you can see what is generated. The collection that the 'group' step generates is a Map. The 'unfold' step turns the HashMap into a series of "entries" which are essentially just key/value pairs.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').unfold()
```

```
BNA=[Nashville]
ANC=[Anchorage]
BOS=[Boston]
ATL=[Atlanta]
AUS=[Austin]
```

If we wanted a list of values, we could use 'unfold' again as shown below. You will recall from our earlier discussion that vertex properties are stored as lists even if there is only one property – a list of length one in other words. Note that this shows that a single 'unfold' step will not recursively unbundle elements from a collection.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').
      select(values).unfold()

[Nashville]
[Anchorage]
[Boston]
[Atlanta]
[Austin]
```

We could add a second 'unfold' to just get the city names back and remove the containing lists.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').
      select(values).unfold().unfold()

Nashville
Anchorage
Boston
Atlanta
Austin
```

As an alternative, 'repeat' can be used when you want to unfold more than once as shown below.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').
      select(values).repeat(unfold()).times(2)

Nashville
Anchorage
Boston
Atlanta
Austin
```

If we were not sure how many times we needed to 'unfold', we could change the query as follows. The 'repeat' loop will unfold until there is only a list of lists left. The final unfold will remove the remaining lists, leaving us with just the text values.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').
      select(values).repeat(unfold()).until(count(local).is(1)).unfold()

Nashville
Anchorage
Boston
Atlanta
Austin
```

Having used 'unfold' to extract just the city names as strings, we could re-fold one time to produce a list of airport names using the 'fold' step. This pattern of unfolding, performing an operation, and refolding is one that comes in handy quite often, especially in more complex queries.

```
g.V().hasLabel('airport').limit(5).group().by('code').by('city').
      select(values).unfold().unfold().fold()

[Nashville,Anchorage,Boston,Atlanta,Austin]
```

It is probably worth pointing out that the 'keys' and 'values' keywords can also be used with something as simple as a 'valueMap' step. Here is a simple example.

```
g.V(3).valueMap().select(keys)

[country,code,longest,city,elev,icao,lon,type,region,runways,lat,desc]

g.V(3).valueMap().select(values)

[[US],[AUS],[12250],[Austin],[542],[KAUS],[-97.6698989868164],[airport],[US-TX],[2
],[30.1944999694824],[Austin Bergstrom International Airport]]
```

4.10.4. Ways to merge collections

We've seen how 'fold' can be used to collect objects in a stream into a List. If the objects themselves are Lists and you wish to merge them to a new List, you can also do that with 'fold' after you've first flattened those individual List objects with 'unfold'.

```
g.V(3,4,5,6,7).map(out().values('code').limit(3).fold())

[PHL,PDX,DTW]
[ATL,AUS,BOS]
[ATL,AUS,BNA]
[YYZ,LHR,RSW]
[DEN,YOW,HPN]

g.V(3,4,5,6,7).map(out().values('code').limit(3).fold()).unfold().fold()
```

```
[PHL,PDX,DTW,ATL,AUS,BOS,ATL,AUS,BNA,YYZ,LHR,RSW,DEN,YOW,HPN]
```

Merging maps is slightly more complex than just a simple 'fold' because duplicate keys among the Maps must be resolved in some way. If you are not concerned about duplicates or simply want the last pair duplicate to be in the resulting Map, you could just use a specific form of 'fold' that includes the 'addAll' operator:

```
g.V(2304,2305,2306).valueMap("code","desc")

[code:[MJD],desc:[Moenjodaro Airport]]
[code:[GIL],desc:[Gilgit Airport]]
[code:[GWD],desc:[Gwadar International Airport]]

g.V(2304,2305,2306).valueMap("code","desc").fold([:], addAll)

[code:[MJD],desc:[Moenjodaro Airport]]
```

You also have the option to use the 'merge' step with collections:

```
g.V().has("code","IAD").valueMap().select(keys).merge(["name"])

[country,code,longest,city,lon,type,elev,name,icao,region,runways,lat,desc]

g.V().has("code","IAD").valueMap("code","desc").
  merge(V().has("code","DUB").valueMap("runways"))

[runways:[2],code:[IAD],desc:[Washington Dulles International Airport]]
```

A more complex case for merging Maps occurs when you want to aggregate together values of the same keys into a list, rather than having them overwrite one another. You can do this with Gremlin with a commonly used pattern of Gremlin steps, the basics of which have been explained earlier in this section.

```
g.V().has("code",within("IAD","DUB")).valueMap("code","desc","runways").
  unfold().
  group().by(keys).by(select(values).unfold().fold())

[code:[IAD,DUB],runways:[4,2],desc:[Washington Dulles International Airport,Dublin
International Airport]]
```

The previous example uses 'unfold' incoming Maps to entries and then groups on the keys of those entries thereby building a list of each of the values in the final Map.

4.10.5. Using 'local' scope with collections

Sometimes if you want to work on the contents of a collection, whether to sort it or perhaps select

some subset of it, it is often necessary to use 'local' scope. There are other ways the following examples could be written, but we wanted to show some ways that 'local' scope can be combined with the 'order', 'range', 'limit,' and 'tail' steps while working with collections.

First, let's, once again produce a collection using airports in Ireland. This time we produce a map where the keys are the airport codes and the values are the number of runways at that airport.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways')

[DUB:[2],SNN:[5],NOC:[1],KIR:[2],ORK:[2],CFN:[1],WAT:[1]]
```

We already know how to select the keys from the map using our prior examples, but for completeness let's take another look.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways').select(keys)

[DUB,SNN,NOC,KIR,ORK,CFN,WAT]
```

If we wanted to sort the keys by ascending order, we could use an 'order' step with 'local' scope.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways').select(keys).order(local)

[CFN,DUB,KIR,NOC,ORK,SNN,WAT]
```

It is worth noting that this is a case where we could have used the 'unfold' and 'fold' pattern instead as shown below.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways').select(keys).
      unfold().order().fold()

[CFN,DUB,KIR,NOC,ORK,SNN,WAT]
```

The next example also uses 'local' scope along with a 'limit' step to retrieve the first two airport keys.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways').select(keys).limit(local,2)

[DUB,SNN]
```

We can use the same 'local' scope with the 'tail' step to select the last three keys.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways').select(keys).tail(local,3)

[ORK,CFN,WAT]
```

As you would expect, we can also specify 'local' scope on a 'range' step.

```
g.V().has('airport','country','IE').
      group().by('code').by('runways').select(keys).range(local,3,5)

[KIR,ORK]
```

You can combine the 'limit' and 'tail' steps with 'local' scope to extract the beginning or ending entries from a list of values as shown below. The following query creates a list containing the IATA codes for all airports in Texas.

```
g.V().has('region','US-TX').values('code').fold()

[AUS,DFW,IAH,SAT,HOU,ELP,DAL,LBB,HRL,MAF,CRP,ABI,ACT,CLL,BPT,AMA,BRO,GGG,GRK,LRD,MFE,S
JT,SPS,TYR,VCT,AFW,DRT]
```

Using 'local' scope with a 'limit' step, we can extract just the first two entries from the list.

```
g.V().has('region','US-TX').values('code').fold().limit(local,2)

[AUS,DFW]
```

Likewise, using 'local' scope with a 'tail' step, we can extract just the last two entries from the list.

```
g.V().has('region','US-TX').values('code').fold().tail(local,2)

[AFW,DRT]
```

It is worth noting that 'local' scope can also be used with a 'dedup' step. The query below finds airports in the US and produces a sorted list of the unique region codes by only allowing one airport from each region to proceed to the next steps of the traversal.

```
g.V().has('country','US').dedup().by('region').values('region').order().fold()
```

The rewritten version of the query below allows the region codes for every airport to be collected, and then the 'dedup' is applied to the resultant collection using 'local' scope. The prior query is likely to be the more efficient way of doing this, but we wanted to make it clear that existing collections of values can have 'dedup' applied to them using 'local' scope.

```
g.V().has('country','US').values('region').order().fold().dedup(local)
```

When either query is run, here are the results that are returned.

```
[US-AK,US-AL,US-AR,US-AZ,US-CA,US-CO,US-CT,US-DC,US-DE,US-FL,US-GA,US-HI,US-IA,US-ID
,US-IL,US-IN,US-KS,US-KY,US-LA,US-MA,US-MD,US-ME,US-MI,US-MN,US-MO,US-MS,US-MT,US-NC
,US-ND,US-NE,US-NH,US-NJ,US-NM,US-NV,US-NY,US-OH,US-OK,US-OR,US-PA,US-RI,US-SC,US-SD
,US-TN,US-TX,US-UT,US-VA,US-VT,US-WA,US-WI,US-WV,US-WY]
```

Lastly, we can combine some of the prior examples to limit, order and deduplicate the contents of a group. The query below does not include a 'limit' step, so it retrieves all possible results. The goal of the query is to build a group, with labels as the keys and vertex 'code' properties as the values for both incoming and outgoing edges connected to AUS (Austin). The results are deduplicated so that no airport code appears twice.

```
g.V().has('code','AUS').
      both().
      group().
        by(label).
        by(values('code').fold().dedup(local).order(local))
```

When run, it returns the following results.

```
[continent:[NA],country:[US],airport:[ABQ,AMA,AMS,ASE,ATL,BHM,BKG,BNA,BOI,BOS,BTR,BUF,
BUR,BWI,BZN,CHS,CLE,CLT,CMH,CUN,CVG,CZM,DAL,DCA,DEN,DFW,DSM,DTW,ECP,ELP,EWR,FLL,FRA,GD
L,GRR,HDN,HNL,HOU,HRL,IAD,IAH,IND,JAX,JFK,LAS,LAX,LBB,LGB,LGW,LHR,LIR,LIT,MCI,MCO,MDW,
MEM,MEX,MIA,MKE,MSP,MSY,NAS,OAK,OKC,OMA,ONT,ORD,PDX,PHL,PHX,PIE,PIT,PNS,PVD,PVR,RDU,RN
O,SAN,SAT,SDF,SEA,SFB,SFO,SJC,SJD,SLC,SMF,SNA,STL,TPA,TUL,TUS,TYS,VPS,XNA,YVR,YYC,YYZ]
]
```

However, let's assume we wanted to limit the query to a maximum of ten results for any of the keys in the group. We can do so by adding a 'limit' step with 'local' scope as shown below.

```
g.V().has('code','AUS').
      both().
      group().
        by(label).
        by(values('code').fold().dedup(local).order(local).limit(local,10))
```

This time when run, the number of results returned is restricted.

```
[continent:[NA],country:[US],airport:[ABQ,AMA,AMS,ASE,ATL,BHM,BKG,BNA,BOI,BOS]]
```

In the next section we will continue our look at Gremlin's collections and how they can be used in conjunction with 'reducing barrier' steps to still achieve a desired result. We will also see other cases where 'unfold' is needed to access the parts of a collection that we care about.

4.11. Collections and reducing barrier steps

If you look at commonly asked questions about writing Gremlin queries on the Gremlin Users discussion list, one area that repeatedly seems to cause people confusion is the behavior of certain traversal steps that are known as 'reducing barrier' steps. What these steps in essence do is reduce the results of the traversal so far to a single traversal, often just a value or a collection of some kind and from that point on you cannot refer back to things you did earlier in the query.

Table 8. Reducing barrier steps

max	Returns the maximum value from a set of values.
min	Returns the minimum value from a set of values.
sum	Returns the sum of a set of values.
count	Counts the number of current elements.
fold	Aggregates current traversal into a map.

Take a look at the example below. On the surface, you might expect the query to count all the routes originating from the DFW airport and return the count '"b"' along with the airport vertex '"a"'.

```
g.V().has('code','DFW').as('a').outE().count().as('b').select('a','b')
```

What actually happens is that nothing is returned. This is because the 'count' step is a so-called 'reducing barrier' step. Once the 'count' has been processed, you have crossed the 'barrier' and the traversal variable '"a"' is no longer available to us. We can still access '"b"' if we reference it by itself as it is defined after the 'count' step as shown below.

```
g.V().has('code','DFW').as('a').outE().count().as('b').select('b')

253
```

In cases such as this, it is almost always possible to achieve the results that you want by changing the way you write the query. It is important to gain an understanding of how different traversal steps work. A great way to do that is to experiment using the Gremlin Console and look at the way different steps operate. The rewritten query below achieves our original goal.

```
g.V().has('code','DFW').group().by().by(outE().count())

[v[8]:253]
```

If this was all we needed, then our job is done. We have a map containing the vertex as the key, and

its outgoing route count as the value. However, there is still an issue if we want to go further with this query. Take a look at the example below. Because the query has reduced the prior traversal to essentially a small map, including a count, we can no longer refer back to '"a"'.

```
g.V().has('code','DFW').as('a').group().by().by(outE().count()).select('a')
```

If for some reason, we wanted to retrieve the vertex that we had stored in '"a"', we should instead pull it from the map that the 'group' step created. You can access the keys of a map using the 'keys' keyword as a parameter to a 'select' step.

```
g.V().has('code','DFW').group().by().by(outE().count()).
                        select(keys)

[v[8]]
```

We have still not quite got the result we wanted as the vertex is still returned in a list. So we can modify the query again to 'unfold' the map before we select the keys from it.

```
g.V().has('code','DFW').group().by().by(outE().count()).
                        unfold().select(keys)
v[8]
```

If we want to get the value back instead of the key, we can use the 'values' keyword as follows.

```
g.V().has('code','DFW').group().by().by(outE().count()).
                        unfold().select(values)

253
```

To prove we could carry on adding to the query from here, let's get back the airport code that we started with.

```
g.V().has('code','DFW').group().by().by(outE().count()).
                        unfold().select(keys).values('code')

DFW
```

So, let's now add to our query and create a new vertex with a label 'dfwcount' that is going to store the number of routes originating in DFW using a property called 'current_count'.

```
g.V().has('code','DFW').group().by().by(outE().count()).as('ct').
      addV('dfwcount').
      property('current_count',select('ct').unfold().select(values))
```

```
v[61394]
```

We can inspect the new vertex to double-check that our query worked as intended.

```
g.V(61394).valueMap(true)

[id:61394,label:dfwcount,current_count:[253]]
```

Hopefully, you are starting to see a pattern here. It is important to understand which steps are 'reducing barrier' steps and be able to work with them in a way that allows you to write queries that do what you need.

4.11.1. Calculating the 'sum' of a collection

The query below returns a map where the keys are airport IATA codes and the values are the number of runways at the airport. The results are ordered to aid readability.

```
g.V().hasLabel('airport').limit(10).
      group().by('code').by(values('runways')).
      order(local).by(values)

[FLL:2,AUS:2,DCA:3,BWI:3,ANC:3,BNA:4,IAD:4,ATL:5,BOS:6,DFW:7]
```

Given such a collection of airports and runways, we might want to calculate the total number of runways present. The query below achieves that. Note that in order to select the values from the collection, an 'unfold' step is used to turn the collection back into a stream from which the values can be selected.

```
g.V().hasLabel('airport').limit(10).
      group().by('code').by(values('runways')).
      unfold().
      select(values).
      sum()

39
```

We can also write the query using 'local' scope rather than an 'unfold' step as shown below.

```
g.V().hasLabel('airport').limit(10).
      group().by('code').by(values('runways')).
      local(select(values).sum(local))

39
```

4.11.2. Using the 'math' step with collections

Building on the examples from the previous section, let's now take these experiments one step further and look at ways to apply the 'math' step to values from one or more collections. Given we know that there are 10 values in our collection, we can easily use a 'math' step to calculate the average of those values.

```
g.V().hasLabel('airport').limit(10).
      group().by('code').by(values('runways')).
      unfold().
      select(values).
      sum().
      math('_ / 10')

3.9
```

We may not always know ahead of time how many entries a collection has. The modified example below uses a 'project' step to feed the 'math' step with two values representing the total number of runways and the number of members in the collection.

```
g.V().hasLabel('airport').limit(10).
      group().by('code').by(values('runways')).
      project('total','number').
        by(select(values).unfold().sum()).
        by(count(local)).
      math('total / number')

3.9
```

Sometimes you may want to perform computations on the sums of multiple collections. The two queries shown below create maps of airport and runway key/value pairs for all the airports in New Mexico and Arizona respectively.

```
g.V().has('region','US-NM').
      group().by('code').by(values('runways'))

[ABQ:4,SVC:4,CNM:4,FMN:2,SAF:3,LAM:1,HOB:3,CVN:3,ROW:3]
```

If you were to add up the runways at the New Mexico airports, you would find there are 27, and likewise there are 26 runways across the Arizona airports.

```
g.V().has('region','US-AZ').
      group().by('code').by(values('runways'))

[YUM:4,PRC:3,FLG:1,PHX:3,IFP:1,TUS:3,GCN:1,AZA:3,SOW:2,PGA:2,IGM:3]
```

Given these two collections, we might want to divide the sum of one set of values by the other to calculate the ratio between the total number of runways in Arizona and New Mexico. The query below does just that. While at first glance this query looks a bit complicated, it is in fact just the result of combining the prior few queries we have looked at into a single query.

```
g.V().has('region','US-NM').
      group().by('code').by(values('runways')).
      select(values).
      unfold().
      sum().
      aggregate('a').
      V().has('region','US-AZ').
      group().by('code').by(values('runways')).
      select(values).
      unfold().
      sum().
      aggregate('b').
      project('first','second').
        by(select('a').unfold()).
        by(select('b').unfold()).
      math('first / second')

1.0384615384615385
```

Given we calculated that there were 27 runways in New Mexico and 26 in Arizona, we can verify that we have the right result using the Gremlin Console.

```
gremlin> 27/26

==>1.0384615385
```

4.12. Introducing 'sack' as a way to store values

As is hopefully now becoming apparent, to efficiently write certain types of queries, you need a way to build up a collection of items as the traversal takes place. We have already looked at steps like 'aggregate' and 'store' that can create collections of items during a traversal. However, the 'sack' step offers an additional capability in that we can specify how items are added to the collection. For example, they can be added using addition, multiplication, subtraction, or division. Alternatively, we can store the minimum or maximum value of a pair of values. These and other sack operators will be used in different parts of this book.

4.12.1. Basic 'sack' operations

The 'sack' step, as shown in the examples below, is a side effect step, meaning it can store values during a traversal but has no effect on what is passed on to the next step.

By way of an introduction, take a look at the example below. All the query does as it stands is create

a list of the number of runways that each airport that you can fly to from Santa Fe (SAF) has. We have not introduced a 'sack' step into the equation yet.

```
g.V().has('code','SAF').out().values('runways').fold()

[7,4,3,6]
```

Now let's start to introduce some usage of sacks into the query. When working with a 'sack', we typically initialize the sack in some way. The example below initializes a sack with a value of zero but does not yet do anything with it, so the result is unchanged.

```
g.withSack(0).V().has('code','SAF').out().
              values('runways').fold()

[7,4,3,6]
```

This time we add the runways to our sack using a sum operation, but as our starting value is zero, we are not changing the result in any way. This is because we essentially perform the operation '0 + runways' for each runway value. The final call to 'sack' with no parameters causes the current contents of the 'sack' to be returned. The 'fold' step, as before, puts whatever results we got from the 'sack' into a list.

```
g.withSack(0).V().has('code','SAF').out().
              values('runways').sack(sum).sack().fold()

[7,4,3,6]
```

So let's finally do something that will change the result. Let's make the starting value one rather than zero and see what happens.

```
g.withSack(1).V().has('code','SAF').out().
              values('runways').sack(sum).sack().fold()

[8,5,4,7]
```

Now, each time the number of runways was added to the sack using a 'sum' operator, the operation that was performed was '1 + runways'. As you can see from the results, in each case, the value returned is one higher than those from the previous query. This is a simple example, but hopefully, you can start to see how useful sacks can be.

Before getting into some more interesting examples, it is worth pointing out that you can also initialize a sack using the 'assign' operator as shown below. It is also important to note that this sack initialization does not have to happen at the start of the query when done in this way. In the example below, a constant value of one is used, but we could equally well have used a traversal to initialize that sack, as we shall see in the next example.

```
g.V().sack(assign).by(constant(1)).has('code','SAF').
      out().values('runways').sack(sum).sack().fold()

[8,5,4,7]
```

Let's now make our query a bit more interesting. There are a couple of interesting new twists shown in the query below. Firstly, the sack is initialized to contain the number of runways from the AUS vertex, whereas before we just used a simple constant. Secondly, notice that rather than make an explicit call to 'values' before adding to the sack, we can just use a 'by' modulator to specify what we want added to our sack.

```
g.V().has('code','AUS').sack(assign).by('runways').
  V().has('code','SAF').out().
      sack(sum).by('runways').sack().fold()

[9,6,5,8]
```

This time, as the Austin (AUS) airport has two runways, our calculation in effect became '2 + runways'.

Before looking at some slightly more complex queries that use multiple 'sack' steps, we should take a look at some of the other operators. So far we have just used 'sum'. The query below uses 'mult, which,' as its name implies, will multiply the values together rather than add them.

```
g.V().has('code','AUS').sack(assign).by('runways').
  V().has('code','SAF').out().
      sack(mult).by('runways').sack().fold()

[14,8,6,12]
```

Likewise, minus will subtract the values before putting them into the sack. Note that the values are subtracted **from** the sack's initialization value.

```
g.V().has('code','AUS').sack(assign).by('runways').
  V().has('code','SAF').out().
      sack(minus).by('runways').sack().fold()

[-5,-2,-1,-4]
```

If we want to subtract the sack's initial value from the other values, we can simply initialize it with a negative value and perform a 'sum' operation.

```
g.V().sack(assign).by(constant(-1)).has('code','SAF').
      out().values('runways').sack(sum).sack().fold()
```

```
[6,3,2,5]
```

4.12.2. Using 'min' and 'max' with a 'sack'

There are many different operators that can be used with sacks. They are defined as part of the TinkerPop Java Enum called 'Operator'. Two such operators that we can use are 'min' and 'max'. The example below looks at the distances of all routes that start at SAF and in each case returns the minimum of the distance or 400 which is assigned to the sack at the start of the query.

```
g.V().sack(assign).by(constant(400)).has('code','SAF').
      outE().sack(min).by('dist').sack().fold()

[400,400,369,303]
```

In a similar vein, this query picks the maximum of the actual distance or 400.

```
g.V().sack(assign).by(constant(400)).has('code','SAF').
      outE().sack(max).by('dist').sack().fold()

[549,708,400,400]
```

4.12.3. Doing calculations using a 'sack'

Now let's look at a more complex, and hopefully more interesting, example. The challenge is to write a query that shows 10 routes that start at Santa Fe (SAF) and have one stop. We also want to return the distance between each hop and the total distance of the two hops. This is a perfect example of a query where using a 'sack' can help. For completeness the results are sorted by overall route distance.

```
g.withSack(0).
  V().has('code','SAF').
      repeat(outE().sack(sum).by('dist').inV()).times(2).limit(10).
      order().by(sack()).
      sack().path().
      by('code').by('dist').by('code').by('dist').by('code').by()
```

If we run our query, we should get back something that looks like the output shown below. Notice how the output from the 'sack' (the last item in each row) contains the sum of the two prior route distances. So, for example, we can see that the total distance from SAF to ATL with a stop in DFW is 1278 miles.

```
[SAF,549,DFW,1197,YYZ,1746]
[SAF,549,DFW,1751,YVR,2300]
[SAF,549,DFW,4458,DUB,5007]
[SAF,549,DFW,4736,LHR,5285]
```

```
[SAF,549,DFW,4933,CDG,5482]
[SAF,549,DFW,5127,FRA,5676]
[SAF,549,DFW,5208,HEL,5757]
[SAF,549,DFW,6410,NRT,6959]
[SAF,549,DFW,8022,DXB,8571]
[SAF,549,DFW,8574,SYD,9123]
```

What may not be obvious from the query above is that we are in fact using multiple 'sack' steps within the same query. If we were to remove the 'repeat' and write the query out in full, this becomes more obvious.

```
g.withSack(0).
  V().has('code','SAF').
      outE().limit(10).sack(sum).by('dist').inV().
      outE().limit(10).sack(sum).by('dist').inV().
      sack().path().
      by('code').by('dist').by('code').by('dist').by('code').by()
```

Later, in the "Using 'sack' to calculate the shortest AUS-LHR route with one stop" section, we will again use a 'sack' to help calculate multi hop route distances using an approach similar to the example above.

So far we have just used simple integer values to initialize our sacks. However, it is also possible to use non-primitive types such as maps when working with a sack. You will find an example of 'sack' being used to generate a map in the "Another example of how 'sack' can be used" section.

4.12.4. Doing calculations without using a 'sack'

A similar result to those from the queries in the previous section can actually be achieved without using a 'sack'. We find the 'sack' form to be quite concise, but you can also use a 'project' step to achieve similar results as shown below. The key thing to notice about this query is that the path generated by the first half of the query is unfolded to get the distances from the edges. As only edges have a 'dist' property in the `air-routes` graph, a 'coalesce' step is used to generate the distance from the edges or a constant value of zero otherwise. If we did not do this, the query would generate an error message as airport vertices do not have a 'dist' property.

```
g.V().has('code','SAF').
      repeat(outE().inV().simplePath()).times(2).limit(10).
        project('path','total').
          by(path().by('code').by('dist')).
          by(path().unfold().coalesce(values('dist'),constant(0)).sum()).
          order().by(select('total'))
```

When run, the query produces the following results. Note that the output includes the 'path' and 'total' key names and also that the path is in a list nested inside another list.

```
[path:[SAF,549,DFW,1197,YYZ],total:1746]
[path:[SAF,549,DFW,1751,YVR],total:2300]
[path:[SAF,549,DFW,4458,DUB],total:5007]
[path:[SAF,549,DFW,4736,LHR],total:5285]
[path:[SAF,549,DFW,4933,CDG],total:5482]
[path:[SAF,549,DFW,5127,FRA],total:5676]
[path:[SAF,549,DFW,5208,HEL],total:5757]
[path:[SAF,549,DFW,6410,NRT],total:6959]
[path:[SAF,549,DFW,8022,DXB],total:8571]
[path:[SAF,549,DFW,8574,SYD],total:9123]
```

The query can be refined a bit more to remove the 'path' and 'total' key names from the result by just selecting the values for each.

```
g.V().has('code','SAF').
      repeat(outE().inV().simplePath()).times(2).limit(10).
        project('path','total').
          by(path().by('code').by('dist')).
          by(path().unfold().coalesce(values('dist'),constant(0)).sum()).
          order().by(select('total')).
          select(values)
```

Now the results are exactly the same as from the version of the query that used 'sack'. This was a longer query to write, but in some cases, depending upon the graph database implementation, could be more efficient as processing of results is left until the second half of the query.

```
[[SAF,549,DFW,1197,YYZ],1746]
[[SAF,549,DFW,1751,YVR],2300]
[[SAF,549,DFW,4458,DUB],5007]
[[SAF,549,DFW,4736,LHR],5285]
[[SAF,549,DFW,4933,CDG],5482]
[[SAF,549,DFW,5127,FRA],5676]
[[SAF,549,DFW,5208,HEL],5757]
[[SAF,549,DFW,6410,NRT],6959]
[[SAF,549,DFW,8022,DXB],8571]
[[SAF,549,DFW,8574,SYD],9123]
```

We can add one more refinement to make the results exactly the same as those from the version of the query that used 'sack' by unfolding the results and then refolding them with 'local' scope.

```
g.V().has('code','SAF').
      repeat(outE().inV().simplePath()).times(2).limit(10).
        project('path','total').
          by(path().by('code').by('dist')).
          by(path().unfold().coalesce(values('dist'),constant(0)).sum()).
          order().by(select('total')).
```

```
          select(values).local(unfold().unfold().fold())
```

Now the results look just the same as those we got using 'sack'.

```
[SAF,549,DFW,1197,YYZ,1746]
[SAF,549,DFW,1751,YVR,2300]
[SAF,549,DFW,4458,DUB,5007]
[SAF,549,DFW,4736,LHR,5285]
[SAF,549,DFW,4933,CDG,5482]
[SAF,549,DFW,5127,FRA,5676]
[SAF,549,DFW,5208,HEL,5757]
[SAF,549,DFW,6410,NRT,6959]
[SAF,549,DFW,8022,DXB,8571]
[SAF,549,DFW,8574,SYD,9123]
```

The query can also be written using the same 'path' and 'coalesce' approach but this time using a 'union' step with 'local' scope. This is more concise than the version that uses 'project' and may be sufficient if you do not need to select parts of the result individually using a key name.

```
g.V().has('code','SAF').
      repeat(outE().inV().simplePath()).times(2).limit(10).
      local(union(path().by('code').by('dist').unfold(),
                  path().unfold().coalesce(values('dist'),constant(0)).sum()).fold()).
      order().by(tail(local,1))
```

Once again we have the same results as we had when using 'sack' or 'project'.

```
[SAF,549,DFW,1197,YYZ,1746]
[SAF,549,DFW,1751,YVR,2300]
[SAF,549,DFW,4458,DUB,5007]
[SAF,549,DFW,4736,LHR,5285]
[SAF,549,DFW,4933,CDG,5482]
[SAF,549,DFW,5127,FRA,5676]
[SAF,549,DFW,5208,HEL,5757]
[SAF,549,DFW,6410,NRT,6959]
[SAF,549,DFW,8022,DXB,8571]
[SAF,549,DFW,8574,SYD,9123]
```

As we have seen, the 'sack' form is quite concise; however, the 'project' and 'union' forms have a nice feature that they will work unchanged no matter how long the resultant path is. As you may have noticed in the prior section, the 'sack' version of the query was tailored to produce results for a two-hop query. We could, of course, rewrite the 'sack' version to be equally flexible as shown below. The key difference is that the result from the sack is not included in the path but factored in later inside a 'union' step.

```
g.withSack(0).
```

```
  V().has('code','SAF').
      repeat(outE().sack(sum).by('dist').inV()).times(2).limit(10).
      order().by(sack()).
      local(union(path().by('code').by('dist'),
                  sack()).fold()).
      local(unfold().unfold().fold())
```

Again we have the same results.

```
[SAF,549,DFW,1197,YYZ,1746]
[SAF,549,DFW,1751,YVR,2300]
[SAF,549,DFW,4458,DUB,5007]
[SAF,549,DFW,4736,LHR,5285]
[SAF,549,DFW,4933,CDG,5482]
[SAF,549,DFW,5127,FRA,5676]
[SAF,549,DFW,5208,HEL,5757]
[SAF,549,DFW,6410,NRT,6959]
[SAF,549,DFW,8022,DXB,8571]
[SAF,549,DFW,8574,SYD,9123]
```

So in summary, as is almost always the case, there is more than one way to get the results you need using a Gremlin query.

4.12.5. Computing hop counts using a 'sack'

The next query we are going to look at uses a 'sack' to keep track of how many flight segments (or hops) a 'path' consists of. This means that along with the 'path' result, we can also return the path's 'hop count'.

The query below looks for routes between San Francisco and Wellington and returns the first 10 results found along with the number of flights that would need to be taken between the source and destination airports. As part of the 'repeat' step, each time an 'out' step is taken, a 'constant' value of one is added to the sack for that individual path. Finally, a 'union' step is used to combine the individual path with its hop count.

```
g.withSack(0).V().
  has('code','SFO').
  repeat(out().simplePath().sack(sum).by(constant(1))).
    until(has('code','WLG')).
  limit(10).
  local(union(path().by('code'),sack()).fold())
```

When run, the query returns ten lists containing the airports visited and the number of hops in each individual list.

```
[[SFO,SYD,WLG],2]
[[SFO,MEL,WLG],2]
```

```
[[SFO,AKL,WLG],2]
[[SFO,BNE,WLG],2]
[[SFO,YVR,SYD,WLG],3]
[[SFO,YVR,MEL,WLG],3]
[[SFO,YVR,AKL,WLG],3]
[[SFO,YVR,BNE,WLG],3]
[[SFO,NRT,SYD,WLG],3]
[[SFO,NRT,MEL,WLG],3]
```

To return the results with the longer routes coming first, we could use the values in the sack along with an 'order' step.

```
g.withSack(0).V().
  has('code','SFO').
  repeat(out().simplePath().sack(sum).by(constant(1))).
    until(has('code','WLG')).
  limit(10).
  order().by(sack(),desc).
  local(union(path().by('code'),sack()).fold())
```

This time the three hop routes appear first in the results.

```
[[SFO,YVR,SYD,WLG],3]
[[SFO,YVR,MEL,WLG],3]
[[SFO,YVR,AKL,WLG],3]
[[SFO,YVR,BNE,WLG],3]
[[SFO,NRT,SYD,WLG],3]
[[SFO,NRT,MEL,WLG],3]
[[SFO,SYD,WLG],2]
[[SFO,MEL,WLG],2]
[[SFO,AKL,WLG],2]
[[SFO,BNE,WLG],2]
```

As a side note, if we wanted to include the path's length rather than the hop count, we could change the query as shown below. In this case a 'sack' step is not needed as we can just count the length of each path.

```
g.V().has('code','SFO').
      repeat(out().simplePath()).
        until(has('code','WLG')).
      limit(10).
      local(union(path().by('code'),path().count(local)).fold())
```

This time the number shown is one bigger than in the previous example as the starting airport is included in the overall count.

```
[[SFO,SYD,WLG],3]
[[SFO,MEL,WLG],3]
[[SFO,AKL,WLG],3]
[[SFO,BNE,WLG],3]
[[SFO,YVR,SYD,WLG],4]
[[SFO,YVR,MEL,WLG],4]
[[SFO,YVR,AKL,WLG],4]
[[SFO,YVR,BNE,WLG],4]
[[SFO,NRT,SYD,WLG],4]
[[SFO,NRT,MEL,WLG],4]
```

4.12.6. Using boolean operators with a 'sack'

You can also use the boolean operators 'or' and 'and' when working with a 'sack'. The examples below just test the basic functionality using constants of 'true' and 'false'. In the first example the result returned in the 'sack' is 'false' as we used an 'and' operator to 'and' together the sack's initial value of 'true' and a constant value of 'false'.

```
g.V(3).sack(assign).by(constant(true)).sack(and).by(constant(false)).sack()

false
```

This time we replace the 'and' operator with an 'or' and the result, as we would expect, is 'true'.

```
g.V(3).sack(assign).by(constant(true)).sack(or).by(constant(false)).sack()

true
```

While proving we can do boolean operations using constants is interesting, it is perhaps not that useful. Where this functionality becomes more interesting is if the values being used come from, for example, a vertex property. So let's create a couple of vertices that each have a boolean property.

```
g.addV('happy').property('happy',true)

v[61394]

g.addV('sad').property('happy',false)

v[61396]
```

We can now write a query that uses a boolean operator to generate a result in our sack. The example below is a bit arbitrary, but it shows how the boolean operators can be used. We start at the vertex with an ID of 3, initialize a 'sack' with the constant 'true,' and the use the 'and' operator against the 'happy' property of the vertex with a label of 'happy'. We return the results in a path,

which will contain the starting vertex and the results of the 'and' operation. The result of the 'and' is 'true' as the 'happy' vertex has a value of 'true' for its 'happy' property.

```
g.V(3).sack(assign).by(constant(true)).sack(and).
       by(V().hasLabel('happy').values('happy')).sack().path()

[v[3],true]
```

If we repeat the query but this time use the 'sad' vertex, we get the expected result of 'false' from the 'and' operation.

```
g.V(3).sack(assign).by(constant(true)).sack(and).
       by(V().hasLabel('sad').values('happy')).sack().path()

[v[3],false]
```

4.12.7. Using 'addAll' and lists with a 'sack'

So far we have looked at using numbers and boolean values with a sack. However, sacks can also contain lists and, as we shall see later, maps.

The example below initializes a sack with an empty list '"[]"' and then uses the 'addAll' operator to store the same results we have seen generated above in a list. Note that a 'fold' step is used to create a list of values that can then be added to the sack. Later we shall look at other ways to build up lists with sacks that do not first fold all the traversal results into a list.

```
g.withSack([]).V().has('code','SAF').out().
               values('runways').fold().sack(addAll).sack()

[7,4,3,6]
```

You might be thinking, and you would be right, that we could have achieved the same result without using a 'sack' at all and just using a fold 'step' as shown below. However, the power of using a 'sack' becomes apparent when you need to build up a list containing the results of various parts of the query.

```
g.V().has('code','SAF').out().values('runways').fold()

[7,4,3,6]
```

The next example, below, shows how a list can be built up using more than one 'sack' step. The 'sack' is initialized with an empty list, and the runway counts of the airports reachable from SAF are again added initially to the list using the 'addAll' operator. Having done that, we add the runway counts for the airports reachable from AUS to the sack. You can see that the output starts with the same 7,4,3,6 sequence we have seen before but is then followed by all the other values

added by the second 'sack' step.

```
g.withSack([]).V().has('code','SAF').out().
                values('runways').fold().sack(addAll).
                V().has('code','AUS').out().values('runways').fold().
                sack(addAll).sack()

[7,4,3,6,4,3,6,4,2,4,2,4,4,3,3,3,4,4,5,5,3,3,1,2,3,2,2,2,2,3,2,3,3,1,3,3,2,2,2,2,3,2,2
,3,1,2,1,2,2,1,1,4,2,1,3,1,4,5,4,6,3,3,7,2,4,5,4,4,4,4,4,7,3,3,3,4,3,3,1,3,2,4,4,6,3,5
,4,3,2,2,3,2,4,4,3,6,3,3,4,2,3,4]
```

Finally, here is the previous query again, but this time the 'sack' is initialized with a list that already has some values in it. You can see from the output that the values we generated above are added after the 1,1,1,1 sequence that the sack was initialized with.

```
g.withSack([1,1,1,1]).
     V().has('code','SAF').out().
     values('runways').fold().sack(addAll).
     V().has('code','AUS').out().values('runways').fold().
     sack(addAll).sack()

[1,1,1,1,7,4,3,6,4,3,6,4,2,4,2,4,4,3,3,3,4,4,5,5,3,3,1,2,3,2,2,2,2,3,2,3,3,1,3,3,2,2,2
,2,3,2,2,3,1,2,1,2,2,1,1,4,2,1,3,1,4,5,4,6,3,3,7,2,4,5,4,4,4,4,4,7,3,3,3,4,3,3,1,3,2,4
,4,6,3,5,4,3,2,2,3,2,4,4,3,6,3,3,4,2,3,4]
```

So far, all the examples have used a 'fold' step before the 'sack' step. While this gives us a useful result, it may not always be the result that we want. To put it another way, what we get back is a single list containing all the values that we generated during the traversal. In some cases what we actually want might be a set of lists where each list contains whatever the 'sack' was initialized with plus just the values specific to each path the traverser takes. In other words, we want the 'sack' to have more of a local scope and not act like a global variable.

When we were thinking about this and doing some experiments using the Gremlin Console, our first thought was "We can use a 'local' step for this". So, we initially tried the query shown below. However, as you can see, while this definitely generated some different output, it did not generate what we wanted.

```
g.withSack([1,1,1,1]).V().has('code','SAF').out().
     values('runways').local(fold().sack(addAll)).sack()

[1,1,1,1,7]
[1,1,1,1,7,4]
[1,1,1,1,7,4,3]
[1,1,1,1,7,4,3,6]
```

What is happening above is that the 'sack' is still acting more like a global variable than a local one. Each 'sack' step generated a list that was based on the previous one.

To get the results we wanted, we needed to use a 'clone' operation. This query uses the Groovy closure or 'Lambda' syntax. We will investigate that syntax more a bit later in the "Using Lambda functions" section but for now all we need to know is that the 'clone' will ensure that each traverser gets its own copy of the original sack and is not affected by what other traversers do to their sacks. Remember that a Gremlin query, or traversal, in essence, causes a set of traversers to follow the paths through the graph that your query demands.

```
g.withSack{[1,1,1,1]}{it.clone()}V().has('code','SAF').
       out().values('runways').local(fold().sack(addAll)).sack()

[1,1,1,1,7]
[1,1,1,1,4]
[1,1,1,1,3]
[1,1,1,1,6]
```

Later in the "Using lambdas with 'sack' steps" section we will revisit the topic of 'sacks' and lambda expressions and see other ways that we could have written the previous query.

4.12.8. Comparing properties and constants to the value of a sack

During a traversal you may want to incrementally modify a sack and then compare a vertex or edge property against the current contents of a sack. The examples below show ways of doing this. To keep things simple, initially we will use a sack that is initialized to a value of six and does not change. This demonstrates how to compare each airport's runway count against the value stored in the sack.

```
g.withSack(6).V().
  hasLabel('airport').as('a').
  where(gt('a')).
    by('runways').
    by(sack()).
  values('code')

DFW
ORD
```

As we mentioned in the "A warning that the 'path' and 'as' steps can also be memory intensive" section, using an 'as' step to remember a prior part of a traversal can be expensive in terms of memory usage. This will not be an issue with small graphs such as `air-routes`, but can be problematic when working with larger graphs. The query can be rewritten without the use of an 'as' step as shown below.

```
g.withSack(6).V().
  hasLabel('airport').
  filter(project('a','b').
           by('runways').
           by(sack()).
```

```
          where('a', gt('b'))).
  values('code')

DFW
ORD
```

Obviously, while these examples demonstrate the concept, in reality we have not done anything that truly warrants use of a 'sack' step so far. The query could easily have been written as shown below. However, hopefully this gives you some basic building blocks that enable sack and property values to be compared.

```
g.V().has('airport','runways',gt(6)).values('code')

DFW
ORD
```

Let's change our query to make the use of a sack more interesting. The query below starts at Santa Fe (SAF) and traverses outgoing edges until the distance traveled exceeds 10,000 miles. Only five results are returned.

```
g.withSack(0).V().
  has('code','SAF').
  repeat(outE().
         sack(sum).by('dist').
         inV()).
  until(sack().is(gt(10000))).
  limit(5).
  path().
    by('code').
    by('dist')
```

When we run the query, we get back a series of routes that stop as soon as we have traveled more than 10,000 miles.

```
[SAF,549,DFW,1751,YVR,7762,SYD]
[SAF,549,DFW,1751,YVR,8197,MEL]
[SAF,549,DFW,4736,LHR,6686,MNL]
[SAF,549,DFW,4736,LHR,5947,BKK]
[SAF,549,DFW,4736,LHR,5956,HND]
```

Finally, let's modify the query again to keep following routes starting at SAF, but this time adding an additional constraint that each route must be no more than 2,500 miles. We keep going until we have traveled more than 8,000 miles. This again demonstrates how we can use a sack to store a running total over the course of a graph traversal. This time, the path that was followed along with the total distance are combined using a 'union' step. We also added a 'simplePath' step to the query to make sure we do not revisit airports.

Splitting long queries over multiple lines makes them easier to read and understand.

You will notice that we have split the queries in this section over multiple lines to aid readability. As your queries become more complex, this becomes more important.

```
g.withSack(0).
  V().
  has('code','SAF').
  repeat(outE().
         has('dist',lte(2500)).
         sack(sum).by('dist').
         inV().
         simplePath()).
  until(sack().is(gt(8000))).
  limit(5).
  local(union(path().
                by('code').
                by('dist').
              unfold(),
              sack()).
        fold())
```

The results from running the modified query are shown below.

```
[SAF,708,LAX,2470,JFK,2410,SEA,2490,BOS,8078]
[SAF,708,LAX,2470,JFK,2440,SAN,2425,EWR,8043]
[SAF,708,LAX,2470,JFK,2440,SAN,2454,YUL,8072]
[SAF,708,LAX,2470,JFK,2460,LGB,2460,ATL,8098]
[SAF,708,LAX,2470,JFK,2447,SNA,2482,ISP,8107]
```

4.13. Using Lambda functions

Gremlin allows you to include a code fragment (sometimes called a Lambda function or a closure) as part of a query. This is typically done as part of a 'filter', 'map,' or 'sideEffect' step, but there are other places where you will find this concept used, such as when working with 'sacks'. This technique provides a lot of additional flexibility in how queries can be written. However, care should be used. When processing your query, Gremlin will try to optimize it as best as it can. For regular traversal steps such as 'out' and 'has' Gremlin will do this optimization for you. However, for closures (code inside braces '{}') Gremlin cannot do this and will just pass the closure on to the underlying runtime. With people just getting started with Gremlin, there is a great temptation to overuse in-line code. This is a natural thing to want to do as for programmers it feels like the programming they are used to. However, there is often, if not always, a pure Gremlin traversal step that can be used to do what is needed. Of course, with all rules there are exceptions.

For reasons of security, not all graph databases, especially those that are managed

> as hosted services, allow lambdas to be included in queries. You should always check the documentation for the graph database that you are using.

By way of a very simple example, the code below declares a variable '"c"' and initializes it to zero. The Gremlin query that follows then adds one to '"c"' each time it finds an airport vertex located in Oregon. We then use a 'println' to display the updated value for '"c"'.

```
c = 0
g.V().has('region','US-OR').sideEffect{ c += 1 }.values('code').fold()

println "I found ${c} airports in Oregon"
```

When we run our query, here is what comes back.

```
[PDX,EUG,LMT,MFR,OTH,RDM,PDT]
I found 7 airports in Oregon
```

Of course, in reality we would probably just use a 'count' when counting things or put them into a list and look at the size of the list returned, but the above example gives a nice, and hopefully easy to understand, example of a closure being used as part of a 'sideEffect' step.

For completeness, here is the query re-written a couple of different ways without the use of a 'sideEffect' or closures. If all we wanted was the count, we could do this, of course.

```
num = g.V().has('region','US-OR').values('code').count().next()
println "I found ${num} airports in Oregon"

I found 7 airports in Oregon
```

If we wanted to save a list of the airport codes found and also count them, we could do this.

```
oregon = []
g.V().has('region','US-OR').values('code').fill(oregon);[]
println oregon
println "I found ${oregon.size} airports in Oregon"
```

Here is the output, this time with the airport codes and the count.

```
[PDX, EUG, LMT, MFR, OTH, RDM, PDT]
I found 7 airports in Oregon
```

Here is another example of a closure being used where a 'has' step could and should have been used instead.

```
// What airports are located in London?
g.V().hasLabel('airport').filter{it.get().property('city').value() =="London"}
```

Here is the same query just using the 'has' step. This is a case where we should not be using a lambda function as Gremlin can handle this just fine all by itself.

```
// What airports are located in London?
g.V().hasLabel('airport').has('city','London')
```

We think you will agree that the second version is a lot simpler to read and enables Gremlin to do its thing.

Here is one more example of a query that contains a 'sideEffect' step. The main part of the query finds airports with 6 runways and counts them. That result will still be returned, but the side effect will also cause the codes of those airports to also be printed. This is a bit of a contrived example, but it shows how 'sideEffect' behaves when combined with a closure.

```
// Example of the 'sideEffect' step
g.V().has('runways',6).sideEffect{print it.get().values('code').next()+" "}.count()
```

The moral here is, avoiding closures unless you can genuinely find no other way to achieve what you need. It is fair to observe that sometimes coming up with a closure to do what you want is easier than figuring out the pure Gremlin way of doing it. However, using just Gremlin steps is still the recommended path to take. Lambda functions in general are discouraged.

```
g.V().hasLabel('airport').as('a').values('desc').
     filter{it.toString().contains('F.')}.select('a').
     local(values('code','desc').fold())

[JFK,New York John F. Kennedy International Airport]
[BDA,Bermuda, L.F. Wade International International Airport]
[SLU,George F. L. Charles Airport]
[EUX,F. D. Roosevelt Airport]
```

4.13.1. Introducing the 'Map' step

The 'map' step will be familiar to users of programming languages such as Ruby, Python, or indeed Groovy. It is often useful to be able to take a set of results, or in the case of Gremlin, the current state of a graph traversal, and modify it in some way before passing on those results to the next part of the traversal. This is what the 'map' step allows us to do.

The 'map' step can accept a traversal or a closure as input. The results of the traversal or closure will be passed on to the next step in the overall traversal. Below is a simple example of a 'map' step

using a traversal to modify what is passed on.

```
g.V().hasLabel('airport').limit(10).map(properties('city'))
```

When this query is run, the output returned is the selected vertex properties for each of the 10 airports that were selected.

```
vp[city->Atlanta]
vp[city->Anchorage]
vp[city->Austin]
vp[city->Nashville]
vp[city->Boston]
vp[city->Baltimore]
vp[city->Washington D.C.]
vp[city->Dallas]
vp[city->Fort Lauderdale]
vp[city->Washington D.C.]
```

We could go one step further and have the 'map' step produce a key and value map for us.

```
g.V().hasLabel('airport').limit(10).
      map(properties('city').group().by(key()).by(value()))

[city:Atlanta]
[city:Anchorage]
[city:Austin]
[city:Nashville]
[city:Boston]
[city:Baltimore]
[city:Washington D.C.]
[city:Dallas]
[city:Fort Lauderdale]
[city:Washington D.C.]
```

As we mentioned above, in many cases, a 'map' step is used in the middle of a query to change what is passed on to the next step. The example below takes the output from the 'map' step and sorts the results in descending order based on the city names. Obviously, there are simpler ways we could write this query, but this demonstrates what 'map' does quite well.

```
g.V().hasLabel('airport').limit(10).
      map(properties('city').group().by(key()).by(value())).
      unfold().order().by(values,asc)
```

Here is the output from running the query showing the sorted city names.

```
city=Anchorage
city=Atlanta
city=Austin
city=Baltimore
city=Boston
city=Dallas
city=Fort Lauderdale
city=Nashville
city=Washington D.C.
city=Washington D.C.
```

In some cases there are other steps, such as the 'values' step that can be used as a shorthand form of a 'map' step. The following two queries yield the same results, for example. There is no need to write an explicit 'map' step when a shorthand form exists.

```
g.V().hasLabel('airport').limit(10).map(values('city')).fold()

[Atlanta,Anchorage,Austin,Nashville,Boston,Baltimore,Washington D.C.,Dallas,Fort
Lauderdale,Washington D.C.]

g.V().hasLabel('airport').limit(10).values('city').fold()

[Atlanta,Anchorage,Austin,Nashville,Boston,Baltimore,Washington D.C.,Dallas,Fort
Lauderdale,Washington D.C.]
```

Now let's look at an example where a lambda function (closure) is used. First, take a look at the query below. It simply returns us a list containing the IDs of the first ten airports in the graph.

```
g.V().hasLabel('airport').limit(10).id().fold()

[1,2,3,4,5,6,7,8,9,10]
```

Imagine we wanted to write a query, similar to the one above, that will modify each of those IDs returned in the query above by adding one to each. Take a look at the query below. We have introduced a 'map' step. The 'map' step takes as a parameter a closure (or lambda) function telling it how we want it to operate on the values flowing in to it. The 'it' is Groovy syntax for '"the thing that came in"' (in this case a traversal). The 'get' is needed to gain access to the current vertex and its properties. Lastly, we get the 'id' of the vertex and add one to it. The modified values are then passed on to the next step of the traversal where they are made into a list by the 'fold' step.

```
g.V().hasLabel('airport').limit(10).map{it.get().id() + 1}.fold()

[2,3,4,5,6,7,8,9,10,11]
```

This is an area where Groovy and Java have a similar but different syntax. If you

> wanted to use the query above in a Java program, you would need to use the Java lambda function syntax.

What is nice about the 'map' step is that it allows us to do within the query itself what we would otherwise have to do after the query was over using a 'for each' type of loop construct.

One other thing to note about the 'map' step is that the closure provided can have multiple steps, separated by semicolons. The following query demonstrates this.

```
g.V().hasLabel('airport').limit(10).map{a=1;b=2;c=a+b;it.get().id() + c}.fold()

[4,5,6,7,8,9,10,11,12,13]
```

Note that only the value from the last expression in the closure is returned from the 'map'. So in the example above the result of 'c=a+b', 3, is added to each ID.

As we have already seen, there are often multiple ways to achieve the same result when working with Gremlin. It is also true that some ways are almost always better than others in terms of performance or some other metric. In the "Introducing 'sack' as a way to store values" we used a 'sack' step to add one to a set of results as follows.

```
g.withSack(1).V().has('code','SAF').out().
              sack(sum).by('runways').sack().fold()

[8,5,4,7]
```

We could achieve the same result using a 'map' step. However, doing so introduces the need to use a closure which the version using 'sack' avoids. Avoiding unnecessary use of closures is a Gremlin best practice.

```
g.V().has('code','SAF').out().values('runways').map{it.get() + 1}.fold()

[8,5,4,7]
```

In the next section we will take a look at some other cases where lambda expressions are very useful as well as a few more examples of the 'map' step being used.

4.13.2. Using lambdas with 'sack' steps

In the "Using 'addAll' and lists with a 'sack'" section we introduced ways in which 'sack' steps could operate on lists. Now that we know a bit more about the 'map' step and how Gremlin can take advantage of Groovy closures (lambda functions) we can explore some additional ways of working with sacks.

When we looked at 'sack' steps and lists earlier, we used the query below as one of the examples.

```
g.withSack{[1,1,1,1]}{it.clone()}V().has('code','SAF').
       out().values('runways').local(fold().sack(addAll)).sack()

[1,1,1,1,7]
[1,1,1,1,4]
[1,1,1,1,3]
[1,1,1,1,6]
```

Now that we have looked at 'map' steps, another way we could write the query, not necessarily the best way, but it does illustrate use of a 'map', is shown below.

```
g.withSack{[1,1,1,1]}{it.clone()}V().has('code','SAF').out().
     values('runways').map{x->[x]}.sack(addAll).sack()

[1,1,1,1,7]
[1,1,1,1,4]
[1,1,1,1,3]
[1,1,1,1,6]
```

Just for completeness, here is what would happen if we ran the same query, without the 'clone' (or split) being used.

```
g.withSack([1,1,1,1]).V().has('code','SAF').out().
     values('runways').map{x->[x]}.sack(addAll).sack()

[1,1,1,1,7]
[1,1,1,1,7,4]
[1,1,1,1,7,4,3]
[1,1,1,1,7,4,3,6]
```

A different way we could write the query, and this is something that we have not yet examined in this book is to use a closure directly with the 'sack' step itself.

```
 g.withSack([1,1,1,1]).V().has('code','SAF').out().
      values('runways').sack{a,v->a+=[v]}.sack()

[1,1,1,1,7]
[1,1,1,1,4]
[1,1,1,1,3]
[1,1,1,1,6]
```

We could also initialize the sack with a map '"[:]"' instead of a list and use a lambda function to manipulate it as shown below.

```
g.withSack{[:]}{it.clone()}.V().has('code','SAF').out().
```

```
      sack {m,v->m[v.values('code').next()]=v.values('runways').next();m}.sack()

[DFW:7]
[LAX:4]
[PHX:3]
[DEN:6]
```

Just to show what happens, here is the same query with the 'clone' step removed.

```
g.withSack([:]).V().has('code','SAF').out().
      sack {m,v->m[v.values('code').next()]=v.values('runways').next();m}.sack()

[DFW:7]
[DFW:7,LAX:4]
[DFW:7,LAX:4,PHX:3]
[DFW:7,LAX:4,PHX:3,DEN:6]
```

If all we wanted back was a single list of key value pairs, such as the one in the last line of the output above, we can write a query to do that. One way we could do it is to use a 'map' and not use a 'sack' step at all as shown below.

```
g.V().has('code','SAF').out().
      map{m=[:];m[it.get().values('code').next()]=
      it.get().values('runways').next();m}.unfold().fold()

[DFW=7,LAX=4,PHX=3,DEN=6]
```

However, doing it using a 'sack' feels cleaner in our view, as shown below. Note that in this case what is returned is a Java LinkedHashMap data structure, whereas the previous query generated an ArrayList of LinkedHashMap.Entry objects. Also, it is worth noting that all we had to do to get the result that we wanted in this case was to add a 'fold' step between the 'sack' steps.

```
g.withSack([:]).V().has('code','SAF').out().
                    sack{m,v -> m[v.value('code')]=v.values('runways').next()}.
                    fold().sack()

[DFW:7,LAX:4,PHX:3,DEN:6]
```

There are other ways that you might choose to write queries like these, that avoid the use of closures altogether, but hopefully these examples show some interesting ways that closures can be combined with 'sack' steps.

4.13.3. Introducing the 'flatMap' step

There are a set of fundamental steps that the other Gremlin query language steps build upon. One of those is 'map'. Another, that we have not looked at so far in this book, is 'flatMap'. The two steps

are similar but have a fundamental difference that may not be obvious from the examples we have looked at so far. The key difference is as follows. If a 'map' step receives multiple inputs, it only passes on the first one it received to the next step, whereas a 'flatMap' step passes them all on. Let's look at a couple of examples that demonstrate this and then look at how we can take advantage of this in practice.

The example below shows what happens when an 'out' step is used inside a 'map' step. Only the first vertex that was encountered is returned.

```
g.V().has('code','SAF').map(out())

v[8]
```

If we do the same experiment but using a 'flatMap' step, you can see that all the vertices are returned.

```
g.V().has('code','SAF').flatMap(out())

v[8]
v[13]
v[20]
v[31]
```

Where a flatMap can be useful is in a case like the one below. The example is a bit contrived, but there are situations where being able to do things like this come in quite handy. Take a look at the query below. It starts off by finding the AUS vertex. Next it traverses all the outgoing edges, finds the vertices at the end of those edges, and returns all the paths traveled. Notice that both the city names and the edge are included in the results.

```
g.V().has('code','AUS').outE().inV().
      path().by('city').by().limit(3)

[Austin,e[3840][3-route->45],Philadelphia]
[Austin,e[5376][3-route->149],Portland]
[Austin,e[3841][3-route->46],Detroit]
```

Let's imagine we have a good reason for needing to look at the edge but that we don't want the edge to be part of the result. What we can do is put the 'outE().inV()' part of the traversal inside a 'flatMap'. As we know from our tests above, a 'flatMap' will return all the results passed into it, which in this case will be the incoming vertices connected to the edges. When we now perform the 'path' step, the edge details are no longer part of the path because they were essentially removed from the path by the 'flatMap' step. This is a useful pattern to be aware of as it can come in handy in some cases.

```
// Hide the edge from the path!
g.V().has('code','AUS').flatMap(outE().inV()).
```

```
      path().by('city').limit(3)

[Austin,Philadelphia]
[Austin,Portland]
[Austin,Detroit]
```

We cannot use a 'map' step to achieve the same result as it will only return the first path as shown below.

```
g.V().has('code','AUS').map(outE().inV()).
      path().by('city').limit(3)

[Austin,Philadelphia]
```

Of course, if all we really wanted was the result from the prior query, we could just do this.

```
g.V().has('code','AUS').out().path().by('city').limit(3)

[Austin,Philadelphia]
[Austin,Portland]
[Austin,Detroit]
```

Lastly, here is one more case where a 'flatMap' can be useful. In the example below we wanted to check the property on an edge as part of a 'repeat' step but not have the edge itself be included in the resultant paths. The query uses the route distance to filter out routes between airports that are shorter than 2,000 miles.

```
g.V().has('code','AUS').
      repeat(flatMap(outE().has('dist',gt(2000)).inV())).
      times(2).
      path().
      limit(5)
```

When the query is run, the results returned only include the vertices. Note that once again, the 'flatMap' step differs from the 'map' step in that for each path being explored, it allows the last result generated, in this case the result of the 'inV' step, to pass to the next step in the traversal.

```
[v[3],v[37],v[66]]
[v[3],v[37],v[67]]
[v[3],v[37],v[71]]
[v[3],v[37],v[105]]
[v[3],v[37],v[122]]
```

The query below has the 'flatMap' step removed.

```
g.V().has('code','AUS').
      repeat(outE().has('dist',gt(2000)).inV()).
      times(2).
      path().
      limit(5)
```

When run, this time the results do indeed include the edges as well as the vertices.

```
[v[3],e[3832][3-route->37],v[37],e[8448][37-route->66],v[66]]
[v[3],e[3832][3-route->37],v[37],e[8449][37-route->67],v[67]]
[v[3],e[3832][3-route->37],v[37],e[8450][37-route->71],v[71]]
[v[3],e[3832][3-route->37],v[37],e[8451][37-route->105],v[105]]
[v[3],e[3832][3-route->37],v[37],e[8452][37-route->122],v[122]]
```

4.14. Turning graphs into trees

TinkerPop defines a Tree API, but it is not that well fleshed out and has not been updated in a long time. The 'tree' step allows you to create a tree from part of a graph using a Gremlin traversal. The example below creates a tree, of depth 3, where the Austin (AUS) vertex is the root of the tree. The next level of the tree is all vertices directly connected to AUS. The third level is made up of all the vertices connected by routes to the vertices in the previous level.

```
//Generate a tree the AUS vertex and its neighbors and their neighbors
tree = g.V().has('code','AUS').
             repeat(out()).
               times(2).
             tree().by('code').next()
```

The object returned to our variable 'tree' will be an instance of the 'org.apache.tinkerpop.gremlin.process.traversal.step.util.Tree' class. That class provides a set of methods that can be used when working with a Tree.

```
// Look at part of the tree directly
tree['AUS']['DFW']

// You can also use the TinkerPop Tree API to work with the tree
tree.getLeafObjects()
tree.getObjectsAtDepth(1)
tree.getObjectsAtDepth(1)
tree.getObjectsAtDepth(2)
```

We will see the Tree API used again in the "Modeling an ordered binary tree as a graph" section later on.

4.15. Creating a subgraph

Using Gremlin, you can create a subgraph, which is a subset of the vertices and edges in a larger graph you are working with. Once created, to work with a subgraph, you create a traversal source object specific to that new graph and distinct from the one being used to process the main graph. Subgraphs are created using the 'subgraph' traversal step. Note that 'subgraph' works with edges (not vertices) and adds both those edges and the vertices that they connect with to the new subgraph being created.

You can find all the sample data in the book's GitHub repository. https://github.com/krlawrence/graph/tree/main/sample-data

Let's start with a simple example. One of the sample data sets shipped with this book is a small version of the main "air-routes" graph called `air-routes-small.graphml`. It contains routes between just the first 46 airports in the full graph. We can quite easily write a query to generate the subgraph representing the flights between those airports by extracting the edges and vertices from the full graph. Take a look at the query below. First, we find all the vertices that have an ID between 1 and 46 inclusive. Then we find all of their outgoing edges but filter out any that do not also end up at an airport within the same ID range of 1 through 46. Lastly, we use a 'subgraph' step to add those edges and vertices to a new subgraph. Note that the 'subgraph' step has to be given a label. In this case we just used "a". This allows us, should we need to, to add to a new subgraph from more than one part of a traversal. In this case we have no more to add, so a 'cap' step is used to complete the creating of the 'subgraph'. The variable 'subg' will now contain a reference to the newly created graph.

```
subg=g.V(1..46).outE().
       filter(inV().hasId(within(1L..46L))).
       subgraph('a').cap('a').next()
```

Note that when we run the query, Gremlin shows that we created a new TinkerGraph containing 46 vertices and 1326 edges.

```
tinkergraph[vertices:46 edges:1391]
```

Now that the subgraph is created, we need to create a traversal source object for it so that we can issue queries against it. Gremlin shows us details of the new traversal source object once it has been created.

```
sgt = subg.traversal()

graphtraversalsource[tinkergraph[vertices:46 edges:1391], standard]
```

Now that we have a traversal source object for our newly created subgraph, we can run some queries against it.

```
// What airports are in the subgraph?
sgt.V().values('code').fold()

[ATL,ANC,AUS,BNA,BOS,BWI,DCA,DFW,FLL,IAD,IAH,JFK,LAX,LGA,MCO,MIA,MSP,ORD,PBI,PHX,RDU,S
EA,SFO,SJC,TPA,SAN,LGB,SNA,SLC,LAS,DEN,HPN,SAT,MSY,EWR,CID,HNL,HOU,ELP,SJU,CLE,OAK,TUS
,SAF,PHL,DTW]

// How many of the 46 airports can you fly to from LAX?
sgt.V().has('code','LAX').out().count()

40
```

Here are some more examples of working with subgraphs. The query below will create a subgraph of all vertices and edges directly connected to the Austin (AUS) vertex. Note that using 'bothE' means we get incoming and outgoing edges. In these examples the more meaningful label 'subGraph' is used to label the subgraph being created.

```
subg = g.V().has('code','AUS').bothE().
            subgraph('subGraph').cap('subGraph').next()
```

If we only wanted the outgoing routes from Austin, we could change the query to just use an 'outE' step instead.

```
subg = g.V().has('code','AUS').outE('route').
            subgraph('subGraph').cap('subGraph').next()
```

The next example is a little more sophisticated. It will create a subgraph starting with the Austin vertex but this time going out two hops. We achieve this using a 'repeat' step.

```
subg = g.V().has('code','AUS').
       repeat(bothE().subgraph('subGraph').outV()).times(2).
       cap('subGraph').next()

[tinkergraph[vertices:1429 edges:14345]
```

As before, we can now work with the newly created subgraph.

```
// Get a traversal source object so that we can traverse
// the newly created subgraph.
sgt = subg.traversal()

// What sort of vertices ended up in the subgraph?
sgt.V().groupCount().by(label)
```

```
[continent:2,country:8,airport:1419]
```

Here is a more complicated example. The query will create a subgraph just of airports and routes that are inside Europe. Effectively, this will make the Europe only version of the air-routes main graph. At first glance, this query looks a bit overwhelming, but if you read it slowly and look at each step, you should be able to make sense of what it is doing. It is quite a bit more sophisticated than the previous examples in that as well as extracting the routes and airports into the subgraph, it extracts all the relevant countries and continents as well. Note that this query also uses multiple 'subgraph' steps.

```
// Create a subgraph only of airports in Europe and routes between those airports
subg = g.V().hasLabel('continent').has('code','EU').
             outE('contains').subgraph('eu-air-routes').inV().as('a').
             inE('contains').subgraph('eu-air-routes').
             outV().hasLabel('country').
             select('a').outE().as('r').
             inV().hasLabel('airport').in().hasLabel('continent').
             has('code','EU').select('r').subgraph('eu-air-routes').
             cap('eu-air-routes').next()

tinkergraph[vertices:652 edges:16150]
```

As before, we can now work with the newly created subgraph.

```
// Create a traversal source object for the subgraph
sgt = subg.traversal()

// How many routes are there in the subgraph?
sgt.E().hasLabel('route').count()

14940

// What sort of vertices ended up in the subgraph?
sgt.V().groupCount().by(label)

[continent:1,country:46,airport:605]
```

The following query uses the new Europe only subgraph to find out where we can get to from London Heathrow (LHR) within Europe.

```
sgt.V().has('code','LHR').out().values('code').fold()

[CGN,GOT,VCE,SNN,OSL,ARN,EDI,GLA,DME,SVO,ORY,NCE,MXP,ATH,ZAG,BUD,BIO,IBZ,WAW,MLA,SOF,B
EG,IST,HAM,STR,PSA,TRN,BLQ,NTE,CPH,LUX,DUS,LIS,GIB,KEF,PMI,AGP,LBA,ABZ,NCL,GCI,BSL,SVG
,BGO,TLL,ORK,VKO,SPU,BHD,HAJ,LIN,LYS,MRS,OTP,RTM,FAO,JMK,JTR,KBP,RJK,TLS,LED,OPO,LCG,I
NN,INV,SZG,BLL,IOM,NQY,KRK,TFS,LEI,MJV,VLC,BIA,MPL,EFL,PVK,LJU,PUY,BDS,FSC,BER,CDG,FRA
```

```
,HEL,DUB,FCO,AMS,PRG,BCN,MAD,VIE,ZRH,GVA,BRU,MUC,MAN]
```

The ability to create a subgraph from a larger graph is a very powerful feature that Gremlin provides for us. If you have a large graph but only want to work with a part of it, it is nice to be able to create a subgraph from it and perhaps even work with that subgraph locally in memory while running some queries.

4.16. Working with GraphML and GraphSON

Apache TinkerPop supports the loading and saving of entire graphs using GraphML and GraphSON. GraphML is a broadly supported XML standard that can be used to represent entire graphs. GraphSON is a JSON format defined as part of the Apache TinkerPop project that also allows whole graphs to be represented. It is also possible to use GraphSON to represent the results of a Gremlin graph query in JSON format. In this section we will take a look at all of these topics. The whole subject of using GraphML and GraphSON will be revisited a few more times later in the book. Knowledge of GraphSON becomes especially important once you start working with the Gremlin Server. That topic will be covered in detail as part of the "INTRODUCING GREMLIN SERVER" section quite a bit later. Both GraphML and GraphSON are covered in detail as part of the "COMMON GRAPH SERIALIZATION FORMATS" section.

The official Apache TinkerPop documentation includes some good coverage of this topic. That documentation can be found at https://tinkerpop.apache.org/docs/current/reference/#_gremlin_i_o and in https://tinkerpop.apache.org/docs/current/dev/io/

There are currently three versions of GraphSON, and each can include embedded type information or to rely on standard JSON types. The original 1.0 version has a complicated embedded type format that is challenging to parse and is typically not used anymore. Version 2.0 introduces a new embedded type format that is much more compact and easier to parse, and 3.0 added a few additional types to the format. The default format, unless explicitly specified, is currently GraphSON 3.0.

4.16.1. Saving (serializing) a graph as GraphML (XML) or GraphSON (JSON)

Using TinkerPop, you can save a graph either in GraphML or GraphSON format. GraphML is an industry standard XML format for describing a graph and is recognized by many other applications such as Gephi. GraphSON was defined as part of the TinkerPop project and is less broadly supported outside TinkerPop enabled tools and graphs. However, whereas GraphML is considered lossy for some graphs (it does not support all the data types and structures used by TinkerPop), GraphSON is not considered so when configured to include embedded types.

Saving a graph to a GraphML file can be done using the following Gremlin expression. You might want to try it on one of your graphs and look at the output generated. You can also take a look at the `air-routes.graphml` file distributed with this book if you want to look at a well laid out (for human readability) GraphML file. Bear in mind that by default, TinkerPop will save your graph in a way that is not easily human-readable without using a code beautifier first. Most modern text editors can also beautify XML files well.

Gremlin has an `io()` step which provides a way to write a graph to file:

```
// Using the .xml (or .graphml) file extension will save the graph as GraphML
g.io('my-graph.xml').write()
```

TinkerPop offers two different JSON packaging options. These are not to be confused with the three different syntax versions. The default encoding option stores each vertex in a graph and all of its edges as a single JSON document. This is repeated for every vertex in the graph. This is essentially what is known as 'adjacency list' format. If you serialize a graph to file using this method and look at the file afterward, you will see that each line (which could be very wide) is a standalone JSON object.

The second variant is referred to as a 'wrapped adjacency list' format. In this flavor all the vertices and edges are stored in a single large JSON object inside an enclosing 'vertices' object.

The GraphSON file generated will not be very human-readable without doing pretty printing on it using something like the Python JSON tool ('python -m json.tool my-graph.json') or using a text editor that can beautify JSON.

The Gremlin line below will create a file containing the 'adjacency list' form of GraphSON.

```
// Using the .json file extension will save the graph as unwrapped JSON
g.io("my-graph.json").write()
```

The following will create a file containing the 'wrapped adjacency list' form of GraphSON.

```
// Create a single (wrapped) JSON file
writer = GraphSONWriter.build().wrapAdjacencyList(true).create()
g.io("my-graph.json").with(IO.writer, writer).write()
```

If you are ingesting large amounts of data into a TinkerPop enabled graph, the unwrapped flavor of GraphSON is probably a much better choice than GraphML or wrapped GraphSON. Using this format, you can stream data into a graph one vertex at a time rather than having to try and send the entire graph as a potentially huge JSON file all in one go.

Note that by default your graph will be saved using the GraphSON 3.0 format. Should you wish to use one of the older formats, you can still do so but will need to explicitly specify which version you want Gremlin to generate. In the example below the graph is saved using GraphSON 1.0 format. Doing this requires the creation and use of a mapper that will produce the format we need.

```
mapper = GraphSONMapper.build().version(GraphSONVersion.V1_0).create();
writer = GraphSONWriter.build().wrapAdjacencyList(true).mapper(mapper).create()
g.io("my-graph.json").with(IO.writer, writer).write()
```

If you want to learn more about the specifics of the GraphML and GraphSON formats, they are covered in detail near the end of this book in the "COMMON GRAPH SERIALIZATION FORMATS" section.

4.16.2. Loading a graph stored as GraphML (XML) or GraphSON (JSON)

In section 2 we saw how to load the `air-routes.graphml` file. In case you skipped that part, let's do a quick recap on loading GraphML files and also look at loading a GraphSON JSON format file.

The only difference between loading a GraphML file or a GraphSON file is in the file name extension, '.xml' and '.json' respectively.

```
// Using the .xml file extension will read the graph as GraphML
g.io('my-graph.xml').read()

// Using the .json file extension will save the graph as unwrapped JSON
g.io("my-graph.json").write()
```

4.16.3. Turning the results of a query into JSON

In the sections above we explored how to save and load an entire graph using GraphSON (JSON) or GraphML (XML). However, what we have not looked at so far are any ways to see query results expressed as JSON objects within the Gremlin Console. As we shall explore in the "INTRODUCING GREMLIN SERVER" section when you communicate with a Gremlin Server from an application or from the command line using a tool such as 'curl' and send queries to a graph over HTTP or WebSockets, the results are returned as JSON objects.

There is not an equivalent XML object mapping capability. This is because GraphML is designed to contain whole graphs and not query results.

When using the Gremlin Console, the results of queries are presented to us in a nice and fairly terse way, and we are not shown any JSON. Most of the time this is exactly what we want. However, if for any reason you want to see what the JSON for a query result looks like, it is possible to do just that. You can do this using the console and a TinkerGraph on your laptop. You do not need to be connected to a Gremlin Server to use the examples that we are about to present.

The first thing you need to do is create an instance of a GraphSON mapper that can be used to generate JSON for us from a query result. There are currently three versions of GraphSON, and each can include embedded type information or to rely on standard JSON types. The original 1.0 version has a complicated embedded type format that is challenging to parse and is typically not used anymore. Version 2.0 introduces a new embedded type format that is much more compact and easier to parse, and 3.0 added a few additional types to the format. The default format, unless explicitly specified, is currently GraphSON 3.0.

The example below creates a 'GraphSONMapper' object that will generate GraphSON 3.0 format JSON where the 'typeInfo(TypeInfo.NO_TYPES)' indicates that embedded types will be omitted.

```
jsonMapper = GraphSONMapper.build().
                version(GraphSONVersion.V3_0).
                typeInfo(TypeInfo.NO_TYPES).
                create().
                createMapper()
```

Next let's run a simple query that finds the vertex representing the Los Angeles (LAX) airport.

```
lax = g.V().has('code','LAX').next()
```

The JSON mapper that we just created can now be used to display the query results as JSON

```
jsonMapper.writeValueAsString(lax)
```

Here is the JSON that was generated. We have pretty printed it a bit to make it more readable. Notice that the JSON includes the ID values for every property.

```
{
    "id": "13",
    "label": "airport",
    "type": "vertex",
    "properties": {
        "country": [{ "id": 149, "value": "US" }],
        "code": [{ "id": 150, "value": "LAX" }],
        "longest": [{ "id": 151, "value": 12091 }],
        "city": [{ "id": 152, "value": "Los Angeles" }],
        "lon": [{ "id": 155, "value": -118.4079971 }],
        "type": [{ "id": 156, "value": "airport" }],
        "elev": [{ "id": 153, "value": 127 }],
        "icao": [{ "id": 154, "value": "KLAX" }],
        "region": [{ "id": 157, "value": "US-CA" }],
        "runways": [{ "id": 158, "value": 4 }],
        "lat": [{ "id": 159, "value": 33.94250107 }],
        "desc": [{ "id": 160, "value": "Los Angeles International Airport" }]
    }
}
```

Let's create another 'GraphSONMapper' instance. We will again be generating GraphSON 3.0, but this time we'll be including the embedded types. Including the types is the default behavior, but we'll be explicit in defining the configuration with 'typeInfo(TypeInfo.PARTIAL_TYPES)'.

```
jsonMapper = GraphSONMapper.build().
                version(GraphSONVersion.V3_0).
                typeInfo(TypeInfo.PARTIAL_TYPES).
                create().
```

```
                createMapper()
```

As there is a lot more information contained in the GraphSON with embedded types, we decided to use a somewhat simpler query and just generate a 'valueMap' for a few of the properties contained in the LAX vertex to avoid having to show too much output!

```
lax = g.V().has('code','LAX').valueMap(true,'code','city').next()
```

As before, we can use the mapper to generate the JSON.

```
jsonMapper.writeValueAsString(lax)
```

This time, as you can see, the JSON returned contains a lot more information. The key thing to notice is the presence of all the '@type' and '@value' keys.

```
{
    "@type": "g:Map",
    "@value": [{
        "@type": "g:T",
        "@value": "id"
    }, "13", {
        "@type": "g:T",
        "@value": "label"
    }, "airport", "code", {
        "@type": "g:List",
        "@value": ["LAX"]
    }, "city", {
        "@type": "g:List",
        "@value": ["Los Angeles"]
    }]
}
```

As we mentioned, the subject of JSON will come up again when we start looking at working with a Gremlin Server. This section has hopefully shown you how to save and load an entire graph as either GraphML or GraphSON and should you so desire to see the results of your queries as JSON. Remember that if you do save an entire graph as JSON, unless you specify otherwise, the default format is GraphSON 3.0 with embedded types.

4.17. Using null in Gremlin

Gremlin does allow null values to move through a traversal. Whether or not nulls are meaningful to you with Gremlin tends to depend on the graph database that you are using and whether or not it supports storing null values. If the graph does not support that capability, then you won't encounter a null in your traversal pipeline or your results unless you introduce the null value yourself. You might do that by way of a side effect or some other Gremlin step that allows you to

supply a value to the stream. The following examples demonstrate a few ways that you might choose to do this:

```
g.inject(null)

null

g.V().limit(10).coalesce(has('elev',gt(500)), constant(null)).fold()

[null,v[1],null,v[3],v[4],null,null,null,v[8],null]
```

The prior examples don't showcase any particular common use case, and unless the graph itself supports storing null values, there is little foundation for injecting them in this fashion. For graphs that do support storing null, such as TinkerGraph, you can treat nulls in much the same manner that you do other values in Gremlin.

TinkerGraph is not configured to support null storage by default. You must provide set the 'gremlin.tinkergraph.allowNullPropertyValues' to true in its configuration to enable it.

Before we look too closely at how null is used in a graph that supports it, let's first take a look at what happens with null for graphs that do not. In particular, we should look at the 'property' step.

```
g.addV('airport').property('code',null)

v[61394]

g.V().has('code',null)

// no result

g.V().hasNot('code')

v[61394]
```

For better portability, prefer 'drop' when removing properties.

In the prior example, Gremlin semantics expect that calls to 'property' with a null value assignment will result in the step being ignored if the property does not exist or removed if it does. The following demonstrates the latter:

```
g.addV('airport').property('code','XYZ')

v[61394]

g.V(61394).property('code',null)
```

```
v[61394]

g.V().hasNot('code')

v[61394]
```

Now that we've looked at graphs that don't support null, let's look at how Gremlin behaves for those that do:

```
g.addV('airport').property('code',null)

v[61394]

g.V().has('code',null)

v[61394]

g.V().has('code',within('IAD',null)).values('code')

null
IAD
```

As you can see in the prior example, the null value is being stored in and retrieved from the graph. By electing to use this feature, you now have the additional burden of accounting for null in your query. Gremlin steps tend to behave in null-safe ways as shown in the the following examples:

```
g.V().has('code',within('IAD',null)).values('code').substring(0,1).fold()

[null,I]

g.V().has('code',within('IAD',null)).values('code').length().fold()

[null,3]

g.V().has('code',within('IAD',null)).values('code').groupCount()

[null:1,IAD:1]

g.V().has('code',within('IAD',null)).values('code').order()

null
IAD

g.V().has('code',within('IAD',null)).values('code').order().by(desc)

IAD
null
```

The choice to use null is often made for you as many graphs do not support storing nulls at which point injecting them yourself into your queries doesn't tend to provide much added value. In addition, when the graph you are using does support it, you should take care in the choice to use it because it will reduce the portability of your application by limiting your graph choices as you will have to find another that also supports the feature. On the flip side, if you choose a graph that does not support null values and use the 'property('key',null)' syntax to remove properties, then keep in mind that this syntax will behave quite differently if you switch to a graph that suddenly supports storing nulls!

4.18. Understanding TraversalStrategies

When you write a Gremlin query and iterate it to get a result, the Gremlin execution engine will take a moment to examine the query itself to determine if any registered 'TraversalStrategy' implementations meet their criteria for application. If one or more do meet the criteria, then the strategy will modify the traversal according to its rules.

A `TraversalStrategy` can serve many kinds of functions but often serve one of four functions:

- Decoration – These strategies embed application-level features into traversal logic.
- Verification – This is a strategy that does checks to ensure that the traversal is legal for the executing engine.
- Optimization – These strategies provide a more efficient way to express traversal logic than the form originally written.
- Finalization – These strategies make final internal adjustments after all other strategies are executed to ensure the traversal is ready for execution.

TinkerPop automatically registers a number of optimization, verification, and finalization strategies by default. Graph database providers who implement TinkerPop will usually have some strategies of those types to automatically register as well. You would usually choose to add your own decoration strategies should your use case call for one. We will discuss that in further detail momentarily, but let's first consider a basic optimization strategy and what it does to provide a more concrete example.

Let's assume that you write the following Gremlin, which calculates the degree of each vertex it encounters:

```
g.V().map(both().count())
```

This Gremlin will get you the count you seek, but it is a bit inefficient because it chooses to count the adjacent vertices using 'both' when it could simply count incident edges with 'bothE' and achieve the same answer without requiring an extra traversal to the vertex on the other side. It would have been better to write:

```
g.V().map(bothE().count())
```

As you work more with Gremlin, you will come to think in terms of those efficiencies, but Gremlin has a 'TraversalStrategy' for that called 'AdjacentToIncidentStrategy' which will automatically rewrite the Gremlin you wrote in the first example to the one in the second when you execute it. Strategies like this one, and particularly ones registered automatically by the graph database you choose, can have a dramatic effect on the performance of your queries. Generally speaking, you don't need to know much about what these types of strategies are doing or feel the need to remove any of the pre-registered strategies.

While those types of strategies aren't critical for most users to understand, it is definitely worth learning about decorative and some verification strategies. These strategies offer actual features that can be automatically applied to a traversal that you write which will alter its behavior in ways that might be quite useful to you. In the following sections, we will examine some of the more useful user-oriented strategies provided by TinkerPop.

Some strategies cannot work in remote contexts. 'EventStrategy' and 'ElementIdStrategy' are two such strategies that will not work this way.

4.18.1. Verification strategies

As a quick reminder, verification strategies validate the contents of a traversal to ensure that it conforms with the strategies guidelines. If it does not, then it throws an exception rather than executing. The following strategies that TinkerPop provides tend to be useful:

- 'EdgeLabelVerificationStrategy' – ensures that labels are always used when using 'out', 'in,' or 'both' steps
- 'ReadOnlyStrategy' – ensures that the traversal contains no mutation steps that could modify the graph
- 'ReservedKeysVerificationStrategy' – ensures that certain strings are not used for property keys, treating them as reserved words

```
verificationStrategy = EdgeLabelVerificationStrategy.build().
                                                    throwException().create()
// results in VerificationException - as out() does not have a label specified
g.withStrategies(verificationStrategy).V(1).out().iterate();

verificationStrategy = ReadOnlyStrategy.instance();
// results in VerificationException since a mutation step, addV(), is used
g.withStrategies(verificationStrategy).addV('airport').iterate();

// by default ReservedKeysVerificationStrategy blocks use of "id" and "label"
// which are commonly mistaken with T.id and T.label in Gremlin
verificationStrategy = ReservedKeysVerificationStrategy.build().
                                                        throwException().create()
// results in VerificationException since the "id" property key was used
g.withStrategies(verificationStrategy).addV('airport').property("id",123).iterate();
```

4.18.2. PartitionStrategy

As its name suggests, 'PartitionStrategy' can be used to partition the graph into named groups that allow it to blind traversals from traveling to particular parts of the graph. The key advantage to using 'PartitionStrategy' is that it automatically handles the insertion of various filter and mutation steps that would otherwise be tedious to write and potentially easy to forget, leading to mistakes. Moreover, the entire partitioning abstraction encapsulated in a strategy means that your Gremlin remains more readable in your code as the partition logic isn't applied until traversal execution time.

For simplicity's sake, let's look at an example of "PartitionStrategy' using an empty graph and consider a multi-tenant scenario where there are two different users accessing the graph who should not be able to see the other's data.

```
graph = TinkerGraph.open()

// create two partitions, one for each tenant. the "partitionKey" refers to the
// property in the graph to use to store the partition value for tenant A or B.
// the "writePartition" is the value to assign to the "partitionKey" when writing
// to the graph and the "readPartition" is the set of values that the partition is
// allowed to see when reading from the graph.
tenantA = PartitionStrategy.build().partitionKey("_partition").
                                    writePartition("a").
                                    readPartitions("a").create()
tenantB = PartitionStrategy.build().partitionKey("_partition").
                                    writePartition("b").
                                    readPartitions("b").create()

// create two instances of "g" each using a different strategy
gA = traversal().with(graph).withStrategies(tenantA)
gB = traversal().with(graph).withStrategies(tenantB)

gA.addV() // this vertex has a property of {_partition:"a"}
gB.addV() // this vertex has a property of {_partition:"b"}
gB.addV() // this vertex has a property of {_partition:"b"}


gA.V().count()

1

gB.V().count()

2
```

Let's look at how you would have written this same code if you didn't have 'PartitionStrategy' and needed this sort of functionality.

```
graph = TinkerGraph.open()
```

```
g = traversal().with(graph)

g.addV().property('_partition', 'a')
g.addV().property('_partition', 'b')
g.addV().property('_partition', 'b')


g.V().has('_partition', 'a').count()

1

g.V().has('_partition', 'b').count()

2
```

As you can see in the above example, you would have to insert logic that was mostly irrelevant to what the traversal itself is doing. It isn't so hard in the example where you need to add a simple step or two, but consider a more complicated example, and you can quickly see how useful this strategy can be if you have this use case.

```
// if air-routes was partitioned and you used PartitionStrategy, you could write
// normal Gremlin like this
g.V().out('route')
  filter(bothE('route').count().is(lt(3))).
  union(both('route').has('code',within('AUS','TUS','YYZ')),
        bothE('route').has('dist', eq(100).otherV())).
  valueMap()

// but if you didn't have PartitionStrategy you'd have to write all the partitioning
// logic yourself in every query. you can see how much harder it is to read, requires
// more repetitive typing and might be error prone to write
g.V().has('_partition', 'a').
  out('route').has('_partition', 'a')
  filter(bothE('route').has('_partition', 'a').count().is(lt(3))).
  union(both('route').
          has('_partition', 'a').
          has('code',within('AUS','TUS','YYZ')),
        bothE('route').
          has('_partition', 'a').
          has('dist', eq(100).
          otherV().
          has('_partition', 'a'))).
  valueMap()
```

'PartitionStrategy' is not always a fit for every situation. Standard rules about Gremlin and graphs still apply. For instance, a vertex with millions of edges, physically still has millions of edges even when using this strategy where you

> expect to only use it to traverse a small fraction of those edges. The underlying graph database will really just be filtering those edges using the partition key, so your query will still be limited to the speed with which it can do that.

4.18.3. SeedStrategy

'SeedStrategy' is mostly helpful when writing tests. Certain Gremlin features aren't determinisitic in what they do, which can make it hard to write good assertions for tests. By using 'SeedStrategy', you can ensure that 'coin', 'sample,' and 'Order.shuffle' will all behave in a deterministic fashion.

```
seedStrategy = SeedStrategy.build().seed(999998L).create() // specify the seed to
reuse
g.withStrategies(seedStrategy).V().limit(10).values('code').
  fold().
  order(local).by(shuffle)

// repeated executions will always shuffle the same way to return the same result
[DFW,BOS,ANC,AUS,FLL,1.0,DCA,BNA,BWI,ATL]
```

4.18.4. SubgraphStrategy

'SubgraphStrategy' is quite similar to 'PartitionStrategy', and it would be worth reading PartitionStrategy first to understand the benefits that are discussed there as they are quite similar to the benefits gained here. As with 'PartitionStrategy', 'SubgraphStrategy' is defined to blind traversals from traveling to particularly defined portions of the graph. Unlike 'PartitionStrategy', which restricts those places by way of a single property key, i.e., the "partitionKey", this strategy allows you to define complex filtering rules using Gremlin itself to help define the subgraph that is available.

The basic idea for using SubgraphStrategy is to define filters for vertices, edges, or vertex properties to constrain the traversal paths. As previously mentioned, you define the filters with Gremlin.

```
// define a subgraph that describes "short flights" where you only traverse
// routes with distances of less than 40 miles
strategy = SubgraphStrategy.build().edges(__.has('dist',lt(40))).create()
g.withStrategies(strategy).V().out().count()

379

// extend the previous subgraph to include a subset of vertices, which now will limit
// queries to traverse edges under 40 miles and also among vertices in the specified
// set
strategy = SubgraphStrategy.build().
                vertices(__.has('code', within(['ADQ','OBU','SHG','AUK','KOT',
                                                'KWK','RSH','RSH','PQS','PQS',
                                                'KPN','IRC','AET','KWN','ORV',
                                                'KGX','VAK','SGY','ANV','HNS',
                                                'HNH','SVA','WMO','ELI','KSM',
```

```
                                                    'MNT','WBB','ANI','KYK','SKK']))).
                edges(__.has('dist',lt(40))).create()
g.withStrategies(strategy).V().out().count()

7
```

4.19. Analyzing the performance of your queries

Apache TinkerPop includes a class called 'TimeUtil' that provides methods that you can use to time how long your queries are taking to run. A second class called ProfileStep provides a way to get a more fine-grained analysis of where the time is spent during execution of a query. In this section, we are going to provide a few examples of how to use the methods provided to analyze the execution time of a few queries. We ran the tests on a laptop using the Gremlin Console with the 'air-routes' data loaded into an in-memory TinkerGraph.

4.19.1. Timing a query – introducing 'clock' and 'clockWithResult'

The 'TimeUtil' class provides several methods that can be used when working with time. We are just going to focus on two of them, namely, 'clock' and 'clockWithResult'. These methods allow you to have a query run one or more times while the execution time is tracked. After all iterations have completed, the average time the query took in milliseconds is returned. Note that, especially for longer queries, the time returned will not appear to be the same as if you tried to measure the same 'clock' procedure using a stopwatch. This is because these methods both perform a warm-up pass before doing the actual timing. The warmup simply consists of running the query one time before timing starts. This means that for a single timing iteration, the 'human perceived time' will be roughly double the time returned by the 'clock' analysis. Let's take a look at a few examples.

Below is a basic example of using a 'clock' step to measure the time it takes to find all airport vertices in the graph that are in China. Note that the clock step takes two parameters. The first is an integer telling it how many times to run the query. The second is a query to execute wrapped in braces '"{...}"'. In other words, a closure. In the example below, the query is only run once as a parameter of 1 is passed to the 'clock' method. From the result we can see that running this query one time on our laptop took a bit more than 1.3 milliseconds.

```
clock(1) {g.V().has('airport','country','CN').next()}

1.364199
```

To get a more accurate assessment of how long a query takes, it is sometimes a good idea to average out the time over multiple attempts. This should reduce the impact of any random system events from impacting your overall result. The example below repeats the previous query but measures the average time taken across 100 iterations. Note that, especially for longer running queries, the time returned will not be the same as if you tried to time this process yourself using a stopwatch. This is because the clock steps do a warm-up pass before doing the actual testing. The warm-up step basically runs the query one time before it starts timing anything. This means that for a single iteration the 'human perceived time' will be roughly double the time returned by the 'clock' analysis. You can see from the results that when averaged over 100 iterations, the time taken is a bit

less. You should not use these numbers as hard and fast answers but rather use them to get a sense of in general how long a query takes. If you repeated this test five times, you would definitely get five different results of a similar but not identical magnitude.

```
clock(100) {g.V().has('country','CN').next()}

0.70694013
```

Here is the same query run 1000 times just to verify that the timings stay more or less the same.

```
clock(1000) {g.V().has('country','CN').next()}

0.6931818670000001
```

We have seen the following query before in the "Shortest paths (between airports) - introducing 'repeat'" section. As you may recall, it finds 10 routes between Austin and Agra. Because there are not that many ways to get to Agra, this query requires a lot of graph traversals and so takes a while to complete. Let's time how long it takes to execute this query one time.

```
clock(1){
    g.V().has('airport','code','AUS').
          repeat(out()).until(has('code','AGR')).
          limit(10).path().by('code').toList()}

4643.605562
```

As you can see, it took quite a bit longer to run than our search for Chinese airports. In fact, it took well over 4 seconds to run. Let's run the same query 100 times and see what the average time taken looks like.

```
clock(100){
    g.V().has('airport','code','AUS').
          repeat(out()).until(has('code','AGR')).
          limit(10).path().by('code').toList()}

4816.6515936999995
```

So running the query 100 times gave a very similar result in this case. By way of an interesting observation, notice the difference in time taken if we start in Agra and look for routes to Austin. The average time over 100 iterations is less than 10 milliseconds. So we have uncovered an interesting possible optimization. Because there are a lot more ways to get to Austin, finding 10 routes did not take very long. This may not always be possible, but keep in mind as you model your graph and design your queries that where you start from can make a big difference!

```
clock(100){
```

```
   g.V().has('airport','code','AGR').
         repeat(out()).until(has('code','AUS')).
         limit(10).path().by('code').toList()}

9.068097369999998
```

You may have noticed that the 'clock' method only shows us the time that a query takes to run and does not show us the actual result of running the query. This is where the 'clockWithResult' method comes into play. We can reuse our 'airports in China' query, but this time notice that we also get the result of the query back.

```
clockWithResult(1) {g.V().has('country','CN').count().next()}

0.918276
209
```

If we run the 'Austin to Agra' query using 'clockWithResult', you can see that we do indeed get the routes back as well as the timing.

```
clockWithResult(1){
    g.V().has('airport','code','AUS').
          repeat(out()).until(has('code','AGR')).
          limit(10).path().by('code').toList()}

4266.584279

[[AUS,JFK,BOM,AGR],[AUS,YYZ,BOM,AGR],[AUS,LHR,BOM,AGR],[AUS,EWR,BOM,AGR],[AUS,FRA,BOM,
AGR],[AUS,AMS,BOM,AGR],[AUS,PHL,YYZ,BOM,AGR],[AUS,PHL,LHR,BOM,AGR],[AUS,PHL,CDG,BOM,AG
R],[AUS,PHL,FRA,BOM,AGR]]
```

Just for fun, the next four queries look for all the ways you can fly between Austin (AUS) and Los Angeles (LAX) with zero, one, two, or three stops. The query avoids any routes that revisit Austin and uses 'simplePath' to avoid repeating the same route twice. As you can see by the time we get to the last query, that takes three stops, the query takes quite a bit of time to complete.

```
clockWithResult(1){
    g.V().has('code','AUS').as('a').
          repeat(out().where(neq('a'))).times(1).simplePath().
          has('code','LAX').path().by('code').count().next()}

1.698357
1
```

As you would expect, looking for routes with exactly one stop does not take too long.

```
clockWithResult(1) {
```

```
    g.V().has('code','AUS').as('a').
          repeat(out().where(neq('a'))).times(2).simplePath().
          has('code','LAX').path().by('code').count().next()}

7.988092999999999
73
```

Looking for routes with exactly two stops takes quite a bit longer but is still fairly fast.

```
clockWithResult(1) {
    g.V().has('code','AUS').as('a').
          repeat(out().where(neq('a'))).times(3).simplePath().
          has('code','LAX').path().by('code').count().next()}

435.423921
4717
```

As you can see by the time we get to the last query, that looks for exactly three stops, things take a lot longer to complete.

```
clockWithResult(1) {
    g.V().has('code','AUS').as('a').
          repeat(out().where(neq('a'))).times(4).simplePath().
          has('code','LAX').path().by('code').count().next()}

34306.386952
325165
```

Notice that for the 'clock' and 'clockWithResult' methods to work correctly, you need to end the query with a termination step such as 'next' or 'toList'. Alternatively, you can end the query with 'iterate'. In our testing, we found things did not always work correctly when using 'iterate' so we tend to avoid it.

4.19.2. Analyzing where time is spent – introducing 'profile'

You can use the 'profile' step to ask Gremlin to give you a more fine-grained summary of where the time is spent processing your query. Take a look at the example below.

```
g.V().has('region','US-TX').out().has('region','US-CA').
                            out().has('country','DE').profile()
```

After you run the query, instead of showing you the results, Gremlin will show you where the time was spent processing the components of the query.

```
Traversal Metrics
Step                                                               Count  Traversers
```

```
Time (ms)    % Dur
=============================================================================================
======================
TinkerGraphStep(vertex,[region.eq(US-TX)])                          27          27
1.651    36.23
VertexStep(OUT,vertex)                                             816         816
0.274     6.01
HasStep([region.eq(US-CA)])                                         58          58
0.270     5.94
VertexStep(OUT,vertex)                                            4248        4248
0.900    19.76
NoOpBarrierStep(2500)                                             4248         237
1.348    29.58
HasStep([country.eq(DE)])                                           61           4
0.112     2.47
                                           >TOTAL                    -           -
4.557        -
```

If we profile the 'Austin to Agra' query from the previous section, you can see that almost all the time was spent inside the 'repeat' loop looking for routes. In this case that is not surprising, but for more complex queries 'profile' can help you to refine them.

```
g.V().has('airport','code','AUS').
      repeat(out()).until(has('code','AGR')).limit(10).
      path().by('code').profile()
```

Here is the 'profile' report. We truncated some the text with '"..." so that it will fit on a single page.

```
Traversal Metrics
Step                                                               Count  Traversers
Time (ms)    % Dur
=============================================================================================
======================
TinkerGraphStep(vertex,[~label.eq(airport), cod...                     1           1
2.178     0.13
RepeatStep([VertexStep(OUT,vertex), RepeatEndSt...                    11          11
1665.667    99.85
  HasStep([code.eq(AGR)])
497.169
  VertexStep(OUT,vertex)                                          958787      958787
176.579
  RepeatEndStep                                                       11          11
1488.996
RangeGlobalStep(0,10)                                                 10          10
0.231     0.01
PathStep([value(code)])                                               10          10
0.143     0.01
                                           >TOTAL                     -           -
```

```
1668.220        -
```

4.19.3. Introducing TinkerGraph indexes

TinkerGraph provides a rudimentary indexing capability, but using it can still improve the overall performance of your queries. Two methods 'createIndex' and 'dropIndex' are provided for creating and deleting indexes. Vertex and Edge properties can be indexed as needed. A third method, 'getIndexedKeys' can be used to query what indexes have been created. The example below runs the query we used in some of our prior tests with and without an index being present for the 'code' vertex property.

```
clock(1){
    g.V().has('code','AUS').repeat(out().simplePath()).
          until(has('code','AGR')).limit(10).path().by('code').toList()}

2019.8820959999998
```

Let's now create an index for the 'code' property of every vertex and try the query again. Note that 'Vertex' as used below is shorthand for 'Vertex.class'.

```
graph.createIndex('code',Vertex)

 clock(1){
     g.V().has('code','AUS').repeat(out().simplePath()).
           until(has('code','AGR')).limit(10).path().by('code').toList()}

1298.386245
```

We can query what indices we have created as follows.

```
graph.getIndexedKeys(Vertex)

code
```

Now let's drop the index.

```
graph.dropIndex('code',Vertex)
```

Because the air-routes graph is small, a Vertex property index has little effect on overall performance. However, as there are a lot more edges, perhaps creating an edge property index could help some queries. Let's try an experiment. The following query looks for all edges that have a 'dist' property of 1000.

```
clockWithResult(100) {g.E().has('dist',1000).count().next()}
```

```
9.80906926
28
```

Let's now create an index for the edge property called 'dist' and run the query again. We can also, as before, check to see what edge indexes have been created.

```
graph.createIndex('dist',Edge)

graph.getIndexedKeys(Edge)

dist
```

This time you can see that our index has made a big difference.

```
clockWithResult(100) {g.E().has('dist',1000).count().next()}

0.046712609999999995
28
```

The timing difference between the two queries can be attributed to the index. In the first case every edge in the graph (over 50,000 of them) had to be inspected. In the second case the index was used to go directly to the edges with a 'dist' of 1000, and no searching of all the edges was required. Note that the index only helps with exact comparisons. A query such as the one below will not benefit from the index.

```
clockWithResult(100) {g.E().has('dist',gt(1000)).count().next()}

11.980037119999999
20422
```

4.20. Using graph variables to associate metadata with TinkerGraph

A graph variable is a key/value pair that can be associated with a TinkerGraph. Graph variables are not considered part of the graph that you would process with Gremlin traversals but are, in essence, a way to associate metadata with a graph. You can set and retrieve graph variables using the 'variables' method of the graph object. Let's assume we wanted to add some metadata that allowed us to record who the maintainer of the 'air-routes' graph is and when it was last updated. We could do that as follows.

```
graph.variables().set('maintainer','Kelvin')
graph.variables().set('updated','July 18th 2017')
```

You can use any string that makes sense to you when naming your graph variable keys. We can use the 'keys' method to retrieve the names of any keys currently in place as graph variables.

```
graph.variables().keys()

updated
maintainer
```

The 'asMap' method will return any graph variables that are currently set as a map of key/value pairs.

```
graph.variables().asMap()

updated=July 18th 2017
maintainer=Kelvin
```

We can use the 'get' method to retrieve the value of a particular key. Note that the value returned is an instance of the 'java.util.Optional' class.

```
graph.variables().get('updated')

Optional[July 18th 2017]
```

If you want to delete a graph variable, you can use the 'remove' method. In this next example we will delete the 'maintainer' graph variable and re-query the variable map to prove it has been deleted.

```
graph.variables().remove('maintainer')
graph.variables().asMap()

updated=July 18th 2017
```

4.21. OLTP vs OLAP

When discussing graph processing, two terms regularly come up. These terms are Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). Most of the queries we have looked at so far definitely fall into the OLTP category. Queries that only look at a small part of a graph, often starting at a single vertex or a small group of vertices and traversing out a few hops from there, are considered OLTP operations. Typically, an OLTP query takes a few seconds or less to run. They are often used in response to a user query where an almost real-time answer is needed. Because the air-routes graph is small – we can do even quite complex queries using little more than a TinkerGraph and the Gremlin Console. This is not the case if your graph contains billions of vertices and edges. That said, We have seen some OLTP type queries, such as those looking for routes to remote airports, that can take several minutes to run in the TinkerGraph environment.

The TinkerPop documentation has in-depth coverage of the various OLAP capabilities that are supported. https://tinkerpop.apache.org/docs/current/reference/#graphcomputer

Where OLAP comes into play is when you want to do detailed analysis across an entire graph. This is especially true as graphs get large and require powerful clusters of vertices to underpin them. While OLTP queries typically take seconds or less, OLAP queries can often take many minutes or even hours to complete.

Apache TinkerPop provides significant support for OLAP graph processing out of the box and is designed to work well with distributed backend systems and software such as Apache Spark and Apache Hadoop.

While OLAP processing is typically done using powerful clusters, you can run simple experiments using nothing more than a standalone in memory TinkerGraph.

Detailed coverage of doing OLAP processing is beyond the current scope of this book, but we are going to give a few simple examples below to at least provide a little insight into what can be done. The TinkerPop documentation includes in-depth coverage of how to perform OLAP operations on a TinkerPop-enabled graph, and we recommend reading it if you plan to perform OLAP style graph processing.

4.21.1. Introducing the TinkerPop Graph Computer

GraphComputer is a Java interface that other TinkerPop classes implement to provide the capability for different back end environments. Along with 'GraphComputer', the concept of a 'VertexProgram' is also introduced. Vertex Programs allow us to specify the operations that we want to be performed across a graph, potentially in conjunction with map reduce operations. TinkerPop comes with a set of pre-configured Vertex Programs and vertex program steps. Of course, you can also write your own.

One of the pre-configured Vertex Programs is Page Rank. Let's take a look next at one way it can be used.

4.21.2. Experiments with Page Rank

Probably one of, if not the most famous algorithms in the big data world is 'Page Rank'. Originally developed at Google, the Page Rank algorithm was created as a way to measure the relative importance of web pages. At a high level the algorithm looks at the number of connections to a web page and the quality of those connections (as in are they from a prominent place). That algorithm ports well to a graph database environment as a graph, like the Web, is a heavily connected data structure. In our graph database environment, we can use a page rank algorithm to assess a graph and compute the likelihood in any given traversal that a particular vertex will get visited. Although the new Graph Computer capabilities have been designed from the ground up with distributed systems in mind, it is possible to use a standalone TinkerGraph to experiment with a few of the capabilities.

The example below only assumes that you have a TinkerGraph running locally inside the Gremlin Console with the 'air-routes' data loaded. You will notice a few differences from the examples we have seen so far in this book. First, when we create our graph traversal source object, we add a call to 'withComputer' to indicate that we plan to be doing some things that require the Graph Computer capabilities. Having created the traversal source, we can set up our page rank. The query initially filters out any vertex that is not an airport as will only want to rank airport vertices. Next a call is made to the 'pageRank' step. The 'by' modulators tell the page rank algorithm how we want the ranking done and what label to use for the ranking results. In this case we want to look at outgoing 'route' edges and label all results 'r'. Before returning the results, we order based on descending ranking value.

```
// Page rank based on outgoing routes
g3 = traversal().with(graph).withComputer()

g3.V().hasLabel('airport').
   pageRank().
     with(PageRank.edges, outE('route')).
     with(PageRank.propertyName, 'r').
   order().by('r',desc).valueMap('code','r').limit(10)
```

Here is the output from running the page rank algorithm. The results show the airport code along with the page rank value that was calculated.

```
[code:[IST],r:[16.40226392763168]]
[code:[DFW],r:[15.80681490439588]]
[code:[ORD],r:[14.944606331063445]]
[code:[DEN],r:[14.339886917974134]]
[code:[PEK],r:[13.587167366449114]]
[code:[ATL],r:[13.491138568380745]]
[code:[FRA],r:[13.443145391187 39]]
[code:[DXB],r:[13.440370180633318]]
[code:[CDG],r:[13.327429760219651]]
[code:[DME],r:[12.845152955480042]]
```

Just as a point of reference, the query below finds the top 10 airports with the most incoming routes.

```
g.V().hasLabel('airport').
      order().by(inE('route').count(),desc).limit(10).
      project('a','b').by('code').by(inE('route').count())
```

As you can see, there is some correlation between the page rank results and the airports with the most incoming routes, which is perhaps not surprising. However, notice that the page rank algorithm came up with some airports that a simple vertex degree test did not come up with.

```
[a:FRA,b:310]
```

```
[a:IST,b:309]
[a:CDG,b:294]
[a:AMS,b:285]
[a:MUC,b:271]
[a:ORD,b:264]
[a:DFW,b:253]
[a:PEK,b:249]
[a:DXB,b:248]
[a:ATL,b:242]
```

The 'pageRank' step used in the prior query is a nice and convenient way for us to quickly generate some rankings. However, it is important to understand how the query would be written if we were to use the Graph Computer more directly and submit a Vertex Program. You may find yourself taking this approach for your own custom 'VertexProgram' implementations when working with a graph that supports OLAP in an embedded mode. The example below sets up a 'PageRankVertexProgram' and runs it. Notice that the result we get back includes a new graph with 3624 vertices and no edges.

```
dcr = graph.compute().
      program(PageRankVertexProgram.build().
      edges(hasLabel('airport').outE('route')).
      create()).submit().get()

result[tinkergraph[vertices:3749 edges:0],memory[size:0]]
```

In case you are curious, we can check the precise type of the result object that is returned from creating our vertex program.

```
dcr.getClass()

class org.apache.tinkerpop.gremlin.
      process.computer.util.DefaultComputerResult
```

Given the result we got back was a new graph, we can create a new traversal and inspect it. Notice that the Istanbul (IST) vertex in this graph contains an additional property called 'gremlin.pageRankVertexProgram.pageRank'. This property contains the page rank score returned for this vertex.

```
g2 = dcr.graph().traversal()

g2.V().has('code','IST').valueMap(true)

[id:161,label:airport,country:[TR],longest:[13451],code:[IST],gremlin.pageRankVertexPr
ogram.pageRank:[0.004679970391715955],city:[Istanbul],lon:[28.751944],type:[airport],e
lev:[325],icao:[LTFM],region:[TR-34],runways:[4],lat:[41.275278],desc:[Istanbul
International Airport]]
```

Just to prove the original graph was untouched, let's check the IST vertex in that graph. As you can see, there are no new properties present.

```
g.V().has('code','IST').valueMap(true)

[id:161,label:airport,country:[TR],longest:[13451],code:[IST],city:[Istanbul],lon:[28.
751944],type:[airport],elev:[325],icao:[LTFM],region:[TR-34],runways:[4],lat:
[41.275278],desc:[Istanbul International Airport]]
```

Finally, let's look at the 10 airports sorted by page rank score in descending order. As you will observe, the scoring system is different, but the selected airports are the same as the ones returned by the in-line 'pageRank' step.

```
g2.V().order().by('gremlin.pageRankVertexProgram.pageRank',desc).limit(10).
       valueMap('code','gremlin.pageRankVertexProgram.pageRank')

[code:[IST],gremlin.pageRankVertexProgram.pageRank:[0.004679970391715955]]
[code:[DFW],gremlin.pageRankVertexProgram.pageRank:[0.004510826102525679]]
[code:[ORD],gremlin.pageRankVertexProgram.pageRank:[0.004264610652820994]]
[code:[DEN],gremlin.pageRankVertexProgram.pageRank:[0.0040921716650514847]]
[code:[PEK],gremlin.pageRankVertexProgram.pageRank:[0.00387754243351180 53]]
[code:[ATL],gremlin.pageRankVertexProgram.pageRank:[0.00384988329863 77]]
[code:[DXB],gremlin.pageRankVertexProgram.pageRank:[0.0038354789213581 97]]
[code:[FRA],gremlin.pageRankVertexProgram.pageRank:[0.0038353030232550645]]
[code:[CDG],gremlin.pageRankVertexProgram.pageRank:[0.0038025032480673913]]
[code:[DME],gremlin.pageRankVertexProgram.pageRank:[0.0036647477033891 01]]
```

We have barely scratched the surface in this section on these new TinkerPop OLAP capabilities. If this is an area that you are interested in, we strongly recommend reading the official TinkerPop reference documentation.

4.22. Translating Gremlin to different programming languages

This book mostly focuses on writing Gremln in its Groovy syntax, where it is easy to run inside the Gremlin Console, which is a great convenience when learning or just trying to work out a query. The Groovy syntax is actually quite close to the native form of the Gremlin language. As most developers tend to write Gremlin in the language they are most comfortable, it isn't always well known that Gremlin has its own grammar that is used for parsing queries. For historical reasons, particularly where Gremlin long ago was wholly built on top of Groovy in versions prior to TinkerPop 3, the grammar for native Gremlin matches quite closely to Groovy's grammar. We refer to this native form of gremlin as 'gremlin-lang'.

The only time that you might encounter gremlin-lang is with Gremlin Server, which uses the grammar by default to process queries. The grammar may also be used internally by TinkerPop-enabled graph systems that support the Gremlin Server protocols. Even in those cases, the Gremlin

drivers hide that syntax away by translating the Gremlin you write in your programming language to the form expected by the grammar. If you are not using the drivers and are instead using raw HTTP requests, then you must write gremlin-lang yourself as a string to send to the server. When sending direct HTTP requests, it would be possible to use the drivers to write Gremlin in your favorite programming language, get the gremlin-lang form and then manually submit that as a direct HTTP request, but it's hard to say what the benefit of that approach would be since the drivers are designed to hide that complexity from the application.

If you want to see the gremlin-lang form, you can get it from the traversal itself. In the following example, we write our Gremlin in Groovy and then use its 'gremlinLang' property (which is shorthand for Java's 'getGremlinLang()') to get the Gremlin native form after giving it the "g" representing the alias for the 'GraphTraversalSource':

```
t = g.V().outE('knows');[]
t.gremlinLang.getGremlin("g")

g.V().outE("knows")
```

You can see in the prior example that gremlin-lang is not so different from Groovy. The only difference is that the translation has normalized the single quotes to double quotes to define a string. That said, gremlin-lang can also equally accept single quotes for its string representation. The influence from Groovy on gremlin-lang is evident here. We can see where the divergence from Groovy though with a slightly longer example:

```
t = g.V().outE('knows').has('weight', 1 as byte);[]
t.gremlinLang.getGremlin("g")


g.V().outE("knows").has("weight",1B)
```

In Groovy, we define a 'byte' using the 'as' keyword. You can see in the translation that gremlin-lang allows for shorthand of a capital "B" suffix to the number to define a 'byte'.

If you have gremlin-lang compliant Gremlin, you can translate it to any of the programming languages that TinkerPop supports using the 'GremlinTranslator'. This can be helpful if you find an example written in a language you aren't familiar with and wish to convert it to one that you are.

```
GremlinTranslator.translate("g.V().outE(\"knows\")", Translator.GO)

g.V().OutE("knows")
```

Chapter 5. MISCELLANEOUS QUERIES AND THEIR RESULTS

In this chapter you will find more Gremlin queries that operate on the 'air-routes' graph. All of these queries build upon the topics covered in the prior sections. In this section we have included lots of examples of the output returned by running queries. In cases where the output is rather lengthy, we have either truncated it or laid it out in columns to make it easier to read and to save space. It is our hope also that from reading the examples in this section that you will get a sense of how good data modeled as a graph can be when used for analysis. We also think that the queries in this section show that you can achieve useful results from a graph using nothing more than some OLTP style queries and a TinkerGraph. This is actually a great example of an ideal use case for TinkerGraph. Even if you have your main data in a massive hosted graph, extracting parts of it and doing analysis locally using TinkerGraph is a technique that can make you very productive.

5.1. Counting more things

To get things started, let's look at a few more examples that basically just count occurrences and distributions of things but are a little more complex than the examples we looked at earlier in the book.

5.1.1. Which countries have no airports?

This first query looks for any 'country' vertices that have no outgoing edges. This indicates that there are no airports in the graph for those countries.

```
// Are there any countries that have no airports?
g.V().hasLabel('country').not(out()).values('desc')
```

So it seems there are five countries for which no airports were found.

```
Andorra
Liechtenstein
Monaco
Pitcairn
San Marino
```

The previous query is a slightly shorter form of the two queries below which would both yield the same results. Note the use of the '"__"' prefix in front of the 'not' step in the second query. This is because 'not' is a reserved word in Groovy and has to be prefixed in this way when not directly connected to a prior step by a dot.

```
// Are there any countries that have no airports?
g.V().hasLabel('country').where(out().count().is(0)).values('desc')
```

```
g.V().hasLabel('country').where(__.not(out())).values('desc')
```

5.1.2. Which airports have no routes?

There are a few airports in the graph that currently have no commercial routes. This is either because they used to have service, and it was discontinued, or they are new airports still awaiting service to start. We can write a query to easily find these "orphan" airports.

In more recent updates of the data set, some of these airports do now have commercial airline service.

```
g.V().hasLabel('airport').not(bothE('route')).values('code').fold()
```

When we run the query, you can see we find quite a few orphan airport vertices that have no outgoing or incoming routes.

```
[TXL,SXF,ILG,TWB,TUA,BVS,KGG,RIG,INT,APA,BWU,BID,NBW,SFH,CVT,AFW,PSY,EKA,ISL,APK,CRC,G
YZ,HFN,HZK,NFO,SFD,VCV,YEI]
```

5.1.3. What is the distribution of runways?

We can easily count the distribution per airport of runways. We can observe from the results of running the query that the vast majority of airports in the graph have either one or two runways.

```
// What is the distribution of runways in the graph
g.V().hasLabel('airport').groupCount().by('runways')

[1:2429,2:775,3:227,4:53,5:14,6:4,7:2]
```

5.1.4. Airports with the most routes

This next query finds all airports that have more than 180 outgoing routes and returns their IATA codes.

```
// Airports with more than 180 outgoing routes
g.V().hasLabel('airport').
      where(out('route').count().is(gt(180))).values('code').fold()

[ATL,DFW,IAH,JFK,LAX,MIA,ORD,DEN,EWR,YYZ,LHR,LGW,CDG,FRA,DXB,DUB,PEK,PVG,FCO,AMS,BCN,M
AD,VIE,BRU,MUC,MAN,STN,DME,SVO,DOH,IST,CPH,CLT,DUS,BER]
```

We could improve our query a bit to include the IATA code and the exact number of outgoing routes in the returned result. There are a few different ways that we could do this. One way that is quite convenient is to use 'group'.

```
// Same basic query but return the airport code and the route count
g.V().hasLabel('airport').
      where(out('route').count().is(gt(180))).
      group().by('code').by(out().count())
```

We have laid the results out in a grid to make them easier to read.

```
[ORD:265,PVG:213,LAX:214,BRU:194,
 CDG:293,CLT:184,STN:211,SVO:181,
 JFK:204,DUB:185,DFW:253,LHR:221,
 MUC:270,DOH:187,DME:232,EWR:203,
 DUS:199,MIA:196,AMS:283,IST:309,
 DEN:217,BCN:203,BER:202,DXB:248,
 IAH:199,MAD:206,FCO:201,VIE:206,
 FRA:310,PEK:248,ATL:242,CPH:194,
 YYZ:195,MAN:216,LGW:232]
```

We can add one further refinement to the query. This time the results are ordered by the number of routes in descending order. Note the use of 'local' to make sure that 'order' is applied to the contents of the collection that was generated by the 'group' step.

```
// Same query with ordered results
g.V().hasLabel('airport').
      where(out('route').count().is(gt(180))).
      group().by('code').by(out().count()).
      order(local).by(values,desc)
```

We have again laid the results out in a grid.

```
[FRA:310, IST:309, CDG:293, AMS:283,
 MUC:270, ORD:265, DFW:253, DXB:248,
 PEK:248, ATL:242, DME:232, LGW:232,
 LHR:221, DEN:217, MAN:216, LAX:214,
 PVG:213, STN:211, MAD:206, VIE:206,
 JFK:204, EWR:203, BCN:203, BER:202,
 FCO:201, DUS:199, IAH:199, MIA:196,
 YYZ:195, BRU:194, CPH:194, DOH:187,
 DUB:185, CLT:184, SVO:181]
```

As we mentioned earlier in the book, the number of incoming and outgoing routes for any given airport will not always be the same due to how airlines operate their flight routing. The query below looks for any airports that have more than 400 total routes (inbound and outbound).

```
// Airports with more than 400 total routes
g.V().hasLabel('airport').
```

```
      where(both('route').count().is(gt(400))).
      values('code').fold()

[ATL,DFW,JFK,LAX,ORD,DEN,EWR,LHR,LGW,CDG,FRA,DXB,PEK,PVG,FCO,AMS,BCN,MAD,VIE,MUC,MAN,S
TN,DME,IST,BER]
```

5.1.5. Airports with just one route

There are, perhaps surprisingly, a large number of airports that only have one route. The query below will figure out just how many fall into that category.

```
// How many airports have only one route?
g.V().hasLabel('airport').
  where(out().count().is(eq(1))).count()

777
```

5.1.6. Single runway airports with the most routes

It is interesting to look at how busy some single runway airports are. The query below looks for the ten airports with just one runway that have the most outgoing routes. You will notice that we have included London Gatwick in the query using an 'or' step. This is because while technically Gatwick is listed in the graph as an airport with two runways, in practice the second runway is primarily used as a taxiway and reserved for emergency use only. Therefore, Gatwick is really a single runway airport. This also makes the query a bit more interesting!

```
g.V().or(has('airport','runways',1),has('code','LGW')).
      order().by(out().count(),desc).limit(10).
      project('apt','city','routes').
      by('code').by('city').by(out().count())
```

When we run the query, here are the results we get back. One interesting observation is that three of the top five busiest single runway airports are in England.

```
[apt:LGW,city:London,routes:232]
[apt:STN,city:London,routes:211]
[apt:LIS,city:Lisbon,routes:139]
[apt:CTU,city:Chengdu,routes:139]
[apt:LTN,city:London,routes:130]
[apt:BHX,city:Birmingham,routes:130]
[apt:SZX,city:Shenzhen,routes:129]
[apt:CKG,city:Chongqing,routes:122]
[apt:STR,city:Stuttgart,routes:121]
[apt:CRL,city:Brussels,routes:117]
```

5.1.7. Another way of counting runways

Let's assume we wanted to count the runways in the graph and for each number of runways produce a simple map result where the runway number and total count of airports having that number of runways are each meaningfully labeled. One way we could do that is to use a 'groupCount' step to calculate the distribution of runways and then use a 'project' step to produce the nicely labeled result. That is what the query below does. Notice that an 'unfold' step is used so that the results of the 'groupCount' which itself produces a map can be further processed.

```
g.V().hasLabel('airport').
      groupCount().by('runways').
      unfold().
      project('runways','count').by(keys).by(values)
```

When run we get a nice map back showing us the number of runways and for each the total count. Each value has a meaningful key name of 'runways' and 'count' respectively.

```
[runways:1,count:2429]
[runways:2,count:775]
[runways:3,count:227]
[runways:4,count:53]
[runways:5,count:14]
[runways:6,count:4]
[runways:7,count:2]
```

Notice that if the unfold step had not been used, a very different result would have been generated.

```
g.V().hasLabel('airport').
      groupCount().by('runways').
      project('runways','count').by(keys).by(values)

[runways:[1,2,3,4,5,6,7],count:[2429,775,227,53,14,4,2]]
```

5.1.8. Airports with the most routes in Canada

The following query finds the top 10 airports in Canada sorted by descending number of outgoing routes. Just for fun, this time we used the 'index' method to include a one-based index as part of the results.

```
g.V().has('country','code','CA').out().
      order().by(out().count(),desc).limit(10).
      project('apt','city','routes').
      by('code').by('city').by(out().count()).indexed(1)
```

Here are the results of running the query along with the index that we added.

```
[1,[apt:YYZ,city:Toronto,routes:195]]
[2,[apt:YUL,city:Montreal,routes:123]]
[3,[apt:YVR,city:Vancouver,routes:106]]
[4,[apt:YYC,city:Calgary,routes:75]]
[5,[apt:YEG,city:Edmonton,routes:48]]
[6,[apt:YHZ,city:Halifax,routes:45]]
[7,[apt:YWG,city:Winnipeg,routes:38]]
[8,[apt:YOW,city:Ottawa,routes:36]]
[9,[apt:YZF,city:Yellowknife,routes:21]]
[10,[apt:YQB,city:Quebec City,routes:20]]
```

5.1.9. Distribution of UK airports

How many airports are there in each of the UK regions of England, Scotland, Wales, and Northern Ireland?

```
g.V().has('country','code','UK').out('contains').groupCount().by('region')

[GB-ENG:27,GB-WLS:3,GB-NIR:3,GB-SCT:25]
```

5.1.10. Distribution of airports by country

This query uses 'groupCount' to produce a map of key value pairs where the key is the two-character ISO country code and the value is the number of airports that country has.

```
// How many airports does each country have in the graph?
g.V().hasLabel('airport').
      groupCount().by('country')
```

When run, the query produces quite a lot of output. As the values are not sorted, it is hard to find the countries with the most airports. Note that the 'groupCount' step does not include countries that had no airports. In other words, if the count is zero, the country was skipped.

```
[PR:7,PT:19,PW:1,PY:2,QA:1,AE:10,AF:5,AG:1,AI:1,AL:1,AM:2,AO:14,AR:38,AS:1,AT:6,RE:2,A
U:132,AW:1,AZ:5,RO:14,BA:4,BB:1,RS:3,BD:7,BE:5,RU:129,BF:2,BG:4,RW:2,BH:1,BI:1,BJ:1,BL
:1,BM:1,BN:1,BO:15,SA:26,BQ:3,SB:17,BR:117,SC:2,BS:18,SD:5,SE:39,BT:1,SG:2,BW:4,SH:2,S
I:2,BY:3,BZ:13,SK:2,SL:1,SN:4,SO:6,CA:205,SR:1,SS:1,CC:1,CD:11,ST:1,SV:1,CF:1,CG:3,CH:
6,SX:1,CI:2,SY:2,SZ:1,CK:6,CL:17,CM:5,CN:217,CO:51,CR:13,TC:4,TD:5,CU:12,CV:7,TG:1,TH:
33,CW:1,CX:1,CY:3,TJ:4,CZ:5,TL:1,TM:1,TN:8,TO:5,TR:52,TT:2,DE:34,TV:1,TW:9,TZ:11,DJ:1,
DK:8,DM:1,DO:7,UA:15,UG:4,UK:58,DZ:30,US:586,EC:15,EE:3,EG:11,EH:2,UY:2,UZ:11,ER:1,VC:
1,ES:43,ET:16,VE:27,VG:2,VI:2,VN:22,VU:26,FI:20,FJ:10,FK:2,FM:4,FO:1,FR:59,WF:2,GA:2,W
S:1,GD:1,GE:3,GF:1,GG:2,GH:6,GI:1,GL:14,GM:1,GN:1,GP:1,GQ:2,GR:39,GT:2,GU:1,GW:1,GY:2,
HK:1,HN:6,HR:8,HT:2,YE:9,HU:2,ID:70,YT:1,IE:7,IL:6,IM:1,IN:77,ZA:20,IQ:6,IR:45,IS:7,IT
:38,ZM:9,JE:1,ZW:3,JM:2,JO:2,JP:65,KE:14,KG:2,KH:3,KI:2,KM:1,KN:2,KP:1,KR:15,KS:1,KW:1
,KY:3,KZ:21,LA:8,LB:1,LC:2,LK:6,LR:2,LS:1,LT:3,LU:1,LV:1,LY:11,MA:16,MD:1,ME:2,MF:1,MG
:13,MH:2,MK:2,ML:1,MM:17,MN:10,MO:1,MP:2,MQ:1,MR:3,MS:1,MT:1,MU:2,MV:8,MW:7,MX:60,MY:3
```

```
5,MZ:11,NA:5,NC:1,NE:1,NF:1,NG:19,NI:1,NL:5,NO:49,NP:11,NR:1,NZ:25,OM:4,PA:5,PE:22,PF:
39,PG:26,PH:40,PK:21,PL:13,PM:1]
```

If we wanted to sort the list in descending order using the numeric values, we could adjust the query as follows. Once again, note the use of 'local' to specify how the ordering is applied.

```
g.V().hasLabel('airport').
      groupCount().by('country').
      order(local).by(values,desc)
```

This time it is much easier to see which countries have the most airports.

```
[US:586,CN:217,CA:205,AU:132,RU:129,BR:117,IN:77,ID:70,JP:65,MX:60,FR:59,UK:58,TR:52,C
O:51,NO:49,IR:45,ES:43,PH:40,SE:39,GR:39,PF:39,AR:38,IT:38,MY:35,DE:34,TH:33,DZ:30,VE:
27,SA:26,VU:26,PG:26,NZ:25,VN:22,PE:22,KZ:21,PK:21,FI:20,ZA:20,PT:19,NG:19,BS:18,SB:17
,CL:17,MM:17,ET:16,MA:16,BO:15,UA:15,EC:15,KR:15,AO:14,RO:14,GL:14,KE:14,BZ:13,CR:13,M
G:13,PL:13,CU:12,CD:11,TZ:11,EG:11,UZ:11,LY:11,MZ:11,NP:11,AE:10,FJ:10,MN:10,TW:9,YE:9
,ZM:9,TN:8,DK:8,HR:8,LA:8,MV:8,PR:7,BD:7,CV:7,DO:7,IE:7,IS:7,MW:7,AT:6,SO:6,CH:6,CK:6,
GH:6,HN:6,IL:6,IQ:6,LK:6,AF:5,AZ:5,BE:5,SD:5,CM:5,TD:5,CZ:5,TO:5,NA:5,NL:5,PA:5,BA:4,B
G:4,BW:4,SN:4,TC:4,TJ:4,UG:4,FM:4,OM:4,RS:3,BQ:3,BY:3,CG:3,CY:3,EE:3,GE:3,ZW:3,KH:3,KY
:3,LT:3,MR:3,PY:2,AM:2,RE:2,BF:2,RW:2,SC:2,SG:2,SH:2,SI:2,SK:2,CI:2,SY:2,TT:2,EH:2,UY:
2,VG:2,VI:2,FK:2,WF:2,GA:2,GG:2,GQ:2,GT:2,GY:2,HT:2,HU:2,JM:2,JO:2,KG:2,KI:2,KN:2,LC:2
,LR:2,ME:2,MH:2,MK:2,MP:2,MU:2,PW:1,QA:1,AG:1,AI:1,AL:1,AS:1,AW:1,BB:1,BH:1,BI:1,BJ:1,
BL:1,BM:1,BN:1,BT:1,SL:1,SR:1,SS:1,CC:1,ST:1,SV:1,CF:1,SX:1,SZ:1,TG:1,CW:1,CX:1,TL:1,T
M:1,TV:1,DJ:1,DM:1,ER:1,VC:1,FO:1,WS:1,GD:1,GF:1,GI:1,GM:1,GN:1,GP:1,GU:1,GW:1,HK:1,YT
:1,IM:1,JE:1,KM:1,KP:1,KS:1,KW:1,LB:1,LS:1,LU:1,LV:1,MD:1,MF:1,ML:1,MO:1,MQ:1,MS:1,MT:
1,NC:1,NE:1,NF:1,NI:1,NR:1,PM:1]
```

If we wanted to sort by the country code, the 'key' in other words, we could change the query accordingly. In this case we will use 'by(keys,asc)' to get a sort in ascending order. If we wanted to sort in descending order by key, we could use 'by(keys,desc)' instead.

```
g.V().hasLabel('airport').
      groupCount().by('country').
      order(local).by(keys,asc)
```

This time the results are now sorted using the country codes in ascending alphabetical order.

```
[AE:10,AF:5,AG:1,AI:1,AL:1,AM:2,AO:14,AR:38,AS:1,AT:6,AU:132,AW:1,AZ:5,BA:4,BB:1,BD:7,
BE:5,BF:2,BG:4,BH:1,BI:1,BJ:1,BL:1,BM:1,BN:1,BO:15,BQ:3,BR:117,BS:18,BT:1,BW:4,BY:3,BZ
:13,CA:205,CC:1,CD:11,CF:1,CG:3,CH:6,CI:2,CK:6,CL:17,CM:5,CN:217,CO:51,CR:13,CU:12,CV:
7,CW:1,CX:1,CY:3,CZ:5,DE:34,DJ:1,DK:8,DM:1,DO:7,DZ:30,EC:15,EE:3,EG:11,EH:2,ER:1,ES:43
,ET:16,FI:20,FJ:10,FK:2,FM:4,FO:1,FR:59,GA:2,GD:1,GE:3,GF:1,GG:2,GH:6,GI:1,GL:14,GM:1,
GN:1,GP:1,GQ:2,GR:39,GT:2,GU:1,GW:1,GY:2,HK:1,HN:6,HR:8,HT:2,HU:2,ID:70,IE:7,IL:6,IM:1
,IN:77,IQ:6,IR:45,IS:7,IT:38,JE:1,JM:2,JO:2,JP:65,KE:14,KG:2,KH:3,KI:2,KM:1,KN:2,KP:1,
KR:15,KS:1,KW:1,KY:3,KZ:21,LA:8,LB:1,LC:2,LK:6,LR:2,LS:1,LT:3,LU:1,LV:1,LY:11,MA:16,MD
```

```
:1,ME:2,MF:1,MG:13,MH:2,MK:2,ML:1,MM:17,MN:10,MO:1,MP:2,MQ:1,MR:3,MS:1,MT:1,MU:2,MV:8,
MW:7,MX:60,MY:35,MZ:11,NA:5,NC:1,NE:1,NF:1,NG:19,NI:1,NL:5,NO:49,NP:11,NR:1,NZ:25,OM:4
,PA:5,PE:22,PF:39,PG:26,PH:40,PK:21,PL:13,PM:1,PR:7,PT:19,PW:1,PY:2,QA:1,RE:2,RO:14,RS
:3,RU:129,RW:2,SA:26,SB:17,SC:2,SD:5,SE:39,SG:2,SH:2,SI:2,SK:2,SL:1,SN:4,SO:6,SR:1,SS:
1,ST:1,SV:1,SX:1,SY:2,SZ:1,TC:4,TD:5,TG:1,TH:33,TJ:4,TL:1,TM:1,TN:8,TO:5,TR:52,TT:2,TV
:1,TW:9,TZ:11,UA:15,UG:4,UK:58,US:586,UY:2,UZ:11,VC:1,VE:27,VG:2,VI:2,VN:22,VU:26,WF:2
,WS:1,YE:9,YT:1,ZA:20,ZM:9,ZW:3]
```

Note that we can use 'select' to only return one or more of the full set of key/value pairs returned. Here is an example of doing just that.

```
// Only return the values for Germany, China, Holland and the US.
g.V().hasLabel('airport').
      groupCount().by('country').
      select('DE','CN','NL','US')

[DE:34,CN:217,NL:5,US:586]
```

Because the 'air-routes' graph also has country-specific vertices, we could choose to write the previous queries a different way. We could start by finding country vertices and then see how many airport vertices each one is connected to. In this instance, because a 'group' step is being used, countries with no airports will be included.

```
// Another way to ask the question above, this time by counting the
// edges (out degree) from each country

g.V().hasLabel('country').
      group().by('code').by(outE().count()).
      order(local).by(values,desc)
```

This time the countries with no airports **are** included in the results.

```
[US:586,CN:217,CA:205,AU:132,RU:129,BR:117,IN:77,ID:70,JP:65,MX:60,FR:59,UK:58,TR:52,C
O:51,NO:49,IR:45,ES:43,PH:40,SE:39,GR:39,PF:39,AR:38,IT:38,MY:35,DE:34,TH:33,DZ:30,VE:
27,SA:26,VU:26,PG:26,NZ:25,VN:22,PE:22,KZ:21,PK:21,FI:20,ZA:20,PT:19,NG:19,BS:18,SB:17
,CL:17,MM:17,ET:16,MA:16,BO:15,UA:15,EC:15,KR:15,AO:14,RO:14,GL:14,KE:14,BZ:13,CR:13,M
G:13,PL:13,CU:12,CD:11,TZ:11,EG:11,UZ:11,LY:11,MZ:11,NP:11,AE:10,FJ:10,MN:10,TW:9,YE:9
,ZM:9,TN:8,DK:8,HR:8,LA:8,MV:8,PR:7,BD:7,CV:7,DO:7,IE:7,IS:7,MW:7,AT:6,SO:6,CH:6,CK:6,
GH:6,HN:6,IL:6,IQ:6,LK:6,AF:5,AZ:5,BE:5,SD:5,CM:5,TD:5,CZ:5,TO:5,NA:5,NL:5,PA:5,BA:4,B
G:4,BW:4,SN:4,TC:4,TJ:4,UG:4,FM:4,OM:4,RS:3,BQ:3,BY:3,CG:3,CY:3,EE:3,GE:3,ZW:3,KH:3,KY
:3,LT:3,MR:3,PY:2,AM:2,RE:2,BF:2,RW:2,SC:2,SG:2,SH:2,SI:2,SK:2,CI:2,SY:2,TT:2,EH:2,UY:
2,VG:2,VI:2,FK:2,WF:2,GA:2,GG:2,GQ:2,GT:2,GY:2,HT:2,HU:2,JM:2,JO:2,KG:2,KI:2,KN:2,LC:2
,LR:2,ME:2,MH:2,MK:2,MP:2,MU:2,PW:1,QA:1,AG:1,AI:1,AL:1,AS:1,AW:1,BB:1,BH:1,BI:1,BJ:1,
BL:1,BM:1,BN:1,BT:1,SL:1,SR:1,CC:1,SS:1,ST:1,CF:1,SV:1,SX:1,SZ:1,TG:1,CW:1,CX:1,TL:1,T
M:1,TV:1,DJ:1,DM:1,ER:1,VC:1,FO:1,WS:1,GD:1,GF:1,GI:1,GM:1,GN:1,GP:1,GU:1,GW:1,HK:1,YT
:1,IM:1,JE:1,KM:1,KP:1,KS:1,KW:1,LB:1,LS:1,LU:1,LV:1,MD:1,MF:1,ML:1,MO:1,MQ:1,MS:1,MT:
```

```
1,NC:1,NE:1,NF:1,NI:1,NR:1,PM:1,AD:0,SM:0,LI:0,MC:0,PN:0]
```

If we only wanted to see the last few of the sorted results, we could add a 'tail' step with 'local' scope to the query.

```
g.V().hasLabel('country').
      group().by('code').by(outE().count()).
      order(local).by(values,desc).
      tail(local,20)

[LU:1,LV:1,MD:1,MF:1,ML:1,MO:1,MQ:1,MS:1,MT:1,NC:1,NE:1,NF:1,NI:1,NR:1,PM:1,AD:0,SM:0,
LI:0,MC:0,PN:0]
```

Distribution of airports by continent We can use a similar query to the one above to find out how many airports are located in each of the seven continents. As you can see from the output, a key/value map is again returned where the key is the continent code and the value is the number of airports in that continent. Note that currently there are no airports with regular scheduled service in Antarctica!

```
// How many airports are there in each continent?
g.V().hasLabel('continent').group().by('code').by(out().count())

[EU:605,AS:971,NA:989,OC:305,AF:321,AN:0,SA:313]
```

5.1.11. Distribution of routes per airport

We have already examined various ways to calculate the distribution of routes in the graph. The following query will, for each airport, return a key value pair where the key is the airport code and the value is the number of outgoing routes from that airport. Because there are over 3,000 airports in the graph, this query will produce a large result. We decided not to include those results here. The second query just picks the results from the map for a few airports. Those results are shown.

```
// How many flights are there from each airport?
g.V().hasLabel('airport').out().groupCount().by('code')

// count the routes from all the airports and then select a few.
g.V().hasLabel('airport').out().groupCount().by('code').
      select('AUS','AMS','JFK','DUB','MEX')

[AUS:98,AMS:285,JFK:203,DUB:185,MEX:116]
```

This next query essentially asks the same question about how many outgoing routes each airport has. However, rather than return the count for each airport individually, it groups the ones with the same number of routes together. As this query returns a lot of data, we just included a few lines from the full result below the query.

```
// Same query except sorted into groups by ascending count
g.V().hasLabel('airport').
      group().by(out().count()).by('code').
      order(local).by(keys)
```

As can be seen, this time the count value is the key and the airport codes are the values.

```
[..., 73:[ALG],74:[MEL,KWI,SVQ,SVX],75:[BNA,CLE,LPL,YYC,SOF,EIN],76:[AMM,CTA],77:[
CGK],78:[STL,RHO,HAJ,TLS],79:[NTE],80:[LCA,PDX,PSA,KIX,OVB],81:[LGA,SAN,HER],82:[BNE,B
SL],83:[FAO,RAK],84:[HND,BOG,XMN,WUH],85:[MDW,KEF],86:[DMK],87:[MNL],88:[PTY,SKG],89:[
IBZ,JNB],90:[TPA,VLC],91:[BWI,CAI,HGH,CSX],92:[GLA,BOD],94:[BOM,ADD,BLQ,LPA,RUH],95:[O
PO,SFB],96:[DCA,RIX],97:[GRU],98:[AUS],99:[VCE],100:[AYT,BRS,KBP],102:[SYD,OTP,SHJ],10
3:[AUH,CMN],104:[SLC,KRK],106:[YVR,TFS],110:[NAP],111:[CUN],112:[JED],113:[LYS],115:[N
CE,SAW],116:[ALC,MEX,WAW,KMG],117:[MRS,CRL,XIY],118:[PHX,DEL],119:[MLA],121:[STR],122:
[CKG],123:[YUL,VKO],125:[HAM],126:[SEA],127:[BUD,TPE],128:[HEL,GVA,OSL],129:[SZX,BGY],
130:[LTN,BHX],136:[EDI],138:[CGN],139:[ORY,LIS,CTU],140:[AGP],141:[NRT,PRG],143:[BOS,T
LV],145:[DTW,KUL],147:[PHL],151:[MXP],153:[BKK,PMI],155:[ATH],156:[MSP],157:[SFO],158:
[FLL,IAD],159:[LED],161:[LAS],165:[MCO],168:[ARN],169:[SIN],173:[ICN],174:[HKG],175:[C
AN],180:[ZRH],181:[SVO],184:[CLT],185:[DUB],187:[DOH],194:[BRU,CPH],195:[YYZ],196:[MIA
],199:[IAH,DUS],201:[FCO],202:[BER],203:[EWR,BCN],204:[JFK],206:[MAD,VIE],211:[STN],21
3:[PVG],214:[LAX],216:[MAN],217:[DEN],221:[LHR],232:[LGW,DME],242:[ATL],248:[DXB,PEK],
253:[DFW],265:[ORD],270:[MUC],283:[AMS],293:[CDG],309:[IST],310:[FRA]]
```

As the above query returns a lot of data, we can also extract specific values we are interested in as follows. Only airports with 62 outgoing routes are selected.

```
// Which of these airports have 105 outgoing routes?
g.V().hasLabel('airport').
      group().by(out().count()).by('code').
      select(constant(62L))
```

The 'select' step does not allow direct selection of non-string keys. To work around this, you use its 'Traversal' form to access a numeric key from a result.

This time the results only include the airports with 62 outgoing routes.

```
[OAK,BEG,CVG,CAG]
```

5.1.12. Using groupCount with a traversal variable

So far we have just used 'groupCount' with no parameters. When used in that way, 'groupCount' behaves like a 'map' step in that it passes the transformed data on to the next step. However, if you specify the name of a traversal variable as a parameter, the results of the count will be stored in that variable, and 'groupCount' will act the same way as a 'sideEffect' would, nothing is passed on from 'groupCount' to the next step. We can use this capability to keep track of things during a query

while not changing the overall state of the traversal.

The example below starts at the vertex 'V(3)' and goes 'out' from there. A 'groupCount' step is then used to group the vertices we visited by a count of the runways each has. We then go 'out' again and count how many vertices we found and save that result in the variable 'b'. Note that the 'groupCount' when used in this way did not pass anything on to the following step. Finally, we use 'select' to return our two variables as the results of the query.

```
g.V(3).out().groupCount('a').by('runways').
       out().count().as('b').select('a','b')

[a:[1:9,2:24,3:30,4:24,5:5,6:4,7:2],b:8354]
```

In the next section we will see another example of a 'groupCount' step that uses a traversal variable.

5.1.13. Combining 'groupCount' and 'constant'

You can use a 'constant' value in combination with a 'groupCount' step as one way of setting the name of the key that will be used in the result. The example below shows a 'groupCount' step that is provided a traversal variable '"a"' as a parameter and a constant as part of the 'by' modulator. The query starts by finding any airports in the US state of Oklahoma. A 'constant' step is used to tell the 'groupCount' step what to use as the key name in the result. At the end of the query a 'cap' step is used to close and return the contents of '"a"'.

```
g.V().has('airport','region','US-OK').groupCount('a').by(constant('OK')).cap('a')
```

If we run the query, here is what we will get back. Notice the key is our constant value '"OK"', and the value is the number of airports found in Oklahoma.

```
[OK:4]
```

The example above is intended purely to demonstrate the fact that you can use a 'constant' value with 'groupCount'. However, for this specific example, you would probably code a query something like the one below instead.

```
g.V().has('airport','region','US-OK').groupCount().by('region')

[US-OK:4]
```

5.1.14. Combining 'choose' and 'groupCount'

Using the ideas discussed above, we can write a query that chains several 'choose' steps together to build a result set made up of various key:value pairs representing different things that we are interested in counting. The query below looks at all vertices that represent an airport. For each

vertex found, various properties are examined using a 'choose' step. If the test returns 'true', a count, stored in the traversal variable "'a"' is incremented. Note how each of the 'choose' steps is 'dot chained' together. When used in this way, 'choose' behaves in the same way as a 'sideEffect' in that all the airport vertices are passed to the next 'choose' rather than just those that pass the *if* test.

```
g.V().hasLabel('airport').
      choose(has('runways',4), groupCount('a').by(constant('four'))).
      choose(has('runways',lte(2)), groupCount('a').by(constant('low'))).
      choose(has('runways',gte(6)), groupCount('a').by(constant('high'))).
      choose(has('country','FR'), groupCount('a').by(constant('France'))).
      groupCount('a').by(constant('total')).cap('a')
```

If we run the query, this is what we might get back. We counted 3504 total airports, found six airports that have six or more runways, 53 that had exactly four runways, 3204 that have two or fewer, and found 59 airports in France.

```
[high:6,total:3504,low:3204,four:53,France:59]
```

Using 'choose' and 'groupCount' in this way provides a very nice pattern for counting somewhat disparate things during a traversal.

5.1.15. Nesting one 'group' step inside another

Many examples of both the 'groupCount' and 'group' steps have already been presented in this book. However, something we have not touched on so far, which by now may be obvious but perhaps not, is that you can nest one 'group' step inside another one. This is made possible by the fact that the 'by' modulators that are used to tell the 'group' step precisely what you want grouped can take arbitrary traversals. Take a look at the example below. Hopefully, the indentation makes it easier to read. We begin by simply finding five airports and starting a group. The key for each group will be one of the codes for each of these airports. This is specified by the first 'by' modulator. For the value part of each group, specified by the second 'by' modulator, we start a second traversal. That traversal begins by finding five places you can fly to from each of the five airports that we initially found. Then another 'group' step is used to group each of those airports by the number of total outgoing routes they have.

```
g.V().hasLabel("airport").limit(5).
      group().
        by('code').
        by(out("route").limit(5).
           group().
             by('code').
             by(out("route").count()))
```

When the query is run, here is what we get back. As expected, the outer group has keys that represent the first five airports that were found. The inner group has the five airport destinations as the keys, and their total outgoing route counts as their values. We pretty printed the output a bit

to make it easier to read.

```
[BNA:[DCA:96,BWI:91,BOS:143,ATL:242,AUS:98],
 ANC:[ORD:265,IAH:199,LAX:214,DFW:253,MSP:156],
 BOS:[DCA:96,BNA:75,BWI:91,ATL:242,AUS:98],
 ATL:[NRT:141,FRA:310,LHR:221,CDG:293,YYZ:195],
 AUS:[ONT:23,PHL:147,PDX:80,DTW:145,OKC:33]]
```

So what we have created is a group, essentially a map, that has airport codes as the key and an additional group as the values. Therefore, given each group is made up of key value pairs, we can use a 'select' step to only pick a few of the results.

```
g.V().hasLabel("airport").limit(5).
      group().
        by('code').
        by(out("route").limit(5).
           group().
             by('code').
             by(out("route").count())).
      select('ANC','ATL')
```

This time only the selected results are returned

```
[ANC:[ORD:265,IAH:199,LAX:214,DFW:253,MSP:156],
 ATL:[NRT:141,FRA:310,LHR:221,CDG:293,YYZ:195]]
```

If you want to select results from one of the inner groups, you can do that as well using an additional 'select' step.

```
g.V().hasLabel("airport").limit(5).
      group().
        by('code').
        by(out("route").limit(5).
           group().
             by('code').
             by(out("route").count())).
      select('ANC').
      select('IAH','LAX')
```

This will first select the result with a key of 'ANC' and then from that group select the results for the keys of 'IAH' and 'LAX' only.

```
[IAH:199,LAX:214]
```

5.1.16. Analysis of routes between Europe and the USA

The next few queries show how you can use a graph like 'air-routes' to perform analysis on a particular industry segment. The following queries analyze the distribution and availability of routes between airports across Europe and airports in the United States. First, let's just find out how many total routes there are between airports anywhere in Europe and airports in the USA.

```
// How many routes from anywhere in Europe to the USA?
g.V().has('continent','code','EU').
      out().out().has('country','US').
      count()

435
```

So we now know that there are 435 different routes. Remember, though, that the 'air-routes' graph does not track the number of airlines that operate any of these routes. The graph just stores the data that at least one airline operates each of these unique route pairs. Let's dig a bit deeper into the 435 and find out how many US airports have flights that arrive from Europe.

```
// How many different US airports have routes from Europe?
g.V().has('continent','code','EU').
      out().out().has('country','US').
      dedup().count()

45
```

So we can now see that the 435 routes from European airports arrive at one of 45 airports in the United States. We can dig a bit deeper and look at the distribution of these routes across the 45 airports.

```
//What is the distribution of the routes amongst those US airports?
g.V().has('continent','code','EU').
      out().out().has('country','US').
      groupCount().by('code').
      order(local).by(values,asc)
```

John F. Kennedy airport (JFK) in New York appears to have the most routes from Europe, with Newark (EWR) having the second most.

```
[CHS:1,MCI:1,BNA:1,ANC:1,CLE:1,IND:1,STL:2,BDL:2,SJC:2,BWI:2,PHX:2,
 MSY:2,CVG:2,SAN:3,RDU:3,PDX:3,SLC:4,AUS:4,RSW:4,PIT:4,TPA:5,SFB:5,SWF:5,
 DTW:6,MSP:6,OAK:6,FLL:7,DEN:7,PVD:7,CLT:8,IAH:8,LAS:11,DFW:12,SEA:12,MCO:14,
 ATL:15,SFO:20,PHL:22,IAD:22,LAX:26,BOS:26,ORD:27,MIA:28,JFK:42,EWR:43]
```

Now let's repeat the process but look at the European end of the routes. First, we can calculate how

many European airports have flights to the United States.

```
// How many European airports have service to the USA?
g.V().has('continent','code','EU').
      out().as('a').
      out().has('country','US').
      select('a').dedup().count()

62
```

Just as we did for the airports in the US, we can figure out the distribution of routes for the European airports.

```
// What is the distribution of US routes amongst
// the European airports?
g.V().has('continent','code','EU').
      out().as('a').
      out().has('country','US').
      select('a').groupCount().by('code').
      order(local).by(values,asc)
```

It appears that London Heathrow (LHR) offers the most US destinations and Frankfurt (FRA) the second most.

```
[TER:1,RIX:1,BRS:1,DSA:1,DBV:1,TFS:1,KRK:1,BLQ:1,ORK:1,KBP:1,DME:1,BEG:1,AGP:1,
 PMI:1,HAM:1,RZE:1,NAP:1,PRG:2,OPO:2,STR:2,NCE:2,BHX:2,BFS:3,GVA:3,VCE:3,VKO:3,
 BUD:3,PDL:3,STN:4,BGO:4,SVO:4,ORY:4,GLA:4,WAW:5,SNN:5,ATH:5,MXP:6,CGN:6,HEL:6,
 VIE:6,BER:7,BRU:8,OSL:8,EDI:9,ARN:9,LIS:9,DUS:11,IST:11,CPH:11,BCN:12,MAD:13,
 FCO:13,MAN:14,LGW:14,ZRH:15,MUC:18,DUB:18,AMS:22,CDG:25,KEF:26,FRA:28,LHR:30]
```

Lastly, we can find out what the list of routes flown is. For this example, we decided to just return 10 of the 435 routes. Note how the 'path' step returns all parts of the traversal, including the continent code 'EU'. We could remove that part of the result by adding a 'from' modulator as shown earlier in the "What vertices and edges did I visit? — Introducing 'path'" section.

```
// Selected routes from Europe to the USA.
g.V().has('continent','code','EU').
      out().out().
      has('country','US').
      path().by('code').
      limit(10)

[EU,KRK,ORD]
[EU,TFS,EWR]
[EU,PDL,BOS]
[EU,PDL,JFK]
```

```
[EU,PDL,EWR]
[EU,DSA,SFB]
[EU,TER,BOS]
[EU,RZE,EWR]
[EU,LHR,PIT]
[EU,LHR,PDX]
```

5.1.17. Using 'fold' to do simple Map-Reduce computations

Earlier in the book we saw examples of 'sum' being used to count a collection of values. You can also use 'fold' to do something similar but in a more 'map-reduce' type of fashion.

First, here is a query that uses 'fold' in a way that we have already seen. It will find all routes from Austin and uses a 'fold' step to return a list of those names.

```
g.V().has('code','AUS').
      out('route').
      values('city').fold()
```

As expected, the results show all the cities that you can fly to from Austin collected into a single list.

```
[Philadelphia,Portland,Detroit,Oaklahoma City,Ontario,Charlotte,Cancun,Memphis,
 Cincinnati,Indianapolis,Kansas City,Dallas,St Louis,Albuquerque,Milwaukee,Chicago,
 Omaha,Tulsa,Puerto Vallarta,Nassau,Little Rock,Jacksonville,Providence,Sacramento,
 Birmingham,Louisville,Buffalo,Boise,Lubbock,Panama City Beach,Harlingen,Reno,
 Columbus,Des Moines,Cozumel,Amarillo,Baton Rouge,Charleston,Guadalajara,
 Grand Rapids,Liberia,Pensacola,San José del Cabo,Knoxville,Valparaiso,
 Fayetteville/Springdale/,Hayden,Orlando,Burbank,Aspen,Bozeman,Branson,St
 Petersburg-Clearwater,Atlanta,Nashville,Boston,Baltimore,Washington D.C.,Dallas,
 Fort Lauderdale,Washington D.C.,Houston,New York,Los Angeles,Orlando,Miami,
 Minneapolis,Chicago,Phoenix,Raleigh,Seattle,San Francisco,San Jose,Tampa,San Diego,
 Long Beach,Santa Ana,Salt Lake City,Las Vegas,Denver,San Antonio,Toronto,Honolulu,
 Vancouver,New Orleans,London,Newark,London,Houston,Frankfurt,El Paso,Amsterdam,
 Cleveland,Calgary,Oakland,Mexico City,Tucson,Pittsburgh]
```

However, what if we wanted to reduce our results further? Take a look at the modified version of our query below. It finds all routes from Austin and looks at the names of the destination cities. However, rather than return all the names, this time the 'fold' step is used differently and effectively reduces the city names to a single value. That value being the total number of characters in all of those city names. We have seen 'fold' used elsewhere in the book, but this time we provide 'fold' with a parameter and a closure. The parameter is passed to the closure as the first variable and the name of the city as the second. The closure then adds the zero and the length of each name, effectively producing a running total.

```
g.V().has('code','AUS').out('route').values('city').
      fold(0) {a,b -> a + b.length()}
```

```
891
```

While this query will work as-is on TinkerGraph within the Gremlin Console, some graph systems are stricter about their type checking and sandboxing of Groovy closures. To be on the safe side, you can always explicitly type cast the closure as follows.

```
g.V().has('code','AUS').out('route').values('city').
                      fold(0) {a,b -> (int)a + ((String)b).length()}
```

The point of the previous example with 'fold' using a closure was to demonstrate a subtle feature of it in the context of reducing results. The more direct way to get the same result is with standard Gremlin steps:

```
g.V().has('code','AUS').out('route').values('city').
      length().sum()
```

5.1.18. Distribution of routes in the graph (mode and mean)

An example of a common question we might want to answer with a network graph, of which air-routes are an example, is "how are the routes in our graph distributed between airport vertices?". We can also use this same query to find the statistical 'mode' (most common number) for a set of routes.

Take a look at the next query that shows how we can do analysis on the distribution of routes throughout the graph. We are only interested in vertices that are airports, and for those vertices we want to count how many outgoing routes each airport has. We want to return the results as a set of ordered 'key:value' pairs where the key is the number of outgoing routes and the value is the number of airports that have that number of outgoing routes.

```
g.V().hasLabel('airport').
      groupCount().by(out('route').count()).
      order(local).by(values,desc)
```

When we run the query, we get back the results below. As the results are sorted in descending order by value, we can see that the 'mode' (most common) number of outgoing routes is actually just one route and that 786 airports have just one outgoing route. We can see that 654 airports have just two routes and so on. We can also see at the other end of the scale that one airport has 237 outgoing routes.

```
[1:777,2:649,3:357,4:234,5:149,6:140,7:100,8:73,9:64,10:59,11:55,12:49,13:43,16:36,
15:31,0:29,20:27,18:24,14:21,17:19,19:18,21:18,22:18,24:15,25:15,23:14,27:14,41:13,
34:12,36:12,26:11,35:11,33:10,38:10,45:10,28:9,32:9,57:9,67:9,42:8,44:8,46:8,49:8,
29:7,37:7,51:7,30:6,52:6,53:6,56:6,75:6,39:5,40:5,43:5,48:5,58:5,61:5,66:5,71:5,80:5,
```

```
94:5,31:4,47:4,54:4,55:4,60:4,62:4,63:4,68:4,74:4,78:4,84:4,91:4,116:4,59:3,69:3,
81:3,100:3,102:3,117:3,128:3,139:3,64:2,65:2,76:2,82:2,83:2,85:2,88:2,89:2,90:2,
92:2,95:2,96:2,103:2,104:2,106:2,115:2,118:2,123:2,127:2,129:2,130:2,141:2,143:2,
145:2,153:2,158:2,194:2,199:2,203:2,206:2,232:2,248:2,265:1,270:1,283:1,293:1,309:1,
310:1,72:1,73:1,77:1,79:1,86:1,87:1,97:1,98:1,99:1,110:1,111:1,112:1,113:1,119:1,
121:1,122:1,125:1,126:1,136:1,138:1,140:1,147:1,151:1,155:1,156:1,157:1,159:1,161:1,
165:1,168:1,169:1,173:1,174:1,175:1,180:1,181:1,184:1,185:1,187:1,195:1,196:1,201:1,
202:1,204:1,211:1,213:1,214:1,216:1,217:1,221:1,242:1,253:1]
```

We could change our query above, replacing 'out()' with '__.in()', and we could find out the distribution of incoming routes. Remembering that in an air route network there is not always a one to one equivalent number of outgoing to incoming routes due to the way airlines plan their routes.

Another change we could make to our query is to change the ordering to use the key field for each key:value pair and this time sort in ascending order.

```
g.V().hasLabel('airport').
      groupCount().by(out('route').count()).
      order(local).by(keys,asc)
```

When we run our query again, we get the results below. Looking at the data sorted this way helps some new interesting facts stand out. The most interesting thing we can immediately spot is that there are 16 airports that currently have no outgoing routes at all!

```
[0:29,1:777,2:649,3:357,4:234,5:149,6:140,7:100,8:73,9:64,10:59,11:55,12:49,13:43,
14:21,15:31,16:36,17:19,18:24,19:18,20:27,21:18,22:18,23:14,24:15,25:15,26:11,27:14,
28:9,29:7,30:6,31:4,32:9,33:10,34:12,35:11,36:12,37:7,38:10,39:5,40:5,41:13,42:8,
43:5,44:8,45:10,46:8,47:4,48:5,49:8,51:7,52:6,53:6,54:4,55:4,56:6,57:9,58:5,59:3,
60:4,61:5,62:4,63:4,64:2,65:2,66:5,67:9,68:4,69:3,71:5,72:1,73:1,74:4,75:6,76:2,
77:1,78:4,79:1,80:5,81:3,82:2,83:2,84:4,85:2,86:1,87:1,88:2,89:2,90:2,91:4,92:2,
94:5,95:2,96:2,97:1,98:1,99:1,100:3,102:3,103:2,104:2,106:2,110:1,111:1,112:1,113:1,
115:2,116:4,117:3,118:2,119:1,121:1,122:1,123:2,125:1,126:1,127:2,128:3,129:2,130:2,
136:1,138:1,139:3,140:1,141:2,143:2,145:2,147:1,151:1,153:2,155:1,156:1,157:1,158:2,
159:1,161:1,165:1,168:1,169:1,173:1,174:1,175:1,180:1,181:1,184:1,185:1,187:1,194:2,
195:1,196:1,199:2,201:1,202:1,203:2,204:1,206:2,211:1,213:1,214:1,216:1,217:1,221:1,
232:2,242:1,248:2,253:1,265:1,270:1,283:1,293:1,309:1,310:1]
```

If we wanted to find the statistical mean number of routes in the graph, we could easily write a query like the one below to tell us how many airports and outgoing routes in total there are in the graph.

```
g.V().hasLabel('airport').union(count(),out('route').count()).fold()

[3504,50637]
```

We could then use the Gremlin Console to do the division for us to calculate the mean.

```
gremlin> 50637/3504

14.4511986301
```

However, Gremlin also has a 'mean' step that we can take advantage of if we can figure out a way to use it in this case that will do the work for us. Take a look at the next query. The key thing to note here is the way 'local' has been used. This will cause Gremlin to essentially do what we did a bit more manually above. If we did not include 'local', the answer would just be the total number of outgoing routes as Gremlin would essentially calculate 50637/1. By using local we force Gremlin to, in essence, create an array containing the number of routes for each airport, add those values up, and divide by the number of elements in the array (the number of airports). We hope that makes sense. If it is confusing, try the query yourself on the Gremlin console with and without 'local' and try it without the 'mean' step. You will see all the interim values instead!

```
g.V().hasLabel('airport').local(out('route').count()).mean()

14.4511986301
```

So it seems there is an average of just over 12 outgoing routes per airport in the graph whichever way we decide to calculate it!

Now that we have a query figured out for calculating the average number of outgoing routes per airport, we can easily tweak it to do the same for incoming routes and combined, incoming and outgoing, routes.

```
// Average number of incoming routes
g.V().has('type','airport').local(__.in('route').count()).mean()

14.4511986301

// Average number of outgoing and incoming routes
g.V().has('type','airport').local(both('route').count()).mean()

28.902397260273972
```

5.1.19. How many routes are there from airports in London (UK)?

This next query can be used to figure out how many outgoing routes each of the airports classified as being in (or near) London, England has. Note that we first find all airports in England using 'has('region','GB-ENG')'. If we did not do this, we would pick up airports in other countries as well, such as London, Ontario, in Canada.

```
g.V().has('region','GB-ENG').has('city','London').
      group().by('code').by(out().count())
```

```
[LCY:51,LHR:221,LTN:130,STN:211,LGW:232]
```

Here is a twist on the above theme. How many places can we get to from London in two hops but not including flights that end up back in London? It turns out there are over 2,000 places! Notice how 'aggregate' is used to store the set of London airports as a collection that can be referenced later on in the query to help with ruling out any flights that would end up back in London.

```
// Leave from London, fly with one stop, not ending back in London, how many places?
g.V().has('region','GB-ENG').has('city','London').aggregate('lon').
      out().out().dedup().where(without('lon')).count()

2353
```

We could have written the previous query like this and avoided using 'aggregate', but to me, this feels more clumsy and somewhat repetitive.

```
g.V().has('region','GB-ENG').has('city','London').out().out().dedup().
      not(and(has('city','London'),has('region','GB-ENG'))).count()

2353
```

5.1.20. How many routes are there between airports in England?

We can use an 'aggregate' step to quite elegantly find the routes that exist between airports located in England. The following query finds all airports in England and orders them by airport code. It then collects those airports using an 'aggregate' step. Finally, the aggregated collection is used to find routes to airports also within the UK. Note that as written, this query finds routes in both directions if they exist.

```
// Flights within England
g.V().has('region','GB-ENG').order().by('code').aggregate('a').
      out().where(within('a')).path().by('code')
```

Here are the routes found when the query is run. We arranged the results into columns to save space.

```
[BHX,NCL]  [BRS,NCL]  [EMA,SOU]  [EXT,MAN]
[EXT,NQY]  [EXT,ISC]  [HUY,NWI]  [ISC,NQY]
[ISC,EXT]  [ISC,LEQ]  [LBA,LHR]  [LBA,SOU]
[LBA,NQY]  [LCY,MAN]  [LCY,NCL]  [LEQ,ISC]
[LGW,NCL]  [LGW,NQY]  [LHR,LBA]  [LHR,NCL]
[LHR,NQY]  [LHR,MAN]  [LPL,NQY]  [MAN,LHR]
[MAN,LCY]  [MAN,SOU]  [MAN,NQY]  [MAN,SEN]
[MAN,NWI]  [MAN,EXT]  [MME,NWI]  [NCL,SOU]
```

```
[NCL,BRS]  [NCL,BHX]  [NCL,LHR]  [NCL,LGW]
[NCL,LCY]  [NQY,SEN]  [NQY,EXT]  [NQY,ISC]
[NQY,LHR]  [NQY,LGW]  [NQY,MAN]  [NQY,LPL]
[NQY,LBA]  [NWI,MAN]  [NWI,HUY]  [NWI,MME]
[SEN,NQY]  [SEN,MAN]  [SOU,NCL]  [SOU,MAN]
[SOU,EMA]  [SOU,LBA]
```

Here is a different way the query can be written. This time a 'where' step is used to filter out the previously seen airport pairs so we only get routes in one direction.

```
g.V().has('region','GB-ENG').order().by('code').as('a').
      out().has('region','GB-ENG').as('b').
      where('a',lt('b')).by('code').
      path().by('code')
```

As you can see this time we only get each airport pair back once regardless of route direction.

```
[BHX,NCL]
[BRS,NCL]
[EMA,SOU]
[EXT,MAN]
[EXT,NQY]
[EXT,ISC]
[HUY,NWI]
[ISC,NQY]
[ISC,LEQ]
[LBA,LHR]
[LBA,SOU]
[LBA,NQY]
[LCY,MAN]
[LCY,NCL]
[LGW,NCL]
[LGW,NQY]
[LHR,NCL]
[LHR,NQY]
[LHR,MAN]
[LPL,NQY]
[MAN,SOU]
[MAN,NQY]
[MAN,SEN]
[MAN,NWI]
[MME,NWI]
[NCL,SOU]
[NQY,SEN]
```

5.1.21. What are the top ten airports by route count?

Earlier we calculated which airports have the most routes. These next three queries are of a similar nature but produce tables of the top ten airports in terms of incoming, outgoing, and overall routes. As mentioned before, because of the way some airlines route flights, the number of outgoing and incoming routes to an airport will not always be the same. For example, several KLM Airlines flights from Amsterdam to airports in Africa continue on to other African airports before returning to Amsterdam. As a result, there are more inbound routes from than outbound routes to these airports. In each example below, the 'project' step is used to generate the results in an easily readable form.

First, this query will find the ten airports with the most incoming routes, sorted in descending order.

```
// Find the top ten overall in terms of incoming routes
g.V().hasLabel('airport').
      order().by(__.in('route').count(),desc).limit(10).
      project('ap','routes').by('code').by(__.in('route').count())
```

So it seems that Frankfurt and Amsterdam are tied for the most incoming routes. It is worth remembering that the version of the graph used for the examples in the book represents a moment in time. If these queries are re-run at a later date on a newer version of the graph, the results will quite likely be different. This is because airlines regularly add, and in some cases drop, routes.

```
[ap:FRA,routes:310]
[ap:IST,routes:309]
[ap:CDG,routes:294]
[ap:AMS,routes:285]
[ap:MUC,routes:271]
[ap:ORD,routes:264]
[ap:DFW,routes:253]
[ap:PEK,routes:249]
[ap:DXB,routes:248]
[ap:ATL,routes:242]
```

Now let's do the same thing but for outgoing routes.

```
// Find the top ten overall in terms of outgoing routes
g.V().hasLabel('airport').
      order().by(out('route').count(),desc).limit(10).
      project('ap','routes').by('code').by(out('route').count())
```

This time Frankfurt has the most routes. This reinforces a point that we made earlier. The number of incoming and outgoing routes for a given airport will not always be the same. This is because of the way airlines route flights. Some flights continue to other destinations before returning, so there will not always be direct flights between airport pairs in both directions. We know as passengers

this is something that we find frustrating!

```
[ap:FRA,routes:310]
[ap:IST,routes:309]
[ap:CDG,routes:293]
[ap:AMS,routes:283]
[ap:MUC,routes:270]
[ap:ORD,routes:265]
[ap:DFW,routes:253]
[ap:DXB,routes:248]
[ap:PEK,routes:248]
[ap:ATL,routes:242]
```

Lastly, let's find the top ten airports ordered by the total number of incoming and outgoing routes that they have.

```
// Find the top ten overall in terms of total routes
g.V().hasLabel('airport').
      order().by(both('route').count(),desc).limit(10).
      project('ap','routes').by('code').by(both('route').count())
```

As expected, Frankfurt is the airport with the most overall routes. One small side note while on the topic of airline routes. The graph does not track the frequency at which any given route is operated. What this means is that the airport with the most routes is not necessarily also the busiest. This is because many routes are operated multiple times a day. We did not try to include this data in the graph as it changes too often for me to keep up with.

```
[ap:FRA,routes:620]
[ap:IST,routes:618]
[ap:CDG,routes:587]
[ap:AMS,routes:568]
[ap:MUC,routes:541]
[ap:ORD,routes:529]
[ap:DFW,routes:506]
[ap:PEK,routes:497]
[ap:DXB,routes:496]
[ap:ATL,routes:484]
```

5.1.22. Using 'local' while counting things

In some of the earlier sections we saw examples of 'local' scope being used. Here is another example of how 'local' scope can be used while counting things to achieve a desired result.

Take a look at the query below. It finds any airports that have six or more runways and then returns the airport's IATA code along with the number of runways it has.

```
g.V().has('airport','runways',gte(6)).values('code','runways').fold()
```

Here is the output from running the query. As you can see, what is returned is a list of airport codes with each followed by its runway count.

```
[BOS,6,DFW,7,ORD,7,DEN,6,DTW,6,AMS,6]
```

While the output returned by the previous query is not bad, it might be nice to have what is returned be a set of code and runway pairs each in its own list. We can achieve this result by having the 'fold' step applied to the interim or 'local' results of the query.

Take a look at the modified form of the query below. Part of the query is now wrapped inside of a 'local' step.

```
g.V().has('airport','runways',gte(6)).local(values('code','runways').fold())
```

Here is the output from running our modified form of the query. Each airport code and runway value pair is now in its own individual list.

```
[BOS,6]
[DFW,7]
[ORD,7]
[DEN,6]
[DTW,6]
[AMS,6]
```

5.1.23. How many vertices have no edges?

In the air-routes graph there are some vertices that have no outgoing edges, some that have no incoming edges, and a few, such as the vertex for the continent of Antarctica, that have neither. We can use some simple queries to count how many of each type exist.

```
// Vertices with no outgoing edges.
g.V().not(outE()).count()

36

// Vertices with no incoming edges.
g.V().not(inE()).count()

245

// Vertices with no edges.
g.V().not(bothE()).count()
```

```
7
```

The queries above provide a nice shorthand way of writing queries that we could also write using 'where' steps. As shown below.

```
g.V().where(outE().count().is(0)).count()

36
```

5.2. Where can I fly to from here?

In this section you will find some more examples of queries that explore different questions that describe: "Where can I fly to from here?".

5.2.1. Does any route exist between two airports?

We have already looked at different ways to discover the shortest routes (by hops) between two airports using queries such as the one below. This query looks for a single route between Austin (AUS) and the Canadian city of Peawanuck in Ontario. It just so happens that Peawanuck is one of the hardest places to get to from Austin in the whole air-routes graph. In fact, to get there requires six stops along the way! With that in mind, the query below can easily run out of memory or time out trying to find even a single route as there is so much work to do to analyze all the possible routes.

```
g.V().has('code','AUS').
      repeat(out().simplePath()).
        until(has('code','YPO')).
      limit(1).
      path().by('code')
```

In cases such as this we can take a slightly different approach that will let us know if any route exists and in fact will execute very efficiently. Note that this query cannot find multiple routes but is very useful when trying to answer the question "does any route exist?".

```
g.V().has('code','AUS').
      repeat(out().dedup()).
        until(has('code','YPO')).
      path().by('code')

[AUS,YYZ,YTS,YMO,YFA,ZKE,YAT,YPO]
```

The difference between the two queries is the addition of a 'dedup' step. This ensures we only ever visit any airport along the route once no matter how we got there. This is different from the 'simplePath' step which makes sure we do not loop back on ourselves during a traversal but allows us to visit the same airport multiple times so long as we got there using a different path. Even if we

were to make the task of finding a route harder by explicitly avoiding a specific stop, the execution remains very efficient due to the 'dedup' step being used.

```
g.V().has('code','AUS').
      repeat(out().has('code',without('YYZ')).dedup()).
        until(has('code','YPO')).
      path().by('code')

[AUS,MDW,YTZ,YTS,YMO,YFA,ZKE,YAT,YPO]
```

Keep in mind that this trick is only useful when trying to figure out if any route (or path) exists in a group between a pair of vertices. It cannot be used to find multiple routes but is still a very useful pattern to be aware of. You will see a variation of this technique used again in the "Looking for the journey requiring the most stops" section.

5.2.2. Where in the USA or Canada can I fly to from any airport in India?

This first query looks for routes to the USA or Canada from any airport in India.

```
// Where in the USA or Canada can I fly to from any airport in India?
g.V().has('country','code','IN').out().out().
      has('country',within('US','CA')).path().by('code')
```

Here are the results of running the query. Note that because we used the 'path' step, the country code for India "IN" is included in the output.

```
[IN,DEL,IAD]
[IN,DEL,JFK]
[IN,DEL,ORD]
[IN,DEL,SFO]
[IN,DEL,EWR]
[IN,DEL,YYZ]
[IN,DEL,YVR]
[IN,BOM,JFK]
[IN,BOM,EWR]
[IN,BOM,YYZ]
```

5.2.3. Which cities in Europe can I fly to from Fort Lauderdale in Florida?

This query looks for routes from Fort Lauderdale in Florida to cities in Europe.

```
// Where can I fly to in Europe from Ft. Lauderdale?
g.V().has('code','FLL').out().as('a').in('contains').
      has('code','EU').select('a').values('city')
```

Here are the results of running the query.

```
London
Paris
Barcelona
Madrid
Oslo
Stockholm
Copenhagen
```

5.2.4. Where can I fly to from Charlotte, to cities in Europe or South America?

This query looks for routes from Charlotte, North Carolina to cities in Europe or South America.

```
// Flights from Charlotte to airports in Europe or South America
g.V().has('code','CLT').out().as('a').in('contains').
      has('code',within('EU','SA')).select('a').by('code')
```

Here are the results of running the query.

```
LHR
CDG
FRA
DUB
FCO
BCN
MAD
MUC
GIG
GRU
```

5.2.5. Where in the United States can I fly from to airports in London?

This query looks for routes from any one of the five airports in the London area in the UK that end up in the United States.

```
// Where in the United States can I fly to non-stop from any of the
// airports in and around London in the UK?
g.V().has('airport','code',within('LHR','LCY','LGW','LTN','STN')).
      out().has('country','US').path().by('code')
```

Here are the results of running the query.

```
[LHR,PIT]  [LHR,PDX]  [LHR,CLT]
```

```
[LHR,CHS]  [LHR,ATL]  [LHR,AUS]
[LHR,BNA]  [LHR,BOS]  [LHR,BWI]
[LHR,DFW]  [LHR,IAD]  [LHR,IAH]
[LHR,JFK]  [LHR,LAX]  [LHR,MIA]
[LHR,MSP]  [LHR,ORD]  [LHR,PHX]
[LHR,RDU]  [LHR,SEA]  [LHR,SFO]
[LHR,SJC]  [LHR,SAN]  [LHR,SLC]
[LHR,LAS]  [LHR,DEN]  [LHR,MSY]
[LHR,EWR]  [LHR,PHL]  [LHR,DTW]
[LGW,AUS]  [LGW,BOS]  [LGW,FLL]
[LGW,JFK]  [LGW,LAX]  [LGW,MCO]
[LGW,MIA]  [LGW,ORD]  [LGW,SEA]
[LGW,SFO]  [LGW,TPA]  [LGW,LAS]
[LGW,DEN]  [LGW,OAK]  [STN,BOS]
[STN,IAD]  [STN,EWR]  [STN,SFB]
```

5.2.6. Where in New York state can I fly to from any of the airports in Texas?

This query looks for all the routes from any airport in Texas that end up in any of the New York state airports.

```
// Where in New York state can I fly to from any airport in Texas?
g.V().has('airport','region','US-TX').out().has('region','US-NY').path().by('code')
```

Here are the results of running the query.

```
[AUS,BUF]
[AUS,JFK]
[AUS,EWR]
[DFW,BUF]
[DFW,JFK]
[DFW,LGA]
[DFW,EWR]
[IAH,JFK]
[IAH,LGA]
[IAH,EWR]
[SAT,JFK]
[SAT,EWR]
[HOU,JFK]
[HOU,LGA]
[HOU,EWR]
[DAL,LGA]
```

5.2.7. Which cities in Mexico can I fly to from Denver?

This query looks for routes from Denver to anywhere in Mexico.

```
// Where in Mexico can I fly to from Denver?
g.V().has('code','DEN').out().has('country','MX').values('city')
```

Here are the results of running the query.

```
Mexico City
Cancun
Puerto Vallarta
Cozumel
Monterrey
Guadalajara
San José del Cabo
```

5.2.8. Which cities in Europe can I fly to from Delhi in India?

This query looks for routes from Delhi that go to any airport in Europe.

```
// Where in Europe can I fly to from Delhi?
g.V().has('code','DEL').out().as('a').in("contains").
      has('code','EU').select('a').by('city')
```

Here are the results of running the query.

```
London
Paris
Frankfurt
Helsinki
Rome
Amsterdam
Madrid
Vienna
Zurich
Brussels
Munich
Stockholm
Moscow
Milan
Istanbul
Copenhagen
Reykjavik
Kiev
Birmingham
```

5.2.9. Finding all routes between London, Munich, and Paris

In the following example we find all the routes between airports in London, Munich, and Paris. Notice how by using 'aggregate' to collect the results of the first 'within' test that we don't have to repeat the names in the second 'within', we can just refer to the aggregated collection.

```
g.V().has('city',within('London','Munich','Paris')).aggregate('a').out().
      where(within('a')).path().by('code')
```

Here is what we might get back from the query. We have laid the results out in columns to save space.

```
[LHR,ORY] [LHR,CDG] [LHR,MUC] [LGW,CDG]
[LGW,MUC] [CDG,LTN] [CDG,LHR] [CDG,LGW]
[CDG,MUC] [CDG,LCY] [MUC,LTN] [MUC,LHR]
[MUC,LGW] [MUC,CDG] [MUC,LCY] [MUC,STN]
[MUC,ORY] [LCY,CDG] [LCY,MUC] [LCY,ORY]
[STN,MUC] [ORY,LHR] [ORY,MUC] [ORY,LCY]
[LTN,CDG] [LTN,MUC] [LTN,ORY]
```

5.3. More analysis of distances between airports

In this section you will find some more queries that examine distances between airports. The query below returns a nice list of all the routes from Austin (AUS) along with their distances. The results are sorted in ascending order by distance.

```
// Distances of all routes from AUS along with destination IATA CODE
g.V().has('code','AUS').outE().order().by('dist',asc).
      inV().path().by('code').by('dist')
```

Here are the results of running the query. For ease of reading we again broke the results into four columns.

```
[AUS,66,SAT]   [AUS,142,IAH]  [AUS,152,HOU]  [AUS,189,DAL]  [AUS,190,DFW]
[AUS,274,HRL]  [AUS,341,LBB]  [AUS,359,OKC]  [AUS,389,BTR]  [AUS,419,AMA]
[AUS,427,TUL]  [AUS,444,MSY]  [AUS,446,LIT]  [AUS,463,XNA]  [AUS,508,BKG]
[AUS,527,ELP]  [AUS,558,MEM]  [AUS,618,ABQ]  [AUS,625,PNS]  [AUS,650,MCI]
[AUS,664,VPS]  [AUS,681,BHM]  [AUS,708,ECP]  [AUS,722,STL]  [AUS,748,MEX]
[AUS,755,GDL]  [AUS,755,BNA]  [AUS,768,DEN]  [AUS,773,OMA]  [AUS,795,TUS]
[AUS,809,PVR]  [AUS,809,ATL]  [AUS,812,ASE]  [AUS,814,DSM]  [AUS,866,PHX]
[AUS,875,SDF]  [AUS,881,TYS]  [AUS,887,SJD]  [AUS,890,HDN]  [AUS,917,PIE]
[AUS,919,IND]  [AUS,922,CUN]  [AUS,925,TPA]  [AUS,945,CZM]  [AUS,952,JAX]
[AUS,957,CVG]  [AUS,972,MDW]  [AUS,973,ORD]  [AUS,992,SFB]  [AUS,994,MCO]
[AUS,1030,CLT] [AUS,1032,MKE] [AUS,1040,MSP] [AUS,1053,CHS] [AUS,1072,CMH]
[AUS,1080,SLC] [AUS,1080,LAS] [AUS,1101,MIA] [AUS,1103,GRR] [AUS,1110,FLL]
[AUS,1140,DTW] [AUS,1159,RDU] [AUS,1160,SAN] [AUS,1173,CLE] [AUS,1193,ONT]
```

```
[AUS,1206,SNA] [AUS,1209,PIT] [AUS,1220,LGB] [AUS,1230,LAX] [AUS,1238,BUR]
[AUS,1284,NAS] [AUS,1294,IAD] [AUS,1298,BZN] [AUS,1313,DCA] [AUS,1339,BWI]
[AUS,1357,YYZ] [AUS,1364,BUF] [AUS,1373,BOI] [AUS,1402,RNO] [AUS,1430,PHL]
[AUS,1476,SJC] [AUS,1477,SMF] [AUS,1493,OAK] [AUS,1500,SFO] [AUS,1500,EWR]
[AUS,1520,JFK] [AUS,1562,LIR] [AUS,1660,PVD] [AUS,1671,YYC] [AUS,1690,BOS]
[AUS,1712,PDX] [AUS,1768,SEA] [AUS,1869,YVR] [AUS,3755,HNL] [AUS,4901,LHR]
[AUS,4921,LGW] [AUS,5074,AMS] [AUS,5294,FRA]
```

This query finds all routes from DFW that are longer than 4,000 miles and returns the airport codes and the distances. Notice the use of two 'by' modulators in this query to decide which values are returned from the source vertex, the edge, and the destination vertex respectively. Also note that only two were specified but three values are returned. This works because 'by' is processed in a round-robin fashion if there are more values than 'by' modulators.

```
// Where can I fly to from DFW that is more than 4,000 miles away?
g.V().has('code','DFW').outE('route').has('dist',gt(4000)).inV().
                        path().by('code').by('dist')
```

Here are the results of running the query.

```
[DFW,4736,LHR]
[DFW,4933,CDG]
[DFW,5127,FRA]
[DFW,5208,HEL]
[DFW,6410,NRT]
[DFW,8574,SYD]
[DFW,8022,DXB]
[DFW,4458,DUB]
[DFW,8105,HKG]
[DFW,6951,PEK]
[DFW,7332,PVG]
[DFW,5597,FCO]
[DFW,4905,AMS]
[DFW,4950,MAD]
[DFW,5313,MUC]
[DFW,7914,DOH]
[DFW,6822,ICN]
[DFW,5228,GIG]
[DFW,5119,GRU]
[DFW,5299,EZE]
[DFW,4884,SCL]
[DFW,6257,IST]
[DFW,8053,AUH]
[DFW,5015,DUS]
```

The previous results are not sorted in any way. We could modify the query to include an 'order' step so that the results are sorted in descending order by distance.

```
g.V().has('code','DFW').outE('route').has('dist',gt(4000)).
      order().by('dist',desc).inV().
      path().by('code').by('dist')
```

Here are the now sorted results.

```
[DFW,8574,SYD]
[DFW,8105,HKG]
[DFW,8053,AUH]
[DFW,8022,DXB]
[DFW,7914,DOH]
[DFW,7332,PVG]
[DFW,6951,PEK]
[DFW,6822,ICN]
[DFW,6410,NRT]
[DFW,6257,IST]
[DFW,5597,FCO]
[DFW,5313,MUC]
[DFW,5299,EZE]
[DFW,5228,GIG]
[DFW,5208,HEL]
[DFW,5127,FRA]
[DFW,5119,GRU]
[DFW,5015,DUS]
[DFW,4950,MAD]
[DFW,4933,CDG]
[DFW,4905,AMS]
[DFW,4884,SCL]
[DFW,4736,LHR]
[DFW,4458,DUB]
```

This next query also finds all routes longer than 4,000 miles but this time originating in London Gatwick. Note also the use of 'where' to query the edge distance. The 'has' form is simpler, but we show 'where' being used just to demonstrate an alternative way we could do it. Note that this query uses three 'by' modulators as each of the values returned is from a different property of the respective vertices and edges.

```
// Routes longer than 4,000 miles starting at LGW
g.V().has('code','LGW').outE().where(values('dist').is(gt(4000L))).
      inV().path().by('code').by('dist').by('city')
```

Here are the results from running the query.

```
[LGW,4921,Austin]               [LGW,4410,Fort Lauderdale]
[LGW,5463,Los Angeles]          [LGW,4341,Orlando]
[LGW,4429,Miami]                [LGW,4807,Seattle]
```

```
[LGW,5374,San Francisco]        [LGW,4416,Tampa]
[LGW,5236,Las Vegas]            [LGW,4678,Denver]
[LGW,5364,Oakland]              [LGW,4731,Vancouver]
[LGW,6751,Singapore]            [LGW,5982,Hong Kong]
[LGW,5070,Beijing]              [LGW,5745,Shanghai]
[LGW,4380,Calgary]              [LGW,5987,Cape Town]
[LGW,5736,Rio de Janeiro]       [LGW,6908,Buenos Aires]
[LGW,4680,Kingston]             [LGW,4953,Cancun]
[LGW,5399,Colombo]              [LGW,6080,Taipei]
[LGW,4197,Bridgetown]           [LGW,4076,St. George]
[LGW,4408,Port of Spain]        [LGW,4699,Montego Bay]
[LGW,4283,Punta Cana]           [LGW,5419,San Jose]
[LGW,4662,Havana]               [LGW,5156,Chengdu]
[LGW,6053,Port Louis]           [LGW,4222,Vieux Fort]
[LGW,5287,Malé]                 [LGW,4077,Kigali]
[LGW,4618,Varadero]             [LGW,5147,Tianjin]
[LGW,5303,Chongqing]            [LGW,6299,Langkawi]
[LGW,6008,Rayong]               [LGW,6264,Duong Dong]
```

This next query is similar to the previous ones. We look for any routes from DFW that are longer than 4,500 miles. However, there are a few differences in this query worthy of note. First, it uses the preferred 'has' technique again to test the distance, whereas in the previous query we used 'where'. Also, this time we just list the distance and the destination airport's code, and we sort the result using a 'sort'. We also use 'select' and 'as' rather than the perhaps more succinct 'path' and 'by' to show a different way of achieving effectively the same results. The 'path' and 'by' combination were introduced more recently into TinkerPop, and we find that to be a more convenient syntax to use most of the time, but both ways work and both have their benefits. We should also note that we show 'sort' being used here just to show a different way of ordering results, but in most cases we can re-write our query to use 'order'. This use of 'sort' requires a Groovy closure to be provided. This may not work when working with commercial graph databases that disallow closures. You will find several examples of 'order' being used throughout this book. The 'order' step is introduced in the "Sorting things – introducing 'order'" section. As much as possible, staying with the pure Gremlin syntax and avoiding closures is recommended.

```
// Routes from DFW that are over 4,500 miles in length.
// Sorted into ascending order
g.V().hasLabel('airport').has('code','DFW').outE().has('dist',gt(4500)).as('e').
      inV().as('c').select('e','c').by('dist').by('code').sort(){it.e}
```

In general, we find using 'path' rather than 'select' and 'as' to be cleaner. However, as discussed in the "A warning that path finding can be memory and CPU intensive" section, there are issues with 'path' consuming memory that you will likely run into with more complex queries so it is always good to have some options as to how you write your Gremlin queries. Here is the output produced by this query.

```
[e:4736,c:LHR]
[e:4884,c:SCL]
```

```
[e:4905,c:AMS]
[e:4933,c:CDG]
[e:4950,c:MAD]
[e:5015,c:DUS]
[e:5119,c:GRU]
[e:5127,c:FRA]
[e:5208,c:HEL]
[e:5228,c:GIG]
[e:5299,c:EZE]
[e:5313,c:MUC]
[e:5597,c:FCO]
[e:6257,c:IST]
[e:6410,c:NRT]
[e:6822,c:ICN]
[e:6951,c:PEK]
[e:7332,c:PVG]
[e:7914,c:DOH]
[e:8022,c:DXB]
[e:8053,c:AUH]
[e:8105,c:HKG]
[e:8574,c:SYD]
```

For the sake of completeness, here is the query re-written to use an 'order' step rather than a call to 'sort' as a post-processing step at the end of the query. In general, this is the recommended way of achieving sorted results.

```
g.V().hasLabel('airport').has('code','DFW').outE().has('dist',gt(4500)).
      order().by('dist').as('e').inV().as('c').
      select('e','c').by('dist').by('code')
```

Note that the results are ordered before being collected using the 'select' step. Here is the output from running the modified query

```
[e:4736,c:LHR]
[e:4884,c:SCL]
[e:4905,c:AMS]
[e:4933,c:CDG]
[e:4950,c:MAD]
[e:5015,c:DUS]
[e:5119,c:GRU]
[e:5127,c:FRA]
[e:5208,c:HEL]
[e:5228,c:GIG]
[e:5299,c:EZE]
[e:5313,c:MUC]
[e:5597,c:FCO]
[e:6257,c:IST]
[e:6410,c:NRT]
```

```
[e:6822,c:ICN]
[e:6951,c:PEK]
[e:7332,c:PVG]
[e:7914,c:DOH]
[e:8022,c:DXB]
[e:8053,c:AUH]
[e:8105,c:HKG]
[e:8574,c:SYD]
```

The query can also be written with the 'order' step coming after the 'select' step. As follows:

```
g.V().hasLabel('airport').has('code','DFW').outE().has('dist',gt(4500)).as('e').
  inV().as('c').select('e','c').by('dist').by('code').order().by(select('e'))
```

5.3.1. Finding routes longer than 8,000 miles

This next set of queries shows various ways of finding and presenting all routes longer than 8,000 miles. Each query improves upon the one before by adding some additional feature or using a step that simplifies the query. First, let's just find all routes longer than 8,000 miles. This will include routes in both directions between airport pairs.

```
// All routes longer than 8,000 miles
g.V().as('src').outE('route').
      has('dist',gt(8000)).inV().as('dest').
      select('src','dest').by('code')
```

Here is what our query produces. As before, we have arranged the output in columns to aid readability.

```
[src:ATL,dest:JNB]    [src:DFW,dest:SYD]    [src:DFW,dest:DXB]
[src:DFW,dest:HKG]    [src:DFW,dest:AUH]    [src:IAD,dest:HKG]
[src:IAH,dest:SYD]    [src:IAH,dest:DXB]    [src:IAH,dest:DOH]
[src:JFK,dest:SIN]    [src:JFK,dest:HKG]    [src:JFK,dest:MNL]
[src:LAX,dest:SIN]    [src:LAX,dest:DXB]    [src:LAX,dest:DOH]
[src:LAX,dest:AUH]    [src:LAX,dest:JED]    [src:LAX,dest:RUH]
[src:ORD,dest:AKL]    [src:SEA,dest:SIN]    [src:SFO,dest:SIN]
[src:SFO,dest:DXB]    [src:SFO,dest:AUH]    [src:EWR,dest:SIN]
[src:EWR,dest:HKG]    [src:YVR,dest:MEL]    [src:LHR,dest:PER]
[src:SYD,dest:DFW]    [src:SYD,dest:IAH]    [src:SIN,dest:JFK]
[src:SIN,dest:LAX]    [src:SIN,dest:SEA]    [src:SIN,dest:SFO]
[src:SIN,dest:EWR]    [src:MEL,dest:YVR]    [src:DXB,dest:DFW]
[src:DXB,dest:IAH]    [src:DXB,dest:LAX]    [src:DXB,dest:SFO]
[src:DXB,dest:AKL]    [src:HKG,dest:DFW]    [src:HKG,dest:IAD]
[src:HKG,dest:JFK]    [src:HKG,dest:EWR]    [src:PER,dest:LHR]
[src:AKL,dest:ORD]    [src:AKL,dest:DXB]    [src:AKL,dest:DOH]
[src:PEK,dest:PTY]    [src:MNL,dest:JFK]    [src:DOH,dest:IAH]
[src:DOH,dest:LAX]    [src:DOH,dest:AKL]    [src:JNB,dest:ATL]
```

```
[src:SCL,dest:TLV]    [src:MEX,dest:CAN]    [src:TLV,dest:SCL]
[src:AUH,dest:DFW]    [src:AUH,dest:LAX]    [src:AUH,dest:SFO]
[src:JED,dest:LAX]    [src:RUH,dest:LAX]    [src:CAN,dest:MEX]
[src:PTY,dest:PEK]
```

Now let's improve the query by including the distance of each route in the query results.

```
// Find routes longer than 8,000 miles.
// Include the distance in returned values.
g.V().as('src').
      outE().has('dist',gt(8000)).as('e').
      inV().as('dest').
      select('src','e','dest').by('code').by('dist')
```

Here is the modified output showing the distance between the airports.

```
[src:ATL,e:8434,dest:JNB]        [src:DFW,e:8574,dest:SYD]        [src:DFW,e:8022
,dest:DXB]
[src:DFW,e:8105,dest:HKG]        [src:DFW,e:8053,dest:AUH]        [src:IAD,e:8135
,dest:HKG]
[src:IAH,e:8591,dest:SYD]        [src:IAH,e:8150,dest:DXB]        [src:IAH,e:8030
,dest:DOH]
[src:JFK,e:9526,dest:SIN]        [src:JFK,e:8054,dest:HKG]        [src:JFK,e:8504
,dest:MNL]
[src:LAX,e:8756,dest:SIN]        [src:LAX,e:8321,dest:DXB]        [src:LAX,e:8287
,dest:DOH]
[src:LAX,e:8372,dest:AUH]        [src:LAX,e:8314,dest:JED]        [src:LAX,e:8246
,dest:RUH]
[src:ORD,e:8186,dest:AKL]        [src:SEA,e:8059,dest:SIN]        [src:SFO,e:8433
,dest:SIN]
[src:SFO,e:8085,dest:DXB]        [src:SFO,e:8139,dest:AUH]        [src:EWR,e:9523
,dest:SIN]
[src:EWR,e:8047,dest:HKG]        [src:YVR,e:8197,dest:MEL]        [src:LHR,e:9009
,dest:PER]
[src:SYD,e:8574,dest:DFW]        [src:SYD,e:8591,dest:IAH]        [src:SIN,e:9526
,dest:JFK]
[src:SIN,e:8756,dest:LAX]        [src:SIN,e:8059,dest:SEA]        [src:SIN,e:8433
,dest:SFO]
[src:SIN,e:9523,dest:EWR]        [src:MEL,e:8197,dest:YVR]        [src:DXB,e:8022
,dest:DFW]
[src:DXB,e:8150,dest:IAH]        [src:DXB,e:8321,dest:LAX]        [src:DXB,e:8085
,dest:SFO]
[src:DXB,e:8818,dest:AKL]        [src:HKG,e:8105,dest:DFW]        [src:HKG,e:8135
,dest:IAD]
[src:HKG,e:8054,dest:JFK]        [src:HKG,e:8047,dest:EWR]        [src:PER,e:9009
,dest:LHR]
[src:AKL,e:8186,dest:ORD]        [src:AKL,e:8818,dest:DXB]        [src:AKL,e:9025
,dest:DOH]
[src:PEK,e:8884,dest:PTY]        [src:MNL,e:8504,dest:JFK]        [src:DOH,e:8030
```

```
,dest:IAH]
[src:DOH,e:8287,dest:LAX]         [src:DOH,e:9025,dest:AKL]         [src:JNB,e:8434
,dest:ATL]
[src:SCL,e:8208,dest:TLV]         [src:MEX,e:8754,dest:CAN]         [src:TLV,e:8208
,dest:SCL]
[src:AUH,e:8053,dest:DFW]         [src:AUH,e:8372,dest:LAX]         [src:AUH,e:8139
,dest:SFO]
[src:JED,e:8314,dest:LAX]         [src:RUH,e:8246,dest:LAX]         [src:CAN,e:8754
,dest:MEX]
[src:PTY,e:8884,dest:PEK]
```

Next, let's simplify things a bit. While using 'as' and 'select' gets the job done, using 'path' and 'by' shortens the query and makes it more readable. As before, remember the warning that in some cases using 'path' can consume large amounts of memory. That should not be an issue for us here as we are writing a fairly simple query still.

```
// Note that this also changes the way the result is returned
g.V().outE().has('dist',gt(8000)).
       inV().path().by('code').by('dist')
```

The output, indeed, now looks more readable.

```
[ATL,8434,JNB]  [DFW,8574,SYD]  [DFW,8022,DXB]
[DFW,8105,HKG]  [DFW,8053,AUH]  [IAD,8135,HKG]
[IAH,8591,SYD]  [IAH,8150,DXB]  [IAH,8030,DOH]
[JFK,9526,SIN]  [JFK,8054,HKG]  [JFK,8504,MNL]
[LAX,8756,SIN]  [LAX,8321,DXB]  [LAX,8287,DOH]
[LAX,8372,AUH]  [LAX,8314,JED]  [LAX,8246,RUH]
[ORD,8186,AKL]  [SEA,8059,SIN]  [SFO,8433,SIN]
[SFO,8085,DXB]  [SFO,8139,AUH]  [EWR,9523,SIN]
[EWR,8047,HKG]  [YVR,8197,MEL]  [LHR,9009,PER]
[SYD,8574,DFW]  [SYD,8591,IAH]  [SIN,9526,JFK]
[SIN,8756,LAX]  [SIN,8059,SEA]  [SIN,8433,SFO]
[SIN,9523,EWR]  [MEL,8197,YVR]  [DXB,8022,DFW]
[DXB,8150,IAH]  [DXB,8321,LAX]  [DXB,8085,SFO]
[DXB,8818,AKL]  [HKG,8105,DFW]  [HKG,8135,IAD]
[HKG,8054,JFK]  [HKG,8047,EWR]  [PER,9009,LHR]
[AKL,8186,ORD]  [AKL,8818,DXB]  [AKL,9025,DOH]
[PEK,8884,PTY]  [MNL,8504,JFK]  [DOH,8030,IAH]
[DOH,8287,LAX]  [DOH,9025,AKL]  [JNB,8434,ATL]
[SCL,8208,TLV]  [MEX,8754,CAN]  [TLV,8208,SCL]
[AUH,8053,DFW]  [AUH,8372,LAX]  [AUH,8139,SFO]
[JED,8314,LAX]  [RUH,8246,LAX]  [CAN,8754,MEX]
[PTY,8884,PEK]
```

Our query is looking pretty good, but it would be nice to not report the same route pair twice. In other words, the distance between two airports in just one direction is all we really want. This adds a little complexity to things as we have to find a way to 'filter' out the routes that we want to ignore.

One way to do this is to filter by making sure the code for the source airport is less than the code for the destination airport. This may seem a bit odd, but it is a way of saying we only want route pairs we have not already seen. Consider the case of the route between ATL and JNB. The less than ('lt') test works as we will allow the route between ATL and JNB through (as ATL is alphabetically less than JNB) but the route between JNB and ATL will be filtered out. This is a useful technique that can be very helpful in many situations where you are filtering out unwanted results. So let's modify the query as shown below to apply this filter.

```
// We could avoid returning both directions of
// travel by refining our query as follows.
g.V().as('a').outE().has('dist',gt(8000)).
      inV().as('b').
      filter(select('a','b').by('code').where('a', lt('b'))).
      path().by('code').by('dist')
```

Here is the output from running the modified query. As you can see, the route from ATL to JNB is still shown, but the reverse route from JNB to ATL is now gone. The same is true for all the other 'return journey' routes.

```
[ATL,8434,JNB]
[DFW,8574,SYD]
[DFW,8022,DXB]
[DFW,8105,HKG]
[IAH,8591,SYD]
[JFK,9526,SIN]
[JFK,8504,MNL]
[LAX,8756,SIN]
[LAX,8246,RUH]
[SEA,8059,SIN]
[SFO,8433,SIN]
[EWR,9523,SIN]
[EWR,8047,HKG]
[LHR,9009,PER]
[MEL,8197,YVR]
[DXB,8150,IAH]
[DXB,8321,LAX]
[DXB,8085,SFO]
[HKG,8135,IAD]
[HKG,8054,JFK]
[AKL,8186,ORD]
[AKL,8818,DXB]
[AKL,9025,DOH]
[PEK,8884,PTY]
[DOH,8030,IAH]
[DOH,8287,LAX]
[SCL,8208,TLV]
[AUH,8053,DFW]
[AUH,8372,LAX]
[AUH,8139,SFO]
```

```
[JED,8314,LAX]
[CAN,8754,MEX]
```

Lastly, now that we have the routes we want, let's tweak the query so that the routes are sorted by descending order of distance. We can do this by adding an 'order' step after finding the routes we are interested in.

```
// As above but sorted by route lengths.
g.V().as('a').outE().has('dist',gt(8000)).
      order().by('dist',desc).
      inV().as('b').
      filter(select('a','b').by('code').where('a', lt('b'))).
      path().by('code').by('dist')
```

Here are the results again, this time sorted by route distance in descending order.

```
[JFK,9526,SIN]
[EWR,9523,SIN]
[AKL,9025,DOH]
[LHR,9009,PER]
[PEK,8884,PTY]
[AKL,8818,DXB]
[LAX,8756,SIN]
[CAN,8754,MEX]
[IAH,8591,SYD]
[DFW,8574,SYD]
[JFK,8504,MNL]
[ATL,8434,JNB]
[SFO,8433,SIN]
[AUH,8372,LAX]
[DXB,8321,LAX]
[JED,8314,LAX]
[DOH,8287,LAX]
[LAX,8246,RUH]
[SCL,8208,TLV]
[MEL,8197,YVR]
[AKL,8186,ORD]
[DXB,8150,IAH]
[AUH,8139,SFO]
[HKG,8135,IAD]
[DFW,8105,HKG]
[DXB,8085,SFO]
[SEA,8059,SIN]
[HKG,8054,JFK]
[AUH,8053,DFW]
[EWR,8047,HKG]
[DOH,8030,IAH]
[DFW,8022,DXB]
```

The 'where' step can be followed by a 'by' modulator. This allows us, should we so desire, to simplify our query a bit more as follows.

```
//Query changed to take advantage of the where().by() construct
g.V().as('s').
      outE().has('dist',gt(8000)).
      order().by('dist',desc).inV().as('f').
      where('f',lt('s')).by('code').
      path().by('code').by('dist')
```

As you can see, we got the same results from our modified query. Well, almost the same results! Notice that we actually got the 'return routes' returned. So, for example, we got the route from DXB to AKL and not the route from AKL to DXB that we got in the prior case. This is because we compared the values in the opposite order!

```
[SIN,9526,JFK]
[SIN,9523,EWR]
[DOH,9025,AKL]
[PER,9009,LHR]
[PTY,8884,PEK]
[DXB,8818,AKL]
[SIN,8756,LAX]
[MEX,8754,CAN]
[SYD,8591,IAH]
[SYD,8574,DFW]
[MNL,8504,JFK]
[JNB,8434,ATL]
[SIN,8433,SFO]
[LAX,8372,AUH]
[LAX,8321,DXB]
[LAX,8314,JED]
[LAX,8287,DOH]
[RUH,8246,LAX]
[TLV,8208,SCL]
[YVR,8197,MEL]
[ORD,8186,AKL]
[IAH,8150,DXB]
[SFO,8139,AUH]
[IAD,8135,HKG]
[HKG,8105,DFW]
[SFO,8085,DXB]
[SIN,8059,SEA]
[JFK,8054,HKG]
[DFW,8053,AUH]
[HKG,8047,EWR]
[IAH,8030,DOH]
[DXB,8022,DFW]
```

As an interesting side note, and this would be true for the queries above as well, if we replace the 'lt' with a 'gt', we will get the routes returned in the reverse order.

```
g.V().as('s').
      outE().has('dist',gt(8000)).
      order().by('dist',desc).inV().as('f').
      where('f',gt('s')).by('code').
      path().by('code').by('dist')
```

Using the prior query, the first result was '[DOH,9025,AKL]'. As you will see below, our first result is now '[AKL,9025,DOH]'.

```
[JFK,9526,SIN]
[EWR,9523,SIN]
[AKL,9025,DOH]
[LHR,9009,PER]
[PEK,8884,PTY]
[AKL,8818,DXB]
[LAX,8756,SIN]
[CAN,8754,MEX]
[IAH,8591,SYD]
[DFW,8574,SYD]
[JFK,8504,MNL]
[ATL,8434,JNB]
[SFO,8433,SIN]
[AUH,8372,LAX]
[DXB,8321,LAX]
[JED,8314,LAX]
[DOH,8287,LAX]
[LAX,8246,RUH]
[SCL,8208,TLV]
[MEL,8197,YVR]
[AKL,8186,ORD]
[DXB,8150,IAH]
[AUH,8139,SFO]
[HKG,8135,IAD]
[DFW,8105,HKG]
[DXB,8085,SFO]
[SEA,8059,SIN]
[HKG,8054,JFK]
[AUH,8053,DFW]
[EWR,8047,HKG]
[DOH,8030,IAH]
[DFW,8022,DXB]
```

5.3.2. Finding the 20 longest routes in the graph

We could write the query from the previous section a different way, looking at edges first and using

the 'project' step instead of using the 'as' and 'select' steps. While in general it is not recommended to start a query with 'g.E()' as there are typically a lot more edges than vertices in a graph, this does illustrate a useful pattern. Also, just to change things a bit, this time we just look for the 20 longest routes rather than looking for routes longer than 8,000 miles. We again filter the results to only show the route in one direction. Note that the limit step is passed a parameter of 40 rather than 20 as we know we will be filtering out the same route in the return direction, so we will actually only get 20 routes back.

```
g.E().hasLabel('route').
      order().by('dist',desc).limit(40).
      project('a','b','c').
        by(inV().values('code')).
        by('dist').
        by(outV().values('code')).
      filter(select('a','c')).where('a',lt('c'))
```

Here are the results from running the query.

```
[a:JFK,b:9526,c:SIN]
[a:EWR,b:9523,c:SIN]
[a:AKL,b:9025,c:DOH]
[a:LHR,b:9009,c:PER]
[a:PEK,b:8884,c:PTY]
[a:AKL,b:8818,c:DXB]
[a:LAX,b:8756,c:SIN]
[a:CAN,b:8754,c:MEX]
[a:IAH,b:8591,c:SYD]
[a:DFW,b:8574,c:SYD]
[a:JFK,b:8504,c:MNL]
[a:ATL,b:8434,c:JNB]
[a:SFO,b:8433,c:SIN]
[a:AUH,b:8372,c:LAX]
[a:DXB,b:8321,c:LAX]
[a:JED,b:8314,c:LAX]
[a:DOH,b:8287,c:LAX]
[a:LAX,b:8246,c:RUH]
[a:SCL,b:8208,c:TLV]
[a:MEL,b:8197,c:YVR]
```

For completeness, here is the query rewritten slightly to again start with airport vertices rather than with all 'route' edges but using 'as' and 'select' steps rather than using a 'path' step as used in the previous section. A 'where' step followed by a 'by' modulator is still used in this case. So once again, it is clear there are often many ways to get the results that you are looking for. The key is to think about which form of a query will be the most effective for the data that you are working with.

```
g.V().hasLabel('airport').as('a').
      outE().as('b').
```

```
      order().by('dist',desc).limit(40).
      inV().as('c').
      where('a',lt('c')).by('code').
      select('a','b','c').by('code').by('dist')
```

When the query is run, the results look like the ones from the prior query.

```
[a:JFK,b:9526,c:SIN]
[a:EWR,b:9523,c:SIN]
[a:AKL,b:9025,c:DOH]
[a:LHR,b:9009,c:PER]
[a:PEK,b:8884,c:PTY]
[a:AKL,b:8818,c:DXB]
[a:LAX,b:8756,c:SIN]
[a:CAN,b:8754,c:MEX]
[a:IAH,b:8591,c:SYD]
[a:DFW,b:8574,c:SYD]
[a:JFK,b:8504,c:MNL]
[a:ATL,b:8434,c:JNB]
[a:SFO,b:8433,c:SIN]
[a:AUH,b:8372,c:LAX]
[a:DXB,b:8321,c:LAX]
[a:JED,b:8314,c:LAX]
[a:DOH,b:8287,c:LAX]
[a:LAX,b:8246,c:RUH]
[a:SCL,b:8208,c:TLV]
[a:MEL,b:8197,c:YVR]
```

5.3.3. Finding the longest route from each airport

The following query can be used to find the longest route from each airport in the graph. We included a 'limit' step so that just a few results are returned, but if you were to remove it, then every airport in the graph would be represented in the query results. A local step is used so that for each airport all of its outgoing routes are analyzed. Those routes are ordered in descending order, and only the first one is selected in each case.

```
g.V().hasLabel('airport').limit(10).
      local(outE().
            order().by('dist',desc).
            inV().
            path().
              by('code').
              by('dist').
              limit(1))
```

When run, the query produces the following results.

```
[ATL,8434,JNB]
[ANC,3368,KEF]
[AUS,5294,FRA]
[BNA,4168,LHR]
[BOS,7952,HKG]
[BWI,3622,LHR]
[DCA,2434,SFO]
[DFW,8574,SYD]
[FLL,7808,DXB]
[IAD,8135,HKG]
```

5.3.4. Combining 'aggregate', 'union' and 'filter' to compute distances

This next query is similar to the one we looked at in the "Finding all routes between London, Munich, and Paris" section. It uses 'aggregate', 'union', 'filter,' and 'where' to factor certain airports in and out of a query. We find all airports in London, Munich, and Paris and then count the total distance of all routes from those airports as the first half of a 'union'. The second half of the 'union' only counts the distances of routes that end up in one of our three selected cities.

```
g.V().has('city',within('London','Munich','Paris')).
      aggregate('a').
      outE().
      union(values('dist').sum(),
            filter(inV().where((within('a')))).
              values('dist').sum())
```

Here are the results from running the query. As expected, the first number is a lot bigger than the second one as it is the total distance of all routes from the selected airports, whereas the second number only reflects the total distance of all routes between those airports.

```
2739039
10282
```

5.3.5. More queries that analyze distances

Calculating the distance between two directly connected airports is very easy. All we have to do is look at the 'dist' property of the edge that connects them. We can do this using the 'select' and 'as' steps, or in more recent versions of TinkerPop we can use 'path' and 'by'. We prefer the latter technique. Both ways are shown in the examples below that find the distance between Austin and Mexico City

```
// Distance between the AUS and MEX airports
g.V().has('code','AUS').outE().as('e').
      inV().has('code','MEX').
      select('e').values('dist')
```

```
748
```

As above but using 'path' and 'by'.

```
// Distance between the AUS and MEX airports
g.V().has('code','AUS').outE().
      inV().has('code','MEX').
      path().by('code').by('dist')

[AUS, 748, MEX]
```

Here are some more queries that are based on the distance between airports. The first query calculates how many routes there are between 100 and 200 miles. As an exercise, If you remove the call to 'count', the query will list the routes. As you can see, there are a lot of them!

```
// Routes Between 100 and 200 miles in length
g.V().outE().has('dist',within(100..200)).
      inV().
      path().by('code').by('dist').
      count()

3237
```

The next query is similar to the previous one but only counts routes that are between airports located in the United States. As before, if you remove the call to 'count', the routes will be returned.

```
// Routes Between 100 and 200 miles in length, but only within the US.
g.V().has('airport','country','US').
      outE().has('dist',within(100..200)).
      inV().has('country','US').
      path().by('code').by('dist').
      count()

597
```

Lastly, this query returns a list of all the routes from San Antonio along with their distances. Some results are shown.

```
// Return a list of routes and their distances,
// starting from San Antonio (SAT)
g.V().has('code','SAT').
  outE('route').inV().
  path().by('code').by('dist')
```

The results of this query are as follows:

```
[SAT,1423,YYZ]  [SAT,698,MEX]   [SAT,1712,PDX]
[SAT,408,OKC]   [SAT,1163,ONT]  [SAT,1093,CLT]
[SAT,931,CUN]   [SAT,624,MEM]   [SAT,1023,CVG]
[SAT,707,MCI]   [SAT,248,DAL]   [SAT,786,STL]
[SAT,608,ABQ]   [SAT,1036,MDW]  [SAT,825,OMA]
[SAT,484,TUL]   [SAT,1005,JAX]  [SAT,732,COS]
[SAT,234,HRL]   [SAT,1138,CMH]  [SAT,135,CRP]
[SAT,278,MTY]   [SAT,692,GDL]   [SAT,1039,SFB]
[SAT,707,TLC]   [SAT,872,ATL]   [SAT,66,AUS]
[SAT,820,BNA]   [SAT,1410,BWI]  [SAT,248,DFW]
[SAT,1143,FLL]  [SAT,1360,IAD]  [SAT,190,IAH]
[SAT,1580,JFK]  [SAT,1210,LAX]  [SAT,1038,MCO]
[SAT,1140,MIA]  [SAT,1097,MSP]  [SAT,1040,ORD]
[SAT,841,PHX]   [SAT,1222,RDU]  [SAT,1772,SEA]
[SAT,1480,SFO]  [SAT,1451,SJC]  [SAT,970,TPA]
[SAT,1129,SAN]  [SAT,1174,SNA]  [SAT,1086,SLC]
[SAT,1069,LAS]  [SAT,795,DEN]   [SAT,493,MSY]
[SAT,1569,EWR]  [SAT,192,HOU]   [SAT,495,ELP]
[SAT,1239,CLE]  [SAT,1473,OAK]  [SAT,1490,PHL]
[SAT,1210,DTW]
```

5.3.6. How far is it from AUS to LHR with one stop?

The next query begins to address the question: "How far is it from AUS to LHR with one stop?". We can quite easily come up with a query given what we now know about Gremlin that will show us all possible options with one stop between AUS and LHR along with the respective route distances.

```
// Return all ways of getting from AUS to LHR with one stop.
// Include the distances between each of the airports in the
// query result.
g.V().has('airport','code','AUS').
      outE().inV().outE().inV().
      has('code','LHR').
      path().by('code').by('dist')
```

The output from running the query produces a nice set of data showing the starting, intermediate, and destination airports with the distances in miles between each. It would be nice, though, if we could find a way to show the total distance that we will have to travel in each case. That is the topic of the next section!

```
[AUS,1430,PHL,3533,LHR]         [AUS,1712,PDX,4896,LHR]
[AUS,1140,DTW,3753,LHR]         [AUS,1030,CLT,3980,LHR]
[AUS,1284,NAS,4332,LHR]         [AUS,1053,CHS,4053,LHR]
[AUS,809,ATL,4198,LHR]          [AUS,755,BNA,4168,LHR]
[AUS,1690,BOS,3254,LHR]         [AUS,1339,BWI,3622,LHR]
```

```
[AUS,190,DFW,4736,LHR]          [AUS,1294,IAD,3665,LHR]
[AUS,142,IAH,4820,LHR]          [AUS,1520,JFK,3440,LHR]
[AUS,1230,LAX,5439,LHR]         [AUS,1101,MIA,4414,LHR]
[AUS,1040,MSP,4001,LHR]         [AUS,973,ORD,3939,LHR]
[AUS,866,PHX,5255,LHR]          [AUS,1159,RDU,3860,LHR]
[AUS,1768,SEA,4783,LHR]         [AUS,1500,SFO,5350,LHR]
[AUS,1476,SJC,5352,LHR]         [AUS,1160,SAN,5469,LHR]
[AUS,1080,SLC,4850,LHR]         [AUS,1080,LAS,5213,LHR]
[AUS,768,DEN,4655,LHR]          [AUS,1357,YYZ,3544,LHR]
[AUS,1869,YVR,4706,LHR]         [AUS,444,MSY,4616,LHR]
[AUS,1500,EWR,3453,LHR]         [AUS,5294,FRA,406,LHR]
[AUS,5074,AMS,230,LHR]          [AUS,1671,YYC,4356,LHR]
[AUS,748,MEX,5529,LHR]          [AUS,1209,PIT,3707,LHR]
```

5.3.7. Using 'sack' to calculate the shortest AUS-LHR route with one stop

Consider a typical use case for air travel. We want to calculate the shortest distance we will have to travel to go from one airport to another with only one stop. Let's take a real example. What are the ten shortest distances from AUS to LHR with one stop on the way? Given what we know of Gremlin so far, we can pretty easily come up with a query that will give us routes from AUS to LHR with just one stop like we did in the previous section. However, what is not so obvious, and this is an area where the TinkerPop documentation is a little weak, is working out how to keep a running total of values as we traverse a graph. This is where the 'sack' step comes in. As you will hopefully recall from some of the earlier sections, a 'sack' allows us to define a place where we can put things as the graph traversal proceeds. We can give the sack an in initial value and can add to it during the traversal and then use it as part of the information that our query will return. Take a look at the rather lengthy query below. We have laid it out in a way that we hope makes it easy to read.

```
// Shortest distances from AUS to LHR with one stop
g.withSack(0).
  V().has('code','AUS').
      outE().
      sack(sum).by('dist').
      inV().
      outE().
      sack(sum).by('dist').
      inV().has('code','LHR').
      sack().
      order().by(asc).limit(10).
      path().
        by('code').
        by('dist').
        by('code').
        by('dist').
        by('code').
        by()
```

On the first line of the query, we initialize our sack with the value zero. During the query, each time

we take an outgoing edge, we add the distance value for that edge to the sack. We filter out routes in the normal way by only keeping destinations that are 'LHR'. After finding the routes we care about, we take the values that are stored in our sack and use them to sort our results in ascending order. Finally on we process the paths that we have taken. Note that the sack value is included as part of the path output and referenced using a 'by' modulator that has no parameter. Running the query will produce the following results:

```
[AUS,1140,DTW,3753,LHR,4893]
[AUS,1357,YYZ,3544,LHR,4901]
[AUS,973,ORD,3939,LHR,4912]
[AUS,1209,PIT,3707,LHR,4916]
[AUS,755,BNA,4168,LHR,4923]
[AUS,190,DFW,4736,LHR,4926]
[AUS,1690,BOS,3254,LHR,4944]
[AUS,1500,EWR,3453,LHR,4953]
[AUS,1294,IAD,3665,LHR,4959]
[AUS,1520,JFK,3440,LHR,4960]
```

We could add a little post-processing to our query to only output the airport codes and the total mileage. Given this is post-processing of a small data set, using a bit of in-line code does not feel too ugly here.

```
g.withSack(0).V().
  has('code','AUS').
  outE().
  sack(sum).by('dist').
  inV().
  outE().
  sack(sum).by('dist').
  inV().has('code','LHR').
  sack().
  order().by(asc).limit(10).
  path().
    by('code').
    by('dist').
    by('code').
    by('dist').
    by('code').
    by().
  toList().
  each(){println "${it[0]} --> ${it[2]} --> ${it[4]} ${it[5]} miles"}[];
```

Here is what our modified query, with the Groovy post-processing added, produces.

```
AUS --> DTW --> LHR 4893 miles
AUS --> YYZ --> LHR 4901 miles
AUS --> ORD --> LHR 4912 miles
```

```
AUS --> PIT --> LHR 4916 miles
AUS --> BNA --> LHR 4923 miles
AUS --> DFW --> LHR 4926 miles
AUS --> BOS --> LHR 4944 miles
AUS --> EWR --> LHR 4953 miles
AUS --> IAD --> LHR 4959 miles
AUS --> JFK --> LHR 4960 miles
```

5.3.8. Another example of how 'sack' can be used

The query below finds all airports with more than 200 routes and returns them as a map of airport code and route count pairs.

```
// Print a table of airports with more than 200 routes and the number of routes
g.V().hasLabel('airport').
      where(out().count().is(gt(200))).
      group().
        by('code').
        by(outE().count())
```

Here are the results that the query generates.

```
[ORD:265,PVG:213,LAX:214,CDG:293,STN:211,JFK:204,DFW:253,LHR:221,MUC:270,DME:232,
 EWR:203,AMS:283,IST:309,DEN:217,BCN:203,BER:202,DXB:248,MAD:206,FCO:201,VIE:206,
 FRA:310,PEK:248,ATL:242,MAN:216,LGW:232]
```

The previous query is a perfectly good way to achieve the desired result. Just for fun let's produce the same results using a 'sack'. This is intended just as an example of how 'sack' works and is not the way you would actually want to perform this specific query. That said, it is useful to have an example where the contents of the sack is more complex than a simple integer value. At the start of the query we initialize our sack with an empty map '[:]'. Later in the query, for each airport that has more than 200 outgoing routes, we update the map by adding the airport code and the route count to the map. Note that the value part of each map entry is produced using a traversal. So while this query is most definitely overkill for the task (as demonstrated by the much simpler query above), it does provide us with another example of how you can use sacks in powerful ways to store data as your query iterates.

```
// The same query but done using the sack() step in TinkerPop shown as an
// example only. The prior query works just fine for this.
g.withSack([:]).
  V().hasLabel('airport').
  where(out().count().is(gt(200))).
  sack{m,v -> m[v.value('code')]=g.V(v).out().count().next()}.
  fold().sack()
```

Here are the results that the new query generates. Other than the fact that the results came back in

a different order, they are the same.

```
[ATL:242,DFW:253,JFK:204,LAX:214,ORD:265,DEN:217,EWR:203,LHR:221,LGW:232,CDG:293,
 FRA:310,DXB:248,PEK:248,PVG:213,FCO:201,AMS:283,BCN:203,MAD:206,VIE:206,MUC:270,
 MAN:216,STN:211,DME:232,IST:309,BER:202]
```

The 'fold' step is needed in the query above to make sure that the result we get back is returned as a map. If we left the 'fold' off, we would just get the values stored in the sack returned as a list of integers. Note that while the list of airports is the same as the previous query, the order is different. This is a result of the way the 'group' step did its work in the previous query. Order should never be relied upon. If you need a specific order for the results of a query, it is always recommended to perform an explicit 'order' step as appropriate.

For completeness, a way of sorting the results of our original query by ascending route count (values) is shown below.

```
g.V().hasLabel('airport').
      where(out().count().is(gt(200))).
      group().by('code').by(outE().count()).
      order(local).by(values)

[FCO:201,BER:202,EWR:203,BCN:203,JFK:204,MAD:206,VIE:206,STN:211,PVG:213,LAX:214,
 MAN:216,DEN:217,LHR:221,DME:232,LGW:232,ATL:242,DXB:248,PEK:248,DFW:253,ORD:265,
 MUC:270,AMS:283,CDG:293,IST:309,FRA:310]
```

If you wanted to sort using the airport codes, you could do it as follows. This time we will sort the results of the 'sack' based query.

```
g.withSack([:]).
  V().hasLabel('airport').where(out().count().is(gt(200))).
  sack{m,v -> m[v.value('code')]=g.V(v).out().count().next()}.
  fold().
  sack().
  order(local).by(keys)

[AMS:283,ATL:242,BCN:203,BER:202,CDG:293,DEN:217,DFW:253,DME:232,DXB:248,EWR:203,
 FCO:201,FRA:310,IST:309,JFK:204,LAX:214,LGW:232,LHR:221,MAD:206,MAN:216,MUC:270,
 ORD:265,PEK:248,PVG:213,STN:211,VIE:206]
```

As well as using 'withSack' to initialize a 'sack', you can also use the 'assign' operator to do it. The query below uses a constant value of 0 to initialize the sack and then uses the sack to count the number of runways that the airports you can fly to from Austin have. At the end of the query we perform a 'sum' step against the sack which will contain a list holding the number of runways for each airport, so we need to add all of those to get a single grand total.

```
g.V().has('code','AUS').
```

```
      sack(assign).by(constant(0)).
      out().
      sack(sum).by('runways').
      sack().sum()

306
```

5.4. Using latitude, longitude and geographical region in queries

The 'air-routes' graph stores the geographic coordinates (latitude and longitude) of each airport as floating point numbers. Some graph systems such as JanusGraph have geographic coordinate and shape (geospatial) functions built in, but TinkerGraph does not. Moreover, GraphML does not offer any specific geospatial support. To keep things simple and flexible, the air-routes data set does not assume any specific back end capabilities. Having coordinates provided as basic floating point numbers still allows us to write some interesting geospatial queries.

So far, we have not taken advantage of these latitude and longitude coordinates in queries. In this section we have included some queries that perform interesting geospatial calculations using just the standard Gremlin steps.

Here is a simple query that finds any airports located North of 77 degrees latitude. Note that a 'where' step could be used instead of 'filter' as they are synonymous in this case.

```
g.V().filter(values('lat').is(gt(77))).valueMap('city','lat')
```

As discussed earlier in the book, you can often just use 'has' steps instead of 'where' or 'filter' steps. We could have written the previous query as follows.

```
g.V().has('lat',gt(77)).valueMap('city','lat')
```

It turns out that just two airports in the graph are located that far North as shown below in the results from running either form of the query.

```
[city:[Longyearbyen],lat:[78.2461013793945]]
[city:[Qaanaaq],lat:[77.4886016846]]
```

We could modify our query to look for airports with a latitude value outside a provided upper and lower bound, thus finding the most Northerly and Southerly airports with a single query. Here is a simple example of such a query.

```
g.V().has('lat',outside(-50,77)).order().by('lat',asc).valueMap('city','lat')
```

Here are the airports found by running the query.

```
[city:[Ushuahia],lat:[-54.8433]]
[city:[Rio Grande],lat:[-53.7777]]
[city:[Punta Arenas],lat:[-53.0026016235352]]
[city:[Mount Pleasant],lat:[-51.8227996826172]]
[city:[Stanley],lat:[-51.6856994628906]]
[city:[Puerto Natales],lat:[-51.671501159668]]
[city:[Rio Gallegos],lat:[-51.6089]]
[city:[El Calafate],lat:[-50.2803001404]]
[city:[Qaanaaq],lat:[77.4886016846]]
[city:[Longyearbyen],lat:[78.2461013793945]]
```

Note that while writing the above query it might have seemed appropriate to use a 'without' step with a range such as '-50..77' but that will not work as without looks for exact matches against the values in the range and the range generated is of the form '-50,-49,-48' and so on, so the only airports that would get filtered out would be the ones having a latitude value that exactly matches one of the values generated by the range. This is why the 'outside' step is so useful in cases like this.

This next query just returns the coordinates for London Heathrow.

```
// Query latitude and longitude for LHR
g.V().has('airport','code','LHR').valueMap('lat','lon')

[lon:[-0.461941003799], lat:[51.4706001282]]
```

This next query returns the code, latitude, and longitude for all airports in London, England. Note that because there are other cities in the world also called London, such as London, Ontario in Canada, we have to take advantage of the region code 'GB-ENG' to only return airports in London, England.

```
g.V().has('airport','city','London').has('region','GB-ENG').valueMap('code','lat'
,'lon')

[code:[LHR], lon:[-0.461941003799], lat:[51.4706001282]]
[code:[LGW], lon:[-0.190277993679047], lat:[51.1481018066406]]
[code:[LCY], lon:[0.055278], lat:[51.505278]]
[code:[STN], lon:[0.234999999404], lat:[51.8849983215]]
[code:[LTN], lon:[-0.368333011865616], lat:[51.874698638916]]
```

Now that we know how to query the geographic coordinates, we can write a query to find out which airports in the graph are very close to the Greenwich Meridian. In this case we will look for any airports that have a longitude between -0.1 and 0.1

```
// Which airports are very close to the Greenwich Meridian ?
g.V().hasLabel('airport').has('lon',between(-0.1,0.1)).valueMap('code','lon')

[code:[LCY], lon:[0.055278]]
```

```
[code:[LDE], lon:[-0.006438999902457]]
[code:[LEH], lon:[0.0880559980869293]]
[code:[CDT], lon:[0.0261109992862]]
```

This next query can be used to find out which airports are closest to the equator.

```
// Which airports are closest to the Equator ?
g.V().hasLabel('airport').has('lat',between(-0.1,0.1)).valueMap('code','lat')

[code:[EBB], lat:[0.0423859991133213]]
[code:[MDK], lat:[0.0226000007242]]
[code:[KIS], lat:[-0.0861390009522438]]
[code:[MCP], lat:[0.0506640002131]]
[code:[LGQ], lat:[0.0930560007691]]
```

The code below will find all the airports in the geographic area defined by a one-degree box around London Heathrow. Various graph databases may offer functionality to better support this type of query. It's important to be familiar with all the underlying features of your graph database in addition to what TinkerPop offers generally so that you can make the best design choice.

```
lat = g.V().has('code','LHR').values('lat').next()
lon = g.V().has('code','LHR').values('lon').next()

g.V().hasLabel('airport').where(and(
                          values('lon').is(between(lon-1,lon+1)),
                          values('lat').is(between(lat-1,lat+1)))).
                          valueMap('code','lat','lon')
```

As we have discussed earlier in this book, it is often possible to avoid use of 'and' step by chaining 'has' steps together. The code below is equivalent to the code above but avoids the use of 'where' and 'and'.

```
lat = g.V().has('code','LHR').values('lat').next()
lon = g.V().has('code','LHR').values('lon').next()

g.V().hasLabel('airport').has('lon',between(lon-1,lon+1)).
                          has('lat',between(lat-1,lat+1)).
                          valueMap('code','lat','lon')
```

Here is the output produced by running either of the snippets of code above inside the Gremlin Console.

```
[code:[LHR],lon:[-0.461941003799],lat:[51.4706001282]]
[code:[LGW],lon:[-0.190277993679047],lat:[51.1481018066406]]
[code:[LCY],lon:[0.055278],lat:[51.505278]]
[code:[STN],lon:[0.234999999404],lat:[51.8849983215]]
```

```
[code:[LTN],lon:[-0.368333011865616],lat:[51.874698638916]]
[code:[SOU],lon:[-1.35679996013641],lat:[50.9502983093262]]
```

In the "Testing values and ranges of values with P" section we came up with a query (shown below) to find routes with one stop between Austin and Las Vegas, using only airports in the United States or Canada and avoiding PHX and LAX for plane changes.

```
g.V().has('airport','code','AUS').out().
      has('country',within('US','CA')).
      has('code',without('PHX','LAX')).out().
      has('code','LAS').path().by('code')
```

Now that we know how to use the longitude and latitude coordinates stored in the air routes graph, we could, for fun, write this query a different way. If you take a look at the query below, you will see we have added a 'where' step. We still check that we are only looking at airports in the United States or Canada, but then, in the 'where' step, we further limit the airports we want to consider further by saying we are only interested if their longitude value is less than that of Austin. In other words, we only want to change planes at an airport that is to the West of Austin. This is actually an improvement on the previous query that would have returned routes that included plane changes in New York and Nashville among other places. With our new query, no airport that is East of Austin will be considered as a place to change planes.

```
// AUS to LAS with one stop but the stop has to be in the US or Canada
// and West of Austin while avoiding PHX and LAX.
g.V().has('airport','code','AUS').as('aus').out().
      has('country',within('US','CA')).
      where(lt('aus')).by('lon').
      has('code',without('PHX','LAX')).out().
      has('code','LAS').path().by('code')
```

Below you will find the output from running the query. If you know your airport codes, you will see that all of these airports are indeed to the West of Austin. We might want to improve the query even more, however, to factor in sensible nearby airports that are not to the West of Austin. For example, Dallas Fort Worth (DFW) is not included in the results as it is situated North of Austin but also a little to the East. We will leave it as an exercise to come up with a refinement to the query so that Dallas is also included!

```
[AUS,PDX,LAS]
[AUS,ONT,LAS]
[AUS,ABQ,LAS]
[AUS,SMF,LAS]
[AUS,BOI,LAS]
[AUS,LBB,LAS]
[AUS,RNO,LAS]
[AUS,AMA,LAS]
[AUS,BUR,LAS]
```

```
[AUS,BZN,LAS]
[AUS,SEA,LAS]
[AUS,SFO,LAS]
[AUS,SJC,LAS]
[AUS,SAN,LAS]
[AUS,LGB,LAS]
[AUS,SNA,LAS]
[AUS,SLC,LAS]
[AUS,DEN,LAS]
[AUS,SAT,LAS]
[AUS,HNL,LAS]
[AUS,YVR,LAS]
[AUS,ELP,LAS]
[AUS,YYC,LAS]
[AUS,OAK,LAS]
[AUS,TUS,LAS]
```

Here is one more example that is similar to the previous one. First, we store the longitude of the Dallas (DFW) airport in the variable 'dfw'. Then we use that variable to find routes to Las Vegas (LAS) from Austin (AUS) that have one stop but avoid Phoenix (PHX) and Los Angeles (LAX) and only have a plane change either in Dallas or to the West of Dallas. Note that we use 'lte' and not 'lt' when testing the longitude value. If we used 'lt' instead, that would also rule out Dallas as an option. This query, as in the prior one, has the effect of ruling out plane changes at airports further to the East of Austin than Dallas.

```
dfw = g.V().has('code','DFW').values('lon').next()

g.V().has('airport','code','AUS').as('aus').out().
      has('country',within('US','CA')).
      has('lon',lte(dfw)).
      has('code',without('PHX','LAX')).out().
      has('code','LAS').path().by('code')
```

Here is what we get back when we run the query. Still lots of choices even if we avoid LAX and PHX, it seems.

```
[AUS,PDX,LAS]
[AUS,OKC,LAS]
[AUS,ONT,LAS]
[AUS,ABQ,LAS]
[AUS,SMF,LAS]
[AUS,BOI,LAS]
[AUS,LBB,LAS]
[AUS,RNO,LAS]
[AUS,AMA,LAS]
[AUS,BUR,LAS]
[AUS,BZN,LAS]
[AUS,DFW,LAS]
```

```
[AUS,SEA,LAS]
[AUS,SFO,LAS]
[AUS,SJC,LAS]
[AUS,SAN,LAS]
[AUS,LGB,LAS]
[AUS,SNA,LAS]
[AUS,SLC,LAS]
[AUS,DEN,LAS]
[AUS,SAT,LAS]
[AUS,HNL,LAS]
[AUS,YVR,LAS]
[AUS,ELP,LAS]
[AUS,YYC,LAS]
[AUS,OAK,LAS]
[AUS,TUS,LAS]
```

Let's look at one last query that uses the 'region' property, present on all airport vertices in the graph.

The query below starts at the DFW airport then looks for all routes to airports also within the United States. Next, those airports are grouped by their region code and airport code. Finally, a few states are selected. The query ends with an 'unfold' step to make the results a bit more readable.

```
g.V().has('airport','code','DFW').out().has('country','US').
      group().by('region').by('code').
      select('US-CA','US-TX','US-FL','US-CO','US-IL').unfold()
```

Below you can see the query results. Each of the selected states is listed along with a list of airports reachable from DFW.

```
US-CA=[ONT, PSP, SMF, FAT, BUR, BFL, MRY, SBA, LAX, SFO, SJC, SAN, SNA, OAK]
US-TX=[LBB, HRL, MAF, CRP, ABI, ACT, CLL, BPT, AMA, BRO, GGG, GRK, LRD, MFE, SJT, SPS,
TYR, DRT, AUS, IAH, SAT, HOU, ELP]
US-FL=[RSW, TLH, EYW, JAX, ECP, PNS, VPS, SRQ, GNV, FLL, MCO, MIA, PBI, TPA]
US-CO=[COS, DRO, GJT, EGE, HDN, ASE, GUC, MTJ, DEN]
US-IL=[PIA, BMI, CMI, MLI, SPI, ORD]
```

In the next section we will look at a more complicated query that builds upon the examples above and performs distance calculations using the latitude and longitude coordinates present on the airport vertices.

5.4.1. Using the 'math' step to calculate Great Circle distances

If the latitude and longitude of two places on the Earth are known, the approximate distance between them can be calculated using the Haversine Great Circle Distance formula. The formula includes some trigonometrical calculations that can be done in Gremlin using the 'math' step.

If you are interested in better understanding the mathematics used by the Haversine formula, details can be found here.

In this section we have included two queries. The first calculates the distance between Austin and Dallas Fort Worth using a somewhat inflexible approach. This reflects our first attempt to create a query that computes Great Circle distances all in Gremlin. The airport codes are embedded in the query. Having got that working, we realized a more generic query would be better. The second example allows an arbitrary source and destination to be specified.

Let's take a look at the queries. At first, the number of steps used may look a bit daunting, but if you work your way through it section by section, it's actually fairly straightforward. The Haversine formula needs a couple of constant values. These are injected into the query using 'withSideEffect' steps. The first constant, 'rdeg' represents one degree in radians. This is the result from dividing PI by 180. The second constant 'gcmiles' represents the average radius of the Earth in miles. This allows for the imperfect spherical shape of the Earth where the radius varies closer to the poles.

In our first attempt at producing a query, we used a 'has' step to find the AUS and DFW airports and then produced a group with the airport codes being the keys and a projection of their latitude and longitude being the values. If we take just that part of the query and run it, this is what we get back.

```
g.withSideEffect("rdeg", 0.017453293).
  withSideEffect("gcmiles",3956).
  V().
  has('code',within('AUS','DFW')).
  group().
    by('code').
    by(project('lat','lon').
      by('lat').
      by('lon'))

[DFW:[lat:32.896800994873,lon:-97.0380020141602],
 AUS:[lat:30.1944999694824,lon:-97.66989868164]]
```

The Haversine formula requires us to calculate the differences between the latitude and longitude of the two coordinates. The next part of the query does that. A project step 'project('ladiff','lgdiff','lat1','lon1','lat2','lon2')' is used to collect all the values we need.

```
g.withSideEffect("rdeg", 0.017453293).
  withSideEffect("gcmiles",3956).
  V().
  has('code',within('AUS','DFW')).
  group().
    by('code').
    by(project('lat','lon').
      by('lat').
      by('lon')).
  as('grp').
  project('ladiff','lgdiff','lat1','lon1','lat2','lon2').
```

```
    by(project('la1','la2').
         by(select('AUS').select('lat')).
         by(select('DFW').select('lat')).
       math('(la2 - la1) * rdeg')).
    by(project('lg1','lg2').
         by(select('AUS').select('lon')).
         by(select('DFW').select('lon')).
       math('(lg2 - lg1) * rdeg')).
    by(select('grp').select('AUS').select('lat')).
    by(select('grp').select('AUS').select('lon')).
    by(select('grp').select('DFW').select('lat')).
    by(select('grp').select('DFW').select('lon'))
```

Here are the results. A map has been generated containing the differences as well as the original coordinates.

```
[ladiff:0.04716405157034253,
 lgdiff:0.011028683009581777,
 lat1:30.1944999694824,
 lon1:-97.6698989868164,
 lat2:32.896800994873,
 lon2:-97.0380020141602]
```

The purpose of collecting the values like this is so they can be fed into the 'math' step to complete the calculation. One key takeaway from this example is the way that a 'math' step can be fed values from a prior 'project' step. The final calculations perform the necessary trigonometry as required by the Haversine formula. Here is the complete version of our first attempt at solving this problem in Gremlin.

```
g.withSideEffect("rdeg", 0.017453293).
  withSideEffect("gcmiles",3956).
  V().
  has('code',within('AUS','DFW')).
  group().
    by('code').
    by(project('lat','lon').
      by('lat').
      by('lon')).
  as('grp').
  project('ladiff','lgdiff','lat1','lon1','lat2','lon2').
    by(project('la1','la2').
         by(select('AUS').select('lat')).
         by(select('DFW').select('lat')).
       math('(la2 - la1) * rdeg')).
    by(project('lg1','lg2').
         by(select('AUS').select('lon')).
         by(select('DFW').select('lon')).
       math('(lg2 - lg1) * rdeg')).
```

```
    by(select('grp').select('AUS').select('lat')).
    by(select('grp').select('AUS').select('lon')).
    by(select('grp').select('DFW').select('lat')).
    by(select('grp').select('DFW').select('lon')).
  math('(sin(ladiff/2))^2 + cos(lat1*rdeg) * cos(lat2*rdeg) * (sin(lgdiff/2))^2').
  math('gcmiles * (2 * asin(sqrt(_)))')
```

We can see that the distance from Austin to Dallas Fort Worth is approximately 190 miles.

```
190.2483140396514
```

A few adjustments are needed to make the query more flexible. We wanted to have a query where you just provide any two airport codes at the start. This makes it easy to parameterize the query when working with code. The main difference from our first attempt is that the source and target airports are located at the start and individually labeled. The `group` step is replaced by a `select` step and given a label of "grp". The remainder of the query refers to "grp" as needed.

```
g.withSideEffect("rdeg", 0.017453293).
  withSideEffect("gcmiles",3956).
  V().has('code','AUS').as('src').
  V().has('code','DFW').as('dst').
  select('src','dst').
    by(project('lat','lon').
         by('lat').
         by('lon')).
  as('grp').
  project('ladiff','lgdiff','lat1','lon1','lat2','lon2').
    by(project('la1','la2').
         by(select('grp').select('src').select('lat')).
         by(select('grp').select('dst').select('lat')).
       math('(la2 - la1) * rdeg')).
    by(project('lg1','lg2').
         by(select('grp').select('src').select('lon')).
         by(select('grp').select('dst').select('lon')).
       math('(lg2 - lg1) * rdeg')).
    by(select('grp').select('src').select('lat')).
    by(select('grp').select('src').select('lon')).
    by(select('grp').select('dst').select('lat')).
    by(select('grp').select('dst').select('lon')).
  math('(sin(ladiff/2))^2 + cos(lat1*rdeg) * cos(lat2*rdeg) * (sin(lgdiff/2))^2').
  math('gcmiles * (2 * asin(sqrt(_)))')
```

The result from the more general-purpose version of the query is the same as before.

```
190.2483140396514
```

The source code in this section comes from the 'great-circle.groovy' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/groovy.

We can go one step further and parameterize the query. Everything shown below can be done using the Gremlin Console, but it also represents the way you might use this query within an application. This time San Francisco and Tokyo Narita are used as the origin and destination.

```
start = 'SFO'
stop = 'NRT'

g.withSideEffect("rdeg", 0.017453293).
  withSideEffect("gcmiles",3956).
  V().has('code',start).as('src').
  V().has('code',stop).as('dst').
  select('src','dst').
    by(project('lat','lon').
         by('lat').
         by('lon')).
  as('grp').
  project('ladiff','lgdiff','lat1','lon1','lat2','lon2').
    by(project('la1','la2').
         by(select('grp').select('src').select('lat')).
         by(select('grp').select('dst').select('lat')).
       math('(la2 - la1) * rdeg')).
    by(project('lg1','lg2').
         by(select('grp').select('src').select('lon')).
         by(select('grp').select('dst').select('lon')).
       math('(lg2 - lg1) * rdeg')).
    by(select('grp').select('src').select('lat')).
    by(select('grp').select('src').select('lon')).
    by(select('grp').select('dst').select('lat')).
    by(select('grp').select('dst').select('lon')).
  math('(sin(ladiff/2))^2 + cos(lat1*rdeg) * cos(lat2*rdeg) * (sin(lgdiff/2))^2').
  math('gcmiles * (2 * asin(sqrt(_)))')

5108.80166113392
```

5.5. Finding routes that go via a specific airport

We have seen a number of examples that use a 'has' step to filter routes. The example below looks for five routes that start in Austin (AUS) and go via Dallas (DFW).

```
g.V().has('code','AUS').
      out().has('code','DFW').
      out().
      limit(5).
      path().by('code')
```

When run, the query, as expected, finds us five routes that start in AUS and stop at DFW on the way.

```
[AUS,DFW,YYZ]
[AUS,DFW,YVR]
[AUS,DFW,LHR]
[AUS,DFW,CDG]
[AUS,DFW,FRA]
```

This technique works well if we know which stop we want to be in DFW. However, we might want to write a query that can only return results that include a visit to a specific airport anywhere in the journey. Let's build an example in two stages. First, look at the query below. It looks for ten routes that start in Austin and end up in Edinburgh only stopping along the way in US or the UK airports.

```
g.V().
  has('code','AUS').
  repeat(out().simplePath().has('country',within('US','UK'))).
    until(has('code','EDI')).
  path().by('code').
  limit(10)
```

When run, we get the following results.

```
[AUS,PHL,EDI]
[AUS,PVD,EDI]
[AUS,BOS,EDI]
[AUS,IAD,EDI]
[AUS,JFK,EDI]
[AUS,MCO,EDI]
[AUS,LHR,EDI]
[AUS,EWR,EDI]
[AUS,LGW,EDI]
[AUS,PHL,SWF,EDI]
```

However, for the sake of making the example more interesting, let's assume that we only want to get back routes that go via Manchester (MAN). We can do this by adjusting the prior query to look for "MAN" as it traverses the graph and keep track, using a sack of paths that encounter Manchester along the way. Once we arrive at EDI, we use a 'where' step to filter out routes where the value of the sack is anything but "1". A value of "1" means we encountered Manchester along the way.

```
g.withSack(0).V().
  has('code','AUS').
  repeat(out().simplePath().has('country',within('US','UK')).
         choose(has('code','MAN'),sack(sum).by(constant(1)))).
    until(has('code','EDI')).
  where(sack().is(1)).
```

```
  path().by('code').
  limit(10)
```

When the modified query is run, each result includes MAN in the path.

```
[AUS,PHL,MAN,EDI]
[AUS,ATL,MAN,EDI]
[AUS,BOS,MAN,EDI]
[AUS,IAD,MAN,EDI]
[AUS,IAH,MAN,EDI]
[AUS,JFK,MAN,EDI]
[AUS,LAX,MAN,EDI]
[AUS,MCO,MAN,EDI]
[AUS,MIA,MAN,EDI]
[AUS,ORD,MAN,EDI]
```

If we increase the limit step to use a value such as 30, then you will start to see that MAN can appear in any position in the path.

```
[AUS,LHR,GLA,MAN,EDI]
[AUS,LHR,ABZ,MAN,EDI]
[AUS,LHR,BHD,MAN,EDI]
[AUS,LHR,INV,MAN,EDI]
```

5.6. Using 'store' and a 'sideEffect' to make a set of unique values

Take a look at the query below. All it does is return the city names for the airports that have IDs between 1 and 200 (inclusive). The list is sorted by ascending alphabetical order.

```
g.V().hasId(between(1,201)).values('city').order().fold()
```

And here are the names that get returned. If you look closely at the list, you will see that city names like 'Dallas' and 'London' appear more than once.

```
[Abu Dhabi,Addis Ababa,Albuquerque,Alicante,Alice Springs,Amsterdam,Anchorage,Athens,
 Atlanta,Auckland,Austin,Ayers Rock,Baltimore,Bangkok,Barcelona,Beijing,Belgrade,
 Bengaluru,Berlin,Bilbao,Bologna,Boston,Brisbane,Brussels,Budapest,Buenos Aires,
 Cairns,Cairo,Calgary,Calicut,Canberra,Cancun,Cape Town,Cedar Rapids,Charlotte,
 Chennai,Chicago,Chicago,Christchurch,Cincinnati,Cleveland,Cologne,Copenhagen,
 Dallas,Dallas,Denver,Detroit,Doha,Dubai,Dublin,Durban,Dusseldorf,Edinburgh,
 Edmonton,El Paso,Fairbanks,Fort Lauderdale,Fort Myers,Frankfurt,Geneva,Genoa,
 Glasgow,Gold Coast,Gothenburg,Hagta,Halifax,Hamburg,Harrison,Helsinki,
 Ho Chi Minh City,Hong Kong,Honolulu,Houston,Houston,Hyderabad,Ibiza,
 Indianapolis,Istanbul,Johannesburg,Kahului,Kansas City,Kingston,Kolkata,
```

```
Kuala Lumpur,Kuwait,Larnaca,Las Vegas,Lima,Liverpool,London,London,London,London,
Long Beach,Los Angeles,Luqa,Luxembourg,Madrid,Manama,Manchester,Manila,Maroochydore,
Melbourne,Memphis,Menorca,Mexico City,Miami,Milan,Milwaukee,Minneapolis,Mombasa,
Montevideo,Montreal,Moscow,Moscow,Mumbai,Munich,Nairobi,Nantes,Naples,Nashville,
New Delhi,New Orleans,New York,New York,Newark,Nice,Nottingham,Oaklahoma City,
Oakland,Omaha,Ontario,Orlando,Osaka,Oslo,Ottawa,Palm Springs,Paris,Paris,Perth,
Philadelphia,Phnom Penh,Phoenix,Pisa,Pittsburgh,Portland,Portland,Prague,
Puerto Vallarta,Raleigh,Rio de Janeiro,Rochester,Rochester,Rome,Salina,
Salt Lake City,San Antonio,San Diego,San Francisco,San Jose,San Juan,Santa Ana,
Santa Fe,Santiago,Sao Paulo,Seattle,Seoul,Shanghai,Shannon,Singapore,Sofia,St Louis,
St. John's,Stockholm,Stuttgart,Sydney,Tallahassee,Tampa,Tel Aviv,Tenerife,Tokyo,
Tokyo,Toronto,Tucson,Tulsa,Turin,Vancouver,Venice,Venice,Verona,Vienna,Warsaw,
Washington D.C.,Washington D.C.,Wellington,West Palm Beach,White Plains,Winnipeg,
Zagreb,Zurich]
```

Just to be sure, we can count how many city names we got back.

```
g.V().hasId(between(1,201)).values('city').order().count()

200
```

What would be nice is if the duplicate names only appeared once. There are other ways we could write this query such as using 'dedup,' but let's rewrite it using 'store' and a 'Set' as we think doing that demonstrates a capability quite well that is useful in more complex scenarios than this one. By starting a query using 'withSideEffect' we can set up a named place we can 'store' things into later in our query, and we can also give the store a type. In this case we chose to use a 'Set'. What this query does is store the City names of the first 200 vertices in the graph into a Set and then displays them. As we know from our first attempt, city names, like Dallas, appear more than once. However, if we look at the Set that we get back from our modified query (because by default Sets do not store duplicates), we will only see the names Dallas and London appearing once.

```
g.withSideEffect("x", [] as Set).V().hasId(between(1,201)).
  values('city').aggregate('x').cap('x').unfold().order().fold()
```

Here are the city names that we get back after running our modified query. You will notice that all the duplicate names are now gone.

```
[Abu Dhabi,Addis Ababa,Albuquerque,Alicante,Alice Springs,Amsterdam,Anchorage,
 Athens,Atlanta,Auckland,Austin,Ayers Rock,Baltimore,Bangkok,Barcelona,Beijing,
 Belgrade,Bengaluru,Berlin,Bilbao,Bologna,Boston,Brisbane,Brussels,Budapest,
 Buenos Aires,Cairns,Cairo,Calgary,Calicut,Canberra,Cancun,Cape Town,Cedar Rapids,
 Charlotte,Chennai,Chicago,Christchurch,Cincinnati,Cleveland,Cologne,Copenhagen,
 Dallas,Denver,Detroit,Doha,Dubai,Dublin,Durban,Dusseldorf,Edinburgh,Edmonton,
 El Paso,Fairbanks,Fort Lauderdale,Fort Myers,Frankfurt,Geneva,Genoa,Glasgow,
 Gold Coast,Gothenburg,Hagta,Halifax,Hamburg,Harrison,Helsinki,Ho Chi Minh City,
 Hong Kong,Honolulu,Houston,Hyderabad,Ibiza,Indianapolis,Istanbul,Johannesburg,
 Kahului,Kansas City,Kingston,Kolkata,Kuala Lumpur,Kuwait,Larnaca,Las Vegas,Lima,
```

```
Liverpool,London,Long Beach,Los Angeles,Luqa,Luxembourg,Madrid,Manama,Manchester,
Manila,Maroochydore,Melbourne,Memphis,Menorca,Mexico City,Miami,Milan,Milwaukee,
Minneapolis,Mombasa,Montevideo,Montreal,Moscow,Mumbai,Munich,Nairobi,Nantes,Naples,
Nashville,New Delhi,New Orleans,New York,Newark,Nice,Nottingham,Oaklahoma City,
Oakland,Omaha,Ontario,Orlando,Osaka,Oslo,Ottawa,Palm Springs,Paris,Perth,
Philadelphia,Phnom Penh,Phoenix,Pisa,Pittsburgh,Portland,Prague,Puerto Vallarta,
Raleigh,Rio de Janeiro,Rochester,Rome,Salina,Salt Lake City,San Antonio,San Diego,
San Francisco,San Jose,San Juan,Santa Ana,Santa Fe,Santiago,Sao Paulo,Seattle,
Seoul,Shanghai,Shannon,Singapore,Sofia,St Louis,St. John's,Stockholm,Stuttgart,
Sydney,Tallahassee,Tampa,Tel Aviv,Tenerife,Tokyo,Toronto,Tucson,Tulsa,Turin,
Vancouver,Venice,Verona,Vienna,Warsaw,Washington D.C.,Wellington,West Palm Beach,
White Plains,Winnipeg,Zagreb,Zurich]
```

And just to make sure we got fewer names back this time, let's count them again.

```
g.withSideEffect("x", [] as Set).V().hasId(between(1,201)).
  values('city').aggregate('x').cap('x').unfold().count()

186
```

As we mentioned at the start of this section, you could achieve this result other ways. For example, here is a version of the query that uses 'dedup' instead of 'store'.

```
g.V().hasId(between(1,201)).values('city').dedup().count()

186
```

This is clearly a simpler query in this case, but you will find cases where the example that uses 'withSideEffect' and 'store' will come in very handy, especially in cases where you want to store things into a Set or List from multiple parts of a traversal such as the one below that finds and counts all the unique city names across multiple hops from a starting airport.

```
g.withSideEffect("x", [] as Set).V().hasId(3).as('a').values('city').aggregate('x').
   select('a').out().as('b').values('city').aggregate('x').
   select('b').out().values('city').aggregate('x').cap('x').unfold().count()

977
```

Lastly, on this topic, let's look at one more interesting use case. It is quite common to want to get back from a query a collection of vertices and edges. This is often because we want to examine properties on both the vertices and the edges. Imagine a small graph that has the following relationships.

(A → B), (A → C), (A → D), (C → D), (C → E), (D → F)

The code below can be used to create this graph using the Gremlin console and TinkerGraph.

```
graph = TinkerGraph.open()
g = traversal().with(graph)

g.addV("A").as("a").
  addV("B").as("b").
  addV("C").as("c").
  addV("D").as("d").
  addV("E").as("e").
  addV("F").as("f").
  addE("knows").from("a").to("b").
  addE("knows").from("a").to("c").
  addE("knows").from("a").to("d").
  addE("knows").from("c").to("d").
  addE("knows").from("c").to("e").
  addE("knows").from("d").to("f")
```

We can see the IDs were allocated for each vertex by looking at the 'valueMap'.

```
g.V().valueMap(true)

[id:0,label:A]
[id:1,label:B]
[id:2,label:C]
[id:3,label:D]
[id:4,label:E]
[id:5,label:F]
```

Likewise, we can look at the edges to see what IDs each edge was given.

```
g.E()

e[6][0-knows->1]
e[7][0-knows->2]
e[8][0-knows->3]
e[9][2-knows->3]
e[10][2-knows->4]
e[11][3-knows->5]
```

Now that we have our test graph created, let's take a look at the query we will need to develop. The problem we want to solve is to start from vertex A, find all the edges that go out from vertex A and also all the vertices at the other ends of those edges. Lastly, we also want to find any edges between the vertices that are also connected to A bit ignore edges that connect to vertices that are not also connected to A. In simple terms we want a query that will return all the relationships except (C→E) and (D-F) as A is not connected to E or F.

Using the 'withSideEffect' pattern that we used earlier in this section, we can again develop a query that will collect for us the vertices and edges that we are interested in. We added line numbers to

make it easier to discuss what is going on, but please, note that these are not part of the query itself.

```
1: g.withSideEffect('x', [] as Set).
2:     V(0L).aggregate('x').
3:     bothE().local(aggregate('x')).
4:     otherV().local(aggregate('x')).
5:     aggregate('tgtlist').
6:     bothE().as('ref').otherV().where(within('tgtlist')).
7:     select('ref').aggregate('x').cap('x').unfold()
```

Let's look at the query above line by line.

1. Start the query and define 'x' as our, initially empty, Set.
2. Start at vertex 0 and 'aggregate' it into our set 'x'.
3. Aggregate all the edges connected to 'V(0)' into our set. We use 'local' here to force an eager evaluation of the stream.
4. Aggregate the vertices connected to 'V(0)' into our set.
5. Aggregate all of these target vertices into 'tgtlist'.
6. Find more edges but only remember them if they connect to vertices also connected to 'V(0)'.
7. Aggregate the edges we found in 'x' and finally return the set as the overall result the query. The 'unfold' just makes the output a little easier to read.

Here is the output we get from running the query. As you can see, the vertices and edges that we were not interested in have been correctly left out of the result set.

```
v[0]
e[5][0-knows->1]
v[1]
e[6][0-knows->2]
v[2]
e[7][0-knows->3]
v[3]
e[8][2-knows->3]
```

As you start to work with graphs and start to do more complex querying, this pattern of query based around 'withSideEffect' is extremely useful to keep in mind.

5.7. Making a copy of the DFW vertex

It is sometimes useful to be able to create a new vertex using the label and properties from one or more existing vertices. In this section We are going to present a small case study that shows how we investigated the creation of a query that could create a copy of the Dallas Fort Worth (DFW) airport vertex.

Creating a vertex using the label from another vertex is quite straightforward. The two queries

shown below use the label from the DFW Airport vertex as the label for a new vertex.

Our first thought was to do this as follows.

```
g.addV().property(label,V().has('code','DFW').label())
```

The query above does work, but it felt a bit cumbersome, and we also realized we were going to need a way to refer to the DFW vertex more than once, so we changed the query as follows and ran it.

```
g.V().has('code','DFW').as('dfw').
      addV().property(label, select('dfw').label())

v[61395]
```

If we examine the new vertex, we can see that it does indeed have the correct label.

```
g.V(61395).valueMap(true)

[label:airport,id:61395]
```

Now we need to expand the query to also copy the property key names and their values from the DFW vertex into our new vertex. Earlier in the book we looked at ways that a 'select' step combined with the 'keys' and 'values' keywords can be used to select values from a map. We can use that same technique here.

First, let's remind ourselves about the properties of the DFW airport.

```
g.V().has('code','DFW').properties()

vp[country->US]
vp[code->DFW]
vp[longest->13401]
vp[city->Dallas]
vp[elev->607]
vp[icao->KDFW]
vp[lon->-97.0380020141602]
vp[type->airport]
vp[region->US-TX]
vp[runways->7]
vp[lat->32.896800994873]
vp[desc->Dallas/Fort Worth...]
```

Let's also count how many properties there are listed above.

```
g.V().has('code','DFW').properties().count()

12
```

So we need to add some steps to our query that will copy the twelve keys and values from these twelve properties belonging to the DFW vertex into our new one.

Below is the first query we tried when thinking about how to do this. The goal of the query is to first capture the properties from the DFW airport and then to add them to the new vertex being created.

```
g.V().has('code','DFW').as('dfw').
      addV().property(label, select('dfw').label()).as('new').
             property(select('dfw').properties().key(),
                      select('dfw').properties().value())
```

When we ran it, we got this result which looked encouraging as a new vertex had clearly been created.

```
v[61396]
```

However, when we inspected our new vertex, we saw we had only managed to copy one of the twelve properties over. So it was time to delete that vertex and have a bit of a rethink.

```
g.V(61396).valueMap(true).unfold()

country=[US]
label=airport
id=61396

g.V(61396).drop()
```

What we realized was that our first attempt at copying the properties was only ever going to copy one property as that was essentially what we had asked it to do. If it's not quite clear why this is the case, hopefully it will become clearer once you look at the results from the profile step that are shown later in this section. We realized that what we needed to do was to first get all the DFW properties and then add them to the new vertex. So we changed the query as shown below.

```
g.V().has('code','DFW').as('dfw').
      addV().property(label, select('dfw').label()).as('new').
      select('dfw').properties().as('dfwprops').
      select('new').property(select('dfwprops').key(),
                             select('dfwprops').value())
```

When run, this is what we saw in the Gremlin console. This is more output than we wanted to get

from the query, but it has clearly done more work this time. If you count the number of rows below, you will find it also adds up to twelve. So this time we got one result per property it would appear.

```
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
v[61398]
```

In a moment we will explore a way to get rid of the unwanted output, but first let's examine the contents of the new vertex.

```
g.V(61398).valueMap(true).unfold()

country=[US]
code=[DFW]
longest=[13401]
city=[Dallas]
lon=[-97.0380020141602]
type=[airport]
elev=[607]
label=airport
icao=[KDFW]
id=61398
region=[US-TX]
runways=[7]
lat=[32.896800994873]
desc=[Dallas/Fort Worth International Airport]
```

So a large part of our original goal has now been achieved. A vertex has been created that is a copy in all but ID value of the original DFW vertex. So, what can be done about the unwanted list of vertices that came back as the results from the query?

First, let's take a look at what the 'profile' step can tell us about this query. Note that we deleted the new vertex using a 'drop' step before running this query, so it did not pick up both the old and new DFW vertices.

```
g.V(61398).drop()

g.V().has('code','DFW').as('dfw').
      addV().property(label, select('dfw').label()).as('new').
```

```
      select('dfw').properties().as('dfwprops').
      select('new').property(select('dfwprops').key(),
                             select('dfwprops').value()).profile()
```

If you look at the results below, you can see that for each property key/value pair, a traverser was spawned. This explains why we got twelve results back. This also explains why our first attempt at writing this query failed. If you were to profile that query, you would find that it only spawns one traverser rather than the twelve that we need to be successful.

```
Traversal Metrics      Step                      Count Traver-  Time   %Dur
                                                       sers      (ms)
=============================================================================
TinkerGraphStep(vertex,[code.eq(DFW)])@[dfw]         1     1    1.416    28.85
AddVertexStep({label=[[SelectOneStep(last,dfw),...   1     1    1.840    37.47
  SelectOneStep(last,dfw)                            1     1    0.171
  NoOpBarrierStep(2500)                              1     1    0.092
  LabelStep                                          1     1    0.059
SelectOneStep(last,dfw)                              1     1    0.132     2.70
NoOpBarrierStep(2500)                                1     1    0.032     0.66
PropertiesStep(property)@[dfwprops]                 12    12    0.094     1.92
SelectOneStep(last,new)                             12    12    0.127     2.60
NoOpBarrierStep(2500)                               12    12    0.065     1.34
AddPropertyStep({value=[[SelectOneStep(last,dfw...  12    12    1.200    24.45
  SelectOneStep(last,dfwprops)                      12    12    0.168
  NoOpBarrierStep(2500)                             12    12    0.064
  PropertyKeyStep                                   12    12    0.066
  SelectOneStep(last,dfwprops)                      12    12    0.134
  NoOpBarrierStep(2500)                             12    12    0.070
  PropertyValueStep                                 12    12    0.043
                                          >TOTAL     -     -    4.911
```

As you will hopefully recall from our discussion of the 'addV' and 'property' steps earlier in Chapter 4, a 'property' step returns the vertex that the property was added to and not the new property itself. This allows multiple 'addV' and 'property' steps to be chained together.

So we need a way to still create our new properties but not have the twelve traversers return any results to us. Hopefully your reaction to this is something like "This feels like a good place to introduce a 'sideEffect' step into the query". If that was your reaction, you are right!

Before running the new query, we need to clean up the work done by the prior one.

```
g.V().has('code','DFW')

v[8]
v[61411]

g.V(61411).drop()
```

So now let's wrap the creation of the properties into a 'sideEffect' step that should stop the unwanted results from being returned. The modified version of the query is shown below.

```
g.V().has('code','DFW').as('dfw').
      addV().property(label, select('dfw').label()).as('new').
      sideEffect(select('dfw').properties().as('dfwprops').
                 select('new').property(select('dfwprops').key(),
                                        select('dfwprops').value()))
```

When we run the modified version of the query, this is what we get back.

```
[61424]
```

This is a lot better as we now just got back the new vertex one time as a result. Let's double-check that the query has worked as intended.

```
g.V(61424).valueMap(true).unfold()

country=[US]
code=[DFW]
longest=[13401]
city=[Dallas]
lon=[-97.0380020141602]
type=[airport]
elev=[607]
label=airport
icao=[KDFW]
id=61424
region=[US-TX]
runways=[7]
lat=[32.896800994873]
desc=[Dallas/Fort Worth International Airport]
```

In this small case study we have tried to show how we were able to evolve a query in stages and also to adapt to unexpected outcomes along the way. You could use essentially the same technique shown in this section to also copy one or more edges if you needed to.

5.8. Modeling an ordered binary tree as a graph

You can, of course, model a tree structure as a graph. The following code will create a new graph containing an ordered binary tree. A graph like this is sometimes referred to as a 'connected acyclic' graph as there are no cycles in the graph. This means that once you leave a vertex, there is no other path you could take that will allow you to get there again. By contrast, the air-routes graph is an example of a 'cyclic' graph as there are clearly many ways to revisit vertices.

```
// Builds a small ordered Binary (BST) Tree
```

```
graph = TinkerGraph.open()
g = traversal().with(graph)
g.addV('root').property('data',9).as('root').
  addV('vertex').property('data',5).as('b').
  addV('vertex').property('data',2).as('c').
  addV('vertex').property('data',11).as('d').
  addV('vertex').property('data',15).as('e').
  addV('vertex').property('data',10).as('f').
  addV('vertex').property('data',1).as('g').
  addV('vertex').property('data',8).as('h').
  addV('vertex').property('data',22).as('i').
  addV('vertex').property('data',16).as('j').
  addE('left').from('root').to('b').
  addE('left').from('b').to('c').
  addE('right').from('root').to('d').
  addE('right').from('d').to('e').
  addE('right').from('e').to('i').
  addE('left').from('i').to('j').
  addE('left').from('d').to('f').
  addE('right').from('b').to('h').
  addE('left').from('c').to('g')
```

We could, of course, use the 'max' and the 'min' steps to find the largest and smallest values in the graph. However, the queries below show how we can do it using the semantics of an ordered binary tree.

```
// Find the largest value in the graph
g.V().hasLabel('root').repeat(out('right')).
                       until(out('right').count().is(0)).values('data')

22

// Find the smallest value in the graph
g.V().hasLabel('root').repeat(out('left')).
                       until(out('left').count().is(0)).values('data')

1
```

As a side note, here is a different way we could have written the query using 'not' instead of 'count'. As 'not' is a reserved word in Groovy, as we discussed in the "A warning about reserved word conflicts and collisions" section, we have to prefix it with the '__.' notation.

```
g.V().hasLabel('root').repeat(out('left')).
                       until(__.not(out('left'))).values('data')
```

If we wanted to see the values that the 'repeat' is encountering as it traverses the tree, we could add an 'emit' step and get the values of each vertex the 'repeat' visits. Note that this does not include the value from the root vertex.

```
g.V().hasLabel('root').repeat(out('left')).emit().values('data')

5
2
1
```

Perhaps a nicer way to look at all the values we encountered as we traversed the tree would be to use 'path' as follows. Note that this does include the root vertex's value.

```
g.V().hasLabel('root').repeat(out('left')).
                       until(__.not(out('left'))).path().by('data')

[9,5,2,1]
```

We can see all the possible paths through the tree by running the following query.

```
g.V().hasLabel('root').repeat(out()).times(4).emit().path().by('data')

[9,5]
[9,11]
[9,5,2]
[9,5,8]
[9,11,10]
[9,11,15]
[9,5,2,1]
[9,11,15,22]
[9,11,15,22,16]
```

We briefly explored the TinkerPop Tree API in the "Turning graphs into trees" section. We could use what we discussed there to create a Tree object from our Binary Tree graph as follows.

```
t=g.V().hasLabel('root').repeat(out()).emit().tree().by('data').next()
```

Just to be sure, we can query what kind of object we just created.

```
t.getClass()

class org.apache.tinkerpop.gremlin.process.traversal.step.util.Tree
```

If we print the tree, we can see how it has been created from our original graph. If you study the nesting closely, you will see that it does indeed represent the original binary tree data that we used to create the graph.

```
println t
```

```
[9:[5:[2:[1:[:]], 8:[:]], 11:[10:[:], 15:[22:[16:[:]]]]]]
```

We can use the 'getObjectsAtDepth' method to further investigate the tree structure.

```
t.getObjectsAtDepth(1)

9

t.getObjectsAtDepth(2)

5
11

t.getObjectsAtDepth(3)

2
8
10
15

t.getObjectsAtDepth(4)

1
22

t.getObjectsAtDepth(5)

16
```

5.9. Randomly walking a graph

When doing analysis of a graph, sometimes you just want to randomly traverse or 'walk' parts of the graph. The example below shows a query that starts at the Austin (AUS) vertex and then randomly goes to five connected vertices from there. The 'random walk' is achieved by picking a sample of one of the possible edges connected to the vertex we are currently at by sampling using the distance property of the edge and then moving to the vertex at the other end of that edge.

```
// Random walk with five hops
g.V().has('code','AUS').
      repeat(bothE('route').
         sample(1).by('dist').otherV()).
      times(5).
      path().by('code').by('dist')
```

Below are the results of running the query five times. You can see each graph walk starts at AUS and then goes to five places from there. The path shown displays the names of the other airports

and the distances between them.

```
[AUS,4921,LGW,2338,AQJ,1915,BER,992,SVO,6059,LAX]

[AUS,5294,FRA,1859,ACE,2078,HAM,1365,DJE,1001,MUC]

[AUS,1869,YVR,5698,HGH,4156,DOH,8030,IAH,2067,STT]

[AUS,5074,AMS,4507,SFB,1335,AUA,2019,IAH,304,MSY]

[AUS,5074,AMS,4905,DFW,1056,CZM,1653,YYZ,7725,CAN]
```

In the previous example, every random walk began at the same place. However, if we wanted a more random walk that starts from a different airport each time, we could instead use a 'sample' step to at the start of the query to pick a random airport to start from out of the set of all vertices with an 'airport' label.

```
// Random walk with five hops
g.V().hasLabel('airport').sample(1).
      repeat(bothE('route').
        sample(1).by('dist').otherV()).
      times(5).
      path().by('code').by('dist')
```

When run five times, as shown below, each walk begins at a different airport.

```
[CRP,355,DFW,5119,GRU,1675,MAO,1768,GIG,5746,LHR]

[ACA,190,MEX,2333,SEA,4807,LGW,4921,AUS,1072,CMH]

[YFC,443,YOW,751,CLT,4564,MUC,862,VLC,2064,DME]

[AUQ,891,PPT,4200,SFO,7688,DEL,6480,SYD,723,ADL]

[RMA,273,BNE,3999,KUL,6810,HNL,3755,AUS,919,IND]
```

Another way to write the query involves the introduction of a 'local' step. In this case the 'repeat' step is applied against the current local state of the traversal. In this way we can achieve multiple random walks. The 'limit(5)' at the end of the query limits the number of random walks returned to just five. Note that if the 'local' step was removed, only one walk would occur.

```
// Five random walks each of five hops
g.V().hasLabel('airport').
      repeat(local(bothE('route').
        sample(1).by('dist').otherV())).
      times(5).
```

```
      path().by('code').by('dist').limit(5)
```

When run, five results such as those shown below are returned.

```
[ATL,2364,UIO,1619,HAV,6183,IST,1483,JED,1421,TBS]
[ANC,763,SNP,763,ANC,722,BRW,151,NUI,151,BRW]
[AUS,919,IND,816,BOS,1610,HOU,998,SJD,1967,YWG]
[BNA,490,MCI,1343,CUN,1437,CLE,692,MCI,1045,PIE]
[BOS,2580,SAN,2253,IAD,960,NAS,4332,LHR,250,IOM]
```

5.10. Seven degrees of separation!

No, the word seven in the heading above is not a mistake, read on! Most people have likely heard of the famous "Six degrees of separation" theory. The theory essentially states that for any living person, using "friend of a friend" style connections, none of us is farther removed than six friendship relationships from anyone else.

You can read more about the Six degrees of separation theory on Wikipedia at this location https://en.wikipedia.org/wiki/Six_degrees_of_separation.

We decided it would be fun to run an experiment using the `air-routes` data set to see how far any airport, that is not a known orphan vertex, is from a major hub airport. For this experiment we decided to use London Heathrow (LHR) airport as our target.

We came up with the queries below to do this. The first thing we wanted to do was establish how many connections we would be looking for.

First, we double-checked how many airports are in the graph.

```
g.V().hasLabel('airport').count()

3504
```

Next we remembered there are some orphan airports that have no routes, so we have to rule those out. We can do this using a technique similar to the one that was discussed in the "Which airports have no routes?" section.

```
g.V().hasLabel('airport').not(outE('route')).count()

29
```

So we know that there are 16 airports we need to ignore for this experiment. Leaving us with 3475 airports we care about, but one of those will be LHR, so we also need to discount that one as we are not looking for cyclic paths. So, we need a query that proves there is a route between all of these 3474 remaining airports and London Heathrow but ignores the orphan airports and Heathrow

itself as starting points.

The core part of the query is shown below. We started out using just one hop for testing purposes. The query starts by ignoring LHR and the orphan vertices. It next uses a 'union' step within 'local' scope to try and find a single path in one hop between the starting airport and LHR. The result, for each starting airport will either be a path containing the codes of all the airports from the starting airport to LHR, or if LHR was not reached, the path will just contain the starting airport. Just to test this part of the query, we used a hop count of 1 and limited the results to just 10.

```
g.V().has('airport','code',neq('LHR')).
       filter(out('route').count().is(neq(0))).
       local(union(
              identity().values('code'),
              local(repeat(out().simplePath()).
                      emit().times(1).
                      has('code','LHR').
                      limit(1)).
              path().by('code')).
              fold()).
        limit(10)
```

As you can see, the query we have so far seems to be working. We either get back a starting airport and a path or just a starting airport if no route was found. This is why we used the 'union' step to guarantee that at a minimum the starting airport is returned. The 'local' scope is used as we wanted the results for each starting airport to be in a separate list.

```
[ATL,[ATL,LHR]]
[ANC]
[AUS,[AUS,LHR]]
[BNA,[BNA,LHR]]
[BOS,[BOS,LHR]]
[BWI,[BWI,LHR]]
[DCA]
[DFW,[DFW,LHR]]
[FLL]
[IAD,[IAD,LHR]]
```

The final piece missing from the query is a way to decide if we reached LHR and count the number of times we succeeded in doing so. This is done by checking to see that the length of the result returned has more than one item in it. In other words, did we get back more than just the starting airport code. To do this, the 'where' step filters out the paths that did not reach LHR in the specified number of steps. Finally, rather than displaying the paths as we did in our testing, we now count how many of the starting airports were able to get to LHR. For our first round of tests we decided to see how many of the airports can reach LHR in five hops.

```
g.V().has('airport','code',neq('LHR')).
      filter(out('route').count().is(neq(0))).
```

```
      local(union(
             identity().values('code'),
             local(repeat(out().simplePath()).
                    emit().times(5).
                    has('code','LHR').
                    limit(1)).
             path().by('code')).
             fold().
             where(count(local).is(gt(1)))).
      count()

3452
```

As you can see, with five hops we were able to reach LHR from 3452 of the 3474 airports we are testing. Next we ran the query again but used six hops.

```
g.V().has('airport','code',neq('LHR')).
      filter(out('route').count().is(neq(0))).
      local(union(
             identity().values('code'),
             local(repeat(out().simplePath()).
                    emit().times(6).
                    has('code','LHR').
                    limit(1)).
             path().by('code')).
             fold().
             where(count(local).is(gt(1)))).
      count()

3461
```

So with six hops, 3461 of our 3474 airports were able to get to LHR. We then tried seven hops and found that all but two of the airports were able to reach LHR

```
g.V().has('airport','code',neq('LHR')).
      filter(out('route').count().is(neq(0))).
      local(union(
             identity().values('code'),
             local(repeat(out().simplePath()).
                    emit().times(7).
                    has('code','LHR').
                    limit(1)).
             path().by('code')).
             fold().
             where(count(local).is(gt(1)))).
      count()

3462
```

This made me curious. We were fairly convinced that there are no airports in the graph that would not be able to reach LHR in seven hops unless, of course, they were orphans, but we have already ruled those out as part of our query. So, we decided to go with 30 hops just to make sure. However, we still got the answer 3462.

```
g.V().has('airport','code',neq('LHR')).
      filter(out('route').count().is(neq(0))).
      local(union(
             identity().values('code'),
             local(repeat(out().simplePath()).
                    emit().times(30).
                    has('code','LHR').
                    limit(1)).
             path().by('code')).
             fold().
             where(count(local).is(gt(1)))).
      count()

3462
```

This left me with an interesting conclusion. There must be two airports in the graph that are not orphans, in other words they have some routes, but they are isolated from the main network. So we needed a query to find them, which is the subject of the next section. What the queries in this section did prove is that from any airport in the graph, except the two cases we need to investigate further, London LHR can be reached in seven hops or fewer. Hence, the title of this section being "Seven degrees of separation" rather than "Six"!

5.10.1. Finding isolated subnetworks

As discussed above, while we were working on the queries for the previous section we realized that there are airports in the graph that are not true orphans, as they do have routes between each other but that they are part of a small network that is not itself connected to the main route network. In other words, the network is an isolated mini network within the main graph and you can never get to London from either any of them.

To solve the puzzle of the missing airports, we only had to slightly modify the seven degrees of separation query.

```
g.V().has('airport','code',neq('LHR')).
      filter(out('route').count().is(neq(0))).
      local(union(
             identity().values('code'),
             local(repeat(out().simplePath()).
                    emit().times(7).
                    has('code','LHR').
                    limit(1)).
             path().by('code')).
             fold().
```

```
            where(count(local).is(1))).
      unfold()
```

The only change made to this query from the one used in the prior section is that the 'where' step has been changed to 'where(count(local).is(1)))' so that it now looks for paths of length one. These represent the airports for which no route to LHR was found. Then rather than 'count' the paths, an 'unfold' is used to return the codes for the airports that had no route.

When the query is run, the codes for the airports with no route to London are revealed. These airports are truly connected to each other but isolated from the main route network.

```
CAT
VRL
VSE
PRM
BGC
BPG
CQA
BQJ
CMK
GYG
UMS
VUU
```

Just to verify our findings, we ran a couple of queries to double-check on the routes from each of these airports. As you can see, they were just connected to each other.

```
g.V().has('code','VUU').out().path().by('code')

[VUU,CMK]

g.V().has('code','CAT').out().path().by('code')

[CAT,VSE]
[CAT,PRM]
```

As a point of curiosity, we also thought that we'd look at the countries where these airports were.

```
g.V().has('code',within('CAT','VRL','VSE','PRM','BGC','BPG',
                        'CQA','BQJ','CMK','GYG','UMS','VUU')).
  values('country').dedup().fold()

[PT,BR,RU,MW]
```

It is interesting that there are a set of airports in Portugal, Brazil, Russia, and Malawi connected only to each other via airlines but with no way to get from any one of them to the main, worldwide

route network.

5.11. Looking for the journey requiring the most stops

Sometimes it is interesting to ponder questions such as "Starting from a given airport, what are the most challenging places to get to?". For the sake of this example, let's define "difficult to get to" as meaning "requires the most hops". By modifying the query from the previous section, we can come up with a way to detect some longest routes. At the end of the section we have also included examples that show alternative ways to generate the same result.

The query shown below finds every airport that has out going routes and is not Austin (AUS). For each airport found an attempt is made to get to Austin in ten or fewer hops. For each airport the attempt is only made once. We decided to use a value of 10 as we were already fairly confident that there would be no airports further away in terms of stops than that. However, it really does not matter what value is used so long as it is big enough. A value of 100 does not adversely affect the query given that the 'emit' step will return a result as soon as AUS is reached. The query will find a single path between each starting airport and AUS. The 'where' step on the last line of the query only keeps any paths that have a length of eight or more. So essentially that says find me only routes that take at least seven hops to get to Austin as AUS.

```
g.V().has('airport','code',neq('AUS')).
      filter(out('route').count().is(neq(0))).
      local(repeat(out().simplePath()).
              emit(has('code','AUS')).
              times(10).limit(1)).
      path().by('code').
      where(count(local).is(gte(8)))
```

When run, the query may take a few seconds to complete, but should not take more than that on a typical laptop or desktop. Here are the results of running the query.

```
[YPO,YAT,ZKE,YFA,YMO,YTS,YYZ,AUS]
[YZG,YIK,AKV,YPX,YGL,YUL,PHL,AUS]
[THU,NAQ,JUV,JAV,GOH,KEF,PIT,AUS]
```

The query could also be written using 'repeat' and 'until' steps as shown below.

```
g.V().has('airport','code',neq('AUS')).
      filter(out('route').count().is(neq(0))).
      local(repeat(out().simplePath()).
        until(has('code','AUS').or().loops().is(8)).
        has('code','AUS').limit(1)).
      path().by('code').
      where(count(local).is(gte(8)))
```

As before, the same three paths are found.

```
[YPO,YAT,ZKE,YFA,YMO,YTS,YYZ,AUS]
[YZG,YIK,AKV,YPX,YGL,YUL,PHL,AUS]
[THU,NAQ,JUV,JAV,GOH,KEF,PIT,AUS]
```

5.11.1. Quickly finding the hardest to get to airports

There are other ways we could write a query that looks for "hard to get to" airports. The technique shown below essentially relies on using a deduplication strategy to make sure you only visit an airport exactly once. Note this is different from the way the 'sideEffect' step works. That step allows a query to fan out and visit the same vertex from many places, a many-to-one type of pattern, but will not allow a traversal to loop back on itself. By using a deduplication approach, we only visit a vertex exactly once and remove all other instances of it as the query progresses. This gives the query processor a lot less work to do, and as a result the query executes more efficiently. The queries below do a bit less filtering than the ones above but work well if all you are looking for are destinations that, from a given starting point, require at least a specified number of hops to reach.

This time we decided to start with Austin (AUS) and look for any airports that are only reachable in seven hops. The fact that we are removing duplicates along the way guarantees that there is not a shorter path to the destinations found. If you are good at reading backwards, you will see that the results match those found above but in the reverse order.

The first example keeps track of where it has been using a 'aggregate' step and only continues on to places it has not seen before until seven hops have completed. Note that it is important to wrap 'aggregate' inside 'local' to get lazy evaluation behavior in this case.

```
g.V().has('code','AUS').
      repeat(out().where(without('a')).local(aggregate('a'))).
        times(7).
      path().
        by('code')
```

As you can see, the results look familiar.

```
[AUS,YYZ,YUL,YGL,YPX,AKV,YIK,YZG]
[AUS,YYZ,CPH,SFJ,JAV,JUV,NAQ,THU]
[AUS,YYZ,YTS,YMO,YFA,ZKE,YAT,YPO]
```

This alternate version of the query uses a 'dedup' step to achieve the same goal and in a more performant manner. On most Gremlin query engines this is an efficient way to look for targets only reachable in at least the given number of hops. This query is similar to the one used in the "Does any route exist between two airports?" section.

```
g.V().has('code','AUS').
      repeat(out().dedup()).
        times(7).
      path().
```

```
        by('code')
```

Once again, the same results are generated.

```
[AUS,PHL,YUL,YGL,YPX,AKV,YIK,YZG]
[AUS,PHL,KEF,GOH,JAV,JUV,NAQ,THU]
[AUS,YYZ,YTS,YMO,YFA,ZKE,YAT,YPO]
```

When looking for longest routes in a graph, especially in a highly connected one, you need to be careful. Notice how we chose a very specific target or a specific number of hops for our searches.

Looking for longest paths must be done carefully as it can result in very long-running queries.

If you were to try and write a query that arbitrarily looked for the longest route between any two airports, you run the risk of writing a query that runs for an extremely long time. In many ways the concept of "longest path" can be viewed as an antipattern. This is especially true in highly connected graphs of which `air-routes` is an example.

As a final point about the prior example, we recall the "Understanding TraversalStrategies" section, where we learned that TinkerPop applies some changes to the way a traversal is written just before it is iterated to optimize performance where it can. The prior example demonstrates a case where certain query patterns will prevent a particular strategy from being applied. Normally, the 'repeat().times()' pattern will result in the contents of 'repeat' being expanded, where 'repeat(out()).times(2)' becomes, 'out().out()' which is more efficient for TinkerPop to execute. But the presence of 'dedup()' precludes that transformation to get the query semantics that are presented. This sort of information can become useful when you are looking for your own query optimizations.

5.12. Finding unwanted parallel edges

In general terms, there are many reasons in a graph that there could be more than one edge between the same two vertices in the same direction. Most property graph systems allow this, and many data models take advantage of this capability. We call these parallel edges. However, in the 'air-routes' graph we only model the existence of a route using a single edge from airport A to airport B. It is considered an error for there to be more than one edge between the same two airports in the same direction. Note this does not include an edge going the other way (from B to A).

During development of the 'air-routes' graph, when we were still cleaning up the data, we frequently ran into problems with parallel edges getting included in the graph by mistake. we realized we could use Gremlin to help me detect these error cases.

Initially we tried something very basic, that still required quite a bit of manual reading of the output. We used the following query to tell me if for a given airport there was more than one outgoing edge to any other airport. This required me to have a hunch ahead of time which airport vertices might have an issue. Far from ideal when there are over 3,300 airport vertices in the graph.

```
g.V().has('code','LHR').out().groupCount().by('code').
  order(local).by(values,desc).next().values().max()
```

If the answer came back greater than one, we then ran the following query and manually looked at each result to see where the duplicate edge was.

```
g.V().has('code','LHR').out().groupCount().by('code').order(local).by(values,desc)
```

As we said, this was a very manual and time-consuming process. We clearly needed a better query. Given what we know about 'groupCount' we realized we could write an arbitrary query to tell me how many times every single route in the graph exists. However, given there are over 50,000 routes, we were not going to be able to check that manually. So as is often the best way with Gremlin, we built up our query in stages. First, we wrote the query to count the occurrence of all routes. We have not shown all the output from this query as it would take tens of pages, but as you can see from what we have shown, this result would still need a person studying all the results. Far from ideal!

```
g.V().as("a").out().as("b").groupCount().by(select("a","b"))

[[a:v[1],b:v[3]]:1,[a:v[1],b:v[6]]:1,[a:v[1],b:v[7]]:1 ...
```

We then added a filter to only select any routes that occurred more than once. Note that we had to use 'unfold' before we applied the 'filter' to turn the map back into a stream of values that could be filtered.

```
g.V().as("a").out().as("b").groupCount().by(select("a","b")).unfold().
                            filter(select(values).is(gt(1)))
```

Next we added one more step to tell me which routes contained the error.

```
g.V().as("a").out().as("b").groupCount().by(select("a","b")).unfold().
                            filter(select(values).is(gt(1))).select(keys)
```

One of the errors we ran into was that we had erroneously added the LHR to JFK route twice into the graph. LHR has the ID of 49 and JFK has the ID of 12. When we ran the above query, we got the following output which told me exactly which route we needed to correct. This is clearly a much more useful query than the prior ones. We could happily have stopped at this point, but we wanted to see if we could improve the query some more.

```
[a:v[49],b:v[12]]
```

We were still not totally happy with our query as we really wanted it to give me back the airport codes. So we added a few more tweaks to do the 'groupCount' using the airport codes. We have also

left the 'select(keys)' off the end of this query as we think it aids understanding to see what is returned before that step is performed.

```
g.V().as("a").out().as("b").select('a','b').by('code').groupCount().unfold()
                           .filter(select(values).is(gt(1)))
```

So what we get back when we run that query is the airport codes as well of the number of times they have appeared connected to each other by a parallel edge. This is actually a key/value pair where the contents of the '{}' are the keys and the '=2' part is the value. You could reasonably argue that this is sufficient for me to go and fix the mistake in the graph. However, we want to add just a couple of additional steps to demonstrate other possible refinements that you may find useful in other circumstances.

```
{a=LHR, b=JFK}=2
```

If we don't want the '=2' part returned, we can just select the keys part.

```
g.V().as("a").out().as("b").select('a','b').by('code').groupCount().unfold()
                           .filter(select(values).is(gt(1))).select(keys)
```

This is what was now returned. Note that the keys themselves are returned in a map using the 'a' and 'b' terms from our 'as' step.

```
[a:LHR,b:JFK]
```

This is almost exactly what we wanted, but we can add one more tiny step to clean up the output by just selecting the values from the map returned.

```
g.V().as("a").out().as("b").
  select('a','b').by('code').
  groupCount().unfold().
  filter(select(values).is(gt(1))).
  select(keys).select(values)
```

So here is the final output, just the codes for the airports with parallel edges that needed fixing.

```
[LHR,JFK]
```

Note how we built up our Gremlin query in stages to solve this problem. We recommend this as a sensible way to approach all but the most basic of queries that you may need to write. Doing it this way has the advantage that you can also check that each part of the query is working the way you intend it to before you

add more parts to it.

5.12.1. Using 'groupCount' with 'path' to find duplicate edges

The prior example found duplicate edges by working with a map created using 'select' and 'groupCount' steps. It is also possible to use a 'path' step inside a 'groupCount' step to achieve a similar result.

First, let's create a duplicate edge between Austin (AUS) and Cancun (CUN).

```
g.V().has('code','AUS').addE('route').to(V().has('code','CUN'))

e[53791][3-route->180]
```

We can now use a 'groupCount' step containing a 'path' step to look for duplicate edges originating in Austin. We used a 'limit' step just to reduce the amount of output a bit.

```
g.V().has('code','AUS').out().
      groupCount().by(path().by('code')).limit(local,5)
```

As you can see from the results below, the route between AUS and CUN has a count of 2 associated with it.

```
[[AUS,CUN]:2,[AUS,MDW]:1,[AUS,MIA]:1,[AUS,DFW]:1,[AUS,BWI]:1]
```

Note that this technique does not replace the full query shown in the prior section. It is more intended to show that you can place a 'path' step inside a 'groupCount' step and achieve some useful results.

5.13. Finding the longest flight route between two adjacent airports in the graph

This section is in a way a case study in the good and the bad of the Gremlin query language. We have documented here the learning steps we went through to get what we thought was a simple question expressed as a single Gremlin query. This documents well the good, the bad and the sometimes ugly aspects of working with Gremlin. On the good side it is powerful, and some things that should be easy are easy. On the bad side, some things that appear easy are in fact very hard to get right, especially if you want to build a single query to do a job rather than break it up into a set of smaller programmatic steps. Let's look at an example of this with a real-world query.

The query we want to build is to find the route(s) in the graph that are of the maximum length between any two airports. It turns out that this simple-looking query takes a lot of work to get right if you want to handle the case where there could be more than one route that is of the maximum length (return flights between the same two airports don't count as different routes for this example).

While using a 'max' step, as shown below, might seem like the obvious and easy to do what we need, because of the way that 'max' is implemented, it will not work. Currently, the 'max' and 'min' steps cause the prior paths that have been taken in a traversal to be lost. We need to explore other ways of achieving our desired result.

```
g.E().hasLabel('route').as('e').values('dist').max().select('e')
```

First, we could do this (below) The downside of this approach is that both queries have to look at a lot of edges, which is likely to use additional memory and CPU to process.

```
r=g.E().hasLabel('route').values('dist').max().next()
g.E().has('dist',r).bothV().values('code')
```

This next query is more efficient as using an ID means the edges are only searched once, but this approach would miss the case where more than one route was of the same (max) length, so it does not meet our required success criteria.

```
r=g.E().hasLabel('route').as('e').order().by('dist', desc).
    limit(1).select('e').id().next()
g.E(r).values('dist')
g.E(r).bothV().values('code')
```

This query is what Daniel Kuppitz from the TinkerPop team recommended after we discussed this on the mailing list, and it does indeed work, but from where we started our experiments to build up this query from, there is a non-trivial journey!

```
g.E().hasLabel("route").order().by("dist", desc).local(aggregate("d").by("dist")).
  filter(values("dist").as("cd").
         select("d").by(limit(local, 1).unfold()).as("md").
         where("cd",eq("md"))).
  project("from","to","dist").
    by(outV()).
    by(inV()).by("dist")
```

Finally, to output airport codes rather than just vertices, we can tweak the query one more time as follows.

```
g.E().hasLabel("route").order().by("dist",desc).local(aggregate("d").by("dist")).
  filter(values("dist").as("cd").
         select("d").by(limit(local, 1).unfold()).as("md").
         where("cd",eq("md"))).
  project("from","to","dist").
    by(outV().values('code')).
    by(inV().values('code')).
```

```
    by("dist")
```

Now we are ready to run our query. This is the output from it. Notice how both routes are between the same city pairs. We could have further refined our query to only show such combinations once, but we will leave that as an exercise for the reader!

```
[from:SIN,to:JFK,dist:9526]
[from:JFK,to:SIN,dist:9526]
```

5.14. Miscellaneous other queries

The examples in this section demonstrate a few miscellaneous features and queries that we have not yet had a chance to examine. Over time, these queries should probably be moved to other sections of the book.

5.14.1. Using a calculation inside an 'is' step

It is possible to use a mathematical expression inside an 'is' step. At times this comes in quite handy. The example below simply divides 500 by 2 as part of a test. You would normally enter 250, but this shows the capability.

```
g.V().hasLabel('airport').where(out().count().is(gt(500/2))).values('code')

DFW
ORD
CDG
FRA
AMS
MUC
IST
```

The capability becomes more interesting when used in conjunction with a variable.

```
a = 500

g.V().hasLabel('airport').where(out().count().is(gt(a/2))).values('code')

DFW
ORD
CDG
FRA
AMS
MUC
IST
```

Here is one more example where we start by setting the variable 'a' to the value that represents the maximum number of routes from any single airport. We use a query to find all the airports that have at least as many outgoing routes as 50 fewer than our value 'a'.

```
a = g.V().local(out('route').count()).max().next()

g.V().hasLabel('airport').where(out().count().is(gt(a-50))).
      project('apt','routes').by('code').by(out().count())

[apt:ORD,routes:265]
[apt:CDG,routes:293]
[apt:FRA,routes:310]
[apt:AMS,routes:283]
[apt:MUC,routes:270]
[apt:IST,routes:309]
```

Chapter 6. MOVING BEYOND THE GREMLIN CONSOLE

The focus of this book so far has been to teach the Gremlin query language, using the Gremlin Console and TinkerGraph as our learning environment. This is indeed a great way to learn, and that environment can be set up quickly, as we have seen, on a single laptop, or desktop, computer. Even as you move towards more sophisticated environments, keeping the console nearby is recommended as an easy way to experiment with new ideas.

As we start to think about using graphs to solve problems in a production environment, we need some additional tools. We have to consider things such as programming language interfaces, persistent storage, making graphs available using servers and more advanced indexing. Those topics are the focus of this chapter, and the two that follow. Hopefully this will whet your appetite for moving beyond the console, and into the world of graph application programming and graph system deployment!

In this chapter we start the journey by introducing ways to access TinkerGraph directly, embedded with your application code, from programming languages such as Java and Groovy. Later we introduce ways to use these same programming languages, and others, to connect to a Gremlin Server and other remotely hosted graph databases. First, let's take a look at the Apache TinkerPop Java API.

6.1. Working with TinkerGraph from a Java Application

So far in this book we have looked at many ways of working with a TinkerGraph from within the Gremlin Console. As you start to create more sophisticated applications, you will find that the console is just one of the tools to keep in your toolbox. It is very likely, if not certain, that you will want to write standalone applications that can work with graph data. There are a number of different programming language bindings currently available for TinkerPop. One of the most widely used is the Java API. Apache TinkerPop itself is coded mostly in Java.

You will find several Java samples at the GitHub repository associated with this book. https://github.com/krlawrence/graph

When building a commercial application, you may need capabilities such as high availability and durability, and would not use TinkerGraph as your graph database in those cases. There are, however, places where TinkerGraph may be just what you need. One example might be doing analysis on a static graph that can fit into memory on your laptop. The 'air-routes' graph is a good example of such a graph. You might also have a large graph stored on a server but need to export part of it to do local analysis. As a first step towards writing standalone applications that use different graph database implementations, let's look at a few examples of how you can create a Java application that uses Gremlin and TinkerGraph.

6.1.1. The Apache TinkerPop interfaces and classes

There are a number of Java interfaces and classes, defined by the Apache TinkerPop project, that you will want to become familiar with. The most recent Javadoc format API documentation is always available at https://tinkerpop.apache.org/javadocs/current/full

The TinkerPop Javadoc is a bit lacking in terms of English prose but is still a useful source of reference information when it comes to methods, parameters and types. Once you have coded up a couple of test programs and got them running, you should find it gets easier to make progress faster. To that end, we recommend you take a look at the sample code included with the book.

When using the Gremlin Console, your environment is pre-configured for you, so for the most part, you do not have to import any classes or interfaces. However, as soon as you move to the domain of the standalone Java application, you will need to start importing the classes that your application needs. You will probably discover a few other things that the console was doing on your behalf.

By way of a reasonable start, some of the most common imports needed by a Java application working with Gremlin are listed below. As you add more capabilities to your application, you will, of course, need to add the appropriate import statements.

Take particular note of the rather odd import of the class called "__" (underscore underscore) on the second line. This is required to enable calling methods such as 'in' and 'out' in a traversal where there is no prior step to "dot" attach them to (such as inside a 'repeat' step). If you prefer, you can statically import the "__" class which will make explicit use of "__" in your code unnecessary except where you are faced with reserved word conflicts as discussed in prior chapters.

```
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.__;
import org.apache.tinkerpop.gremlin.process.traversal.Path;
import org.apache.tinkerpop.gremlin.process.traversal.*;
import org.apache.tinkerpop.gremlin.structure.Edge;
import org.apache.tinkerpop.gremlin.structure.Vertex;
import org.apache.tinkerpop.gremlin.structure.io.IoCore;
import org.apache.tinkerpop.gremlin.tinkergraph.structure.*;
import java.io.IOException;
import java.util.List;
import java.util.ArrayList;
import java.util.Map;
import java.util.Set;
```

6.1.2. Writing our first TinkerPop Java program

Now that we have some imports in place, we can start to craft the basic outline of a Java application. The code below defines a class called 'TinkerGraphTest' and defines a 'main' method that creates an in-memory TinkerGraph and loads the air routes GraphML data. Note that as we are now going to be running as a Java program, we have to catch exceptions. This is another thing hidden from you when you are working within the Gremlin Console.

The source code in this section comes from the 'TinkerGraphTest.java' sample

> located at https://github.com/krlawrence/graph/tree/main/sample-code/java.

Lastly, in this initial class definition we create a 'GraphTraversalSource' and we make a Gremlin query to get the property 'valueMap' for the Austin airport vertex and print it. The 'toString' method provided by the 'Map' should give us some useful output. Take note of the call to 'next'. This terminates the graph traversal and causes the result to be returned. If this call is left off, you will not get back what you were expecting!

```
public class TinkerGraphTest
{
  public static void main(String[] args)
  {
    TinkerGraph graph = TinkerGraph.open();
    GraphTraversalSource g = traversal().with(graph);

    try
    {
      g.io("./air-routes.graphml").read().iterate();
    }
    catch( IOException e )
    {
      System.out.println("File not found");
      System.exit(1);
    }

    Map<String,?> aus = g.V().has("code","AUS").valueMap().next();
    System.out.println(aus);
  }
}
```

6.1.3. Compiling our code

Before we can test our program, we, of course, need to compile it. The easiest way to do this while experimenting is to set up the Java 'classpath' to include the TinkerPop JAR files. Once you get into writing bigger solutions, you will most likely be using a tool like Apache Maven to control your build. For our experiments here, simple use of the 'classpath' will suffice.

The following lines of a Bash shell script will set up what you need to both build and run our small test program. Note that the 'GREMLIN' variable should be set to point to the root directory of wherever your TinkerPop JAR files are.

```
# Root directory for Gremlin Console install
GREMLIN=...

# Path to Gremlin core JARs
LIBPATH=$GREMLIN/lib/*

# Path to TinkerGraph JARs
```

```
EXTPATH=$GREMLIN/ext/*

#Path to additional JARs
ADDL=$GREMLIN/ext/tinkergraph-gremlin/lib/*

# Classpath
export CP=$CLASSPATH:$LIBPATH:$EXTPATH:$ADDL

# Compile
javac -cp $CP TinkerGraphTest.java
```

Assuming everything we have coded so far compiled OK, then we can run it using the same 'classpath' that we created previously.

```
# Run
java -cp $CP TinkerGraphTest
```

The output we get back should look something like the following. If it does, take 30 seconds to celebrate as you have just successfully built and run your first TinkerPop aware Java app!

```
{country=[US], code=[AUS], longest=[12250], city=[Austin], elev=[542], icao=[KAUS],
lon=[-97.6698989868164], type=[airport], region=[US-TX], runways=[2], lat
=[30.1944999694824], desc=[Austin Bergstrom International Airport]}
```

6.1.4. Adding to our Java program

Now that we have a basic skeleton application that compiles and runs, we can start to add more experiments to it. If we add the two lines below, we can extract the city name from the value map that we just generated. From now on, as we add to our program, we are just going to show the lines that we add and the additional output that those new lines will generate.

```
List city = (List)(aus.get("city"));
System.out.println("The AUS airport is in " + city.get(0));
```

So when we compile and run again, we should now see this additional line of output.

```
The AUS airport is in Austin
```

If we wanted to nicely print out all values returned in our value map, we could do it as follows.

```
aus.forEach( (k,v) -> System.out.println("Key: " + k + ": Value: " + v));
```

Which will generate this output.

```
Key: country: Value: [US]
Key: code: Value: [AUS]
Key: longest: Value: [12250]
Key: city: Value: [Austin]
Key: elev: Value: [542]
Key: icao: Value: [KAUS]
Key: lon: Value: [-97.6698989868164]
Key: type: Value: [airport]
Key: region: Value: [US-TX]
Key: runways: Value: [2]
Key: lat: Value: [30.1944999694824]
Key: desc: Value: [Austin Bergstrom International Airport]
```

So we now know one way to get the property values from a vertex and manipulate them. Let's now add something a bit more interesting to our program. The following two lines count the number of airports you can fly to non-stop starting at Dallas Fort Worth (DFW) and print out that result for us.

```
Long n = g.V().has("code","DFW").out().count().next();
System.out.println("There are " + n + " routes from Dallas");
```

```
There are 221 routes from Dallas
```

Let's now add some code to retrieve the airport IATA codes of these 221 airports that we can fly to non-stop from DFW. Note that this time we ended our query with a call to the 'toList' method. This will terminate the traversal, and as the name implies, return the results to us in a list. An 'order' step is used in the traversal so that we get the airport codes back in ascending order.

```
List fromDfw = g.V().has("code","DFW").out().
                     order().by("code").values("code").toList();

System.out.println(fromDfw);
```

The new output that we get back should look like this.

```
[ABI,ABQ,ACA,ACT,AEX,AGS,AGU,AMA,AMS,ANC,ASE,ATL,AUA,AUG,AUH,AUS,AVL,BDL,BFL,BHM,BIL,
 BIS,BJX,BKG,BMI,BNA,BOG,BOI,BOS,BPT,BRO,BTR,BUF,BUR,BWI,BZE,BZN,CAE,CCS,CDG,CHA,CHS,
 CID,CLE,CLL,CLT,CMH,CMI,CNM,COS,COU,CRP,CUN,CUU,CVG,CVN,CYS,CZM,DAY,DCA,DEN,DGO,DOH,
 DRO,DRT,DSM,DTW,DUB,DUS,DXB,ECP,EGE,ELP,EVV,EWR,EYW,EZE,FAR,FAT,FCO,FLG,FLL,FRA,FSD,
 FSM,FWA,GCK,GCM,GDL,GEG,GGG,GIG,GJT,GLH,GNV,GPT,GRI,GRK,GRR,GRU,GSO,GSP,GUA,GUC,HDN,
 HEL,HKG,HNL,HOU,HRL,HSV,IAD,IAH,ICN,ICT,ILM,IND,IST,JAC,JAN,JAX,JFK,JLN,KEF,KOA,LAS,
 LAW,LAX,LBB,LCH,LEX,LFT,LGA,LHR,LIM,LIR,LIT,LRD,MAD,MAF,MBJ,MCI,MCO,MEI,MEM,MEX,MFE,
 MGA,MGM,MHK,MIA,MID,MKE,MLI,MLM,MLU,MOB,MRY,MSN,MSO,MSP,MSY,MTJ,MTY,MUC,MYR,MZT,NAS,
 NRT,OAK,OAX,OGG,OKC,OMA,ONT,ORD,ORF,PBC,PBI,PDX,PEK,PHL,PHX,PIA,PIB,PIT,PLS,PNS,PSP,
 PTY,PUJ,PVG,PVR,QRO,RAP,RDU,RIC,RNO,ROW,RSW,RTB,SAF,SAL,SAN,SAP,SAT,SAV,SBA,SBN,SCL,
 SDF,SDQ,SEA,SFO,SGF,SHV,SJC,SJD,SJO,SJT,SJU,SKB,SLC,SLP,SMF,SNA,SPI,SPS,SRQ,STL,SUX,
```

```
 SWO,SYD,TGU,TLH,TPA,TRC,TUL,TUS,TVC,TXK,TYR,TYS,UIO,VPS,XNA,YEG,YUL,YUM,YVR,YYC,YYZ,
 ZCL]
```

The next lines will find all routes from London Heathrow to any airport in the United States. A list of paths will be returned where each path contains the airport codes and the distance between them.

```
List <Path> lhrToUsa = g.V().has("code","LHR").outE().inV().
                            has("country","US").
                            path().by("code").by("dist").toList();

lhrToUsa.forEach((k) -> System.out.println(k));
```

The output should look like this. We arranged the results in columns to aid readability.

```
[LHR,3707,PIT]
[LHR,4896,PDX]
[LHR,3980,CLT]
[LHR,4053,CHS]
[LHR,4198,ATL]
[LHR,4901,AUS]
[LHR,4168,BNA]
[LHR,3254,BOS]
[LHR,3622,BWI]
[LHR,4736,DFW]
[LHR,3665,IAD]
[LHR,4820,IAH]
[LHR,3440,JFK]
[LHR,5439,LAX]
[LHR,4414,MIA]
[LHR,4001,MSP]
[LHR,3939,ORD]
[LHR,5255,PHX]
[LHR,3860,RDU]
[LHR,4783,SEA]
[LHR,5350,SFO]
[LHR,5352,SJC]
[LHR,5469,SAN]
[LHR,4850,SLC]
[LHR,5213,LAS]
[LHR,4655,DEN]
[LHR,4616,MSY]
[LHR,3453,EWR]
[LHR,3533,PHL]
[LHR,3753,DTW]
```

The final part of our first Java program shows how to perform a simple 'repeat' operation. The code below will look for any cities in the UK that you can get to from Austin with one stop on the way. A

key thing to note here is that we have to prefix the call to 'out' with the strangely named class "__." that we mentioned at the start of this discussion of using Java with TinkerPop. If you do not include the "__." prefix, you will get a compilation error as the compiler does not know where the 'out' step is from.

```
List <Object> eng =
      g.V().has("code","AUS").repeat(__.out()).times(2).
            has("region","GB-ENG").values("city").dedup().toList();

System.out.println("\nPlaces in England I can get to with one stop from AUS.\n");
eng.forEach( (p) -> System.out.print(p + " "));
```

The code we just added should generate output that looks like this.

```
Places in England I can get to with one stop from AUS.

Birmingham Bristol London Manchester Leeds Newcastle
```

6.1.5. Important Classes and Enums to be aware of

When you are using the Gremlin Console, as we have already mentioned, a few things are done for you that you need to take care of yourself when writing standalone Java code. A key area that we have found that people making the jump from the console to Java find confusing is figuring out which Classes and Enums have been statically imported "behind the scenes" and how to access those same capabilities from Java. You should also bookmark the TinkerPop Javadoc pages as you can find more detail on all classes and enums covered below there. The latest Javadoc is always available at https://tinkerpop.apache.org/javadocs/current/full/

The source code in this section comes from the 'TestImports.java' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/java and demonstrates these constructs being used.

The tables below show some commonly used Gremlin keywords in the left column. The second column shows how you would reference those same keywords explicitly from a Java program. The third column shows an example of the context in which they might be used in a Java program. You may choose to statically import some of these classes into your code. We prefer to use the explicit prefix, but that is a matter of personal preference in many cases. There is a table of all the available predicates in the "Testing values and ranges of values with P" section. For the special "__" class we have just shown a few examples. In general, you use this prefix when you have nothing prior in the query that you can "dot chain" to.

If you need to specify how a property value should be treated when added to a vertex, you can use one of the keywords defined as part of the 'Cardinality' enum, which is part of the 'VertexProperty' interface.

Table 9. Cardinality

single	Cardinality.local	property(Cardinality.single,"mykey","ABC")
list	Cardinality.list	property(Cardinality.list,"mykey","ABC")
set	Cardinality.set	property(Cardinality.set,"mykey","ABC")

When 'local' scope needs to be specified as a parameter or sort order needs to be specified, the statics defined in the 'Scope' and 'Order' Enums can be used.

Table 10. Scope and ordering

local	Scope.local	order(Scope.local)
global	Scope.global	order(Scope.global)
desc	Order.desc	order().by(Order.desc)
asc	Order.asc	order().by(Order.asc)
shuffle	Order.shuffle	order().by(Order.shuffle)

If you need to access the keys or values from a map data structure, you can use the statics defined in the 'Column' enum.

Table 11. Keys and values

keys	Column.keys	order().by(Column.keys)
values	Column.values	order().by(Column.values)

When accessing the 'id' and 'label' values from a 'valueMap', you need to use the statics defined in the 'T' Enum. The same is true if you want to access the keys and values from a set of properties.

Table 12. Label and id

label	T.label	valueMap(true).next().get(T.label)
id	T.id	valueMap(true).next().get(T.id)
key	T.key	properties().order().by(T.key)
value	T.value	properties().order().by(T.value)

When there is no previous step to "dot chain", to then we can use the "__" class as our prefix. If you look at the Javadoc for the class, you will see it defines static methods that we can use to call Gremlin functionality in cases like the ones shown below.

Table 13. Anonymous references

out	__.out	repeat(__.out())
in	__.in	order().by(__.in("contains"))
constant	__.constant	union(__.constant("b"),__.constant("a"))

Whenever we need to use a predicate to perform a test, we can use the static methods defined in the 'P' class. Not all the methods defined are shown below.

Table 14. Predicates

gt	P.gt	has("runways",P.gt(3))
gte	P.gte	has("runways",P.gte(5))
lt	P.lt	has("runways",P.lt(2))
lte	P.lte	has("runways",P.lte(2))
eq	P.eq	has("city",P.eq("Dallas"))
neq	P.neq	has("city",P.neq("Dallas"))
within	P.within	has("city",P.within("Dallas","Austin"))
without	P.without	has("city",P.without("Dallas"))
inside	P.inside	has("runways",P.inside(3,5))
outside	P.outside	has("runways",P.outside(2,5))
between	P.between	has("runways",P.between(2,5))

The 'TextP' class exposes text-search-based predicates.

Table 15. Text Predicates

startingWith	TextP.startingWith	has("city",TextP.startingWith("Dal"))
endingWith	TextP.endingWith	has("city",TextP.endingWith("as"))
containing	TextP.containing	has("city",TextP.containing("all"))
notStartingWith	TextP.notStarting With	has("city",TextP.notStartingWith("Dal"))
notEndingWith	TextP.notEndingW ith	has("city",TextP.notEndingWith("as"))
notContaining	TextP.notContaini ng	has("city",TextP.notContaining("all"))
regex	TextP.regex	has("city",TextP.regex("D.*"))
notRegex	TextP.notRegex	has("city",TextP.notRegex("D.*"))

If a traversal path has multiple values associated with a single label, such as '"x"', then you can use the 'first', 'last', 'all' and 'mixed' statics that are defined as part of the 'Pop' Enum. As the name suggests, 'first' returns the first item in a collection. Specifying 'last' returns the last item, and 'all' returns all items in a collection. Specifying 'mixed' will return a 'List' if the collection has more than one item. Otherwise, an 'Object' will be returned.

Table 16. First, last, all and mixed

first	Pop.first	select(Pop.first,"x")
last	Pop.last	select(Pop.last,"x")
all	Pop.all	select(Pop.all,"x")
mixed	Pop.mixed	select(Pop.mixed,"x")

When working with a 'sack' the operators like 'sum' and 'assign' are defined in the 'Operator' Enum.

Table 17. Operators

sum	Operator.sum	sack(Operator.sum)
minus	Operator.minus	sack(Operator.minus)
mult	Operator.mult	sack(Operator.mult)
div	Operator.div	sack(Operator.div)
assign	Operator.assign	sack(Operator.assign).by(constant(0))
min	Operator.min	sack(Operator.min)
max	Operator.max	sack(Operator.max)
addAll	Operator.addAll	sack(Operator.addAll)
and	Operator.and	sack(Operator.and)
or	Operator.or	sack(Operator.or)

The 'Direction' Enum defines constants that are used in association with edge direction. Note that 'from' and 'to' are friendly aliases of 'OUT' and 'IN' that can help make Gremlin read more fluidly at times.

Table 18. Direction

IN	Direction.IN	V().to(Direction.IN)
OUT	Direction.OUT	V().to(Direction.OUT)
BOTH	Direction.BOTH	V().to(Direction.BOTH)
from	Direction.from	mergeE([(Direction.from):1, ...])
to	Direction.to	mergeE([(Direction.to):2, ...])

The 'Pick' Enum defines constants that are used in association with the 'branch', `choose` and `option` steps. The 'unproductive' option refers to the case where the incoming traverser does not produce an output.

Table 19. Pick

none	Pick.none	option(none,constant('no match'))
any	Pick.any	option(any,constant('any picked'))
unproductive	Pick.unproductive	option(unproductive,constant('traversal unproductive'))

The 'Merge' Enum has constants that are associated to the `mergeV` and `mergeE` steps. 'onCreate' and 'onMatch' relate to the events triggered by these steps, and 'outV' and 'inV' are used to specify a way to late-bind in and out vertices for these operations.

Table 20. Merge

onCreate	Merge.onCreate	option(Merge.onCreate,[name:'alice'])
onMatch	Merge.onMatch	option(Merge.onMatch,[name:'alice'])
outV	Merge.outV	option(Merge.outV,select('v'))

inV	Merge.inV	option(Merge.inV,select('v'))

The 'DT' Enum provides constants representing units of time for the 'dateAdd' step.

Dates

```
[cols="1,1,3"]
|=======================================================================
|second    | Date.second  | dateAdd(Date.second, 10)
|minute    | Date.minute  | dateAdd(Date.second, 10)
|hour      | Date.hour    | dateAdd(Date.second, 10)
|day       | Date.day     | dateAdd(Date.second, 10)
|=======================================================================
```

As discussed earlier, you can always use the 'getClass' method or simply '.class' while using the console to, in many cases, find out where something is defined. As we saw in the examples earlier in this section, a lot of the keywords such as 'values', 'id' and 'local' are defined as Enums so you can use 'getClass' on them directly. A few examples are shown below.

```
gremlin> label.getClass()
==>class org.apache.tinkerpop.gremlin.structure.T$1
gremlin> key.getClass()
==>class org.apache.tinkerpop.gremlin.structure.T$3
gremlin> keys.getClass()
==>class org.apache.tinkerpop.gremlin.structure.Column$1
gremlin> values.getClass()
==>class org.apache.tinkerpop.gremlin.structure.Column$2
gremlin> local.getClass()
==>class org.apache.tinkerpop.gremlin.process.traversal.Scope
gremlin> Order.class
==>class org.apache.tinkerpop.gremlin.process.traversal.Order
gremlin> Column.class
==>class org.apache.tinkerpop.gremlin.structure.Column
```

If you compare this output to the tables above, you can see that we have been able to verify that, for example, that 'label' and 'key' are defined in the 'T' Enum. We can also see that 'keys' and 'values' are indeed defined in the 'Column' Enum. Lastly, we can see that 'local' is, as we expected, defined in the 'Scope' Enum.

6.1.6. Using Gremlin predicates in a Java application

As discussed in the previous section, when you use a Gremlin predicate such as 'eq' or 'neq' from a Java program, you need to prefix it with a '"P."' which is a reference to the TinkerPop class of the same name where a set of static methods, representing the Gremlin predicates, are defined.

The source code in this section comes from the 'GraphRegion.java' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/java.

Take a look at the code below. A method called 'findByRegion' is defined that takes a String representing a three-character airport IATA code as input. The method then uses a Gremlin query to figure out which geographical region the specified airport is in and then returns all airports also in that region. Note the use of 'P.eq' as part of the 'where' step. This is another case of where, because we are not running inside the Gremlin Console, we have to more precisely specify things.

```
// Find all airports in the region of the specified airport
public void findByRegion(String iata) {
  System.out.println("\nRegion code lookup for " + iata );

  List<List<Object>> list =
  g.V().has("code",iata).values("region").as("r").
    V().hasLabel("airport").as("a").values("region").
        where(P.eq("r")).by().
        local(__.select("a").values("city","code","region").fold()).toList();

  for(List t : list) {
    System.out.println(t);
  }
}
```

If we were to call the 'findByRegion' method, passing in a parameter of 'DEN', representing the airport in Denver Colorado, the following output should be returned.

```
Region code lookup for DEN
[DEN,Denver,US-CO]
[COS,Colorado Springs,US-CO]
[DRO,Durango,US-CO]
[GJT,Grand Junction,US-CO]
[EGE,Eagle,US-CO]
[HDN,Hayden,US-CO]
[ASE,Aspen,US-CO]
[ALS,Alamosa,US-CO]
[CEZ,Cortez,US-CO]
[GUC,Gunnison,US-CO]
[MTJ,Montrose,US-CO]
[PUB,Pueblo,US-CO]
[APA,Denver,US-CO]
[TEX,Telluride,US-CO]
```

6.1.7. Checking to see if a query returned a result

It is often important to know if a query returned a result before trying to reference it to avoid those pesky Java 'NoSuchElementException' errors. Without worrying about Java for a second, consider the query below purely from a Gremlin point of view.

```
g.V().has("code","AUS").outE().as("edge").inV().has("code","SYD").
```

```
                        select("edge").by("dist")
```

The query finds the Austin (AUS) airport and then looks for an outgoing edge connecting Austin with Sydney (SYD) and then returns the distance value from that edge. The problem here is that there is no direct route between Austin and Sydney and therefore no edge to retrieve the distance from. In other words, this query returns no result. Now, within the Gremlin Console this is not a problem as we just get nothing back and life goes on. However, take a look at the code below, which is a first attempt at moving the query into Java code.

```
Long result =
      g.V().has("code","AUS").outE().as("edge").inV().has("code","SYD").
            select("edge").by("dist").next();
```

On the surface, this looks fine. However, were we to execute this code, we would get a 'NoSuchElementException' as when we try to call 'next', there is no result to process as there is no edge between Austin and Sydney and hence no distance value to process.

The source code in this section comes from the 'GraphSearch2.java' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/java.

So we need a way to check to see if we go a valid result. One such way is to store the result of the query into a list. Then, in the worst case, if no results are found, we will get an empty list back. So, we can rewrite the query as follows.

```
List result =
      g.V().has("code","AUS").outE().as("edge").inV().has("code","SYD").
            select("edge").by("dist").toList();
```

Now that the result is in a list we can safely check to see if we got any results.

```
if (result.isEmpty()) {
  System.out.println("No results were found");
} else {
  System.out.println("The distance is " + result.get(0));
}
```

If we wanted a "pure Gremlin" solution, without using a List and without doing any post-processing, one way we could do it is to use the 'coalesce' step and return a special constant value, in this case minus one, to indicate that there were no results found.

```
Integer d = (Integer)
    g.V().has("code","AUS").outE().as("edge").inV().has("code","SYD").
          select("edge").by("dist").fold().
          coalesce(__.unfold(),__.constant(-1)).next();
```

If the route exists, the distance will be found and returned, otherwise a value of '"-1"' will be returned. This is really using the same concept as the 'toList' example, except in this case we generate the list using the 'fold' step within the query itself. The 'unfold' will return a result if the list is not empty, otherwise the constant value will be returned as 'coalesce' returns the first to yield a result.

There are, of course, many other ways that you might come up with to solve this problem, but using lists often provides a fairly easy-to-use solution.

6.1.8. Creating a new graph from a Java application

The code below creates a new (empty) TinkerGraph instance, creates a graph traversal source object and then uses a traversal to create a small graph.

The source code in this section comes from the 'CreateGraph.java' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/java.

Note the call to 'iterate' at the end of the traversal. When running as a standalone application, this is necessary. This is another one of those little things that the Gremlin Console does for you without you realizing it that we have to remember to do for ourselves when we are not working inside the console.

```
// Create a new (empty) TinkerGraph
TinkerGraph graph = TinkerGraph.open() ;

// Create a Traversal source object
GraphTraversalSource g = traversal().with(graph);

// Add some vertices and edges - Note the use of "iterate".
g.addV("airport").property("code","AUS").as("aus").
  addV("airport").property("code","DFW").as("dfw").
  addV("airport").property("code","LAX").as("lax").
  addV("airport").property("code","JFK").as("jfk").
  addV("airport").property("code","ATL").as("atl").
  addE("route").from("aus").to("dfw").
  addE("route").from("aus").to("atl").
  addE("route").from("atl").to("dfw").
  addE("route").from("atl").to("jfk").
  addE("route").from("dfw").to("jfk").
  addE("route").from("dfw").to("lax").
  addE("route").from("lax").to("jfk").
  addE("route").from("lax").to("aus").
  addE("route").from("lax").to("dfw").iterate();
```

Having created a new graph, we can run some queries to make sure it looks correct. Firstly, let's check that the vertices were created and look at the IDs that were allocated to them. As with prior examples, a call to 'valueMap' with a parameter of 'true' will return what we need. What we will get back from this code is a list of maps, with each map containing keys for the airport code, the vertex

ID and the vertex label.

```
List<Map<Object,Object>> vm = new ArrayList<Map<Object,Object>>() ;

vm = g.V().valueMap(true).toList();
```

Having got our list of maps back, we can process them. Note that to get the 'id' and 'label' values from the map we had to prefix the key name with a '"T."'. This is because while most of the property keys are Strings, IDs and labels are a special case. If you look at the TinkerPop documentation, you will see that T is a Java Enum that contains definitions for 'T.id', 'T.label', 'T.value' and 'T.key'. When working with Gremlin in Java, it is important to remember that we need to use the '"T."' prefix in cases where when using the console, we would not have to.

```
// Display the code property as well as the label and id.
for( Map m : vm) {
  System.out.println(((List)(m.get("code"))).get(0) + " " +
                             m.get(T.id) + " " + m.get(T.label));
}
```

If all has gone well during graph creation, we should get back a list like the one below that shows us the ID that has been given to each vertex.

```
AUS 0 airport
DFW 2 airport
LAX 4 airport
JFK 6 airport
ATL 8 airport
```

Finally, let's check that the edges were created correctly by displaying all paths between vertices in our new graph.

```
// Display the routes in the graph we just created

List<Path> paths = new ArrayList<Path>();

paths = g.V().out().path().by("code").toList();

for (Path p : paths) {
  System.out.println(p.toString());
}
```

Once again, if everything has worked as expected, here is what we should get back.

```
[AUS, DFW]
[AUS, ATL]
```

```
[DFW, JFK]
[DFW, LAX]
[LAX, JFK]
[LAX, AUS]
[LAX, DFW]
[ATL, DFW]
[ATL, JFK]
```

6.1.9. Saving a graph from a Java application

Having created our new graph, we may want to save it. The code below shows how to save the graph as either GraphSON (TinkerPop's JSON format) or as GraphML (XML). This is another instance where we have to do a bit more work as we are running in a standalone program and not inside the Gremlin Console. Our attempts to save our data have to catch any exceptions that may occur. This code is also included as part of the CreateGraph.java sample program.

```
// Save the graph we just created as GraphML (XML) or GraphSON (JSON)
try {
  // If you want to save the graph as GraphML uncomment the next line. Since this
  // is Java code it is important to include the iterate() terminal step to execute
  // the traversal
  g.io("mygraph.xml").write().iterate();

  // If you want to save the graph as JSON uncomment the next line
  //g.io("mygraph.json").write().iterate();
} catch (IOException ioe) {
  System.out.println("Graph failed to save");
}
```

It's worth adding that with TinkerGraph there is a second way to save your data that allows it to happen a bit more automatically. If you configure your TinkerGraph at startup to include a 'graphLocation' and 'graphFormat', TinkerGraph will automatically save the graph on a call to the 'close'. Earlier, the graph was created with the following code:

```
// Create a new (empty) TinkerGraph
TinkerGraph graph = TinkerGraph.open();
```

The 'open' method has the option to accept a 'Configuration' object. In that configuration we would add those two settings.

```
// Create a new (empty) TinkerGraph with a configuration
Configuration conf = new BaseConfiguration();
conf.setProperty("gremlin.tinkergraph.graphLocation", "mygraph.xml");
conf.setProperty("gremlin.tinkergraph.graphLocation", "graphml");
TinkerGraph graph = TinkerGraph.open(configuration);

...
```

```
// call close() to save as GraphML to a file called mygraph.xml
try {
    graph.close();
} catch (Exception ex) {
  System.out.println("Graph failed to save");
}
```

'Configuration' refers to the Apache Configuration library.

6.2. Working with TinkerGraph from a Groovy application

Earlier in this book, in the "Making Gremlin even Groovier" section we explored the ways that you can use Groovy code within the Gremlin Console. However, we have not yet looked at how you can use a standalone Groovy application to work with a TinkerGraph.

You will find several Groovy samples at the GitHub repository associated with this book. https://github.com/krlawrence/graph

In this section we will rewrite parts of the test application we coded in Java earlier in Groovy. A lot of what was covered in the "Working with TinkerGraph from a Java Application" section is equally relevant here, and we have not duplicated that material. So even if you are writing a Groovy application, please also give that section a read.

The source code in this section comes from the 'TinkerGraphTest.groovy' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/groovy.

It is assumed that you have downloaded and installed Groovy for your environment and have set the PATH to point to wherever the Groovy binaries are located. Just as in the Java example, the first thing we need to do is pull in via 'import' all the TinkerPop classes that our program will use.

```
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.__
import org.apache.tinkerpop.gremlin.process.traversal.Path
import org.apache.tinkerpop.gremlin.process.traversal.*
import org.apache.tinkerpop.gremlin.structure.Edge
import org.apache.tinkerpop.gremlin.structure.Vertex
import org.apache.tinkerpop.gremlin.structure.io.IoCore
import org.apache.tinkerpop.gremlin.tinkergraph.structure.*
import org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerGraph
import org.apache.tinkerpop.gremlin.util.Gremlin
import java.io.IOException
```

Having imported the classes we need, we can make a start on our application. Initially, we are just going to display the version of TinkerPop that we are using.

```
println "The Gremlin version is ${Gremlin.version()}"

def graph = TinkerGraph.open()
def g = traversal()
```

We are now ready to create a new TinkerGraph instance, 'GraphTraversalSource' named "g" and try to load the air-routes graph. We can do this in Groovy just like we did in the Java example. Unlike Java, however, Groovy does not require you to catch exceptions that may get thrown, but as a best practice it is probably a good idea to still do so when you need to take some specific action if an exception does happen.

```
println "Loading the air-routes graph...\n"

// Load the air-routes graph
try {
  g.io("air-routes.graphml").read().iterate()
} catch (IOException e) {
  println "Could not load the graph file"
  System.exit(0)
}
```

First, we just do a simple query to find the vertex representing the AUS airport and use 'println' to display some information about it.

```
def aus = g.V().has('code','AUS').valueMap().next()

println aus
```

Just as when we were looking at building a Java application, you need to terminate your query with a step such as 'next', 'toList' or 'fill' to make sure you get back the results that you expect.

Now that we have the beginnings of an application, it's time to make sure we can compile it. The key thing we need to do is make sure we pick up the TinkerPop specific JAR files as shown in the next section.

6.2.1. Compiling our Groovy application

In the Java example, we included the CLASSPATH on the 'javac' invocation using the '-cp' flag. That can also be done when using Groovy, however, if you are using Microsoft Windows as your build environment, you may find that using '-cp' gives unexpected errors. Therefore, it is recommended to define the CLASSPATH variable in your environment before running the Groovy compiler.

The commands below work in a Bash shell but could be easily ported to other environments. The GREMLIN variable should point to wherever you have installed and unzipped the Gremlin Console download

```
# Root directory for Gremlin Console install
GREMLIN=...

# Path to Gremlin core JARs
LIBPATH=$GREMLIN/lib/*

# Path to TinkerGraph JARs
EXTPATH=$GREMLIN/ext/*

#Path to additional JARs
ADDL=$GREMLIN/ext/tinkergraph-gremlin/lib/*

# Classpath
export CLASSPATH=$CLASSPATH:$LIBPATH:$EXTPATH:$ADDL

# Compile
groovyc TinkerGraphTest.groovy
```

6.2.2. Running our Groovy application

Assuming that the project is compiling cleanly, we can now run our Groovy application.

```
groovy TinkerGraphTest
```

And here is the sort of output we should get back.

```
Gremlin version is 3.8.0
Loading the air-routes graph...

[country:[US], code:[AUS], longest:[12250], city:[Austin], elev:[542], icao:[KAUS],
lon:[-97.6698989868164], type:[airport], region:[US-TX], runways:[2], lat:
[30.1944999694824], desc:[Austin Bergstrom International Airport]]
```

Now that we have a small skeleton of a test program running, we can start to add a few more interesting features to it.

6.2.3. Adding to our Groovy application

The code below demonstrates how to work with the 'valueMap' for the Austin vertex that we created a few lines back in our program.

```
// Retrieve the city name property and display it
def city = aus['city']
println "\nThe AUS airport is in ${city[0]}\n"

// Iterate through the keys we got back and print them along with their values
```

```
aus.each {println "${it.key} : ${it.value[0]}"}
```

If we were to re-compile and run our program now, the lines that we added will generate the following output.

```
The AUS airport is in Austin

country : US
code : AUS
longest : 12250
city : Austin
elev : 542
icao : KAUS
lon : -97.6698989868164
type : airport
region : US-TX
runways : 2
lat : 30.1944999694824
desc : Austin Bergstrom International Airport
```

As we did in our Java program earlier, we can add a query to see how many routes there are that originate at the DFW airport.

```
def n = g.V().has("code","DFW").out().count().next()
println "\nThere are ${n} routes from Dallas"

There are 253 routes from Dallas
```

Next, we can add some code to find the IATA codes representing the places that we can fly to from DFW.

```
// Where can I fly to from DFW?
def fromDfw = g.V().has("code","DFW").out().values("code").toList()
println "\nHere are the places you can fly to from DFW\n"
println fromDfw
```

When we run the code, we should get back some results that look like this.

```
Here are the places you can fly to from DFW

[YYZ, YVR, LHR, CDG, FRA, HEL, NRT, SYD, DXB, DUB, HKG, PEK, PVG, FCO, AMS, MAD, MUC,
 RSW, YUL, YEG, YYC, DOH, ICN, GIG, GRU, EZE, LIM, SCL, MEX, TLH, PIT, PDX, OKC, ONT,
 IST, AUH, CLT, CUN, PSP, MEM, CVG, IND, MCI, STL, ABQ, MKE, OMA, TUL, PVR, OGG, DUS,
 NAS, EYW, KEF, MBJ, LIT, AUA, ORF, JAX, PUJ, SJO, SMF, RTB, TGU, COS, HSV, BHM, RAP,
 SDF, BUF, SHV, BOI, LBB, ECP, HRL, RNO, CMH, ICT, MAF, BDL, BIL, SGF, RIC, CCS, TXK,
 PIA, LEX, GUA, CRP, ABI, ACT, CLL, BMI, BOG, BPT, DSM, MYR, AEX, CZM, AGU, MTY, AMA,
```

```
 BJX, BRO, BTR, BZE, CAE, CHA, CHS, CMI, COU, DAY, CUU, DRO, EVV, FAR, FAT, FSD, FSM,
 FWA, GCK, GDL, GGG, GJT, GPT, GRI, GRK, GRR, GSO, GSP, JAN, JLN, LAW, LCH, LFT, LIR,
 LRD, MFE, MGM, MHK, MLI, MLM, MLU, MOB, MSN, MZT, PBC, PLS, PNS, PTY, QRO, ROW, SAL,
 SAV, SJD, SJT, SLP, SPI, SPS, TRC, TYR, TYS, VPS, XNA, ZCL, AVL, EGE, HDN, SRQ, AGS,
 ILM, UIO, GNV, SDQ, SAP, MID, MGA, GCM, SKB, BUR, MEI, PIB, SBN, KOA, SUX, BFL, MRY,
 SBA, ASE, GEG, YUM, DGO, BZN, JAC, MSO, BKG, BIS, CYS, GUC, MTJ, TVC, AUG, ACA, OAX,
 FLG, CNM, GLH, SWO, CVN, DRT, ATL, ANC, AUS, BNA, BOS, BWI, DCA, FLL, IAD, IAH, JFK,
 LAX, LGA, MCO, MIA, MSP, ORD, PBI, PHX, RDU, SEA, SFO, SJC, TPA, SAN, SNA, SLC, LAS,
 DEN, SAT, MSY, EWR, CID, HNL, HOU, ELP, SJU, CLE, OAK, TUS, SAF, PHL, DTW]
```

The code below will discover all airports in the United States that you can fly to from London's Heathrow airport (LHR). Only the first 10 results are selected using a 'limit' step. A 'path' step is used to nicely return the airport pairs and the distance between them. What we get back is a list of paths, so we can use a simple Groovy 'each' loop to print the results.

```
def lhrToUsa = g.V().has("code","LHR").outE().inV().
                    has("country","US").limit(10).
                    path().by("code").by("dist").toList()

println "\nFrom LHR to airports in the USA (only 10 shown)\n"

lhrToUsa.each {println it}
```

Here are the 10 routes that were returned when we ran the code having added this new query.

```
From LHR to airports in the USA (only 10 shown)

[LHR,3707,PIT]
[LHR,4896,PDX]
[LHR,3980,CLT]
[LHR,4053,CHS]
[LHR,4198,ATL]
[LHR,4901,AUS]
[LHR,4168,BNA]
[LHR,3254,BOS]
[LHR,3622,BWI]
[LHR,4736,DFW]
```

Finally, let's write the code to find the airports in England that you can get to from Austin with no more than one stop.

```
def eng = g.V().has("code","AUS").repeat(__.out()).emit().times(2).
                has("region","GB-ENG").dedup().values("code").toList();

println "\nAirports in England reachable with no more than one stop from AUS"
println "\n${eng}\n"
```

Here are the results, and it looks like we can get to a total of nine different airports if we make no more than one stop on the way.

```
Airports in England reachable with no more than one stop from AUS

[LHR, LGW, MAN, STN, BRS, BHX, DSA, LCY, LBA, NCL,
 NQY, EMA, LPL, LTN, SOU, SEN, HUY, MME, NWI, EXT]
```

Now that we have a somewhat interesting Groovy application up and running, should you so choose, you can build upon this foundation just as we did in the "Working with TinkerGraph from a Java Application" section.

6.2.4. Using Gremlin predicates in a Groovy application

Just as we had to do when writing a standalone Java application, when you use a Gremlin predicate such as 'eq' or 'neq' from a Groovy program, you need to prefix it with a '"P."' which references the TinkerPop class of the same name where a set of static methods, representing the Gremlin predicates, are defined.

The source code in this section comes from the 'GraphRegion.groovy' sample located at https://github.com/krlawrence/graph/tree/main/sample-code/groovy.

In the code below, the 'findByRegion' method that we wrote in Java earlier has been ported to Groovy. As before, a String representing a three-character airport IATA code is expected as input. The method then uses a Gremlin query to figure out which geographical region the specified airport is in and then returns all airports also in that region. Once again, note the use of 'P.eq' as part of the 'where' step.

```
def findByRegion(iata)
{
  println("\nRegion code lookup for " + iata )

  def list =
    g.V().has("code",iata).values("region").as("r").
      V().hasLabel("airport").as("a").values("region").
          where(P.eq("r")).by().
          local(__.select("a").values("city","code","region").fold()).toList()

  list.each {println it}
}
```

Chapter 7. INTRODUCING GREMLIN SERVER

We have now explored a few ways to set up a TinkerPop enabled graph store. We've focused on running TinkerGraph locally as an in-memory graph. It's the easiest place to start because it's packaged with Gremlin Console, but also because it requires minimal configuration and environmental dependencies to begin use.

Not all graphs are this simple. Many graphs have underlying data stores with separate indexing components. Others are backed by extensive cloud infrastructure and run as a managed service. In these cases, it is desirable, or even required, that most of the implementation details be hidden and access secured. This is where Gremlin Server comes in.

Gremlin Server, as its name suggests, offers a way of setting up access to a graph that via a front end web server. In this way the user of the graph only has to know the host name or IP address of Gremlin Server to communicate with a graph. You can set Gremlin Server up on your local machine, which is useful for testing, but you can also use it to set up a graph on a remote server and allow users to access it. A graph may even just expose Gremlin Server interfaces for you to connect to as the only way to have access.

Gremlin Server supports a number of different connection protocols and methods. You can connect to it from a Gremlin Console, from a command line using 'curl' commands, or from an application. Gremlin Server has a second advantage over allowing us to hide the graph implementation details. It allows people using programming languages that do not yet have Apache TinkerPop language bindings to work with a graph using simple HTTP protocols.

The official Apache TinkerPop documentation includes in-depth coverage of configuring and using Gremlin Server. https://tinkerpop.apache.org/docs/current/reference/#gremlin-server

Gremlin Server offers a lot of valuable capabilities. In this section we are going to explain how to customize a local Gremlin Server setup with TinkerGraph. There are many other useful ways that Gremlin Server can be configured, deployed, and used. If you plan to experiment further with Gremlin Server, we very much encourage you to read the official documentation.

For managed graph services that expose a Gremlin Server interface, there typically isn't any access to the low-level configuration options mentioned here. While some options may be exposed, most are hidden and maintained internally.

7.1. Configuring Gremlin Server

We've mentioned that Gremlin Server interfaces are often exposed by various graph databases that have complicated underlying configurations and environments. While a helpful advantage for production workloads, it is less helpful when doing development, particularly when writing tests. Gremlin Server therefore presents a helpful way for programmers to do local development without requiring the associated infrastructure of their production graph. TinkerGraph can be configured within Gremlin Server to try to closely mimic the behavior of the that production graph.

You can download Gremlin Server from the official Apache TinkerPop website at https://tinkerpop.apache.org/

It only takes a few minutes to get Gremlin Server installed and running. You download the ZIP, 'unzip' it, and you are all set. As with Gremlin Console, Gremlin Server also requires Java to be at version 11 or higher.

If you look at the files that were unzipped on your machine, you will find a path of 'conf/gremlin-server'. Inside this directory you will find a set of YAML and properties files that can be used to start a Gremlin Server working with TinkerGraph in and a variety of different ways.

For the rest of this discussion, we are going to use the 'gremlin-server.yaml' file as our starting point and make minor modifications to it. Gremlin Server by default is configured for a WebSockets connection, and that is how Gremlin Console connects to it. Using WebSockets is the recommended approach when possible as it allows for a long-running full duplex connection. However, there are still many use cases where supporting an HTTP connection is desirable. There is also a third option that allows both WebSockets and HTTP connections. In the YAML file used when starting Gremlin Server, you need to specify one of the following.

org.apache.tinkerpop.gremlin.server.channel.WebSocketChannelizer

- The server will expect a WebSockets connection (this is the default).

org.apache.tinkerpop.gremlin.server.channel.HttpChannelizer

- The server will expect an HTTP connection.

org.apache.tinkerpop.gremlin.server.channel.WsAndHttpChannelizer

- The server will accept both WebSockets and HTTP connections.

The first part of the 'gremlin-server.yaml' file, modified to meet our needs, is shown below. We did not modify the parts of the file that are not shown, but we encourage you to look at the whole file and study the settings. For our current needs the defaults are fine, but this may not be true for your environment. The TinkerPop documentation has detailed coverage of the settings and what they do.

OK, so let's look at the parts of the YAML file that are relevant to this experiment. Note that we have chosen to use the 'WsAndHttpChannelizer'. This is because we want our development environment to connect with both TinkerPop's drivers in our Java application over WebSockets and other applications such as 'curl' and 'Ruby' over HTTP to connect to our new Gremlin Server.

Notice also that the 'tinkergraph-empty.properties' file is specified in the 'graphs' section. This is a file provided as part of the Gremlin Server download. This is the file that Gremlin Server will use to configure the TinkerGraph to use for our unit testing. We will examine that more closely a bit later.

The 'evaluationTimeout' setting is important. It tells Gremlin Server how long to let a query run before terminating it. This essentially establishes the maximum amount of time any query will be allowed to run, regardless of whether it has completed or not. For this experiment the default setting of '30000' should be more than adequate. The value represents the number of milliseconds allowed. If you want to allow queries sent to the server to run for longer, you can increase this value. Keep in mind for a production environment that if you have multiple users using the same Gremlin Server, you may not want to allow someone to run a really complex query that might take a long time to complete. If you want to disable the timeout feature, you can do that by specifying a

timeout value of '0'.

gremlin-server.yaml

```
host: 0.0.0.0
port: 8182
evaluationTimeout: 30000
channelizer: org.apache.tinkerpop.gremlin.server.channel.WsAndHttpChannelizer
graphs: {
  graph: conf/tinkergraph-empty.properties }
scriptEngines: {
  gremlin-groovy: {
    plugins: { org.apache.tinkerpop.gremlin.server.jsr223.GremlinServerGremlinPlugin:
{},

org.apache.tinkerpop.gremlin.tinkergraph.jsr223.TinkerGraphGremlinPlugin: {},
               org.apache.tinkerpop.gremlin.groovy.jsr223.GroovyCompilerGremlinPlugin:
{enableThreadInterrupt: true},
               org.apache.tinkerpop.gremlin.jsr223.ImportGremlinPlugin: {classImports:
[java.lang.Math], methodImports: [java.lang.Math#*]},
               org.apache.tinkerpop.gremlin.jsr223.ScriptFileGremlinPlugin: {files:
[scripts/empty-sample.groovy]}}}}
# The rest of the file is not shown
```

As well as the YAML file and the properties file, there is a third file that we need to provide when starting a Gremlin Server. This file can contain Groovy code that will be run when the server starts. For our purposes the default file is all we need. These files should be placed in the 'scripts' directory that is part of the standard Gremlin Server. We will take a look at the default script in a moment.

By default, Gremlin Server includes a Groovy script called 'empty-sample.groovy'. That name is a bit misleading as the file actually does some interesting things. For our purposes the most useful thing that the script does is to configure and make available to us in Gremlin Console, the graph traversal source, 'g' object that hopefully by now you are very familiar with. This provides you with a template for any other 'global' variables that you may want to make available to the user of the console connected to your Gremlin Server. The file also configures some default log messages that will be generated when the server starts and stops. Note that you can add your own code to this script or replace it with your own script entirely. You will find additional example scripts included as part of the Gremlin Server download. These scripts do things such as create a TinkerGraph instance and load some graph data as part of the server startup process. Using this technique, we could easily add a line to the script so that when the server starts, an empty TinkerGraph is created and the 'air-routes' data loaded.

empty-sample.groovy

```
// an init script that returns a Map allows explicit setting of global bindings.
def globals = [:]

// defines a sample LifeCycleHook that prints some output to Gremlin Server console.
// note that the name of the key in the "global" map is unimportant.
globals << [hook : [
```

```
        onStartUp: { ctx ->
            ctx.logger.info("Executed once at startup of Gremlin Server.")
        },
        onShutDown: { ctx ->
            ctx.logger.info("Executed once at shutdown of Gremlin Server.")
        }
] as LifeCycleHook]

// define the default TraversalSource to bind queries to - this one will be named "g".
globals << [g : traversal().with(graph)]
```

Now that we have all of our configuration files in place, we can start the Gremlin Server by typing the following command into a terminal window. The 'gremlin-server.sh' file is located in the 'bin' directory of your Gremlin Server installation.

```
$ gremlin-server.sh conf/gremlin-server/gremlin-server.yaml
```

If all goes well, you should see output from Gremlin Server displayed. The server will keep running until you kill it. In this case a simple CTRL-C is all you need to do to kill the server. After you press CTRL-C, the server will do a bit of cleaning up.

You can use the 'start' keyword to start Gremlin Server as a background task.

You can also start Gremlin Server in the background rather than have it take over your current terminal window by adding the 'start' keyword as part of the invocation command as shown below. The examples below assume that you are starting the server from the place where you installed the Gremlin Server zip file.

```
$ bin/gremlin-server.sh start

Server started 25897
```

By default, the configuration information for the server being started will look for `conf/gremlin-server.yaml`. If you want to override the default, you need to provide an environment variable called 'GREMLIN_YAML' before starting the server as shown below.

```
$ export GREMLIN_YAML='conf/mysettings.yaml'
$ bin/gremlin-server.sh start

Server started 25897
```

As an alternative to defining an environment variable, you can instead create a file called `bin/gremlin-server.conf` and put the name of your YAML file in it. An example is shown below.

```
GREMLIN_YAML='conf/mysettings.yaml'
```

If you want to check whether Gremlin Server is currently running, you can use the 'status' keyword.

```
$ bin/gremlin-server.sh status

Server running with PID 25897
```

To stop the server, you can use the 'stop' keyword as follows.

```
$ bin/gremlin-server.sh stop

Server stopped [25897]
```

7.2. Loading air-routes In Gremlin Server

We've now started and stopped Gremlin Server with an empty TinkerGraph, but we also alluded to being able to load data at the server startup by making changes to Gremlin Server's configuration files. Let's load the air-routes dataset so that our TinkerGraph has some data ready for querying when Gremlin Server starts. To load the data, we need to edit 'empty-sample.groovy', which is the initializer for the server. Since TinkerPop packages the air-routes dataset starting at TinkerPop 3.8.0, we will also use the 'TinkerFactory' as a convenient way to load this data.

```
// an init script that returns a Map allows explicit setting of global bindings.
def globals = [:]

// defines a sample LifeCycleHook that prints some output to the Gremlin Server
console.
// note that the name of the key in the "global" map is unimportant.
globals << [hook : [
        onStartUp: { ctx ->
            ctx.logger.info("Executed once at startup of Gremlin Server.")
            org.apache.tinkerpop.gremlin.tinkergraph.structure.TinkerFactory
.generateAirRoutes(graph)
        },
        onShutDown: { ctx ->
            ctx.logger.info("Executed once at shutdown of Gremlin Server.")
        }
] as LifeCycleHook]

// define the default TraversalSource to bind queries to - this one will be named "g".
globals << [g : traversal().with(graph)]
```

We can see in the above script that we added a line to 'onStartUp' that will load the air-routes data

into the 'graph'. While this example made use of 'TinkerFactory' for an easy way to load the data, we could have equally chosen to use any other means we'd like to set up the graph.

7.3. Connecting to a Gremlin Server from the Gremlin Console

It is fairly straightforward to connect to a running Gremlin Server from a Gremlin Console. Gremlin Server very nicely hides the back end implementation details from us. As far as we are concerned, it is just an HTTP or WebSockets endpoint that can handle Gremlin queries.

There is perhaps one exception to our statement about not needing to worry about server side implementation details. This exception is a result of potential version mismatches. Typically, the TinkerPop download, assuming you have the very latest, will be at least a few minor point releases ahead of any given graph store release. This is purely the case because whenever TinkerPop has a release, it takes a bit of time for the graph database maintainers to catch up. We will give a concrete example of this in a moment.

As with Gremlin Server, YAML files can be used to configure a remote connection from the Gremlin Console. The Gremlin Console includes a set of YAML files that can be used as-is or edited as needed. To connect the Gremlin Console to the Gremlin Server that we just configured, the file 'remote.yaml' can be used. It is essential that the console and the server be using the same version. If they are not, Gremlin Console and Gremlin Server may not be able to correctly communicate. Note that in the 'remote.yaml' file we also specify the name and port of the Gremlin Server host that we will be connecting to. As we are running everything locally, the default host name of 'localhost' is fine. If you are connecting to a remote Gremlin Server, the 'hosts' value needs to be edited to correctly identify the name or IP address of the server where Gremlin Server is running. Also, we can use the default port of 8182. By default, a Gremlin Server listens on port 8182. The only reason you would need to change this value is if you are connecting to a Gremlin Server using a different port.

remote.yaml

```
hosts: [localhost]
port: 8182
serializer: { className:
org.apache.tinkerpop.gremlin.util.ser.GraphBinaryMessageSerializerV1, config: {
serializeResultToString: true }}
```

Be sure that you align the version of TinkerPop you are using with the one supported for the version of the graph database you have chosen. Consult the provider documentation for these details.

7.3.1. Making the remote connection

Now that we have our YAML file ready, all that we have to do to establish a connection between our Gremlin Console and Gremlin Server is to issue the following command once the console is running. Be sure that Gremlin Server is running before issuing this command.

```
gremlin> :remote connect tinkerpop.server conf/remote.yaml
```

Now that we are connected to Gremlin Server, we can send some Gremlin commands. Given that the 'air-routes' graph is already loaded into our remote graph, we can immediately start to issue some queries. To make sure the query goes to the remote graph, the query needs to be prefixed with '":>"'.

```
gremlin> :> g.V().count()

3749
```

7.3.2. Gremlin Console's 'result' variable

When working within Gremlin Console, it is useful to be aware of the fact that results of queries sent to a server when the console is in '"local mode"', are made available in two ways. They are displayed as a result but are also stored in a variable called 'result'. Take a look at the query below.

```
gremlin> :>  g.V().hasLabel('continent').group().by('desc').by(out().count())

==>{South America=313, Asia=971, Europe=605, Africa=321, Antarctica=0, North America
=989, Oceania=305}
```

If we were to print the contents of the 'result' variable, we would find it contains the results from the query.

```
gremlin> println result

[result{object={South America=313, Asia=971, Europe=605, Africa=321, Antarctica=0,
North America=989, Oceania=305} class=java.lang.String}]
```

As the console is still in local mode, we can use some inline Groovy code to post-process, in this case pretty print, the contents of 'result'. This capability is worth keeping in mind. There are some interesting things it allows you to do, such as doing local analysis on the results and saving them to a file locally when working with a remote server.

```
gremlin> for (x in result['object'][0][1..-2].split(', ')) println x

South America=313
Asia=971
Europe=605
Africa=321
Antarctica=0
North America=989
Oceania=305
```

7.3.3. Working in remote mode

As useful as keeping the console in '"local mode"' can be, if you are going to be issuing a lot of queries to the remote graph, We find it more convenient to put the console into '"remote mode"'. This can be done as follows.

```
gremlin> :remote console

All scripts will now be sent to Gremlin Server - [localhost/127.0.0.1:8182] - type
':remote console' to return to local mode
```

The console is now in '"remote mode"'. All queries that you enter will be sent to Gremlin Server, and there is no need to use the '":>"' prefix.

```
gremlin> g.V().count()

3749
```

One thing to notice is that the output that comes back from Gremlin Server looks a little different at times from when you use the commands using the Gremlin Console attached to a local TinkerGraph. This is because Gremlin Console essentially does a 'toString()' on the output before it is shown to the user in these cases.

```
gremlin> g.V().has('code','AUS').valueMap()

{country=[US], code=[AUS], longest=[12250], city=[Austin], elev=[542], icao=[KAUS],
lon=[-97.6698989868164], type=[airport], region=[US-TX], runways=[2], lat
=[30.1944999694824], desc=[Austin Bergstrom International Airport]}
```

As an example of the slight differences in the output format, below you will find the results from the same query when the graph was running as a local, in memory, TinkerGraph.

```
[country:[US],code:[AUS],longest:[12250],city:[Austin],elev:[542],icao:[KAUS],lon:[-
97.6698989868164],type:[airport],region:[US-
TX],runways:[2],lat:[30.1944999694824],desc:[Austin Bergstrom International Airport]]
```

Once you are done sending all commands to Gremlin Server, you can switch out of that mode as follows. Commands will now be sent to your local console. This means that you can work with a local and remote graph at the same time. The ':remote console' command is therefore a toggle. Each time you use the command, the console will switch between local mode and remote mode or vice versa.

```
gremlin> :remote console

==>All scripts will now be evaluated locally - type ':remote console' to return to
```

```
remote mode for Gremlin Server - [localhost/127.0.0.1:8182]
```

If you are completely done with the remote connection for this console session, you can truly close it as follows. Having done this, you will need to reestablish the connection before the ':remote console' will work again.

```
gremlin> :remote close

==>Removed - Gremlin Server - [localhost/127.0.0.1:8182]
```

7.4. Connecting to a Gremlin Server from the command line

Now that we have a Gremlin Server up and running that supports both HTTP and Web Sockets connections, we can, if we wish, communicate with it using nothing more than a 'curl' command. The 'curl' command below uses an HTTP GET to send a query to our Gremlin Server.

```
$ curl "http://localhost:8182?gremlin=g.V().has('code','AUS').valueMap()"
```

In response to the HTTP GET request, the server sends back the result packaged as JSON as follows. We have formatted the output in a way that makes it easier to read. What was actually returned did not have any line breaks in it at all and was quite hard to read.

```
{"requestId":"a8ad654a-a5a3-4bb9-8474-69aca3c3db1e",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"country":["US"],
                       "code":["AUS"],
                    "longest":[12250],
                       "city":["Austin"],
                       "elev":[542],
                       "icao":["KAUS"],
                        "lon":[-97.6698989868164],
                       "type":["airport"],
                     "region":["US-TX"],
                    "runways":[2],
                        "lat":[30.1944999694824],
                       "desc":["Austin Bergstrom International Airport"]}],
           "meta":{}}}
```

The following example shows how to send the same query, but with the 'valueMap' step removed, using an HTTP POST. The Apache TinkerPop documentation states that using POST is the recommended way to send queries over HTTP to a Gremlin Server. Note how in this case we are sending the query packaged as JSON and that we have to escape the quote characters.

```
$ curl -X POST -d "{\"gremlin\":\"g.V().has('code','AUS')\"}" \
           "http://localhost:8182"
```

As with the prior query, the HTTP POST form of the query also returns the result packaged as JSON. However, in this case, because we left off the 'valueMap' step, the JSON includes additional information in the form of the ID values and labels for the vertex and its properties. This is because the result represents a vertex this time rather than a map. We have again formatted the output in a way that is easier to read.


```
{"requestId":"64c757b8-27a6-4509-a54c-ea35ba517667",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{
   "data":[
     {"id":12352,
      "label":"airport",
      "type":"vertex",
      "properties":
        {"country":[{"id":"8p4-9j4-8p1", "value":"US"}],
            "code":[{"id":"93c-9j4-5j9", "value":"AUS"}],
         "longest":[{"id":"9hk-9j4-mx1", "value":12250}],
            "city":[{"id":"9vs-9j4-7wl", "value":"Austin"}],
            "elev":[{"id":"aa0-9j4-but", "value":542}],
            "icao":[{"id":"ao8-9j4-6bp", "value":"KAUS"}],
             "lon":[{"id":"b2g-9j4-dfp", "value":-97.6698989868164}],
            "type":[{"id":"bgo-9j4-745", "value":"airport"}],
          "region":[{"id":"buw-9j4-9hh", "value":"US-TX"}],
         "runways":[{"id":"c94-9j4-b2d", "value":2}],
             "lat":[{"id":"cnc-9j4-cn9", "value":30.1944999694824}],
            "desc":[{"id":"d1k-9j4-a9x",
                  "value":"Austin Bergstrom International Airport"}]}}],
            "meta":{}}}
```


We have included examples of the different types of JSON results that you are likely to have to process in the "More examples of the JSON returned from a Gremlin Server" section that is coming up soon.

7.5. Connecting to a Gremlin Server from Java using 'with'

While it is perfectly possible to work directly with the JSON returned from Gremlin Server, it is often more desirable to have the results placed directly into variables of the appropriate type. If the appropriate Gremlin language driver exists for the programming language that you are using, this is quite easy to set up. In this section we will look at connecting to a Gremlin Server from a Java application and taking advantage of the remote connection capability that TinkerPop provides.

The source code in this section comes from the 'RemoteClient.java' sample located

> at https://github.com/krlawrence/graph/tree/main/sample-code/java.

Let's look at the small application in sections. First, it is necessary to import the required classes that we will need to make the connection to the server and retrieve the query results.

```
import org.apache.tinkerpop.gremlin.driver.Cluster;
import org.apache.tinkerpop.gremlin.process.traversal.dsl.graph.GraphTraversalSource;
import org.apache.tinkerpop.gremlin.driver.remote.DriverRemoteConnection;
import org.apache.tinkerpop.gremlin.util.ser.GraphBinaryMessageSerializerV1;
import java.util.Map;
import java.util.List;
import java.util.ArrayList;

import static
org.apache.tinkerpop.gremlin.process.traversal.AnonymousTraversalSource.traversal;
```

We can now define a small Java class that we will call 'RemoteClient' and set up the connection to Gremlin Server. This is done by first creating a 'Cluster.Builder' instance that will be used to describe the server we are connecting to and the protocol we want to use. It is important that these settings match what Gremlin Server is configured to use. For this basic example we are just using 'localhost' as the host name, but the name of any Gremlin Server that you have access to can be used instead. The default Gremlin Server port of '8182' is specified, and the 'GraphBinaryMessageSerializerV1' serialization format is selected. Again, this needs to match both the protocol and the version of the protocol that your Gremlin Server is supporting.

```
public class RemoteClient
{
  public static void main( String[] args )
  {
    Cluster.Builder builder = Cluster.build();
    builder.addContactPoint("localhost");
    builder.port(8182);
    builder.serializer(new GraphBinaryMessageSerializerV1());
```

Once the 'Cluster.Builder' instance has been set up, we can use it to create our 'Cluster' instance.

```
Cluster cluster = builder.create();
```

Lastly, we need to set up a 'GraphTraversalSource' object for Gremlin Server hosted graph that we will be working with. This object is often named "g" by convention and is typically created using the statically imported 'traversal()' method, which in turn allows the call to 'with' to bind the traversal source to a graph. We've seen 'with' take a graph instance before, but this time we will use it with a reference to a remote connection to a graph in Gremlin Server. Note that the cluster instance that we just created is passed in as a parameter. While this looks a little complicated, it is really not a lot different from when we connect to a local graph using the Gremlin Console. The only difference is that by setting up the remote connection this way, when we start to issue queries against the graph,

rather than getting JSON objects back, the results will automatically be serialized into Java variables for us. This makes our code a lot easier to write and essentially is the same code from this point onwards that would also work with a local graph that we are directly connected to.

```
GraphTraversalSource g = traversal().
        with(DriverRemoteConnection.using(cluster));
```

We can now use our new 'GraphTraversalSource' object to issue a Gremlin query. The results will be placed directly into the 'List' called 'vmaps'. The query finds the first 10 airports with a region code of 'GB-ENG', which is short for Great Britain–England.

```
    List <Map<String,Object>> vmaps =
      g.V().has("airport","region","GB-ENG").limit(10).valueMap().toList();

    System.out.println("\n\nThe following airports were found\n");
    for (Map <String,Object> m : vmaps) {
      ArrayList code = (ArrayList) m.get("code");
      ArrayList desc = (ArrayList) m.get("desc");
      System.out.println(code.get(0) + " , " + desc.get(0));
    }

    cluster.close();
  }
}
```

When the Java application is compiled and run, the output should look similar to that shown below.

```
LEQ , Land's End Airport
LGW , London Gatwick
MAN , Manchester Airport
LHR , London Heathrow
LCY , London City Airport
STN , London Stansted Airport
EMA , East Midlands Airport
LPL , Liverpool John Lennon Airport
LBA , Leeds Bradford Airport
NCL , Newcastle Airport
```

7.6. Connecting to a Gremlin Server from Ruby

As far as we know, at the time of writing, there is currently no formal Gremlin language binding support available for Ruby programmers. This is therefore a perfect use case to show how, using a small amount of code, a Ruby programmer can connect to a Gremlin Server and issue Gremlin Queries.

The source code in this section comes from the 'gremlin-client-http.rb' sample

> located at https://github.com/krlawrence/graph/tree/main/sample-code/ruby.

The code below represents a complete, standalone Ruby application. It uses the standard Ruby libraries. No additional Ruby Gems or third party libraries should be required. The example as shown connects to a Gremlin Server running on your local machine. It packages up an HTTP POST request and sends it to Gremlin Server. The body of the HTTP request is encoded as JSON.

gremlin-client-http.rb

```
# Simple example of how you can connect to a Gremlin Server and
# issue queries from a Ruby application.

require 'net/http'
require 'uri'
require 'json'

uri = URI.parse("http://localhost:8182")

request = Net::HTTP::Post.new(uri)
req_options = { use_ssl: uri.scheme == "https", }

query = {"gremlin" => "g.V().has('code','AUS').out().count()"}
request.body = JSON.dump(query)

response = Net::HTTP.start(uri.hostname, uri.port, req_options) do |http|
  http.request(request)
end

puts "Response code from the server was #{response.code}"
puts response.body
```

Here is the output returned when we ran the program using Ruby version 2.3.1.

As you can see, the result body contains a JSON object just as when we issued requests using the 'curl' command earlier.

```
{"requestId":"0129e905-6903-4658-9cfb-23404842ba12",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[62],"meta":{}}}
```

7.7. Tweaking queries to make the JSON returned easier to work with

Below is a query that we have seen used earlier in this book. It finds all routes longer than 8,000 miles and returns the airport pairs and the distance between them.

```
g.V().as('a').outE().has('dist',gt(8000)).
```

```
      order().by('dist',desc).inV().as('b').
      filter(select('a','b').by('code').where('a', lt('b'))).
      path().by('code').by('dist')
```

When we run this query using the Gremlin Console with TinkerGraph, we get back results that have been to a degree 'pretty printed' by the Console as shown below.

```
[JFK,9526,SIN]
[EWR,9523,SIN]
[AKL,9025,DOH]
[LHR,9009,PER]
[PEK,8884,PTY]
[AKL,8818,DXB]
[LAX,8756,SIN]
[CAN,8754,MEX]
[IAH,8591,SYD]
[DFW,8574,SYD]
[JFK,8504,MNL]
[ATL,8434,JNB]
[SFO,8433,SIN]
[AUH,8372,LAX]
[DXB,8321,LAX]
[JED,8314,LAX]
[DOH,8287,LAX]
[LAX,8246,RUH]
[SCL,8208,TLV]
[MEL,8197,YVR]
[AKL,8186,ORD]
[DXB,8150,IAH]
[AUH,8139,SFO]
[HKG,8135,IAD]
[DFW,8105,HKG]
[DXB,8085,SFO]
[SEA,8059,SIN]
[HKG,8054,JFK]
[AUH,8053,DFW]
[EWR,8047,HKG]
[DOH,8030,IAH]
[DFW,8022,DXB]
```

However, if you were to use a system that returns the full JSON response, as is the case when using a Gremlin Server over an HTTP connection, you will not get the benefit of Gremlin Console '"pretty printing"'. Instead, you will get back something that looks a lot like this from the exact same query as the one we used above.

```
{"requestId":"5acca62c-7351-4b3d-bb20-3660f6feb3cc",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":
```

```
    [{"labels":[["a"],[],["b"]],"objects":["AKL",9025,"DOH"]},
    {"labels":[["a"],[],["b"]],"objects":["AKL",8818,"DXB"]},
    {"labels":[["a"],[],["b"]],"objects":["LAX",8756,"SIN"]},
    {"labels":[["a"],[],["b"]],"objects":["CAN",8754,"MEX"]},
    {"labels":[["a"],[],["b"]],"objects":["IAH",8591,"SYD"]},
    {"labels":[["a"],[],["b"]],"objects":["DFW",8574,"SYD"]},
    {"labels":[["a"],[],["b"]],"objects":["ATL",8434,"JNB"]},
    {"labels":[["a"],[],["b"]],"objects":["SFO",8433,"SIN"]},
    {"labels":[["a"],[],["b"]],"objects":["AUH",8372,"LAX"]},
    {"labels":[["a"],[],["b"]],"objects":["DXB",8321,"LAX"]},
    {"labels":[["a"],[],["b"]],"objects":["JED",8314,"LAX"]},
    {"labels":[["a"],[],["b"]],"objects":["DOH",8287,"LAX"]},
    {"labels":[["a"],[],["b"]],"objects":["LAX",8246,"RUH"]},
    {"labels":[["a"],[],["b"]],"objects":["MEL",8197,"YVR"]},
    {"labels":[["a"],[],["b"]],"objects":["DXB",8150,"IAH"]},
    {"labels":[["a"],[],["b"]],"objects":["AUH",8139,"SFO"]},
    {"labels":[["a"],[],["b"]],"objects":["DFW",8105,"HKG"]},
    {"labels":[["a"],[],["b"]],"objects":["DXB",8085,"SFO"]},
    {"labels":[["a"],[],["b"]],"objects":["HKG",8054,"JFK"]},
    {"labels":[["a"],[],["b"]],"objects":["AUH",8053,"DFW"]},
    {"labels":[["a"],[],["b"]],"objects":["EWR",8047,"HKG"]},
    {"labels":[["a"],[],["b"]],"objects":["DOH",8030,"IAH"]},
    {"labels":[["a"],[],["b"]],"objects":["DFW",8022,"DXB"]}],
    "meta":{}}}
```

What is being returned is useful in some cases. For example, we can see the 'a' and 'b' labels that we used in our query, but in this case all we really wanted was the last part with the airport codes and the distances. We could decide to write code to process this JSON as-is, and that is a valid choice you could make. However, by tweaking the query slightly, we can enable Gremlin to give us back what we really wanted. Let's start by looking at what happens if we add '.toList().toString()' to the end of the query. Take a look at the modified form of the query below.

```
g.V().as('a').outE().has('dist',gt(8000)).
      order().by('dist',desc).inV().as('b').
      filter(select('a','b').by('code').where('a', lt('b'))).
      path().by('code').by('dist').toList().toString()
```

If we were to send this modified form of the query to our Gremlin Server, we should get back something that looks a lot more like the result we got back when working with the Gremlin Console. As shown below, it is certainly a bit easier to process in your application now. However, this is still not an ideal result as what we now have is a list containing a single string with all of our routes in it.

```
{"requestId":"63c660d0-28cf-41fc-86cf-5560a4e2fac0","status":{"message":"","code":200
,"attributes":{}},"result":{"data":["[[AKL, 9025, DOH], [AKL, 8818, DXB], [LAX, 8756,
SIN], [CAN, 8754, MEX], [IAH, 8591, SYD], [DFW, 8574, SYD], [ATL, 8434, JNB], [SFO,
8433, SIN], [AUH, 8372, LAX], [DXB, 8321, LAX], [JED, 8314, LAX], [DOH, 8287, LAX],
[LAX, 8246, RUH], [MEL, 8197, YVR], [DXB, 8150, IAH], [AUH, 8139, SFO], [DFW, 8105,
```

```
HKG], [DXB, 8085, SFO], [HKG, 8054, JFK], [AUH, 8053, DFW], [EWR, 8047, HKG], [DOH,
8030, IAH], [DFW, 8022, DXB]]"],"meta":{}}}
```

We can add a little more post-processing to split up our single string into an array of strings where each string is a single route of the form '[AKL,9025,DOH]'. One way to do this is to trim off the unwanted characters at each end of the string and then use split to divide it up. As there are a lot of commas in the string, we could not just do a simple 'split(",")' as that would not have returned what we wanted. To make the split work, we replaced every occurrence of '],' in the string with ']x' and then did the split using 'split("x")'. Here is the modified query.

```
g.V().as('a').outE().has('dist',gt(8000)).
      order().by('dist',desc).inV().as('b').
      filter(select('a','b').by('code').where('a', lt('b'))).
      path().by('code').by('dist').toList().toString()[1..-2].
      replaceAll('],',']x').split('x')
```

Here is what we now get back in the returned JSON. Each route is now a string in an array of strings. From here it is a simple task to extract the airport names and distances for each route.

```
{"requestId":"9d8324a8-89e4-4c1e-be59-ff433784a3da",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[ " [AKL, 9025, DOH]",
                    " [AKL, 8818, DXB]",
                    " [LAX, 8756, SIN]",
                    " [CAN, 8754, MEX]",
                    " [IAH, 8591, SYD]",
                    " [DFW, 8574, SYD]",
                    " [ATL, 8434, JNB]",
                    " [SFO, 8433, SIN]",
                    " [AUH, 8372, LAX]",
                    " [DXB, 8321, LAX]",
                    " [JED, 8314, LAX]",
                    " [DOH, 8287, LAX]",
                    " [LAX, 8246, RUH]",
                    " [MEL, 8197, YVR]",
                    " [DXB, 8150, IAH]",
                    " [AUH, 8139, SFO]",
                    " [DFW, 8105, HKG]",
                    " [DXB, 8085, SFO]",
                    " [HKG, 8054, JFK]",
                    " [AUH, 8053, DFW]",
                    " [EWR, 8047, HKG]",
                    " [DOH, 8030, IAH]",
                    " [DFW, 8022, DXB]"]
```

It's really a matter of personal preference whether you decide to have the query return less data or just return the full set of data that we got back from the initial query. One advantage to having the

query limit what is returned is that less data, potentially a lot less data, will need to be sent back to your application and stored in memory or on disk. However, as most programming languages have built-in support that makes it easy to deserialize JSON objects into native data structures, you may prefer to just have all the JSON be returned and do the rest of the processing yourself.

Here is a basic example. If we added the following lines to our Ruby application that we created in the previous section and used the original query from before we added any post-processing, we could easily get at the parts of the JSON that we are interested in.

```
res = JSON.parse(response.body)['result']['data']

res.each do |x|
  p x['objects']
end
```

The code uses Ruby's 'JSON' class to convert the JSON response from the Gremlin Server into a map data structure. We can then access each part of the map by the names contained in the JSON. Note that the code as written expects a specific set of keywords to be present in the JSON. Not all query results contain these keywords. Therefore, it would take a little more work to turn this into a more general-purpose piece of code that could handle any of the possible JSON return formats the server could send to us. Here is the output from running the updated Ruby code. Notice that what we have now is a nice collection of lists, each one containing two strings and an integer. The data is now in a form that is really easy and convenient to process further.

```
["AKL", 9025, "DOH"]     ["LAX", 8246, "RUH"]
["AKL", 8818, "DXB"]     ["MEL", 8197, "YVR"]
["LAX", 8756, "SIN"]     ["DXB", 8150, "IAH"]
["CAN", 8754, "MEX"]     ["AUH", 8139, "SFO"]
["IAH", 8591, "SYD"]     ["DFW", 8105, "HKG"]
["DFW", 8574, "SYD"]     ["DXB", 8085, "SFO"]
["ATL", 8434, "JNB"]     ["HKG", 8054, "JFK"]
["SFO", 8433, "SIN"]     ["AUH", 8053, "DFW"]
["AUH", 8372, "LAX"]     ["EWR", 8047, "HKG"]
["DXB", 8321, "LAX"]     ["DOH", 8030, "IAH"]
["JED", 8314, "LAX"]     ["DFW", 8022, "DXB"]
["DOH", 8287, "LAX"]
```

In the next section you will find more examples of the JSON that can be returned by Gremlin Server and also some examples of how to reduce the amount of data that is returned.

7.8. More examples of the JSON returned from a Gremlin Server

The JSON returned by Gremlin Server depends on the query that is used and more specifically, what that query returns. Everything returned in the 'data' part of the 'result' will, at the outermost level, be an array. What is inside that array could be a simple number or a string. It could also be a

list of strings or other objects including maps. If you plan to write some general-purpose code that can handle the different possible formats, it is important to know what they look like. In the examples that follow, we have attempted to show several of the possible response formats that you may encounter. We are mainly going to focus on the parts of the JSON that follow the 'data' key. Each example assumes that the query shown was sent to a Gremlin Server using the HTTP protocol. As always, if you are unsure what JSON a particular query may generate, you should always run some experiments to find out.

Please note that some of the queries that follow may not represent the best way to achieve the specific result. We have deliberately picked queries that show different Gremlin steps to give you a feel for the type of JSON result each generates.

7.8.1. No result

The following query does not return any results. The JSON reflects this in the form of the 'data' returned being an empty list '"[]"'.

```
g.V().has('code','AUS').out('route').has('code','SYD')

{"requestId":"e68ce6d6-29a0-4a70-af35-b4e8bb123458",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[],"meta":{}}}
```

7.8.2. Integer result

A simple query that just returns a single integer result will generate JSON as shown below. The 'result' section of the JSON will contain a 'data' section with the single integer value encoded as a list with one member.

```
g.V().count()

{"requestId":"25fc4d45-3e58-4f72-99b1-fe1c6575fdd0",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[3624],"meta":{}}}
```

7.8.3. String result

As with integer results, a query that just returns a single string result will generate JSON as shown below. The 'result' section of the JSON will contain a 'data' section with the single string value encoded as a list with one member.

```
g.V().has('code','DFW').values('city')

{"requestId":"0ae1e2af-adea-487c-b365-7ef76bb56791",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":["Dallas"],"meta":{}}}
```

7.8.4. List of strings

The query below generates a 'data' array containing a list of strings representing airport codes.

```
g.V().has('code','SAF').out().values('code')

{"requestId":"264cbaf8-6679-43b0-936c-f65b9f6fd0ed",
"status":{"message":"","code":200,"attributes":{}},
"result":{"data":["PHX","DFW","LAX","DEN"],"meta":{}}}
```

7.8.5. List of integers

The query below generates a 'data' array containing a list of integers representing runway counts. Note that in reality you would not use a 'sack' for this, a simple 'values' step will generate the same results, but we wanted to show an example that uses a 'sack' step.

```
g.withSack(0).V().has('code','SAF').out().sack(sum).by('runways').sack()

{"requestId":"23598951-ffa4-440d-910f-eebc6d5f620a",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[3,7,4,6],"meta":{}}}
```

7.8.6. List of mixed types

It is common for a query result to contain a variety of different data types. The example below generates a list containing a string, and integer, and a double. Note, as we have seen before, TinkerPop does not guarantee the order in which results are returned, so does not create any dependencies on that.

```
g.V().has('code','LGW').values('city','lat','runways')

{"requestId":"6043ce66-221b-49b8-a3f9-6131eef3b9c2",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":["London",2,51.1481018066406],"meta":{}}}
```

7.8.7. Value map

As you might expect, when a 'valueMap' is used to generate the result from a query, the JSON generated also contains a map. Note how each property value is encoded in a list even if there is only one value.

```
g.V().has('code','CDG').valueMap()

{"requestId":"c989a182-aa97-4ed7-bddb-7f0e3ad237d6",
    "status":{"message":"","code":200,"attributes":{}},
    "result":{"data":[{
```

```
        "country":["FR"],
           "code":["CDG"],
        "longest":[13829],
           "city":["Paris"],
           "elev":[392],
           "icao":["LFPG"],
            "lon":[2.54999995232],
           "type":["airport"],
         "region":["FR-J"],
        "runways":[4],
            "lat":[49.0127983093],
           "desc":["Paris Charles de Gaulle"]}],"meta":{}}}
```

7.8.8. Single vertex

In Gremlin Console you get back something like "'v[51]'" when your query returns a vertex. Wit Gremlin Server what you get back is a JSON object representing everything that is known about the vertex, including its ID, label, properties, and the ID of each property. If you do not need the entire vertex returned, it might be worth writing your query in a way such that you only get back the properties that you are interested in. This is especially pertinent if your query could potentially return a lot of vertices in the result.

```
g.V().has('code','CDG')

{"requestId":"a70cab32-73a5-492f-a00b-0c7d66485b18",
    "status":{"message":"","code":200,"attributes":{}},
    "result":{"data":
       [{"id":69736,
      "label":"airport",
       "type":"vertex",
"properties":
     {"country":[{"id":"2e4t-1ht4-8p1", "value":"FR"}],
         "code":[{"id":"2ej1-1ht4-5j9","value":"CDG"}],
      "longest":[{"id":"2ex9-1ht4-mx1","value":13829}],
         "city":[{"id":"2fbh-1ht4-7wl","value":"Paris"}],
         "elev":[{"id":"2fpp-1ht4-but","value":392}],
         "icao":[{"id":"2g3x-1ht4-6bp","value":"LFPG"}],
         " lon":[{"id":"2gi5-1ht4-dfp","value":2.54999995232}],
         "type":[{"id":"2gwd-1ht4-745","value":"airport"}],
       "region":[{"id":"2hal-1ht4-9hh","value":"FR-J"}],
      "runways":[{"id":"2hot-1ht4-b2d","value":4}],
          "lat":[{"id":"2i31-1ht4-cn9","value":49.0127983093}],
         "desc":[{"id":"2ih9-1ht4-a9x","value":"Paris Charles de Gaulle"}]}}],
    "meta":{}}}
```

7.8.9. Selected vertex information

One way to limit the amount of JSON we get back is shown below. Let's assume for a selection of

airport vertices, all we are interested in is the ID, airport code and city name. We can construct a query, as shown below, that will return just those values for each vertex.

```
g.V().hasLabel('airport').sample(3).
      union(id(),values('code','city'))

{"requestId":"4d308287-9725-4fa6-8c2b-b7e517ca5009",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[45096,"SCL","Santiago",
                   610336,"YWK","Wabush",
                   163880,"CAK","Akron"],"meta":{}}}
```

7.8.10. Single edge

Just as when we queried a single vertex, when we query a single edge, we get back a lot of information including its label and ID and information about the vertices the edge is connected to.

```
g.V().has('code','SAF').outE().limit(1)

{"requestId":"cac0a975-33a0-4714-a797-1be782201a27",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"
        id":"2xhcd-1560-pat-39s",
    "label":"route",
     "type":"edge","inVLabel":"airport",
                  "outVLabel":"airport","
                         inV":4240,
                      "outV":53352,
                "properties":{"dist":369}}],"meta":{}}}
```

7.8.11. New vertex

When a new vertex and some properties are added, the returned JSON will contain all information about the vertex, including its ID, label and type as well as its properties.

```
g.addV('test').property('fruit','apple')

{"requestId":"accd4354-0db9-417d-927f-c0945e1721dc",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"id":4248,
          "label":"test",
           "type":"vertex",
     "properties":{"fruit":[{"id":"177-3a0-28lh",
                             "value":"apple"}]}}],"meta":{}}}
```

7.8.12. New vertex only returning the ID

When adding a new vertex, if you are not really interested in getting back the entire new vertex and its properties, you can write the query to only return the ID of the new vertex as shown below.

```
g.addV('test').as('a').property('fruit','apple').select('a').id()

{"requestId":"c50d0fa7-5caa-4294-8eca-310f032b1c42",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[8344],"meta":{}}}
```

7.8.13. Path by value (list of strings)

The query below returns a path between two airports as a list of airport codes. Note the new 'objects' key used when the returned JSON represents a path.

```
g.V().has('code','SAF').out().path().by('code').limit(1)

{"requestId":"b9a1655f-1b14-4313-96d0-085858f47de7",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"labels":[[],[]],
                    "objects":["SAF","PHX"]}],"meta":{}}}
```

7.8.14. Path by values (list of strings and integers)

Similar to the previous query, but this time the path also includes the distance between the airports.

```
g.V().has('code','SAF').outE().inV().path().by('code').by('dist').limit(1)


{"requestId":"c4eb3141-be1e-4335-aa04-50843f73838b",
    "status":{"message":"","code":200,"attributes":{}},
    "result":{"data":[{"labels":[[],[],[]],
                       "objects":["SAF",369,"PHX"]}],"meta":{}}}
```

7.8.15. Two vertex path

The query below returns a path but does not include a 'by' modulator, so what is returned is the two vertices along with their IDs, labels, and properties.

```
g.V().has('code','SAF').out().path().limit(1)

{"requestId":"bcdc3113-d1f6-41cf-b2ad-b1409646677e",
    "status":{"message":"","code":200,"attributes":{}},
    "result":{"data":[{"labels":[[],[]],
           "objects":[{
```

```
            "id":53352,
          "label":"airport",
           "type":"vertex",
      "properties":{
         "country":[{"id":"1s0d-1560-8p1","value":"US"}],
            "code":[{"id":"1sel-1560-5j9","value":"SAF"}],
         "longest":[{"id":"1sst-1560-mx1","value":8366}],
            "city":[{"id":"1t71-1560-7wl","value":"Santa Fe"}],
            "elev":[{"id":"1tl9-1560-but","value":6348}],
            "icao":[{"id":"1tzh-1560-6bp","value":"KSAF"}],
             "lon":[{"id":"1udp-1560-dfp","value":-106.088996887}],
            "type":[{"id":"1urx-1560-745","value":"airport"}],
          "region":[{"id":"1v65-1560-9hh","value":"US-NM"}],
         "runways":[{"id":"1vkd-1560-b2d","value":3}],
             "lat":[{"id":"1vyl-1560-cn9","value":35.617099762}],
            "desc":[{"id":"1wct-1560-a9x","value":"Santa Fe"}]}},

            {"id":4240,
          "label":"airport",
           "type":"vertex",
      "properties":{
         "country":[{"id":"176-39s-8p1","value":"US"}],
            "code":[{"id":"1le-39s-5j9","value":"PHX"}],
         "longest":[{"id":"1zm-39s-mx1","value":11489}],
            "city":[{"id":"2du-39s-7wl","value":"Phoenix"}],
            "elev":[{"id":"2s2-39s-but","value":1135}],
            "icao":[{"id":"36a-39s-6bp","value":"KPHX"}],
             "lon":[{"id":"3ki-39s-dfp","value":-112.012001037598}],
            "type":[{"id":"3yq-39s-745","value":"airport"}],
          "region":[{"id":"4cy-39s-9hh","value":"US-AZ"}],
         "runways":[{"id":"4r6-39s-b2d","value":3}],
             "lat":[{"id":"55e-39s-cn9","value":33.4342994689941}],
            "desc":[{"id":"5jm-39s-a9x",
             "value":"Phoenix Sky Harbor International Airport"}]}}]}],
      "meta":{}}}
```


7.8.16. Path with two vertices and an edge

The following query is similar to the previous one but also includes an edge. You can hopefully see here how the JSON can rapidly get large if we are not more specific in our queries about what results we really need back. Notice how, because this is a path result, most of the data is contained inside an 'objects' key.

```
g.V().has('code','SAF').outE().inV().path().limit(1)
```


```
{"requestId":"171d0f30-2f93-4ae6-a421-4601a35388a2",
 "status":{"message":"","code":200,"attributes":{}},
  "result":{"data":[{"labels":[[],[],[]],
  "objects":[
```

```
        {"id":53352,"label":"airport","type":"vertex",
        "properties":
            {"country":[{"id":"1s0d-1560-8p1","value":"US"}],
            "code":[{"id":"1sel-1560-5j9","value":"SAF"}],
            "longest":[{"id":"1sst-1560-mx1","value":8366}],
            "city":[{"id":"1t71-1560-7wl","value":"Santa Fe"}],
            "elev":[{"id":"1tl9-1560-but","value":6348}],
            "icao":[{"id":"1tzh-1560-6bp","value":"KSAF"}],
            "lon":[{"id":"1udp-1560-dfp","value":-106.088996887}],
            "type":[{"id":"1urx-1560-745","value":"airport"}],
            "region":[{"id":"1v65-1560-9hh","value":"US-NM"}],
            "runways":[{"id":"1vkd-1560-b2d","value":3}],
            "lat":[{"id":"1vyl-1560-cn9","value":35.617099762}],
            "desc":[{"id":"1wct-1560-a9x","value":"Santa Fe"}]}},

        {"id":"2xhcd-1560-pat-39s",
         "label":"route",
         "type":"edge",
         "inVLabel":"airport",
         "outVLabel":"airport",
         "inV":4240,"outV":53352,
             "properties":{"dist":369}},

        {"id":4240,"label":"airport","type":"vertex",
        "properties":
            {"country":[{"id":"176-39s-8p1","value":"US"}],
             "code":[{"id":"1le-39s-5j9","value":"PHX"}],
             "longest":[{"id":"1zm-39s-mx1","value":11489}],
             "city":[{"id":"2du-39s-7wl","value":"Phoenix"}],
             "elev":[{"id":"2s2-39s-but","value":1135}],
             "icao":[{"id":"36a-39s-6bp","value":"KPHX"}],
             "lon":[{"id":"3ki-39s-dfp","value":-112.012001037598}],
             "type":[{"id":"3yq-39s-745","value":"airport"}],
             "region":[{"id":"4cy-39s-9hh","value":"US-AZ"}],
             "runways":[{"id":"4r6-39s-b2d","value":3}],
             "lat":[{"id":"55e-39s-cn9","value":33.4342994689941}],
             "desc":[{"id":"5jm-39s-a9x",
               "value":"Phoenix Sky Harbor International Airport"}]}}]}],
      "meta":{}}}
```


7.8.17. Selection map

If a query ends with a 'select' step that references labels defined earlier in the query, what is returned is a map where the labels are the keys and the values are the things that the labels were attached to in the query.

```
g.V().has('code','SAF').as('a').out().has('code','DFW').as('b').
      select('a','b').by('code')
```

```
{"requestId":"af8de8e6-4137-4378-bb31-921e134d0661",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"a":"SAF","b":"DFW"}],"meta":{}}}
```

7.8.18. Projected map

The 'project' step also generates a map just as the 'select' step did in the previous example.

```
g.V().has('code','LGW').project('a','b').by('code').by(out().count())

{"requestId":"57819d0b-c27e-40d7-a89a-e69a6b4872a1",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"a":"LGW","b":204}],"meta":{}}}
```

7.8.19. Strings and a map

The following path query returns a list containing two strings representing airport codes and the full JSON object representing an edge.

```
g.V().has('code','SAF').outE().inV().limit(1).path().by('code').by()

{"requestId":"5102bb14-e594-41ec-8643-89882377b1e7",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[{"labels":[[],[],[]],
           "objects":["SAF",
                      {"id":"2xhcd-1560-pat-39s",
                      "label":"route",
                      "type":"edge",
                      "inVLabel":"airport",
                      "outVLabel":"airport",
                      "inV":4240,
                      "outV":53352,
                      "properties":{"dist":369}},
                      "PHX"]}],"meta":{}}}
```

7.8.20. Nested lists

When using the console or issuing Gremlin commands via the TinkerPop API from an application program, ending a query with a 'fold' step can be a nice way to put all the results into a list. When working with a Gremlin Server, ending a query with a 'fold' step in many cases is redundant as the results will be placed in a list inside the JSON anyway. In the example below, the 'fold' step simply caused an extra list to be nested inside the one that was generated while the JSON was being assembled.

```
g.V().has('region','US-OK').values('code').fold()
```

```
{"requestId":"edcea305-086d-4f8d-b79a-ff72c5a26847",
 "status":{"message":"","code":200,"attributes":{}},
 "result":{"data":[["OKC","TUL","LAW","SWO"]],"meta":{}}}
```

Chapter 8. COMMON GRAPH SERIALIZATION FORMATS

There are a number of ways that graph data can be stored in a file. In this section we have provided a brief overview of a few of them. As you will see, there are a number of ways you can represent graph data using simple CSV files. There are also XML formats, JSON formats and many more. Of these, the one that still seems to be supported across most tools and platforms is GraphML. Not all the features offered by Apache TinkerPop can be expressed using GraphML, however. So let's take a look at some of the more commonly used formats.

8.1. Comma Separated Values (CSV)

There are a number of ways that a graph can be stored using CSV files. There is no single preferred format that we are aware of. However, a common and convenient way, especially when vertices contain lots of properties, is to use two CSV files. One will contain all the vertex data and the other will contain all the edge data.

8.1.1. Using two CSV files to represent the air-routes data

If we were to store the airport data from the 'air-routes' graph in CSV format, we might do something like the example below. Note that to improve readability, we have not included every property (or indeed every airport) in this example. Notice how each vertex has a unique ID assigned. This is important as when we define the edges, we will need the vertex IDs to build the connections.

```
"ID","LABEL","CODE","IATA","CITY","REGION","RUNWAYS","LONGEST","ELEV","COUNTRY"
"1","airport","ATL","KATL","Atlanta","US-GA","5","12390","1026","US"
"2","airport","ANC","PANC","Anchorage","US-AK","3","12400","151","US"
"3","airport","AUS","KAUS","Austin","Austin","US-TX","2","12250","542","US"
"4","airport","BNA","KBNA","Nashville","US-TN","4","11030","599","US"
"5","airport","BOS","KBOS","Boston","US-MA","6","10083","19","US"
"6","airport","BWI","KBWI","Baltimore","US-MD","3","10502","143","US"
"7","airport","DCA","KDCA","Washington D.C.","US-DC","3","7169","14","US"
"8","airport","DFW","KDFW","Dallas Ft. Worth","US-TX","7","13401","607","US"
"9","airport","FLL","KFLL","Fort Lauderdale","US-FL","2","9000","64","US"
```

For the route data, the edges in our graph, we might use a format like the one below. We did not include an edge ID as we typically let the graph system assign those. For completeness, we did include a label. However, when every edge is of the same type, you could choose to leave this out so long as the program ingesting the data knew what label to assign. Most graph systems require edges to have a label even if it is optional for vertices. This is equally true for the airport data. However, in cases where vertices and edges within the same CSV file are of different types, then for those cases it is best to always include the label for each entry.

```
"LABEL","FROM","TO","DIST"
```

```
"route",1,3,811
"route",1,4,214
"route",2,8,3036
"route",3,4,755
"route",4,6,586
"route",5,1,945
```

Some graph systems provide ingestion tools that, when presented with a CSV file like the ones we have shown here, can figure out how to process them and build a graph. However, in many other situations you may also find yourself writing your own scripts or small programs to do it.

We often find ourselves writing Ruby or Groovy scripts that can generate CSV or GraphML files so that a graph system can ingest them. In some cases we have used scripts to take CSV or GraphML data and generate the Gremlin statements that would create the graph. This is very similar to another common practice, namely, using a script to generate 'INSERT' statements when working with SQL databases.

We have also written Java and Groovy programs that will read the CSV file and use the TinkerPop API or the Gremlin Server REST API to insert vertices and edges into a graph. If you work with graph systems for a while, you will probably find yourself also doing similar things.

8.1.2. Adjacency matrix format

The examples shown above of how a CSV file can be used to store data about vertices and edges present a convenient way to do it. However, this is by no means the only way you could do it. For graphs that do not contain properties, you could lay the graph out using an 'adjacency matrix' as shown below. The letters represent the vertex labels, and a 1 indicates there is an edge between them and a zero indicates no edge. This format can be useful if your vertices and edges do not have properties and if the graph is small, but in general it is not a great way to try and represent large graphs.

```
A,B,C,D,E,F,G
A,0,1,1,0,1,0,1
B,1,0,0,1,0,1,0
C,1,1,0,1,1,0,1
D,0,1,1,0,1,0,1
E,0,0,0,1,0,1,0
F,1,1,0,1,0,0,1
G,1,1,1,0,1,1,0
```

8.1.3. Adjacency List format

The adjacency matrix shown above could also be represented as an 'adjacency list'. In this case, the first column of each row represents a vertex. The remaining parts of each row represent all the other vertices that this vertex is connected to.

```
A,C,D,F,G
```

```
B,A,D,F
C,A,B,D,E,1
D,B,C,E,G
E,D,E
F,A,B,D,G
G,A,B,C,E,F
```

While this is a simple example, it is possible to represent a more complex graph such as the 'air-routes' graph in this way. We could build a more complex CSV file where the vertex and its properties are listed first, followed by all the other vertices it connects to and the properties for those edges.

8.1.4. Edge List format

When using an 'edge list' format, each line represents an edge. So our simple example could be represented as follows. Only a few edges are shown.

```
A,C
A,D
A,F
A,G
B,A
B,D
B,F
C,A
C,B
```

There are many ways you could construct an edge list. By way of another simple example, we could represent routes in the 'air-routes' graph in a format similar to that shown below. In this case we also include the label of the edge between each of the vertices. The vertices are represented by their ID value.

```
[1,route,623]
[1,route,624]
[1,route,625]
[1,route,626]
[1,route,627]
[1,route,628]
[1,route,629]
[1,route,630]
[1,route,631]
[1,route,632]
```

If you wanted to export a basic version of the 'air-routes' graph, using just the airport IATA codes and the edge labels, you could write a Gremlin query to do it for you as follows. Only the first 10 results returned are shown.

```
g.V().outE().inV().path().by('code').by(label)

[ATL,route,MBS]
[ATL,route,MCN]
[ATL,route,MEI]
[ATL,route,MLB]
[ATL,route,MSL]
[ATL,route,PHF]
[ATL,route,PIB]
[ATL,route,SBN]
[ATL,route,TRI]
[ATL,route,TTN]
```

There is a sample program called GraphFromCSV.java in the sample programs folder that shows how to read a CSV file like the one above and create a graph from it.

If you wanted to print the list without the containing square brackets, you could take advantage of the Java 'forEachRemaining' method from the Iterator interface to add a bit of post-processing to the end of the query. Once again, only the first 10 results are shown.

```
g.V().outE().inV().path().by('code').by(label).
      forEachRemaining{println it[0] + ',' + it[1] + ',' + it[2]}

ATL,route,MBS
ATL,route,MCN
ATL,route,MEI
ATL,route,MLB
ATL,route,MSL
ATL,route,PHF
ATL,route,PIB
ATL,route,SBN
ATL,route,TRI
ATL,route,TTN
```

8.2. GraphML

GraphML is an XML-based format designed to serialize an entire graph. This format is best used for interoperability with other graph tools and frameworks as it has wide support. The format specification can be found at http://graphml.graphdrawing.org/.

The following sample file illustrates what the format looks like.

```
<?xml version='1.0' ?>
<!-- ****************************************************** -->
<!-- Small sample taken from the air-routes.graphml file.    -->
```

```
<!-- ************************************************** -->

<graphml xmlns='http://graphml.graphdrawing.org/xmlns'>
  <key id='type'    for='node' attr.name='type'    attr.type='string'></key>
  <key id='code'    for='node' attr.name='code'    attr.type='string'></key>
  <key id='icao'    for='node' attr.name='icao'    attr.type='string'></key>
  <key id='desc'    for='node' attr.name='desc'    attr.type='string'></key>
  <key id='region'  for='node' attr.name='region'  attr.type='string'></key>
  <key id='runways' for='node' attr.name='runways' attr.type='int'></key>
  <key id='longest' for='node' attr.name='longest' attr.type='int'></key>
  <key id='elev'    for='node' attr.name='elev'    attr.type='int'></key>
  <key id='country' for='node' attr.name='country' attr.type='string'></key>
  <key id='city'    for='node' attr.name='city'    attr.type='string'></key>
  <key id='lat'     for='node' attr.name='lat'     attr.type='double'></key>
  <key id='lon'     for='node' attr.name='lon'     attr.type='double'></key>
  <key id='dist'    for='edge' attr.name='dist'    attr.type='int'></key>
  <key id='labelV'  for='node' attr.name='labelV'  attr.type='string'></key>
  <key id='labelE'  for='edge' attr.name='labelE'  attr.type='string'></key>

  <graph id='routes' edgedefault='directed'>

  <node id='1'>
    <data key='labelV'>airport</data>
    <data key='type'>airport</data>
    <data key='code'>ATL</data>
    <data key='icao'>KATL</data>
    <data key='city'>Atlanta</data>
    <data key='desc'>Hartsfield - Jackson Atlanta International Airport</data>
    <data key='region'>US-GA</data>
    <data key='runways'>5</data>
    <data key='longest'>12390</data>
    <data key='elev'>1026</data>
    <data key='country'>US</data>
    <data key='lat'>33.6366996765137</data>
    <data key='lon'>-84.4281005859375</data>
  </node>

  <edge id='3610' source='1' target='3'>
    <data key='labelE'>route</data>
    <data key='dist'>811</data>
  </edge>

  </graph>
</graphml>
```

You can learn more about the specifics of the format itself in TinkerPop's IO Documentation https://tinkerpop.apache.org/docs/current/dev/io/#graphml

8.3. GraphSON

As discussed in the "Working with GraphML and GraphSON" section, there are multiple versions of the GraphSON format. There are currently three versions of GraphSON, and each can include embedded type information or rely on standard JSON types. The original 1.0 version has a complicated embedded type format that is tied heavily to the Java type system and is typically not used anymore. Version 2.0 introduced a new embedded type format that is much more compact and easier to parse, and 3.0 added a few additional types to the format. The default format, unless explicitly specified, is currently GraphSON 3.0.

```
graph = TinkerGraph.open()
g = traversal().with(graph)
g.addV('airport').property('code','AUS').as('aus').
  addV('airport').property('code','DFW').as('dfw').
  addV('airport').property('code','LAX').as('lax').
  addV('airport').property('code','JFK').as('jfk').
  addV('airport').property('code','ATL').as('atl').
  addE('route').from('aus').to('dfw').
  addE('route').from('aus').to('atl').
  addE('route').from('atl').to('dfw').
  addE('route').from('atl').to('jfk').
  addE('route').from('dfw').to('jfk').
  addE('route').from('dfw').to('lax').
  addE('route').from('lax').to('jfk').
  addE('route').from('lax').to('aus').
  addE('route').from('lax').to('dfw')
```

8.3.1. Adjacency list format GraphSON

In an earlier section, we introduce the structure of an adjacency list for CSV. We can have a similar format with GraphSON where each line is a GraphSON serialized vertex and its incident edges. Note that in this case, the document as a whole is not well-formed JSON, but each individual line is.

```
{"id":0,"label":"airport","inE":{"route":[{"id":17,"outV":4}]}, ... }
{"id":2,"label":"airport","inE":{"route":[{"id":18,"outV":4}, ... ]}}
{"id":4,"label":"airport","inE":{"route":[{"id":15,"outV":2}]}, ... }
{"id":6,"label":"airport","inE":{"route":[{"id":16,"outV":4}, ... ]}}
{"id":8,"label":"airport","inE":{"route":[{"id":11,"outV":0}]}, ... }
```

```
{
    "id": 0,
    "label": "airport",
    "inE": {
        "route": [{
            "id": 17,
            "outV": 4
        }]
```

```
    },
    "outE": {
        "route": [{
            "id": 10,
            "inV": 2
        }, {
            "id": 11,
            "inV": 8
        }]
    },
    "properties": {
        "code": [{
            "id": 1,
            "value": "AUS"
        }]
    }
}
```

```
{"id":197,"label":"airport","inE":{"contains":[{"id":46566,"outV":3378},{"id":49931,"o
utV":3608}],"route":[{"id":9524,"outV":55,"properties":{"dist":520}},{"id":9753,"outV"
:57,"properties":{"dist":903}},{"id":22158,"outV":231,"properties":{"dist":1036}}]},"o
utE":{"route":[{"id":20448,"inV":231,"properties":{"dist":1036}},{"id":20446,"inV":55,
"properties":{"dist":520}},{"id":20447,"inV":57,"properties":{"dist":903}}]},"properti
es":{"country":[{"id":2356,"value":"AU"}],"code":[{"id":2357,"value":"MCY"}],"longest"
:[{"id":2358,"value":5896}],"city":[{"id":2359,"value":"Maroochydore"}],"elev":[{"id":
2360,"value":15}],"icao":[{"id":2361,"value":"YBSU"}],"lon":[{"id":2362,"value":153.09
1003418}],"type":[{"id":2363,"value":"airport"}],"region":[{"id":2364,"value":"AU-
QLD"}],"runways":[{"id":2365,"value":2}],"lat":[{"id":2366,"value":-26.6033000946
}],"desc":[{"id":2367,"value":"Sunshine Coast Airport"}]}}
```

This newline-delimited format is known as JSON Lines (https://jsonlines.org/).

This format is particularly useful when you have a large graph and need to split the adjacency list into multiple files.

8.3.2. Wrapped adjacency list format GraphSON

The prior section showed a GraphSON format for an adjacency list that did not have well-formed JSON and instead followed JSON Lines format. While that format has its use cases, GraphSON can also be produced as well-formed JSON, where the JSON Line format is essentially just wrapped in an array as part of a JSON object.

```
{
    "vertices": [{
        "id": 0,
        "label": "airport",
        "inE": {
            "route": [{
```

```
                "id": 17,
                "outV": 4
            }]
        },
        "outE": {
            "route": [{
                "id": 10,
                "inV": 2
            }, {
                "id": 11,
                "inV": 8
            }]
        },
        "properties": {
            "code": [{
                "id": 1,
                "value": "AUS"
            }]
        }
    }, {
        "id": 2,
        "label": "airport",
        "inE": {
            "route": [{
                "id": 18,
                "outV": 4
            }, {
                "id": 10,
                "outV": 0
            }, {
                "id": 12,
                "outV": 8
            }]
        },
        "outE": {
            "route": [{
                "id": 14,
                "inV": 6
            }, {
                "id": 15,
                "inV": 4
            }]
        },
        "properties": {
            "code": [{
                "id": 3,
                "value": "DFW"
            }]
        }
    }, {
        "id": 4,
```

```
        "label": "airport",
        "inE": {
            "route": [{
                "id": 15,
                "outV": 2
            }]
        },
        "outE": {
            "route": [{
                "id": 16,
                "inV": 6
            }, {
                "id": 17,
                "inV": 0
            }, {
                "id": 18,
                "inV": 2
            }]
        },
        "properties": {
            "code": [{
                "id": 5,
                "value": "LAX"
            }]
        }
    }, {
        "id": 6,
        "label": "airport",
        "inE": {
            "route": [{
                "id": 16,
                "outV": 4
            }, {
                "id": 13,
                "outV": 8
            }, {
                "id": 14,
                "outV": 2
            }]
        },
        "properties": {
            "code": [{
                "id": 7,
                "value": "JFK"
            }]
        }
    }, {
        "id": 8,
        "label": "airport",
        "inE": {
            "route": [{
```

```
                "id": 11,
                "outV": 0
            }]
        },
        "outE": {
            "route": [{
                "id": 12,
                "inV": 2
            }, {
                "id": 13,
                "inV": 6
            }]
        },
        "properties": {
            "code": [{
                "id": 9,
                "value": "ATL"
            }]
        }
    }]
}
```

You can learn more about the specifics of the format itself in TinkerPop's IO Documentation. https://tinkerpop.apache.org/docs/current/dev/io/#graphson

8.4. GraphBinary

We mention GraphBinary here mostly for informational purposes as GraphBinary is not meant to be used for persisting graphs to file. It is primarily a network transport serialization format used by Gremlin Server compliant systems to communicate with compatible drivers. In general, you do not need to know much about GraphBinary itself unless you are implementing a server or a driver.

You can learn more about the specifics of the format itself in TinkerPop's IO Documentation. https://tinkerpop.apache.org/docs/current/dev/io/#graphbinary

Chapter 9. FURTHER READING

Below you will find a selection of links to additional material related to the topics covered in this book. The links include additional websites maintained by the Apache TinkerPop community as well as some mailing lists where Gremlin and related topics are discussed daily.

Additional Apache TinkerPop community links .Official Apache TinkerPop home page and Gremlin downloads https://tinkerpop.apache.org/

Current Apache TinkerPop Reference Documentation

https://tinkerpop.apache.org/docs/current/reference/

Apache TinkerPop Javadoc API reference material

https://tinkerpop.apache.org/javadocs/current/core/

https://tinkerpop.apache.org/javadocs/current/full/

Gremlin serialization and file formats documentation

https://tinkerpop.apache.org/docs/current/dev/io/

Useful Gremlin "recipes"

https://tinkerpop.apache.org/docs/current/recipes/

Apache TinkerPop documentation describing new features per release

https://tinkerpop.apache.org/docs/current/upgrade/

Apache TinkerPop documentation for graph database providers

https://tinkerpop.apache.org/docs/current/dev/provider/

Apache TinkerPop documentation for making contributions

https://tinkerpop.apache.org/docs/current/dev/developer/

Tutorials There are a series of tutorials that can be found at the link below.

Apache TinkerPop Tutorials

https://tinkerpop.apache.org/docs/current/tutorials/

The tutorials include the following

Getting Started

- https://tinkerpop.apache.org/docs/current/tutorials/getting-started/

Gremlin's Anatomy

- https://tinkerpop.apache.org/docs/current/tutorials/gremlins-anatomy/

The Gremlin Console

- https://tinkerpop.apache.org/docs/current/tutorials/the-gremlin-console/

9.1. Gremlin-related mailing lists and discussion groups

Apache Tinkerpop Discord Server

https://discord.gg/tinkerpop

Public mailing list for Gremlin users

https://groups.google.com/forum/#!forum/gremlin-users

Apache TinkerPop developers mailing list archive

https://lists.apache.org/list.html?dev@tinkerpop.apache.org

Stack Overflow posts tagged with "gremlin"

https://stackoverflow.com/questions/tagged/gremlin