GRAPH DATABASES, GREMLIN and TINKERPOP : A TUTORIAL

Kelvin R. Lawrence – klawrenc@us.ibm.com – Version 266, Oct 17, 2017

Table of Contents

---

1. INTRODUCTION

This book is a work in progress. Feedback is very much encouraged and welcomed!

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 too long to fit on one line for a heading but in a single sentence that describes the focus of this work pretty well.

I hope people find what follows useful. It remains a work in progress and more will be added in the coming weeks and months as time permits. I am hopeful that what is presented so far is of some use to folks, who like me, are learning to use the Gremlin query and traversal language and related technologies.

1.1. How this book came to be

I forget exactly when, but over a year ago I started compiling a list of notes, hints and tips, initially for my own benefit, of things I had found poorly explained elsewhere while using graph databases and especially using Apache TinkerPop, Janus Graph and Gremlin. Over time that document grew (and continues to grow) and has effectively become a book. After some encouragement from colleagues I have decided to release it as a living book in an open source venue so that anyone who is interested can read it. It is aimed at programmers and anyone using the Gremlin query language to work with graphs. Lots of code examples, sample queries, discussion of best practices, lessons I learned the hard way etc. are included.

Thanks to all those that have encouraged me to keep going with this adventure!

Kelvin R. Lawrence
October 5th, 2017

1.2. Providing feedback

Please let me know about any mistakes you find in this material and also please feel free to send me 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 266 of the document.

1.3. A word of thanks

I would like to thank my colleagues at IBM, Graham Wallis, Jason Plurad and Adam Holley for their help in refining and improving several of the queries contained in this document. Gremlin is definitely a bit of a team sport. We spent many hours over the last many months discussing the best way to handle different types of queries and traversals!

I would also be remiss if I did not give a big shout out to all of the folks that spend a lot of time replying to questions and suggestions on the Gremlin Users Google Group (https://groups.google.com/forum/#!forum/gremlin-users). Special thanks should go to Daniel Kuppitz, Marko Rodriguez and Stephen Mallette, key members of the team that created and maintains Apache TinkerPop.

1.4. What is this book about?

This book introduces the Apache TinkerPop 3 *Gremlin* graph query and traversal language via real examples featuring a real world graph that is also provided along with sample code and applications. The graph, *air-routes.graphml*, is a model of the world airline route network between 3,373 airports including 43,400 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 Downloading, installing and launching the console section below.

> **NOTE** The examples in this book have now been updated and tested using Apache TinkerPop version 3.3 which introduced a few breaking changes. If you find any I missed please let me know!

TinkerGraph is an *in-memory* graph, meaning nothing gets saved to disk automatically, that is shipped as part of the Apache TinkerPop 3 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 document we will discuss using additional technologies such as Janus Graph, Apache Cassandra and Apache Elastic Search with Gremlin. We will also discuss writing stand alone Java and Groovy applications as well as using the Gremlin Console.

NOTE

In the first four sections of this this document we have mainly focussed on showing the different types of query that you can issue using Gremlin. We have not tried to show all of the output that you will get back from entering these queries but have selectively shown examples of output. We go a lot deeper into things in chapters 5 and 6.

How this book is organized

Chapter 1 - INTRODUCTION

- We start off by briefly doing a recap on why Graph databases are of interest to us and discuss some good use cases for graphs.

Chapter 2 - GETTING STARTED

- In Chapter two we introduce several of the components of Apache TinkerPop 3 and we also introduce the air-routes.graphml file that will be used as the graph we base most of our examples on.

Chapter 3 - WRITING GREMLIN QUERIES

- In Chapter three we start discussing how to use the Gremlin graph traversal and query language to interrogate the air-routes graph. We begin by comparing how we could have built the air-routes graph using a more traditional relational database and then look at how SQL and Gremlin are both similar in some ways and very different in others. For the rest of the Chapter, we introduce several of the key Gremlin methods, or as they are often called, "steps". We mostly focus on reading the graph (not adding or deleting things) in this Chapter.

Chapter 4 - BEYOND BASIC QUERIES

- In Chapter four we move beyond just reading the graph and describe how to add vertices (nodes), edges and properties as well as how to delete and update them. We also present a discussion of various best practices.We also start to explore some slightly more advanced topics in this chapter.

Chapter 5 - MISCELLANEOUS QUERIES AND THE RESULTS THEY GENERATE

- In Chapter five we focus on using what we have covered in the prior Chapters to write queries that have a more real world feel. We present a lot more examples of the output from running queries in this Chapter. We also start to discuss topics such as analyzing distances, route distribution and writing geospatial queries.

Chapter 6 - MOVING BEYOND THE CONSOLE AND TINKERGRAPH

- In Chapter six we move beyond our focus on the Gremlin Console and TinkerGraph. We start by looking at how you can write stand alone Java and Groovy applications that can work with a graph. We then introduce Janus Graph and take a fairly detailed look at its capabilities such as support for transactions, schemas and external indexes. We also explore various technology choices for back end persistent store and index as well as introducing the Gremlin Server.

Chapter 7 - COMMON GRAPH SERIALIZATION FORMATS

- In Chapter seven a discussion is presented of some common Graph serialization file formats along with coverage of how to use them in the context of TinkerPop 3 enabled graphs.

Chapter 8 - FURTHER READING

- We finish up by providing several links to useful web sites where you can find tools and documentation for many of the technologies covered in this document.

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

This document 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 document 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 (often referred to as nodes), edges and properties. Vertices represent "things" such as people or places. Edges represent connections between those vertices and properties and information 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 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 modelling our data the way we think about it in the real world. Armed with this knowledge you now have all of the basic building blocks you need in order to start thinking about how you might model things you are familiar with as a graph.

So getting back to the question "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 "use 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 of the examples in this document 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 "haven't graphs been around for ages?". 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 Konigsberg" problem.

If you want read a bit more about graph theory and its present day application you can find a lot of good information on-line. 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 has 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 the 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 recently 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 be focussed on in this document can support all of 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 on-line 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 document 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 document, someone wanting to experiment with graph technology can download the Apache TinkerPop package and as long as Java 8 is installed, be up and running with zero configuration, other than doing an unzip 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 complimentary to other databases that you may already have deployed. One common use case is for the graph to be uses as a form of smart index into other data stores. This is sometimes called having a polyglot data architecture.

1.6. A word about terminology

The words *node* and *vertex* are synonymous when discussing a graph. Throughout this book you will find both words used. However, as the Apache TinkerPop documentation almost exclusively uses the word *vertex*, as much as possible when discussing Gremlin queries and other concepts, I will endeavor to stick to the word *vertex* or the plural form *vertices*. As this document has evolved I realized my use of these terms had become inconsistent and in future updates I plan to standardize on *vertex* rather than *node* to be consistent with the TinkerPop documentation.

2. GETTING STARTED

Let's take a look at what you will need to have installed and what tools you will need available to make best use of the examples contained in this guide. The key thing that you will need is the Apache TinkerPop project's Gremlin Console download.

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: http://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.

Gremlin Server

- Allows hosting of graphs remotely via an HTTP/Web Sockets connection.

TinkerGraph

- A small in-memory graph implementation that is great for learning.

Programming Interfaces

- A set of programming interfaces written in Java

Documentation

- A user guide, a tutorial and programming API documentation.

The programming interfaces allow providers of graph databases to build systems that are TinkerPop enabled. Any such databases can be accessed using the Gremlin query language and corresponding API. For most of this document we will be working within the Gremlin console with a local graph. However in Section 6 we will take a look at Gremlin Server and some other TinkerPop 3 enabled environments. Most of Apache Tinkerpop has been developed using Java 8 but there are also bindings available for many other programming languages such as Groovy and Python. Parts of TinkerPop are themselves developed in Groovy, most notably the Gremlin Console. The nice thing about that is that we can use Groovy syntax along with Gremlin when entering queries into the Console or sending them via REST API to a Gremlin Server. All of these topics are covered in detail in this boook.

The queries used as examples in this book have been tested with Apache TinkerPop version 3.3. using the TinkerGraph graph and the Gremlin console as well as some other TinkerPop 3 enabled graph stores.

2.2. The Gremlin console

The Gremlin Console is a fairly standard REPL (Read Execute Print Loop) shell. It is based on the Groovy console 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 low barrier of entry 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 document 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 TinkerPop3/Gremlin enabled graph store such as TinkerGraph or Janus Graph.

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

2.2.1. Downloading, installing and launching the console

You can download the Gremlin console from the official Apache TinkerPop website:

http://tinkerpop.apache.org/

It only takes a few minutes to get the Gremlin Console installed and running. You just download the ZIP file and *unzip* it and you are all set. TinkerPop 3 also requires a recent version of Java 8 being installed. I have done all of my testing using Java 8 version 1.8.0_131. The Gremlin Console will not work with versions prior to 1.8.0_45. If you do not have Java 8 installed it is easy to find and download off the Web.

When you start Gremlin 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 all commands to the console itself are prefixed by a colon *":"*. 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
  :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

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

It is worth noting that as mentioned above, the Gremlin console is based on the Groovy console 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!

```groovy
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
```

NOTE

The Gremlin Console does a very nice job of only showing you a nice and tidy set of query results. If you are working with a graph system that supports TinkerPop 3 but not via the Gremlin console (an example of this would be talking to a Gremlin Server using the HTTP REST API) then what you will get back is going to be a JSON document that you will need to write some code to parse. We will explore that topic much later in this document.

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

Mostly you will run the Gremlin 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 -e mycode.groovy
```

If you wanted to have the console run your script and not exit afterwards, you can use the *-i* option instead of *-e*.

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

```
$ gremlin --help

usage: gremlin.sh [options] [...]
  -C, --color                                Disable use of ANSI colors
  -D, --debug                                Enabled debug Console output
  -Q, --quiet                                Suppress superfluous Console
                                             output
  -V, --verbose                              Enable verbose 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
  -v, --version                              Display the version
```

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

```groovy
// What version of Gremlin am I running?
Gremlin.version()
```

2.2.2. Saving output from the console to a file

Sometimes it is useful to save part or all of a console session to a file. You can turn recording to a file on and off using the *:record* command.

In the following example, we turn 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 (nodes) 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()
==>3618
gremlin> :record stop
Recording stopped; session saved as: "mylog.txt" (157 bytes)
```
GROOVY

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 document I am 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 Gremlin console however.

TIP 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.

2.3. Introducing TinkerGraph

As well as the Gremlin Console, the TinkerPop 3 download includes an implementation of an in-memory graph store called TinkerGraph. This document was mostly developed using TinkerGraph but I also tested everything using Janus Graph. We will introduce Janus Graph later in the "Introducing Janus Graph" section. 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 explain how to get started with the Gremlin Console and TinkerGraph a bit later in this document.

Tinkerpop 3 defines a number of capabilities that a graph store should support. Some are optional others are not. You can query any TinkerPop 3 enabled graph store to see which features are supported. 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 makes sense right away - we'll get there soon!

```
> GraphFeatures
>-- ConcurrentAccess: false
>-- ThreadedTransactions: false
>-- Persistence: true
>-- Computer: true
>-- Transactions: false
> VariableFeatures
>-- Variables: 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
> 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
> VertexPropertyFeatures
>-- UserSuppliedIds: true
>-- StringIds: true
>-- RemoveProperty: true
>-- AddProperty: true
>-- NumericIds: true
>-- CustomIds: false
>-- AnyIds: true
>-- UuidIds: true
>-- 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
> 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 very useful is to crate a sub-graph of a larger graph and work with it locally. TinkerGraph can even be used in production deployments if an all in memory graph fits the bill. Typically, TinkerGraph us used to explore static (no changing) graphs but you can 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 Janus Graph such as ACID Transactions and external indexes. We will cover these topics as part of our discussion of Janus Graph towards the end of the document. One other thing worth noting in the list above is that *UserSuppliedIds* 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 most other graph systems.

When running in the Gremlin 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
```
GROOVY

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.

```
graph = TinkerGraph.open()
```
GROOVY

In many cases you will want to pass parameters to the *open* method that give more information on how the graph is to be configured. We will explore those options later in the document. Before you can start to issue Gremlin queries against the graph you also need to establish a graph traversal source object by calling the new graph's *traversal* method as follows.

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

NOTE

Throughout the remainder of this document we will follow the convention that we will always use the variable name *graph* for any variable that represents a graph instance and we will always use the variable name *g* for any variable that represents an instance of a graph traversal source object.

2.4. Introducing the air-routes graph

Along with these notes I have provided what is, in big data terms,a very small, but nonetheless real World, graph that is written in GraphML, a standard XML format for describing graphs that can be used to move graphs between applications. The graph, *air-routes.graphml* is a model I built of the World airline route network that is fairly accurate. Of course, in the real World, 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 I 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 very 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 algorithms also sill 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 much deeper 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 at the end of this document.

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 I 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 modelled 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 parenthesis 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 airport vertex with an ID of 3 looks like. We will explain the steps that make up the Gremlin query later in this document.

GROOVY

```groovy
// Query the properties of vertex 3
g.V(3).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* I 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 document.

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 document 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 3,612 vertices and 49,894 edges. Of these 3,367 vertices are airports, and 43,160 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 very interesting Gremlin queries.

Lastly, here is 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 (v0.77, 2017-Oct-06) contains:
  3,374 airports
  43,400 routes
  237 countries (and dependent areas)
  7 continents
  3,619 total nodes
  50,148 total edges

Additional observations:
  Longest route is between DOH and AKL (9,025 miles)
  Shortest route is between WRY and PPW (2 miles)
  Average route distance is 1,164.747 miles.
  Longest runway is 18,045ft (BPX)
  Shortest runway is 1,300ft (SAB)
  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 node degree (routes in and out) is 544 (FRA)
  Country with the most airports: United States (579)
  Continent with the most airports: North America (978)
  Average degree (airport nodes) is 25.726
  Average degree (all nodes) is 25.856
```

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  (544)  out:272 in:272
 2    70   AMS  (541)  out:269 in:272
 3   161   IST  (540)  out:270 in:270
 4    51   CDG  (524)  out:262 in:262
 5    80   MUC  (474)  out:237 in:237
 6    64   PEK  (469)  out:234 in:235
 7    18   ORD  (464)  out:232 in:232
 8     1   ATL  (464)  out:232 in:232
 9    58   DXB  (458)  out:229 in:229
10     8   DFW  (442)  out:221 in:221
11   102   DME  (428)  out:214 in:214
12    67   PVG  (402)  out:201 in:201
13    50   LGW  (400)  out:200 in:200
14    13   LAX  (390)  out:195 in:195
15    74   MAD  (384)  out:192 in:192
```

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

2.5. TinkerPop 3 migration notes

There are still a large number of examples on the internet that show the TinkerPop 2 way of doing things. Quite a lot of things changed between TinkerPop 2 and TinkerPop 3. If you were an early adopter and are coming from a TinkerPop 2 environment to a TinkerPop 3 environment you may find some of the tips in this section helpful. As we we will explain below, using the *sugar* plugin will make the migration from TinkerPop 2 easier but it is recommended to learn the full TinkerPop 3 Gremlin syntax and get used to using that as soon as possible. Using the full syntax will make your queries a lot more portable to other TinkerPop 3 enabled graph systems.

TinkerPop 3 requires a minimum of Java 8 v45. It will not run on earlier versions of Java 8 based on my testing.

2.5.1. Creating a TinkerGraph TP2 vs TP3

The way that you create a TinkerGraph changed between TinkerPop 2 and 3.

```groovy
graph = new TinkerGraph()  // TinkerPop 2
graph = TinkerGraph.open() // TinkerPop 3
```

2.5.2. Loading a graphML file TP2 vs TP3

If you have previous experience with TinkerPop 2 you may also have noticed that the way a graph is loaded has changed in TinkerPop 3.

```groovy
graph.loadGraphML('air-routes.graphml') // TinkerPop 2
graph.io(graphml()).readGraph('air-routes.graphml') // TinkerPop 3
```

The Gremlin language itself changed quite a bit between TinkerPop 2 and TinkerPop 3. The remainder of this document only shows TinkerPop 3 examples.

2.5.3. A word about the TinkerPop.sugar plugin

The Gremlin console has a set of plug in modules that can be independently enabled or disabled. Depending upon your use case you may or may not need to manage plugins.

TinkerPop 2 supported by default some syntactic *sugar* that allowed shorthand forms of queries to be entered when using the Gremlin console. In TinkerPop 3 that support has been moved to a plugin and is off by default. It has to be enabled if you want to continue to use the same shortcuts that TinkerPop 2 allowed by default.

You can enable *sugar* support from the Gremlin console as follows:

```groovy
:plugin use tinkerpop.sugar
```

> **TIP** The Gremlin Console remembers which plugins are enabled between restarts.

In the current revision of this document I have tried to remove any dependence on the *TinkerPop.sugar* plugin from the examples presented. By not using Sugar, queries shown in this document should port very easily to other TinkerPop 3 enabled graph platforms. A few of the queries may not work on versions of TinkerPop prior to 3.2 as TinkerPop continues to evolve and new features are being added fairly regularly.

The *Tinkerpop.sugar* plugin allows some queries to be expressed in a more shorthand or lazy form, often leaving out references to *values()* and leaving out parenthesis. For example:

```groovy
// With Sugar enabled
g.V.hasLabel('airport').code

// Without Sugar enabled
g.V().hasLabel('airport').values('code')
```

People Migrating from TinkerPop 2 will find the Sugar plugin helps get your existing queries running more easily but as a general rule it is recommended to become familiar with the longhand way of writing queries as that will enable your queries to run as efficiently as possible on graph stores that support TinkerPop 3. Also, due to changes introduced with TinkerPop 3, using sugar will not be as performant as using the normal Gremlin syntax.

NOTE

In earlier versions of this document many of the examples showed the sugar *form. In the current revision I have tried to remove all use of that form. It's possible that I may have missed a few and I will continue to check for, and fix, any that got missed. Please let me know if you find any that slipped through the net!*

2.6. Loading the air-routes graph using the Gremlin console

Here is some code you can load the air routes graph using the gremlin 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 setup the console environment, create a TinkerGraph graph 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 and edge IDs. TinkerPop 3 introduced the concept of a *traversal* so as part of loading a *graph* we also setup a graph traversal source called *g* which we will then refer to in our subsequent queries of the graph. The *max-iteration* option tells the Gremlin console the maximum number of lines of output that we ever want to see in return from a query. The default, if this is not specified, is 100.

TIP

You can use the *max-iteration* setting to control how much output the Gremlin Console displays.

If you are using a different graph environment, if 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.

GROOVY

```
conf = new BaseConfiguration()
conf.setProperty("gremlin.tinkergraph.vertexIdManager","LONG")
conf.setProperty("gremlin.tinkergraph.edgeIdManager","LONG")
graph = TinkerGraph.open(conf)
graph.io(graphml()).readGraph('air-routes.graphml')
g=graph.traversal()
:set max-iteration 1000
```

If you put the commands given above into a file called something like *mygremlin-setup*, once the console is up and running you can load that file by entering the command below. Doing this will save you a fair bit of time as each time you restart the console you can just reload your configuration file and the environment will be configured and the graph loaded and you can get straight to writing queries.

```
:load mygremlin-setup
```
GROOVY

> **NOTE** 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 Gremlin Console up and running and have the graph loaded, if you feel like it you can cut and paste queries from this document directly into the console 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 tack of vertex and edge counts. We will look at some other ways of doing that later in the document.

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

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

```
tinkergraph[vertices:3610 edges:49490]
```
GROOVY

2.7. 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();[]
```
GROOVY

2.8. A word about indexes and schemas

Some graph implementations such as IBM-Graph have strict requirements on the use of an *index*. This means that a schema and an index must be in place before you can work with a graph and that you can only begin a traversal by referencing a property in the graph that is included in the index. While that is beyond the scope of this document, it should be pointed out that some of the queries included in this material will not work on any graph system that requires all queries to be backed by an index and does not allow what are sometimes called *full graph searches* for cases where a particular item in a graph is not backed by an index. One example of this is vertex and edge *labels* which are typically not indexed but are sometimes very useful items to specify at the start of a query. As the examples in this document are intended to work just fine with a only basic TinkerGraph the subject of indexes is not covered in more detail until Section 6 "MOVING BEYOND THE CONSOLE AND TINKERGRAPH" where we will take a look at some other technologies such as Janus Graph and we do discuss indexing as part of that coverage. 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. We will explain what TinkerGraph is in the next section.

In general for any graph, regardless of whether it is optional or not, use of an index should be considered a best practice. Even TinkerGraph has a way to create an index should you want to.

NOTE

In production systems, especially those where the graphs are large, the task of creating and managing the index is often handed to an additional software component such as Apache Solr or Apache Elastic Search.

3. WRITING GREMLIN QUERIES

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

In this section 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 we will look at some fairly basic queries and finally get into some more advanced concepts. Hopefully each set of examples presented by 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 also 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 in order 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 of the airport data in a single table called *airports*. In a very simple case (the air routes graph actually stores a lot more data than this about each airport) we could setup 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.

```sql
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 document).

```groovy
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 very good choice and can be more performant than relational databases.

Graph databases are usually a good choice for storing and modelling networks. The air-routes graph is an example of a network graph a social network is of course another good example. Networks can be modelled 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 travelling from Austin (AUS) to Agra (AGR) with two stops, we would end up writing a query that looked something like this:

```SQL
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:

```GROOVY
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.

```GROOVY
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 document. 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 very good 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 ID can be accessed using it's 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 = graph.traversal()
```
GROOVY

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.

> **TIP** If it helps with remembering you can think of *g.V()* as meaning "looking at all of the vertices in the graph" and *g.E()* as meaning "looking at all of 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 introduce a lot of different Gremlin steps as we build up our Gremlin examples throughout this document, including may other forms of the *has* step, but these few are enough to get us started.

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

NOTE

The latest TinkerPop 3 documentation is always available at this URL: http://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 (nodes) that have the *airport* label.

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

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

```groovy
// 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.

```groovy
// 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 gt 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.

```groovy
v[8]
```

So, what we actually got back from these queries was a TinkerPop *Vertex* data structure. Later in this document 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 Gremlin 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. The *next* step 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 document.

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

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

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.

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 document. Initially, let's look at a couple of simple ways that we can look up the property values of a given vertex.

```groovy
// 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 document.

```
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 returned the values for the provided key names. The queries below return the values of some specific properties.

```groovy
// 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 simply 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.

```groovy
// 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 of all let's look at some examples of using the *count* step to count how many of various things there are in our air-routes graph. First of all lets find out how many vertices in the graph represent airports.

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

3374
```

Now, looking at edges that have a *route* label, let's find out how many flight routes are stored in the graph.

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

43400
```

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

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

43400
```

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.

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

43400
```

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 of the 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.

```groovy
// 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.

```groovy
[continent:7,country:237,version:1,airport:3374]
```

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

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

[continent:7,country:237,version:1,airport:3374]
```

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.

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

[contains:6748,route:43400]
```

As before we could rewrite the query as follows.

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

[contains:6748,route:43400
```

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

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

[continent:7,country:237,version:1,airport:3374]
```

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.

```groovy
// 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.

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

[EU:583,AS:932,NA:978,OC:284,AF:294,AN:0,SA:303]
```

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.

```groovy
// How many airports in 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:58,GR:39,BE:5]
```

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

3.2.5. 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 very easy 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. To get us started, here are some simple examples.

The *out* step is used to find vertices connected by an outgoing edge to that vertex and the *outE step* is used to explore 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. There are several other steps that we can use when traversing a graph to move between vertices and edges. We will encounter those in the "Other ways to explore vertices and edges using *both*, *bothE*, *bothV* and *otherV*" section of this document.

This first query 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 of 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 of the results into a list for us. This just makes it easier for us to inspect the results in the console.

```groovy
// 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.

```groovy
[YYZ, LHR, FRA, MEX, PIT, PDX, CLT, CUN, MEM, CVG, IND, MCI, DAL, STL, ABQ, MDW, LBB, HRL,
GDL, PNS, VPS, SFB, 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, MSY, EWR, HOU,
ELP, CLE, OAK, PHL, DTW]
```

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 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.

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

Despite having just stated consistently referencing labels is a good idea, unless you truly do want to get back all edges or all connected vertices, I will break my own rule quite a bit in this document. The reason for this is purely to save space and make the queries I present shorter.

Here are a few more simple queries similar to the previous one.

```groovy
// 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!
g.V().has('airport','code','AUS').out('route').out('route').values('code')

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

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

3.2.6. 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 document 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. Here is a simple example of *path* being used. Throughout the document you will see numerous examples of *path* being used including in conjunction with *by* to specify how the path should be formatted. This particular query will return the vertices and outgoing edges starting at the LCY airport vertex. You can read this query like this: "Start at the LCY vertex, find all outgoing edges and also find all of the vertices that are on the other ends of those edges".

```groovy
// 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 13208.

```groovy
[v[88],e[13208][88-route->77],v[77]]
```

While this 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. Take a look at the modified query and an example of the results that it will now return. The *by* modulators are processed in a round robin fashion. So even though there are three values we want to have formatted, we only need to specify two *by* modulators as both the first and third values are the same. If all three were different, say for example that the third value was a different property like a city name then we would have to provide an explicit *by* modulator for it. If this is not fully clear yet don't panic. Both *path* and *by* are used a lot throughout this document.

```groovy
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.

```groovy
[LCY,468,GVA]
```

Note that the example above is equivalent to this longer form of the same query. The *by* modulator steps that follow a *path* are applied in a *round robin* fashion. So if there are not enough specified for the number of steps in the path, it just loops back around to the first *by* step and so on.

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

There are a few things to be aware of when using *path*. Those concerns are explained in the A warning that *path* can be memory and CPU intensive section a bit later.

3.2.7. 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.

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

true

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

false
```

3.2.8. 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 document we also introduce the *coin* step that allows a pseudo random sample of the data to be returned.

```groovy
// 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 less airports are initially returned but it is also possible that an implementation would optimize both the same way.

```groovy
// 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.

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

We can also limit based on time taken. The following query has a maximum limit of ten milliseconds. All of the parts of this query are explained in detail later on in this document but I think what they do is fairly clear.

```groovy
// 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')
```

3.2.9. 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 range of vertices by giving a non zero starting offset and an ending offset. The *range* offsets are zero based, and while the official documentation states that the ranges are inclusive/inclusive it actually appears from my testing that they are inclusive/exclusive.

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

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

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

NOTE

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 Titan and IBM Graph. So do not rely on any sort of expectation of order when using *range* to process sets of vertices.

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" section.

3.2.10. Using *as*, *select* and *project* to name and refer back to prior steps

Sometimes it is useful to be able to remember a point of a traversal and refer back to it later on. This ability was more essential in TinkerPop 2 than it is in TinkerPop 3 but it still has many uses.

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

This query, while a bit contrived, and in reality a poor substitute for using *path* returns the following results.

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

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*.

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

Which would produce the following results.

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

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

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

In the most recent releases of TinkerPop you can also use the new *project* step and achieve the same results that you can get from the combination of *as* and *select* steps. The example below shows the previous query, rewritten to use *project* instead of *as* and *select*.

```groovy
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.

```groovy
[a:ATL,b:US-GA,c:232]
[a:ANC,b:US-AK,c:39]
[a:AUS,b:US-TX,c:59]
[a:BNA,b:US-TN,c:55]
[a:BOS,b:US-MA,c:129]
[a:BWI,b:US-MD,c:89]
[a:DCA,b:US-DC,c:93]
[a:DFW,b:US-TX,c:221]
[a:FLL,b:US-FL,c:141]
[a:IAD,b:US-VA,c:136]
```

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

```groovy
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.

```groovy
[IATA:ATL,Region:US-GA,Routes:232]
[IATA:ANC,Region:US-AK,Routes:39]
[IATA:AUS,Region:US-TX,Routes:59]
[IATA:BNA,Region:US-TN,Routes:55]
[IATA:BOS,Region:US-MA,Routes:129]
[IATA:BWI,Region:US-MD,Routes:89]
[IATA:DCA,Region:US-DC,Routes:93]
[IATA:DFW,Region:US-TX,Routes:221]
[IATA:FLL,Region:US-FL,Routes:141]
[IATA:IAD,Region:US-VA,Routes:136]
```

3.3. Using *valueMap* to explore the properties of a vertex or element

A call to *valueMap* will return all of the properties of a vertex or element as an array of key:value pairs. Basically a hash map. 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.

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

If you are using the Gremlin console, the output from running the previous command should look something like this.

```groovy
[country:[US], code:[AUS], longest:[12248], city:[Austin], elev:[542], icao:[KAUS], lon:
[-97.6698989868164], type:[airport], region:[US-TX], runways:[2], lat:[30.1944999694824],
desc:[Austin Bergstrom International Airport]]
```

NOTE

Notice how each key like *country* is followed by a value that is 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.

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.

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

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

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

```groovy
// 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]
```

TIP

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*.

```groovy
// 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]]
```

If you are reading the output of queries that use *valueMap* on the Gremlin console, it is sometimes easier to read the output if you add an *unfold* step to the end of the query as follows.

```groovy
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]
```

3.4. Assigning query results to a variable

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*.

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

TIP

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.

Once you have some results in a variable you can refer to it as you would in any other programming language that supports key:value type maps and dictionaries. We will explore mixing Java and Groovy code with your Gremlin queries later in this document. For now we will just show the Groovy *println* command being used 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 document when we look at mixing Gremlin and Groovy in the "Making Gremlin even Groovier" section.

```groovy
// 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
```

NOTE

Because 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.

3.5. Working with IDs

Every vertex and every edge in a graph has a unique ID that can be used to reference them individually or as a group. 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 most systems like Janus Graph or IBM-Graph 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 new a vertex or edge to your graph, 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!

TIP

Don't rely on the graph preserving the ID values you provide. Write code that can query the graph itself for ID values.

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 setup it's 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.

```groovy
// 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.

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

DFW
```

We could also have also written the above query as follows.

```groovy
// 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.

```groovy
// 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.

```groovy
// 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.

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

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

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

3.6. Working with labels

It's a good idea when designing a graph to give the vertices and edges meaningful labels. You can use these to help refine searches. For example in the air-routes graph, every airport vertex is labelled *airport* and every country vertex, not surprisingly, is labelled *country*. Similarly, edges

that represent a flight route are labelled *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.

```groovy
// 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.

NOTE

As useful as labels are, in larger graph deployments when indexing technology such as Solr or Elastic Search is often used to speed up traversing the graph, vertex labels typically do not get indexed. Therefore, it is currently recommended that an actual vertex property that can be indexed is used when walking a graph rather than relying on the vertex label. This is especially important when working with large graphs where performance can become an issue.

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

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

// But you would normally just use the hasLabel() method in this case of course
g.V().hasLabel('airport').count()

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

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

// The same basic concepts apply equally to edges
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*".

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

3.7. 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 lets 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.

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

43400.0
```

As you can see the answer we got back, *42760.0* 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 of 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 all of these numbers by the number of members ,airports in this case, int the collection.

Take a look at the modified query below.

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

12.863070539419088
```

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. I truncated the output to just show a few lines.

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

232
38
59
55
129
87
93
220
141
135
```

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.

There are other ways that *local* can be used. We will see examples of those throughout the document including 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 of the document.

3.8. Basic statistical and numerical operations

The following queries demonstrate concepts such as calculating the number 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. 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 document. Some of these examples also take advantage of the *local* step that was introduced in the previous section.

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

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

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

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

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

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

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

3.8.1. Testing values and ranges of values

We have already seen some ways of testing whether a value is within a certain range. Gremlin provides a number of different steps that we can use to do range testing. The list below provides a summary of the available steps. We will see each of these in use throughout this document.

Table 1. Steps 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

The following queries demonstrate these capabilities being used in different ways. First of all, here are some examples of some of the direct compare steps such as *gt* and *gte* being used.

```groovy
// Airports with at least 5 runways
g.V().has('runways',gte(5)).values('code','runways').fold()

// Airports with less 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()
```

Here are examples of *inside* and *outside* being used.

```groovy
// Airports with greater than 3 but less than 6 runways.
g.V().has('runways',inside(3,6)).values('code','runways')

// Airports with less than  3 or more than 6 runways.
g.V().has('runways',outside(3,6)).values('code','runways')
```

Examples of *within* and *without*.

```groovy
// 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 less 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 less than 8 runways.

```groovy
// Airports with at least 5 runways but less than 8
g.V().has('runways',between(5,8)).values('code','runways')
```

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.

```groovy
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()
```

> **NOTE** 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*

```groovy
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.

```groovy
 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. Looks like we can change planes in Dallas or Houston but nothing goes via San Antonio.

```groovy
[AUS,DFW,LAS]
[AUS,IAH,LAS]
[AUS,DAL,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).

```groovy
 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!

```groovy
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.

```groovy
// 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]))
```

You will find more examples of these types of queries in the next section.

3.8.2. 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.

```groovy
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*

```groovy
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

```groovy
g.V().has('airport','code','AUS').
      out().and(has('code',neq('DFW')),has('code',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 easy 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.

```groovy
// 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.

```groovy
// 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 from routes from SAT to SLC with one stop but this time only returns routes that stop in one of the designated airports.

```groovy
// 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.

```groovy
[SAT,LAX,SLC]
[SAT,DFW,SLC]
```

Here are two more examples that use *without* and *within* to find routes based on countries.

```groovy
// 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.

```groovy
London
Frankfurt
Mexico City
Cancun
Guadalajara
```

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.

```groovy
// 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.

```groovy
Toronto
Mexico City
Cancun
Guadalajara
```

3.8.3. 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 20found.

```groovy
// 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 very small 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!

```groovy
// 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()
```
GROOVY

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')
```
GROOVY

3.8.4. Using *Math.random* to more randomly select a single node

While the *sample* step allows you to select one or more vertices at random, in my testing, at least when using a TinkerGraph, it tends to favor vertices with lower index values. So for example, in a test I ran this query 1000 times.

```
g.V().hasLabel('airport').sample(1).id()
```
GROOVY

What I found was that I 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)
```
GROOVY

Look at the code below. Even if I run that simple experiment many times 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
```
GROOVY

Given the air routes graph has over 3,300 airport vertices I wanted to come up with a query that gave a more likely result of picking any one airport from across all of 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 of the possible airports. Take a look at the snippets of Groovy/Gremlin code below.

> **NOTE** 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.

```groovy
// How many airports are there?
numAirports = g.V().hasLabel('airport').count().next()

3374

// Pick a random airport ID
x=Math.round(numAirports*Math.random()) as Integer

2359

// Get the code for our randomly selected airport
g.V(x).values('code')

PHO
```

This simple experiment shows that the numbers being generated using the *Math.random* approach appears to be a lot more evenly distributed across all of the possible airports.

```groovy
(1..10).each { println Math.round(numAirports*Math.random()) as Integer}

1514
18
3087
1292
3062
2772
2401
400
2084
3028
```

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 of 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. Indeed when I ran these queries on Janus graph the *sample* step did indeed 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 do something like this if we wanted to use our *Math.random* technique.

```groovy
airports = g.V().hasLabel('airport').id().toList()

numAirports = airports.size

3374

x=Math.round(numAirports*Math.random()) as Integer

859

g.V(airports[x-1])

v[859]

g.V(airports[x-1]).values('code')

OSR
```

3.9. Sorting things

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. I used *fold* at the end of the query to collect all of the results into a nice 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 document such as in the "Using *fold* to do simple Map-Reduce computations" section.

```groovy
// 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]


// 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()

[AAE,AAL,AAN,AAQ,AAR,AAT,AAX,AAY,ABA,ABB,ABD,ABE,ABI,ABJ,ABL,ABM,ABQ,ABR,ABS,ABT]
```

By default a sort performed using *order* is performed in ascending order. If we wanted to sort in descending order instead we can specify *decr* as a parameter to *order*. We can also specify *incr* if we want to be clear that we intend an ascending order sort.

```groovy
// Sort the first 20 airports returned in descending order
g.V().hasLabel('airport').limit(20).values('code').order().by(decr).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.

```groovy
g.V().hasLabel('airport').limit(20).values('code').order().by(shuffle).fold()

[MCO,LGA,BWI,IAD,ATL,BOS,DCA,BNA,IAH,DFW,MIA,MSP,ANC,AUS,JFK,ORD,PBI,FLL,LAX,PHX]
```

Below is an example where we combine the field we want to sort by *longest* and the direction we want the sort to take, *decr* into a single *by* instruction.

```groovy
// List the 10 airports with the longest runways in decreasing order.
g.V().hasLabel('airport').order().by('longest',decr).valueMap().select('code','longest').l
imit(10)
```

Here is the output from running the query. To save space I have split the results into two columns.

```
[code:[BPX],longest:[18045]]    [code:[DOH],longest:[15912]]
[code:[RKZ],longest:[16404]]    [code:[GOQ],longest:[15748]]
[code:[ULY],longest:[16404]]    [code:[HRE],longest:[15502]]
[code:[UTN],longest:[16076]]    [code:[FIH],longest:[15420]]
[code:[DEN],longest:[16000]]    [code:[ZIA],longest:[15092]]
```
GROOVY

Lastly, 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()
```
GROOVY

Now that we know how to sort things we could write a slightly more complex query that sorts all the airports by 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'),decr).limit(1).valueMap()
```
GROOVY

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], code:[BPX], longest:[18045], city:[Bangda], elev:[14219], icao:[ZUBD], lon:
[97.1082992553711], type:[airport], region:[CN-54], runways:[1], lat:[30.5536003112793],
desc:[Qamdo Bangda Airport]]
```
GROOVY

3.9.1. 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 four ways that we can control the sort in these cases. The four keywords are *keyIncr*, *keyDecr*, *valueIncr* and *valueDecr*.

The following shows the difference between running the query with and without the use of *order*

```
g.V().hasLabel('airport').limit(5).group().by('code').by('runways')

[BNA:[4],ANC:[3],BOS:[6],ATL:[5],AUS:[2]]

g.V().hasLabel('airport').limit(5).group().by('code').by('runways').
      order(local).by(keyIncr)

[ANC:[3],ATL:[5],AUS:[2],BNA:[4],BOS:[6]]
```
GROOVY

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(keyDecr)

[7:[DFW],6:[BOS],5:[ATL],4:[BNA,IAD],3:[ANC,BWI,DCA],2:[AUS,FLL]]
```
GROOVY

3.10. 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 I find that *or* can be avoided by using *within* for example and that *not* can be sometimes 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'))
```
GROOVY

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')
```
GROOVY

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.

```groovy
g.V().hasLabel('airport').or(has('region','US-TX'),
                             has('region','US-LA'),
                             has('region','US-AZ'),
                             has('region','US-OK'))
                         .order().by('region',incr)
                         .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.

```groovy
g.V().hasLabel('airport').has('region',within('US-TX','US-LA','US-AZ','US-OK')).
                          order().by('region',incr).
                          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.

```groovy
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.

```groovy
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?

```groovy
g.V().not(hasLabel('airport')).count()
```

This previous query could also be written as follows.

```groovy
g.V().has(label,neq('airport')).count()
```

TIP

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 *within* and *without* my 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 less than 100 outbound routes but more than 95 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.

```groovy
g.V().hasLabel('airport').
      where(out().count().is(lt(100).and(gt(95)))).
      group().by('code').by(out().count())

[BUD:96,SZX:99,AUH:97,BGY:99]
```

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.

```groovy
g.V().hasLabel('airport').where(and(out().count().is(lt(100)),
                out().count().is(gt(95)))).
      group().by('code').by(out().count())

[BUD:96,SZX:99,AUH:97,BGY:99]
```

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.

```groovy
g.V().hasLabel('airport').
      where(out().count().is(lt(100).and(gt(95)))).
      group().by(out().count()).by('code')

[96:[BUD],97:[AUH],99:[SZX,BGY]]
```

You can also add additional inline *and* steps to a query as shown below. Notice that this time *AUH* is not part of the result set as it has 97 routes which our new *and* test eliminates.

```
GROOVY
g.V().hasLabel('airport').
      where(out().count().is(lt(100).and(gt(95)).and(neq(97)))).
      group().by(out().count()).by('code')

[96:[BUD],99:[SZX,BGY]]
```

We used the *where* step a lot in the examples above. We will explain in more detail how the *where* step works in the "Using *where* to filter things out of a result" section which is coming up soon.

3.11. 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.

> **TIP** 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.

```
GROOVY
// 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. The next example starts to show the power of the *where* step. We can include a traversal inside of 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.

```
GROOVY
// Airports with more than 60 unique routes from them
g.V().hasLabel('airport').where(out('route').count().is(gt(60))).count()

179
```

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 of an *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.

```groovy
// 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.

```groovy
[ATL,4502,HNL]
[JFK,4970,HNL]
[ORD,4230,HNL]
[EWR,4950,HNL]
[HNL,4502,ATL]
[HNL,4970,JFK]
[HNL,4230,ORD]
[HNL,4950,EWR]
```

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.

```groovy
// List 10 places you can fly to with one stop, starting at Austin but
// never ending up back in Austin
g.V().has('code','AUS').as('a').out().out().where(neq('a')).
              path().by('code').limit(10)
```

3.11.1. Using *where* and *by* to filter results

A new capability was added in the Tinkerpop 3.2.4 release that allows a *where* step to be followed with a *by* modulator. This makes writing certain types of queries a lot easier than it was before. 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 as that is handled for us by the query.

```groovy
g.V().has('code','AUS').as('a').out().
      where(eq('a')).by('runways').valueMap('code','runways')
```

TIP

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.

```groovy
[code:[LHR],runways:[2]]
[code:[MEX],runways:[2]]
[code:[CUN],runways:[2]]
[code:[GDL],runways:[2]]
[code:[PNS],runways:[2]]
[code:[VPS],runways:[2]]
[code:[FLL],runways:[2]]
[code:[SNA],runways:[2]]
[code:[MSY],runways:[2]]
```

3.12. 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.

```groovy
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*.

```groovy
// 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'))
```

Here is another example that uses the same constructs.

```groovy
// 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(15)
```

Sometimes it is very useful, as the query 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 less than four runways or "lots" if it has more than four.

```groovy
// 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'))
```

TIP 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 the both the derived and constant parts of the query result that we will ultimately return.

```groovy
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:YYT,b:not so many]
[a:YEG,b:not so many]
[a:LGA,b:lots]
[a:DXB,b:lots]
[a:BLR,b:not so many]
[a:CGN,b:lots]
[a:BOM,b:lots]
[a:SIN,b:lots]
[a:TSF,b:not so many]
[a:HKG,b:lots]
```
GROOVY

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)
```
GROOVY

Here is what the output from the modified form of the query.

```
[YYT,not so many]
[YEG,not so many]
[LGA,lots]
[DXB,lots]
[BLR,not so many]
[CGN,lots]
[BOM,lots]
[SIN,lots]
[TSF,not so many]
[HKG,lots]
```
GROOVY

3.13. 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.

```groovy
// 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'))


// This example has three options - note the default case of 'none' used as the catchall
// Note how for ease of reading the query can be split across multiple lines.
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'))
```

3.14. Using *match* to do pattern matching

Match was added in TinkerPop 3 and allows a more declarative style of pattern based query to be expressed in 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.

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.

```groovy
// Find any cases of where you can fly from JFK non stop to a place you cannot get back
from
// there non stop.This query should return LCY, as the return flight stops in DUB 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.

```groovy
// 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 query to find out just how many there are.

```groovy
// 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'))).
                          select('s','d').by('code').count()


238
```

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.

```groovy
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.

```groovy
[a:ATL,b:MLB,c:ISP]
[a:ATL,b:MLB,c:BIM]
[a:ATL,b:MLB,c:YTZ]
[a:ATL,b:PHF,c:SFB]
[a:ATL,b:SBN,c:SFB]
[a:ATL,b:SBN,c:AZA]
[a:ATL,b:SBN,c:PIE]
[a:ATL,b:SBN,c:PGD]
[a:ATL,b:TRI,c:SFB]
[a:ATL,b:TRI,c:PIE]
```

Here is another example of using *match* along with a *where*. 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. While this is a bit of a contrived query, it does show the power of *match* and also illustrates using values calculated in one part of a query later on in that same query.

```groovy
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 I mentioned, the example above is a bit contrived and it could actually be done without using a *match* step at all and just using a *where* step as shown below. Hopefully, however, these examples give you an idea of what *match* can do.

```groovy
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')
```

3.15. Using *union* to combine query results

The *union* step works just as you would expect from its name. It allows to 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 very useful capability.

By way of a simple example, the following query returns flights that arrive in AUS from the UK of that leave AUS and arrive in Mexico.

```groovy
// 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

```groovy
[AUS,LHR]
[AUS,MEX]
[AUS,CUN]
[AUS,GDL]
```

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.

```groovy
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 we get back from running the query.

```groovy
[LHR,CDG]
[LHR,ORY]
[LHR,TXL]
[LGW,CDG]
[LGW,SXF]
[LCY,CDG]
[LCY,ORY]
[STN,TXL]
[STN,SXF]
[LTN,CDG]
[LTN,SXF]
```

As mentioned above, sometimes, especially for fairly simple queries, there are alternatives to using *union*. The previous query could have been written as follows using the *within* method.

```groovy
// The above query could have been written without using union() as follows
g.V().has('city','London').has('region','GB-ENG').
      out().has('city',within('Paris','Berlin')).path().by('code')
```

This next query is 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 sets of operations as you need to. In this example we combine the results of three sets of traversals.

```groovy
// 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

```groovy
[LHR,227,ORY,513,BCN]
[LHR,216,CDG,533,BCN]
[LGW,591,SXF,1432,LIS]
[LGW,191,CDG,533,BCN]
[LCY,227,ORY,513,BCN]
[STN,563,SXF,1432,LIS]
[LTN,589,SXF,1432,LIS]
[LTN,236,CDG,533,BCN]
```

Here is one last example of using union. This 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.

```groovy
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.

```groovy
397708
396410
```

3.16. 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 stores it in a traversal variable named *a* before counting how many places you can get towith 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.

```
GROOVY
g.V(3).sideEffect(out().count().store('a')).out().out().count().as('b').select('a','b')

[a:[59],b:5911]
```

Later in the document 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 node 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.

```
GROOVY
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 document will will look at other ways that side effects can be used to solve more interesting problems.

3.17. Using *aggregate* to create a temporary collection

At the time of writing this document, there were 59 places you could fly to directly (non stop) from Austin. We can verify this fact using the following query.

```
GROOVY
g.V().has('code','AUS').out().count()

59
```

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.

```
GROOVY
g.V().has('code','AUS').out().out().dedup().count()

865
```

There is however a problem with this query. The 865 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 865 some how. This is where *aggregate* is useful. Take a look at the modified query below

```
GROOVY
g.V().has('code','AUS').out().aggregate('nonstop').
     out().where(without('nonstop')).dedup().count()

806
```

After the first *out* step. all of the vertices that were found are stored in a collection I 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 806 is precisely 59 less than the 865 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!

3.18. 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. In the example below, the string *ABIA*, another acronym commonly used when referring to the Austin Bergstrom International Airport, is injected into the query.

```
GROOVY
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
```
GROOVY

3.19. 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 labelled *contains* will be returned.

```
// Return the first step inside coalesce that returns a node
g.V(3).coalesce(out('fly'),__.in('contains')).valueMap()
```
GROOVY

As there are not any edges labelled *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]]
```
GROOVY

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]]
```

3.19.1. Combining *coalesce* with a *constant* value

The *coalesce* step can also be very 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
```

3.20. 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 counts the number of outgoing edges from the node with an ID of 3.

```
g.V(3).outE().count()

59
```

Likewise, this query counts the number of incoming edges to that same node.

```
g.V(3).inE().count()

61
```

The following query introduces the *bothE* step. What this step does is return all of the edges connected to this node 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()

120
```

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 node with an ID of 3 regardless of whether they are connected by an outgoing or an incoming edge.

```
g.V(3).both().count()

120
```

This next query can be used to show us the 120 vertices that we just counted in the previous query. I 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[34],v[34],v[35],v[35],v[38],v[38],v[39],v[39],v
[41],v[41],v[42],v[42],v[45],v[45],v[46],v[46],v[47],v[47],v[49],v[49],v[52],v[52],v[136],
v[136],v[147],v[147],v[149],v[149],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[190],v[190],v[273
],v[273],v[278],v[278],v[389],v[389],v[416],v[416],v[430],v[430],v[549],v[549],v[929],v[92
9],v[1274],v[1274],v[3591],v[3605]]
```

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 wanted to eliminate any duplicate results we can do that by adding a *dedup* step to our query.

```groovy
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[34],v[35],v[38],v[39],
v[41],v[42],v[45],v[46],v[47],v[49],v[52],v[136],v[147],v[149],v[178],v[180],v[182],v[183]
,v[184],v[185],v[186],v[187],v[188],v[190],v[273],v[278],v[389],v[416],v[430],v[549],v[929
],v[1274],v[3591],v[3605]]
```

We can do another count using our modified query to check we got the expected number of results back.

```groovy
g.V(3).both().dedup().count()

61
```

There are 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 node with the ID of 3 and then looks at all the edges no matter whether they are incoming or outgoing and retrieves all of the vertices at each end of those edges using the *bothV* step. Notice that this time our count is 240. This is because for every one of the 120 edges, we asked for the node at each end so we ended up with 240 of them.

```groovy
g.V(3).bothE().bothV().count()

240
```

We can again add a *dedup* step to get rid of duplicate vertices as we did before and re-do the count but notice this time we get back 62 instead of the 61 we got before. So what is going on here?

```groovy
g.V(3).bothE().bothV().dedup().count()

62
```

Let's run another query and take a look at all of the vertices that we got back this time.

```
GROOVY
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[34],v[35],v[38],v
[39],v[41],v[42],v[45],v[46],v[47],v[49],v[52],v[136],v[147],v[149],v[178],v[180],v[182],v
[183],v[184],v[185],v[186],v[187],v[188],v[190],v[273],v[278],v[389],v[416],v[430],v[549],
v[929],v[1274],v[3591],v[3605]]
```

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 61 again.

```
GROOVY
g.V(3).bothE().otherV().dedup().count()
61
```

So let's again look at the returned vertices and see what the difference is.

```
GROOVY
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[34],v[35],v[38],v[39],
v[41],v[42],v[45],v[46],v[47],v[49],v[52],v[136],v[147],v[149],v[178],v[180],v[182],v[183]
,v[184],v[185],v[186],v[187],v[188],v[190],v[273],v[278],v[389],v[416],v[430],v[549],v[929
],v[1274],v[3591],v[3605]]
```

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]*.

3.21. 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 travelled to be travelled again. We can ask for this behavior this 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!

```groovy
// 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.

```groovy
[AUS,YYZ,BOM,AGR]
[AUS,LHR,BOM,AGR]
[AUS,FRA,BOM,AGR]
[AUS,EWR,BOM,AGR]
[AUS,YYZ,ZRH,BOM,AGR]
[AUS,YYZ,BRU,BOM,AGR]
[AUS,YYZ,MUC,BOM,AGR]
[AUS,YYZ,ICN,BOM,AGR]
[AUS,YYZ,CAI,BOM,AGR]
[AUS,YYZ,ADD,BOM,AGR]
```

You can also place the *repeat* before the *until* as shown below.

```groovy
// 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,LAX,SYD]
[AUS,SFO,SYD]
[AUS,YYZ,HND,SYD]
[AUS,YYZ,ICN,SYD]
[AUS,YYZ,SCL,SYD]
[AUS,YYZ,AUH,SYD]
[AUS,YYZ,TPE,SYD]
[AUS,YYZ,CAN,SYD]
[AUS,YYZ,DFW,SYD]
```
GROOVY

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.

```
g.V().has('code','AUS').repeat(out()).times(2).has('code','SYD').path().by('code')

[AUS,DFW,SYD]
[AUS,LAX,SYD]
[AUS,SFO,SYD]
```
GROOVY

The previous query is equivalent to this next one but doing it this way is less flexible in that we can not as easily vary the number of *out* steps, should, for example we want to next try 5 hops.

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

A bit later, in the Using *emit* to return results during a traversal section, we will look at how *emit* can be used to adjust the behavior of a *repeat...times* loop.

3.21.1. A warning that *path* can be memory and CPU intensive

The *path* step is incredibly useful and I find myself using it a lot. However, there are some downsides to the use of *path*, especially when searching entire graphs for non trivial results. 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.

```
GROOVY
g.V().has('code','PSV').repeat(out().simplePath()).until(has('code','AUS')).
      limit(10).path().by('code')

[PSV,LWK,FIE,KOI,EDI,JFK,AUS]
[PSV,LWK,FIE,KOI,EDI,EWR,AUS]
[PSV,LWK,FIE,KOI,EDI,LHR,AUS]
[PSV,LWK,FIE,KOI,EDI,FRA,AUS]
[PSV,LWK,FIE,KOI,GLA,JFK,AUS]
[PSV,LWK,FIE,KOI,GLA,MCO,AUS]
[PSV,LWK,FIE,KOI,GLA,EWR,AUS]
[PSV,LWK,FIE,KOI,GLA,PHL,AUS]
[PSV,LWK,FIE,KOI,GLA,YYZ,AUS]
[PSV,LWK,FIE,KOI,GLA,LHR,AUS]
```

However, were we to reverse the query as follows 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.

```
GROOVY
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.

NOTE | The *path* step uses a lot of memory and in some cases can cause issues.

The reason that so much memory is consumed is that the *path* step, even if *simplePath* is used to avoid travelling the same path twice, has to store up a very large number of routes before finding the ones we are actually interested in. So while the *path* step is incredibly useful, be aware that in cases like this one, it can get you into trouble if not used with care.

3.22. Calculating node degree

While working with graphs, the word *degree* is used when discussing the number of edges coming into a node, going out from a node or potentially both coming in and going out. It's quite simple with Gremlin to calculate various measures of degree as the examples below show.

```groovy
// Out degree (number of routes) from each node (airport)
g.V().out().groupCount().by('code')

// what is the out degree (number of unique outbound routes) from LHR, JFK and DFW?
 g.V().hasLabel('airport').out().groupCount().by('code').select('LHR','JFK','DFW')

// Calculate degree (in and out) for each node (so counts edges going both ways)
// Note the project() step was introduced in gremlin 3.2 and is ideal for this!
g.V().hasLabel('airport').limit(10).project("v","degree").by('code').by(bothE().count())

// We could also write the same query like this.
g.V().hasLabel('airport').limit(10).group().by('code').by(bothE().count())
```

3.23. More examples using what we have covered so far

The queries in this section build upon the topics that we have covered so far.

```groovy
// Which cities can I fly to from any airport in the Hawaiian islands?
g.V().has('airport','region','US-HI').out().path().by('city')

// Find all the airports that are in Europe (The graph stores continent information
// as "contains" edges connected from a "continent" node to each airport node.
g.V().has('continent','code','EU').out('contains').values('code').order().fold()
```

The next queries show two ways of finding airports with 6 or more runways.

```groovy
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 3.

```groovy
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.

```groovy
// 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.

```groovy
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!

```groovy
// 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]
```

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!

```groovy
g.V().has('code','BNE').out().as('stop').
                        out().has('code','AUS').select('stop').values('code')

LAX
```

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 less than five outgoing routes and counts them. It does this by counting the number of airports that have less than five outgoing edges with a *route* label. There are a surprisingly high number of airports that offer this small number of destinations.

```groovy
// Airports with less than 5 outgoing edges
g.V().hasLabel('airport').where(out('route').count().is(lt(5))).count()

2058
```

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.

```groovy
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 a specific criteria. The first finds routes where the distance is exactly 100 miles. The second one looks for the airports that have an elevation above 10,000 feet.

```groovy
// 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')

// Airports above 10,000ft sorted by ascending elevation
g.V().has('airport','elev', gt(10000)).
      order().by('elev',incr).valueMap('city','elev')
```

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*.

```groovy
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 intresting as the continent information is represented in the graph as edges connecting an airport node with a continent node. 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 node that represents Africa. The third query uses *where* to show an alternate way of achieving the same result.

```groovy
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. I have laid them out in two columns to save space.

```groovy
[a:JNB, b:ATL]    [a:ACC, b:JFK]
[a:JNB, b:JFK]    [a:CMN, b:IAD]
[a:CAI, b:JFK]    [a:CMN, b:JFK]
[a:ADD, b:IAD]    [a:GCK, b:DFW]
[a:ADD, b:EWR]    [a:DKR, b:IAD]
[a:LOS, b:ATL]    [a:DKR, b:JFK]
[a:LOS, b:IAH]    [a:LFW, b:EWR]
[a:LOS, b:JFK]    [a:RAI, b:BOS]
[a:ACC, b:IAD]
```

The query below shows how to use the *project* step that was introduced in TinkerPop 3, 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 document that use variations of this collection of steps.

```groovy
g.V().has('code','AUS').out().project('ap','rw').by('code').by('runways').
      order().by(select('rw'),decr).limit(10)
```

Here are the results we get from running the query.

```groovy
[ap:ORD, rw:8]
[ap:DFW, rw:7]
[ap:BOS, rw:6]
[ap:DEN, rw:6]
[ap:DTW, rw:6]
[ap:YYZ, rw:5]
[ap:MDW, rw:5]
[ap:ATL, rw:5]
[ap:IAH, rw:5]
[ap:FRA, rw:4]
```

4. BEYOND BASIC QUERIES

So far we have looked mostly at querying the graph. In the following sections we will look at some other topics that it is also important to be familiar with when working with Gremlin such as mixing in some Groovy or Java code with your queries, adding vertices (vertices),and edges and properties to a graph and also how to delete them. We will also look at how to create a sub-graph and how to save a graph to and XML or JSON file.

4.1. A word about layout and indentation

As you begin to write more complex Gremlin queries they can get quite lengthy. In order 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. I am not going to propose an indentation standard, I believe this should be left to personal preference however there are a few things I 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.

The following example shows the query we already looked at in the Boolean operations section of this document but this time edited so that it could be copy and pasted directly into the Gremlin console.

```groovy
g.V().hasLabel('airport').has('region',within('US-TX','US-LA','US-AZ','US-OK'))   \
                         .order().by('region',incr)   \
                         .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.

```groovy
g.V().hasLabel('airport').has('region',within('US-TX','US-LA','US-AZ','US-OK')).
                          order().by('region',incr).
                          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.

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.

4.2. A warning about reserved word conflicts and collisions

Most of the time the issue I am 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. One 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()
```
GROOVY

That query does not cause an error and correctly returns all of the vertices that are connected by an incoming edge, to the *AUS* node. 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())
```
GROOVY

In this case the *in()* is on it's 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())
```
GROOVY

Notice that I added the "__." (underscore underscore period) in front of the *in()*. 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 "Modelling an ordered binary tree as a graph" section a bit later on.

4.3. 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 I decided to model continents as their own vertices. So each of the seven continents has a node. Each node is connected to airports within that continent by an edge labeled "contains".

I could have chosen to just make the continent a property of each airport node but had I done that, to answer the question about "routes starting in Africa" I would have to look at every single airport node in the graph just to figure out which continent contained it. By giving each continent it's own node I am 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 node 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.

```groovy
// 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.

```groovy
// 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!

4.3.1. Keeping information in two places within the same graph

Sometimes, to improve query efficiency I 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 I decided to model countries. I have a unique node for each country but I also store the

country code as a property of each airport node. In a small graph this perhaps is overkill but I 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.

```groovy
g.V().has('country','code','PT').out("contains").values('city')

g.V().has('airport','country','PT').values('city')
```

The first query finds the country node for Portugal and then, finds all of 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 less 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 less than 300 country vertices. Also, in a production system with an index in place finding the *Portugal* node should be very fast.

Conversely, if we were already looking at an airport node 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 node.

So there is no golden rule here but it is something to think about while designing your data model.

4.3.2. 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 I 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 I could do is add a URL as a property of each airport node that points to the airports home page or some other resource that has all of 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*.

4.3.3. A few words about *supernodes*

When a node 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 node. Such vertices (nodes) 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 nodes 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 node will have to look at most if not all of the edges connected to that node as part of a traversal.

The air-routes graph does not really have anything that could be classed as a *supernode*. The node with the most edges is the continent node 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 some of the people on Twitter with millions of followers. Without taking some precautions, such a social network, modelled as a graph, could face issues.

As you design your graph model it is worth considering that some things are perhaps better modelled as a node property than as a node 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 of the airports in the graph were in the same country that would still give us less than 3,500 edges connected to that node. 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 same the country that would be a guaranteed way to get a *supernode* if we modelled that relationship by connecting every person node to a country node 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 node.

A detailed discussion of *supernode* mitigation is beyond the scope of this document but I 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.

4.4. 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 of 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 3 classes as needed. This capability makes Gremlin

additionally powerful. You can also take advantages of these features when working with Gremlin Server over HTTP and with other TinkerPop based Graph services like IBM Graph with the caveat that some features may be blocked if viewed as a potential security risk to the server.

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 multi part Gremlin query.

In this section we are going to go one step further and actualaly 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.

```groovy
// 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 node in a nice, human readable, form.

```groovy
// Groovy function to display node 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.

```groovy
// 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.

```groovy
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.

```groovy
for (a in 1..10) print g.V().has(id,a).values('code').next()+" "
```

This example returns a hash of vertices, with node labels as the keys and the code property as the values. It then uses the label names to access the returned hash.

```groovy
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 of 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 in order 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.

```groovy
// 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.

```groovy
// 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 node
a['JFK'][0].values('desc')
```

Another useful way to work with veriables 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 lisy 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.

```groovy
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 stand alone Groovy application.

```groovy
// How many results did we get back?
german.size

32

// 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()

237
```

Towards the end of the document, in the "Working with TinkerGraph from a Groovy application" section, we will explore writing some stand alone Groovy code that can use the TinkerPop API and issue Gremlin queries while running outside of the Gremlin Console as a stand alone application.

4.4.1. 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 second query. Notice how we do this by passing the variable containing the Austin vertex into the *V()* step.

```groovy
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 of 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.

```groovy
// 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.

```groovy
// 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 of a *has* step. We initially create a list containing all of 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 document.

```groovy
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.

```groovy
['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]
```

4.5. Adding vertices, edges and properties

So far in this document 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.

Vertices and edges can be added directly to the graph using the *graph* object or as part of a graph traversal. We will look at both of these techniques over course of the following pages.

4.5.1. Adding an airport (vertex) and a route (edge)

The following code uses the *graph* object that we created when we fist loaded the air-routes graph to create a new airport vertex (node) and then adds a route (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. While using the *graph* object in this way works, it is strongly recommended that the traversal source object *g* be used instead and that vertices and edges be added using a traversal. 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 = graph.addVertex(label,'airport',
                      'code','XYZ',
                      'icao','KXYZ',
                      'desc','This is not a real airport')

// Find the DFW vertex
dfw = g.V().has('code','DFW').next()

// Create a route from our new airport to DFW
xyz.addEdge('route',dfw)
```

In many cases it is more convenient, and also recommendd, to perform each of the previous operations using just the traversal object *g*. The following example does just that. We first create a new aiport vertex for our imaginary airport and store it's vertex in the variable *xyz*. We can then

use that stored value when we create the edge also using a traversal. As with many parts of the Gremlin language, there is more than one way to achieve the same results.

```groovy
// 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, how each property step can be chained to the previous one when adding multiple properties. Whether you need to do it while creating a node or to add and edit properties on a node at a later date you can use the same *property* step.

> **NOTE** It is strongly recommended that the traversal source object *g* be used when adding, updating or deleting vertices and edges. Using the *graph* object directly is not viewed as a TinkerPop best practice.

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 a *to* step and wrapping the variable inside of a *V* step.

```groovy
// Add a route from DFW to XYZ
g.V().has('code','DFW').addE('route').to(V(xyz))
```

We could have written the previous line like this if we had not previously saved anything in a variable.

```groovy
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.

```groovy
// Add the return route back to DFW
g.V().has('code','DFW').addE('route').from(V(xyz))
```

Another way that we could have chosen to create our edge is by naming things using *as* steps. Note also there is more than one way to define the direction of an edge. Two examples are shown below. One uses the *addOutE* step and the other uses the *outE* and *to* combination. Notice also how the *V* step was used below to essentially start a new traversal midway through the current one.

```
g.V().has('code','XYZ').as('a').V().has('code','DFW').addOutE('route','a')

g.V().has('code','XYZ').as('a').V().has('code','DFW').addE('route').to('a')
```

You will see a bigger example that uses *as* to name steps in the "Quickly building a graph for testing" section that is coming up soon.

4.5.2. 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 of 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 it's value a list of codes.

```
g.V().has('code','AUS').valueMap('places')

[places:[[YYZ, LHR, FRA, MEX, PIT, PDX, CLT, CUN, MEM, CVG, IND, MCI, DAL, STL, ABQ, MDW,
LBB, HRL, GDL, PNS, VPS, SFB, 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, MSY,
EWR, HOU, ELP, CLE, OAK, PHL, DTW]]]
```

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()

59
```

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.

```groovy
g.V().has('code','AUS').values('places').next()[2..4]

FRA
MEX
PIT
```

4.5.3. 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 stand alone 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 of the vertices and edges are created in a single query with each step joined together.

```groovy
graph=TinkerGraph.open()
g=graph.traversal()
g.addV(label,'airport','code','AUS').as('aus').
  addV(label,'airport','code','DFW').as('dfw').
  addV(label,'airport','code','LAX').as('lax').
  addV(label,'airport','code','JFK').as('jfk').
  addV(label,'airport','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')
```

4.5.4. 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 node 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 node for us.

```groovy
vertices = [["WYZ","KWYZ"],["XYZ","KXYZ"]]
for (a in vertices) {graph.addVertex(label,"airport","code",a[0],"iata",a[1])}
```

We could also have added the vertices using the traversal object *g* as follows. Notice the call to *next()*. Without this the node creation will not work as expected.

```groovy
vertices = [["WYZ","KWYZ"],["XYZ","KXYZ"]]
for (a in vertices) {g.addV("airport").property("code",a[0],"iata",a[1]).next()}
```

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.

If you prefer a more Groovy like syntax you can also do this.

```groovy
vertices = [["WYZ","KWYZ"],["XYZ","KXYZ"]]
vertices.each {g.addV("airport").property("code",it[0],"iata",it[1]).next()}
```

4.6. Deleting vertices, edges and properties

So far in this document 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.

4.6.1. Deleting a node

In some of our earlier examples we created a fictitious airport node 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 node will also remove any edges we created connected to that node.

```groovy
// Remove the XYZ node
g.V().has('code','XYZ').drop()
```

4.6.2. 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.

```groovy
// 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()
```

4.6.3. Deleting a property

Lastly, we can use *drop* to delete a specific property value from a specific node. 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]]
```
GROOVY

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]]
```
GROOVY

If we wanted to delete all of the properties currently associated with the SFO airport node we could do that as follows.

```
g.V().has('code','SFO').properties().drop()
```
GROOVY

4.6.4. 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()
```
GROOVY

You could also use the *graph* object to do this. The code below uses the graph object to retrieve all of the edges and then iterates over them dropping them one by one. Again for very large graphs this may not be an ideal approach is this requires reading all of the edge definitions into memory. Note that in this case we call the *remove* method rather than use *drop* as we are not using a graph traversal in this case.

```
// Remove all the edges from the graph
graph.edges().each{it.remove()}
```
GROOVY

You could also delete the whole graph, vertices and edges, by deleting all of the vertices!

```groovy
// Delete the entire graph!
g.V().drop()
```

4.7. Property keys and values revisited

We have already looked, earlier in the document, 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.

4.7.1. The *Property* and *VertexProperty* interfaces

In a TinkerPop 3 enabled graph, all properties are implementations of the Property interface. Vertices implement the *VertexProperty* interface which itself implements *Property*. These interfaces are documented as part of the Apache TinkerPop 3 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 3 defines in the "Working with TinkerGraph from Java Application" section a bit later in this document.

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 document. 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 node.

```groovy
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.

```groovy
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 node 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.

```groovy
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 of the property keys associated with a given node we could write a query like the one below that finds all of the property keys associated with the DFW node in the air-routes graph.

```groovy
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 node using this query. Note that edge properties are implementations of *Property* and not *VertexProperty*.

```groovy
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:25497644, runways:4816]
```
GROOVY

4.7.2. The *propertyMap* traversal step

We have previously used the *valueMap* step to produce a map of key/value pairs for all of the properties associated with a node 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()
[country:[vp[country->US]], code:[vp[code->AUS]], longest:[vp[longest->12250]], city:
[vp[city->Austin]], lon:[vp[lon->-97.66989893868164]], type:[vp[type->airport]], places:
[vp[places->[YYZ, LHR, FRA, MEX,]], 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 Int]]]
```
GROOVY

4.7.3. 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 an ID. Unlike node and edge IDs property IDs are not guaranteed to be unique across the graph. Certainly with TinkerGraph I have encountered cases where a node 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')
vp[city->London]
vp[city->London]
vp[city->London]
vp[city->London]
vp[city->London]
vp[city->London]
```
GROOVY

At first glance, each of the values returned above looks identical. However, let's now query their ID values.

```groovy
g.V().properties().hasValue('London').id()

583
595
1051
1123
2467
7783
```

As you can see each property has a different, and unique, ID. 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.

```groovy
g.V().properties().hasId(583)

vp[city->London]
```

We can query the value of this property as you would expect.

```groovy
g.V().properties().hasId(583).value()

London
```

We can retrieve the name of the property key as follows.

```groovy
g.V().properties().hasId(583).key()

city
```

We could also have used *label* instead of *key*

```groovy
g.V().properties().hasId(583).label()

city
```

We can also find out which element (node or edge) that this property belongs to.

```groovy
g.V().properties().hasId(583).next().element()

v[49]
```

We can also look at other property values of the element containing our property with an ID of 583.

```groovy
g.V().properties().hasId(583).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.

```groovy
g.V().has('airport','code','DFW').properties('city').next().graph()

tinkergraph[vertices:3619 edges:50148]
```

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.

```groovy
p = g.V(3).properties().toList()
p.each {println it.key + "\t:" + it.id}

country :28
code    :29
longest :30
city    :31
elev    :32
icao    :33
lon     :34
type    :35
region  :36
runways :37
lat     :38
desc    :39
```

NOTE

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 of all 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.

```groovy
g.V(4).properties('city').id()

43

g.V(4).property('city','newname')

g.V(4).properties('city').id()

53361
```

The fact that every property in a graph has an ID can improve performance of accessing properties, especially in large graphs.

4.7.4. Attaching multiple values (lists or sets) to a single property

A node property value can be a basic type such as a String or an Integer but it can also be something 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 setup when the node is first created or added afterwards. These more complex type of property values are not supported on edges.

If we wanted to store the IATA amd 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 node follows. You can also add properties to an existing node that have lists of values. We will look at how to do that later in this section.

```groovy
g.addV('code','AUS','code','KAUS')
```

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 setup explicit schemas for our property types. However, once we start working with a more sophisticated graph system such as Janus Graph, that is something that we will both want and need to be able to do. We cover the topic of cardinality in detail in the "The Janus Graph management API" section later in the document.

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*.

```groovy
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.

```groovy
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.

```groovy
g.V().has('code','AUS').properties()

vp[code->AUS]
vp[code->KAUS]
```

For completeness we could also do this.

```groovy
g.V().properties().hasValue('AUS')

vp[code->AUS]
```

So now we know how to create a node 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 paramter 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 that is 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*.

```groovy
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.

```groovy
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.

```groovy
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 the value map after dropping *ABIA* we can indeed see that it is gone from the list.

```groovy
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.

```groovy
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.

```groovy
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 node, with some duplicate values associated with the *dups* property.

```groovy
g.addV(label,'test','dups','one','dups','two','dups','one')

g.V().hasLabel('test').valueMap()

[dups:[one,two,one]]
```

We can add additional duplicate values after the node has been created.

```groovy
g.V().hasLabel('test').property(list,'dups','two')

g.V().hasLabel('test').valueMap()

[dups:[one,two,one,two]]
```

So far we have just created values in a list that have *LIST* cardinality which means that duplicate values are allowed. If we wanted 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 inded our set was created.

```groovy
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 aready 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]]
```

Note that the other examples we have shown in this section are not the same as 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]]]
```

4.7.5. Adding properties to other properties (meta properties)

TinkerPop 3 introduced the ability to 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 value of *6/6/2017* to the property with a key of *code* and a value of *AUS*.

```
g.V().has('code','AUS').properties().hasValue('AUS').property('date','6/6/2017')
```

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','6/6/2017')
```

We can retrieve all of the meta properties on a specific property, such as the *code* property as follows.

```
g.V().has('code','AUS').properties('code').properties()

p[date->6/6/2017]
```

If you want to find all the properties associated with the AUS vertex that have a meta property with a date of *6/6/2017* you can do that as follows.

```groovy
g.V().has('code','AUS').properties().has('date','6/6/2017')

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*.

```groovy
g.V().has('code','AUS').properties().hasValue('AUS').properties('date')

p[date->6/6/2017]
```

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.

```groovy
g.V(3).property('comment','I like this airport','date','6/6/2017','user','Kelvin')
```

We can query the graph to make sure everything worked as expected.

```groovy
g.V(3).properties('comment')

vp[Comment->I like this airport]

g.V(3).properties('comment').properties()

p[date->6/6/2017]
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 whole node.

```groovy
g.V().has('code','AUS').properties().hasValue('AUS').property('date','6/6/2017').drop()
```

To remove a single meta property we need to use drop in this way

```groovy
 g.V().has('code','AUS').properties('code').properties('date').drop()
```

Note that you cannot chain meta properties together endlessly. The main properties on a node 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. Using Lambda functions

Gremlin allows you to put 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* or *sideEffect* step but there are other places where you wll find this concept used. This technique provides a lot of additional flexibility in how queries can be written. However, care should be used. When processing your query, Gremlin wil 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 over use 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. Here is an example of a closure being used where a *has* step could and should have been used instead.

```groovy
// 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.

```groovy
// What airports are located in London?
g.V().hasLabel('airport').has('city','London')
```

I think you will agree that the second version is a lot simpler to read and enables Gremlin to do its thing.

For completeness, here is an 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.

```groovy
// Example of the 'sideEffect' step
g.V().has('runways',6).sideEffect{print it.get().values('code').next()+" "}.count()
```

TIP

The moral here is, avoid closures unless you can genuinely find no other way to achieve what you need. It is fair I think to observe that sometimes coming up with a closure to do what you want is easier than figuring out the pure Gremlin way to do it but if at all possible using just Gremlin steps is still the recommended path to take. Lambda functions in general are discouraged.

4.8.1. 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.

Take a look at the query below. It simply returns us a list containing the IDs of the first ten airports in the graph.

```groovy
g.V().hasLabel('airport').limit(10).id().fold()

[1,2,3,4,5,6,7,8,9,10]
```

Let's say 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. gg

```groovy
g.V().hasLabel('airport').limit(10).map{it.get().id() + 1}.fold()

[2,3,4,5,6,7,8,9,10,11]
```

NOTE

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 semi-colons. The following query demonstrates this.

```groovy
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*, 4, is added to each ID.

In the next section we will take a look at some other cases where lambda expressions are very useful.

4.8.2. Using regular expressions to do fuzzy searches

Let's take a look at one case where use of closures might be helpful. It is a common requirement when working with any kind of database to want to do some sort of fuzzy text search or even to search using a regular expression. TinkerPop 3 itself does not provide direct support for this. In other words there currently is no sophisticated text search method beyond the basic *has()* type steps we have looked at above. However, the underlying graph store can still expose such capabilities.

NOTE

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 Elastic Search and a graph engine like Titan. In those cases some amount 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.

When working with Tinkergraph and the Gremlin console if we want to do any sort of text search beyond very basic things like *city == "Dallas"* then we will have to fall back on the Lambda function concept to take advantage of underlying Groovy and Java features. Note that even in graph systems backed by a real index the examples we are about to look at should still work but may not be the preferred way.

So let's look at some examples. First of all, every airport in the air routes graph contains a description which will be something like *Dallas Fort Worth International Airport* in the case of DFW. If we wanted to search the vertices in the graph for any airport that has the word *Dallas* in the description we could take advantage of the Groovy *String.contains()* method and do it like this.

```groovy
// Airport descriptions containing the word 'Dallas'
g.V().hasLabel('airport').filter{it.get().property('desc').value().contains('Dallas')}
```

Where things get even more interesting is when you want to use a regular expression 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!.

```groovy
// Using a filter to search using a regular expression
g.V().has('airport','type','airport').filter{it.get().property('city').value
==~/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').filter{it.get().property('city').value()==~/^Dal\w*/
}.values('city')
```

So in summary it is useful to know about closures and the way you can use them with filters but as stated above - use them sparingly and only when a "pure Gremlin" alternative does not present itself.

NOTE

We could actually go one step further and create a custom predicate (see next section) that handles regular expressions for us.

4.9. Creating custom tests (predicates)

TinkerPop comes with a set of built in methods that can be used for testing values. These methods are commonly referred to as *predicates*. Examples of existing Gremlin predicates include methods like *gte()*, *lte()* and *neq()*. Sometimes, however, it is useful to be able to define your own custom predicate that can be passed in to a *has()*, *where()* or *filter()* step as part of a Gremlin query.

The following example uses the Groovy closure syntax to define a custom predicate, called *f*, that tests the two values passed in to see if *x* is greater than twice *y*. This new predicate can then be used as part of a *has()* step by using it as a parameter to the *test()* method. When *f* is called, it will

be passed two parameters. The first one will be the value returned in response to asking *has()* to return the property called *longest*. The second parameter passed to *f* will be the value of *a* that we provide. This is a simple example, but shows the flexibility that Gremlin provides for extending the basic predicates.

```groovy
// Find the average longest runway length.
a = g.V().hasLabel('airport').values('longest').mean().next()

// Define a custom predicate
f = {x,y -> x > y*2}

// Find airports with runways more than twice the average maximum length.
g.V().hasLabel('airport').has('longest',test(f,a)).values('code')
```

4.9.1. Creating a regular expression predicate

In the previous section we used a closure to filter values using a regular expression. Now that we know how to create our own predicates we could go one step further and create a predicate that accepts regular expressions for us.

```groovy
// Create our method
f = {x,y -> x ==~ y}

// Use it to find any vertices where the description string starts with 'Dal'
g.V().has('desc',test(f,/^Dal.*/)).values('desc')
```

We can actually go one step further and create a custom method called *regex* rather than use the *test* method directly. If the following code seems a bit unclear don't worry too much. It works and that may be all you need to know. However if you want to understand the TinkerPop API in more detail the documentation that can be found on the Apache TinkerPop web page explains things like *P* in detail. Also remember that Gremlin is written in Groovy/Java and we take advantage of that here as well.

In the following example, rather than use *test* directly we use the *BiPredicate* functional interface that is part of Java 8. *BiPredicate* is sometimes referred to was a *two-arity* predicate as it takes two parameters. We will create an implementation of the interface called *bp*. The interface requires that we provide one method called *test* that does the actual comparison between two objects and returns a simple true or false result. Like we did in the previous section we simply perform a regular expression compare using the ==~ operator.

We can then use our *bp* implementation to build a named closure that we will call *regex*. TinkerPop includes a predicate class P that is an implementation of the Java Predicate functional interface. We we can use *P* to build our new *regex* method. We can then pass *regex* directly to steps like *has*.

```groovy
// Create a new BiPredicate that handles regular expression pattern matching
bp = new java.util.function.BiPredicate<String, String>() {
         boolean test(String val, String pattern) {
           return val ==~ pattern  }}

// Create a new closure we can use for regular expression pattern matching.
regex = {new P(bp, it)}

// Use our new closure to find descriptions that start with 'Dal'. As this
// unwinds, the contents of 'desc' are passed to the test method as the first parameter
// and the regex pattern as the second paramter.
g.V().has('desc', regex(/^Dal.*/)).values('desc')
```

4.10. Using graph variables to associate metadata with a graph

TinkerPop 3 introduced the concept of graph variables. A graph variable is a key/value pair that can be associated with the graph itself. 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 I wanted to add some metadata that allowed me to record who the maintainer of the air-routes graph is and when it was last updated. We could do that as follows.

```groovy
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.

```groovy
graph.variables().keys()

updated
maintainer
```

The *asMap* method will return any graph variables that are currently set as a map of key/value pairs.

```groovy
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.

```groovy
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.

```groovy
graph.variables().remove('maintainer')
graph.variables().asMap()

updated=July 18th 2017
```

4.11. Turning node values 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) node 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.

```groovy
//Generate a tree the AUS node and it's neighbors and their neighbors
 t=g.V().has('code','AUS').repeat(out()).times(2).tree().by('code').next()
```

The object returned to our variable *t* 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.

```groovy
// Look at part of the tree directly
t['AUS']['DFW']

// You can also use the TinkerPop Tree API to work with the tree
t.getLeafObjects()
t.getObjectsAtDepth(1)
t.getObjectsAtDepth(1)
t.getObjectsAtDepth(2)
```

We will see the Tree API used again in the "Modelling an ordered binary tree as a graph" section later on.

4.12. Creating a sub graph

You can create a sub graph which is a subset of the vertices and edges in the main graph. Once created, to work with a sub graph, you create a traversal specific to that new graph and distinct from the trversal being used to process the main graph. Subgraphs are created using the *subgraph* traversal step. Note that *subgraph* takes edges (not vertices) as input and returns both those edges and the vertices that they connect with.

```groovy
// Create a subgraph of all vertices and edges directly connected to the AUS node
// Note that using bothE() means we get incoming and outgoing edges.
subg = g.V().has('code','AUS').bothE().subgraph('subGraph').cap('subGraph').next()

// As above but only keep the outgoing edges in the subgraph.
subg = g.V().has('code','AUS').outE().subgraph('subGraph').cap('subGraph').next()

// Create a subgraph starting at AUS but this time going out two levels. We achieve this
// using a repeat() method to go out from Austin 2 times.
subg = g.V().has('code','AUS').repeat(__.bothE().subgraph('subGraph').outV()).times(2)
                              .cap('subGraph').next()

// Get a traversal object so that we can traverse the newly created sub graph
sgt = subg.traversal(standard())

// Which airports ended up in the sub graph?
sgt.V().hasLabel('airport').values('code')
```

In this second example we create a subgraph just of airports and routes that are inside Europe. Effectively we just made 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 a bit more sophisticated that the previous examples in that as well as extracting the routes and airports into the subgraph it also extracts all of the relevant countries as well.

```groovy
// Create a sub graph only of airports in Europe and routes between those airports
subgraph = 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()

// Create a traversal object for the subgraph
sg = subgraph.traversal()

// How many routes are there in the subgraph?
sg.E().hasLabel('route').count()

// How many airports?
sg.V().hasLabel('airport').count()

// Now I can query the subgraph to find out where I can get to from LHR within
// the subgraph.
sg.V().has('code','LHR').out().values('code').join(',')
```

4.13. Saving (serializing) a graph as GraphML (XML) or GraphSON (JSON)

Using TinkerPop 3 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 of TinkerPop enabled tools and graphs. However, whereas GraphML is considered lossee for some graphs (it does not support all of the data types and structures used by TinkerPop 3). GraphSON is not considered lossee.

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 distribued with this document 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.

```groovy
// Save the graph as GraphML
graph.io(graphml()).writeGraph('mygraph.graphml')
```

TinkerPop 3 offers two flavors of JSON. The default flavor 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 afterwards you will see that each line (which could be very wide) is a stand alone JSON obect.

The second flavor is referred to as a *wrapped adjacency list* format. In this flavor all of the vertices and edges are stored in a single large JSON oject inside of an enclosing *vertices* object.

NOTE

The GraphSON file generated will not be easily human readable without doing some pretty printing on it using something like the Python json tool (*python -m json.tool mygraph.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.

```
// Save the graph as unwrapped JSON
graph.io(graphson()).writeGraph("graph.json")
```
GROOVY

The following will create a file containing the *wrapped adjacency list* form of GraphSON.

```
// Create a single (wrapped) JSON file
os = new FileOutputStream("mygraph.json")
GraphSONWriter.build().wrapAdjacencyList(true).create().writeGraph(os,graph)
```
GROOVY

TIP

If you are ingesting large amounts of data into a TinkerPop 3 enables 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.

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 document in the "COMMON GRAPH SERIALIZATION FORMATS" section.

4.14. 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 lets 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 name of the method from the IoCore Tinkerpop 3 class that we use. When we want to load a graphML file we specify *IoCore.graphml()*.

```groovy
// Load a graphML file
graph.io(IoCore.graphml()).readGraph('my-graph.graphml')
```

If we are loading a GraphSON format file we instead specify *IoCore.graphson()*.

```groovy
// Load a grapSON file
graph.io(IoCore.graphson()).readGraph('my-graph.json')
```

5. MISCELLANEOUS QUERIES AND THE RESULTS THEY GENERATE

In this section you will find more Gremlin queries that operate on the air-routes graph. All of these queries build upon the topics already covered in the prior sections. Unlike the prior sections, in this section I have included lots of examples of the output returned by running the query than in the previous sections. In cases where the output is rather lengthy I have either truncated or laid it out in columns to make it easier to read and to save space.

5.1. Counting more things

Here are a few more examples that basically just count occurrences of things but are a little more complex than the examples we looked at earlier in the document.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.

```groovy
// Are there any countries that have no airports?
g.V().hasLabel('country').where(out().count().is(0)).values('desc')
```

Here are the results of running the query.

```groovy
Andorra
Liechtenstein
Monaco
Montserrat
Pitcairn
San Marino
```

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.

```groovy
// What is the distribution of runways in the graph
g.V().hasLabel('airport').groupCount().by('runways')

[1:2288, 2:756, 3:224, 4:51, 5:15, 6:4, 7:1, 8:1]
```

This next query finds all airports that have more than 180 outgoing routes and returns their IATA codes.

```groovy
// 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,ORD,DEN,EWR,LHR,LGW,CDG,FRA,DXB,PEK,PVG,FCO,AMS,BCN,MAD,MUC,DME,IST]
```

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*.

```groovy
// Same basic query but return the airport code and the route count
g.V().where(out('route').count().is(gt(180))).group().by('code').by(out().count())

[ORD:226, DFW:216, FRA:254, CDG:253, PEK:232, AMS:257, ATL:232, MUC:219, IST:259, DME:206,
DXB:225]
```

How many airports are there in each of the UK regions of England, Scotland, Wales and Northern Ireland?

```groovy
g.V().has('country','code','UK').out('contains').groupCount().by('region')

[GB-ENG:27,GB-WLS:3,GB-NIR:3,GB-SCT:21]
```

The next query counts how many airports have more than 400 total routes (inbound and outbound).

```groovy
// Airports with more than 400 total routes
g.V().hasLabel('airport').where(both('route').count().is(gt(400))).values('code')

ATL
ORD
CDG
FRA
PEK
AMS
IST
```

This next query counts how many airports in the graph only have one outgoing route. There are actually a lot of airports that meet this criteria.

```groovy
// How many airports have only one route?
g.V().hasLabel('airport').where(out().count().is(eq(1))).count()
```

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.

```groovy
// How many airports does each country have in the graph?
g.V().hasLabel('airport').groupCount().by('country')

[PR:6, PT:14, PW:1, PY:2, QA:1, AE:9, AF:4, AG:1, AI:1, AL:1, AM:2, AO:14, AR:36, AS:1,
AT:6, RE:2, AU:113, AW:1, AZ:5, RO:13, BA:4, BB:1, RS:2, BD:7, BE:5, RU:115, 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:115, SC:2, BS:18,
SD:5, SE:39, BT:1, SG:1, BW:4, SI:1, BY:1, BZ:13, SK:2, SL:1, SN:3, SO:5, CA:203, SR:1,
SS:1, CC:1, CD:11, ST:1, SV:1, CF:1, CG:3, CH:5, SX:1, CI:1, SY:1, SZ:1, CK:6, CL:16,
CM:5, CN:179, CO:50, CR:13, TC:4, TD:1, CU:10, CV:7, TG:1, TH:32, CW:1, CX:1, CY:3, TJ:4,
CZ:5, TL:1, TM:1, TN:8, TO:1, TR:44, TT:2, DE:32, TV:1, TW:9, TZ:8, DJ:1, DK:8, DM:1,
DO:7, UA:13, UG:4, UK:52, DZ:29, US:566, EC:15, EE:3, EG:10, EH:2, UY:2, UZ:11, ER:1,
VC:1, ES:41, ET:14, VE:24, VG:2, VI:2, VN:21, VU:26, FI:19, FJ:10, FM:4, FO:1, FR:54,
WF:2, GA:2, WS:1, GD:1, GE:3, GF:1, GG:2, GH:5, 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:67, YT:1, IE:7,
IL:4, IM:1, IN:72, ZA:20, IQ:6, IR:44, IS:5, IT:36, ZM:8, JE:1, ZW:3, JM:2, JO:2, JP:63,
KE:14, KG:2, KH:3, KI:2, KM:1, KN:2, KP:1, KR:15, KS:1, KW:1, KY:3, KZ:18, LA:8, LB:1,
LC:2, LK:6, LR:2, LS:1, LT:3, LU:1, LV:1, LY:10, MA:14, MD:1, ME:2, MF:1, MG:13, MH:2,
MK:1, ML:1, MM:14, MN:10, MO:1, MP:2, MQ:1, MR:3, MT:1, MU:2, MV:8, MW:2, MX:57, MY:34,
MZ:10, NA:4, NC:1, NE:1, NF:1, NG:18, NI:1, NL:5, NO:49, NP:10, NR:1, NZ:25, OM:3, PA:4,
PE:20, PF:30, PG:26, PH:37, PK:21, PL:12, PM:1]
```

Because the air-routes graph also has country specific vertices, we could chose to write the previous query a different way. We could start by finding country vertices and then see how many airport vertices each one is connected to.

```groovy
// Another way to ask the question above, this time by counting the edges (degree) from each country
g.V().hasLabel('country').group().by('code').by(outE().count())
```

If we wanted to sort the list in descending order we could adjust the query as follows. Note the use of *local* to specify how the ordering is applied. If we did not use *local* the ordering would not have the desired effect.

```groovy
g.V().hasLabel('airport').groupCount().by('country').order(local).by(valueDecr)
```

```
[US:578, CA:203, CN:200, AU:118, RU:118, BR:115, IN:72, ID:67, JP:63, MX:58, FR:56, UK:54,
CO:50, NO:49, TR:48, IR:44, ES:41, SE:39, GR:39, PH:38, AR:36, IT:36, MY:34, TH:32, DE:32,
PF:30, DZ:29, SA:26, VU:26, PG:26, NZ:25, VE:24, VN:21, PK:21, FI:20, ZA:20, PE:20, BS:18,
KZ:18, NG:18, SB:17, CL:17, BO:15, EC:15, KR:15, PT:14, AO:14, UA:14, ET:14, GL:14, KE:14,
MA:14, MM:14, RO:13, BZ:13, CR:13, MG:13, PL:13, CU:12, CD:11, UZ:11, AE:10, EG:10, FJ:10,
LY:10, MN:10, MZ:10, NP:10, TW:9, YE:9, TN:8, TZ:8, DK:8, HR:8, ZM:8, LA:8, MV:8, BD:7,
CV:7, DO:7, IE:7, PR:6, AT:6, CK:6, HN:6, IQ:6, LK:6, AZ:5, BE:5, SD:5, SO:5, CH:5, CM:5,
CZ:5, GH:5, IL:5, IS:5, NL:5, PA:5, AF:4, BA:4, BG:4, BW:4, TC:4, TJ:4, UG:4, FM:4, NA:4,
BQ:3, SN:3, CG:3, CY:3, EE:3, GE:3, ZW:3, KH:3, KY:3, LT:3, MR:3, OM:3, PY:2, AM:2, RE:2,
RS:2, BF:2, RW:2, SC:2, SH:2, SK: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,
MP:2, MU:2, MW: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, SG:1, SI:1, BY:1, SL:1, SR:1, SS:1, CC:1, ST:1, SV:1, CF:1, SX:1, CI:1,
SZ:1, TD:1, TG:1, CW:1, CX:1, TL:1, TM:1, TO: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, MK: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 *keyIncr* to get a sort in ascending order. If we wanted to dort in descending order by key we could use *keyDecr* instead.

```groovy
g.V().hasLabel('airport').groupCount().by('country').order(local).by(keyIncr)

[AE:10, AF:4, AG:1, AI:1, AL:1, AM:2, AO:14, AR:36, AS:1, AT:6, AU:118, 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:115,
BS:18, BT:1, BW:4, BY:1, BZ:13, CA:203, CC:1, CD:11, CF:1, CG:3, CH:5, CI:1, CK:6, CL:17,
CM:5, CN:200, CO:50, CR:13, CU:12, CV:7, CW:1, CX:1, CY:3, CZ:5, DE:32, DJ:1, DK:8, DM:1,
DO:7, DZ:29, EC:15, EE:3, EG:10, EH:2, ER:1, ES:41, ET:14, FI:20, FJ:10, FK:2, FM:4, FO:1,
FR:56, GA:2, GD:1, GE:3, GF:1, GG:2, GH:5, 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:67, IE:7, IL:5, IM:1, IN:72,
IQ:6, IR:44, IS:5, IT:36, JE:1, JM:2, JO:2, JP:63, KE:14, KG:2, KH:3, KI:2, KM:1, KN:2,
KP:1, KR:15, KS:1, KW:1, KY:3, KZ:18, LA:8, LB:1, LC:2, LK:6, LR:2, LS:1, LT:3, LU:1,
LV:1, LY:10, MA:14, MD:1, ME:2, MF:1, MG:13, MH:2, MK:1, ML:1, MM:14, MN:10, MO:1, MP:2,
MQ:1, MR:3, MS:1, MT:1, MU:2, MV:8, MW:2, MX:58, MY:34, MZ:10, NA:4, NC:1, NE:1, NF:1,
NG:18, NI:1, NL:5, NO:49, NP:10, NR:1, NZ:25, OM:3, PA:5, PE:20, PF:30, PG:26, PH:38,
PK:21, PL:13, PM:1, PR:6, PT:14, PW:1, PY:2, QA:1, RE:2, RO:13, RS:2, RU:118, RW:2, SA:26,
SB:17, SC:2, SD:5, SE:39, SG:1, SH:2, SI:1, SK:2, SL:1, SN:3, SO:5, SR:1, SS:1, ST:1,
SV:1, SX:1, SY:2, SZ:1, TC:4, TD:1, TG:1, TH:32, TJ:4, TL:1, TM:1, TN:8, TO:1, TR:48,
TT:2, TV:1, TW:9, TZ:8, UA:14, UG:4, UK:54, US:578, UY:2, UZ:11, VC:1, VE:24, VG:2, VI:2,
VN:21, VU:26, WF:2, WS:1, YE:9, YT:1, ZA:20, ZM:8, 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.

```groovy
// Only return the values for Germany, China, Holland and the US.
g.V().hasLabel('airport').groupCount().by('country').select('DE','CN','NL','US')

[DE:32, CN:179, NL:5, US:566]
```

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!

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

[EU:561, AS:888, NA:959, OC:274, AF:292, AN:0, SA:300]
```

This next 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 in the graph, this query will produce quite a lot of results so I did not try to include them all here. The second query just picks the results from the map for a few airports. Those results are shown.

```groovy
// 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','ME
X')

[AUS:46,AMS:249,JFK:168,DUB:150,MEX:95]
```

This next query essentially asks the same question about how many outgoing routes each airport has. However rather than return the count for each airports individually it groups the ones with the same number of routes together. As this query returns a lot of data I just included a few lines from the full result below the query.

```groovy
// Same query except sorted into groups by ascending count
g.V().hasLabel('airport').group().by(out().count()).by('code')

77:[MNL,MLA,SXF],79:[BWI,TLV],80:[BUD,CUN],81:[BOM,RUH],82:[CAI,STR,LPA,LYS,SHJ,BGY], 83:
[WAW],84:[JNB,CRL],85:[PHX,MRS,TFS],86:[SYD,LTN,CMN,BHX],87:[SLC,YVR],88:[SZX], 89:
[DCA,CKG,XIY],90:[HEL,EDI,GRU,HAM],92:[NCE],93:[AUH],94:[SEA],95:[CGN,MEX,KMG], 97:
[YUL],98:[ALC,JED],99:[DEL],100:[PRG,SAW],102:[MCO,GVA],103:[LIS,TPE],105:[FLL,OSL], 106:
[BOS,ATH],110:[NRT,TXL,CTU],111:[SFO],112:[KUL,MXP],116:[LED],117:[AGP],120:[ORY], 122:
[PHL,BKK,CPH],125:[ARN],126:[IAD],127:[DOH],129:[PMI],132:[MSP,SIN],133:[DTW], 134:
[LAS],136:[ICN],140:[VIE],142:[HKG,ZRH,CLT],145:[MIA],147:[SVO],148:[BRU,MAN,DUS], 150:
[DUB,CAN],155:[YYZ],156:[STN],157:[PVG],159:[EWR],160:[LAX],163:[FCO],164:[BCN], 166:
[MAD],167:[DEN],169:[JFK],170:[IAH],171:[LGW],174:[LHR],187:[DXB],189:[DME], 195:
[DFW],196:[MUC],207:[ORD],210:[PEK],218:[ATL],229:[IST],236:[CDG],240:[FRA,AMS]
```

As the above query returns a lot of data, we can also extract specific values we are interested in as follows:

```groovy
// Which of these airports have 110 outgoing routes?
g.V().hasLabel('airport').group().by(out().count()).by('code').next().get(110L)

NRT
TXL
CTU
```

NOTE

Currently *select* can only take a string value as the key so we have to use the slightly awkward *next().get()* syntax to get a numeric key from a result.

5.1.1. 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 actually 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 number of 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.

```groovy
g.V(3).out().groupCount('a').by('runways').
       out().count().as('b').select('a','b')

[a:[1:2,2:9,3:17,4:22,5:4,6:3,7:1,8:1],b:5913]
```

5.1.2. Analysis of routes between Europe and the USA

The next few queries show how you can use a graph like *air-routes* do 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 of all let's just find out how many total routes there are between airports anywhere in Europe and airports in the USA.

```groovy
// How many routes from anywhere in Europe to the USA?
g.V().has('continent','code','EU').out().out().has('country','US').count()

345
```

So we now know that there are 345 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 345 and find out how many US airports have flights that arrive from Europe.

```groovy
// How many different US airports have routes from Europe?
g.V().has('continent','code','EU').out().out().has('country','US').dedup().count()

38
```

So we can now see that the 345 routes from European airports arrive at one of 38 airports in the United States. We can dig a bit deeper and look at the distribution of these routes across the 38 airports. John F. Kennedy airport (JFK) in New York appears to have the most routes from Europe with Newark (EWR) having the second most.

```groovy
//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(valueIncr)

[PHX:1,CVG:1,RSW:1,BDL:2,SJC:2,BWI:2,AUS:2,RDU:2,MSY:2,SAN:3,TPA:3,SLC:3,PDX:3,PIT:3,SFB:4
,DEN:4,OAK:4,DTW:5,FLL:5,SWF:5,MSP:5,CLT:7,DFW:7,SEA:7,PVD:7,IAH:8,MCO:9,LAS:10,ATL:14,SFO
:15,PHL:17,IAD:19,ORD:21,BOS:22,LAX:23,MIA:24,EWR:33,JFK:40]
```

Now let's repeat the process but looking at the European end of the routes. First of all, we can calculate how many European airports have flights to the United States.

```groovy
// 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()

53
```

Just as we did for the airports in the US we can figure out the distribution of routes for the European airports. It appears that London Heathrow (LHR) offers the most US destinations and Frankfurt (FRA) the second most.

```groovy
//What is the distribution of these 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(valueIncr)

[RIX:1,TER:1,BRS:1,STN:1,NCE:1,KRK:1,ORK:1,KBP:1,PDL:1,DME:1,BEG:1,AGP:1,HAM:1,OPO:2,STR:2
,VCE:2,BHX:2,ORY:2,ATH:2,MXP:3,BFS:3,BGO:3,GVA:3,VKO:3,HEL:3,WAW:4,SVO:4,GLA:4,EDI:5,SNN:5
,CGN:5,TXL:6,VIE:6,LIS:6,BRU:7,ARN:7,OSL:8,BCN:9,LGW:9,IST:9,MAD:11,DUS:11,CPH:11,FCO:12,Z
RH:13,MAN:13,DUB:14,MUC:16,KEF:17,AMS:18,CDG:21,FRA:24,LHR:27]
```

Lastly, we can find out what the list of routes flown is. For this example we just return 10 of the 345 routes. Note how the *path* step returns all parts of the traversal including the continent code *EU*.

```
// What are some of these routes?
 g.V().has('continent','code','EU').out().out().has('country','US').path().by('code').limi
t(10)

[EU,WAW,JFK]
[EU,WAW,LAX]
[EU,WAW,ORD]
[EU,WAW,EWR]
[EU,BEG,JFK]
[EU,IST,ATL]
[EU,IST,BOS]
[EU,IST,IAD]
[EU,IST,IAH]
[EU,IST,JFK]
```
GROOVY

5.1.3. Using *fold* to do simple Map-Reduce computations

Earlier in the document 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 of all, here is a query that uses *fold* in a way that we have already seen. We find all routes from Austin and use *fold* to return a nice list of those names.

```
g.V().has('code','AUS').out('route').values('city').fold()

[Harlingen,Guadalajara,Orlando,Atlanta,Los Angeles,Orlando, New York,
Nashville,Boston,Baltimore,Washington D.C.,Dallas, Ft. Lauderdale, Washington
D.C.,Houston,Toronto,London,Frankfurt, Mexico City,Pittsburgh, San Diego,Long Beach,Santa
Ana,Miami,Minneapolis,Chicago,Phoenix,Seattle, San Francisco,San Jose, Tampa,St
Louis,Albuquerque,Chicago,Lubbock,Cleveland,Oakland,Philadelphia,Detroit,Portland,Charlott
e,Cancun,Memphis,Cincinnati, Dallas,Salt Lake City,Las Vegas,Denver,New
Orleans,Newark,Houston, El Paso]
```
GROOVY

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, *fold* is used to reduce the names to a value. That value being the total number of characters in all of those city names. We have seen *fold* used elsewhere in the document 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 to a running total.

```
g.V().has('code','AUS').out('route').values('city').fold(0) {a,b -> a + b.length()}

449
```
GROOVY

NOTE

While this query will work as-is on TinkerGraph within the Gremlin Console, some graph systems are more strict 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.

```groovy
g.V().has('code','AUS').out('route').values('city').
                        fold(0) {a,b -> (int)a + ((String)b).length()}
```

5.1.4. 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 my 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 odered *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.

```groovy
g.V().hasLabel('airport').groupCount().by(out('route').count()).order(local).by(valueDecr)
```

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 226 outgoing routes.

```
GROOVY
[1:786, 2:654, 3:361, 4:244, 5:149, 6:110, 7:86, 8:80, 9:67, 10:54, 11:45, 12:40, 13:30,
19:30, 14:28, 15:28, 17:23, 23:19, 20:18, 16:17, 18:17, 0:16, 22:15, 21:13, 29:13, 26:12,
30:12, 35:11, 39:11, 24:10, 32:10, 42:10, 25:9, 33:9, 49:9, 28:8, 31:8, 34:8, 37:8, 27:7,
38:7, 41:7, 58:7, 60:7, 36:6, 50:6, 54:6, 59:6, 40:5, 47:5, 62:5, 43:4, 44:4, 51:4, 52:4,
53:4, 56:4, 57:4, 64:4, 65:4, 66:4, 83:4, 84:4, 92:4, 100:4, 46:3, 69:3, 72:3, 76:3, 78:3,
81:3, 86:3, 97:3, 103:3, 134:3, 142:3, 45:2, 48:2, 55:2, 63:2, 67:2, 70:2, 74:2, 77:2,
82:2, 85:2, 91:2, 93:2, 99:2, 108:2, 115:2, 119:2, 128:2, 174:2, 176:2, 183:2, 191:2,
232:2, 254:2, 257:1, 259:1, 61:1, 71:1, 75:1, 79:1, 80:1, 87:1, 88:1, 89:1, 94:1, 95:1,
96:1, 101:1, 102:1, 104:1, 105:1, 106:1, 109:1, 110:1, 111:1, 112:1, 117:1, 118:1, 121:1,
123:1, 125:1, 127:1, 129:1, 132:1, 140:1, 141:1, 146:1, 148:1, 149:1, 153:1, 154:1, 155:1,
157:1, 158:1, 159:1, 166:1, 172:1, 178:1, 182:1, 184:1, 186:1, 197:1, 206:1, 216:1, 219:1,
225:1, 226:1]
```

We could change our query above, replacing *out()* with *__.in()* and we could find out the distribution of incoming routes. Remember, 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.

```
GROOVY
g.V().hasLabel('airport').groupCount().by(out('route').count()).order(local).by(keyIncr)
```

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!

```
GROOVY
[0:16, 1:786, 2:654, 3:361, 4:244, 5:149, 6:110, 7:86, 8:80, 9:67, 10:54, 11:45, 12:40,
13:30, 14:28, 15:28, 16:17, 17:23, 18:17, 19:30, 20:18, 21:13, 22:15, 23:19, 24:10, 25:9,
26:12, 27:7, 28:8, 29:13, 30:12, 31:8, 32:10, 33:9, 34:8, 35:11, 36:6, 37:8, 38:7, 39:11,
40:5, 41:7, 42:10, 43:4, 44:4, 45:2, 46:3, 47:5, 48:2, 49:9, 50:6, 51:4, 52:4, 53:4, 54:6,
55:2, 56:4, 57:4, 58:7, 59:6, 60:7, 61:1, 62:5, 63:2, 64:4, 65:4, 66:4, 67:2, 69:3, 70:2,
71:1, 72:3, 74:2, 75:1, 76:3, 77:2, 78:3, 79:1, 80:1, 81:3, 82:2, 83:4, 84:4, 85:2, 86:3,
87:1, 88:1, 89:1, 91:2, 92:4, 93:2, 94:1, 95:1, 96:1, 97:3, 99:2, 100:4, 101:1, 102:1,
103:3, 104:1, 105:1, 106:1, 108:2, 109:1, 110:1, 111:1, 112:1, 115:2, 117:1, 118:1, 119:2,
121:1, 123:1, 125:1, 127:1, 128:2, 129:1, 132:1, 134:3, 140:1, 141:1, 142:3, 146:1, 148:1,
149:1, 153:1, 154:1, 155:1, 157:1, 158:1, 159:1, 166:1, 172:1, 174:2, 176:2, 178:1, 182:1,
183:2, 184:1, 186:1, 191:2, 197:1, 206:1, 216:1, 219:1, 225:1, 226:1, 232:2, 254:2, 257:1,
259:1]
```

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.

```groovy
g.V().hasLabel('airport').union(count(),out('route').count()).fold()

[3340,40662]
```

We could then use the Gremlin Console do the division for us to calculate the mean.

```groovy
gremlin> 40662/3340
==>12.1742514970
```

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 40662/1. 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). I 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 of the interim values instead!

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

12.174251497005988
```

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.

```groovy
// Average number of incoming routes
g.V().has('type','airport').local(__.in('route').count()).mean()

12.174251497005988

// Average number of outgoing and incoming routes
g.V().has('type','airport').local(both('route').count()).mean()

24.348502994011977
```

5.1.5. 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.

```groovy
g.V().has('region','GB-ENG').has('city','London').group().by('code').by(out().count())

[LCY:37, LHR:176, LTN:91, STN:162, LGW:179]
```

Here is a twist on the above theme. How many places can I 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.

```groovy
// 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')).coun
t()

2082
```

We could have written the previous query like this and avoided using *aggregate* but to me, this feels more clumsy and somewhat repetetive.

```groovy
 g.V().has('region','GB-ENG').has('city','London').out().out().dedup()
                             .not(and(has('city','London'),has('region','GB-
ENG'))).count()

2082
```

5.1.6. What are the top ten airports by route count?

These next three queries produce a table of the top ten airports in terms of incoming, outgoing and overall routes. Note that 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 out bound routes to these airports.

First of all, this query will find the ten airports with the most incoming routes, sorted by descending order.

```groovy
// Find the top ten overall in terms of incoming routes
g.V().hasLabel('airport').order().by(__.in('route').count(),decr).limit(10)
                         .project('ap','routes').by('code').by(__.in('route').count())
[ap:AMS, routes:249]
[ap:FRA, routes:243]
[ap:CDG, routes:239]
[ap:IST, routes:234]
[ap:ATL, routes:221]
[ap:PEK, routes:213]
[ap:ORD, routes:212]
[ap:MUC, routes:199]
[ap:DFW, routes:198]
[ap:DME, routes:190]
```

Now let's do the same thing but for outgoing routes.

```groovy
// Find the top ten overall in terms of outgoing routes
g.V().hasLabel('airport').order().by(out('routes').count(),decr).limit(10)
                         .project('ap','routes').by('code').by(out('routes').count())

[ap:FRA, routes:243]
[ap:AMS, routes:240]
[ap:CDG, routes:239]
[ap:IST, routes:232]
[ap:ATL, routes:221]
[ap:PEK, routes:213]
[ap:ORD, routes:212]
[ap:MUC, routes:199]
[ap:DFW, routes:198]
[ap:DME, routes:190]
```

Lastly, let's find the top ten airports ordered by the total number of incoming and outgoing routes that they have.

```groovy
// Find the top ten overall in terms of total routes
g.V().hasLabel('airport').order().by(both('route').count(),decr).limit(10).project('ap','routes').by('code').by(both('route').count())

[ap:AMS, routes:489]
[ap:FRA, routes:486]
[ap:CDG, routes:478]
[ap:IST, routes:466]
[ap:ATL, routes:442]
[ap:PEK, routes:426]
[ap:ORD, routes:424]
[ap:MUC, routes:398]
[ap:DFW, routes:396]
[ap:DME, routes:380]
```

5.1.7. Using *local* while counting things

Earlier in this document we saw examples of *local* being used. Here is another example of how *local* 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 airports IATA code along with the number of runways.

```groovy
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.

```groovy
[BOS,6,DFW,7,ORD,8,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 it's 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.

```groovy
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 it's own individual list.

```
[BOS,6]
[DFW,7]
[ORD,8]
[DEN,6]
[DTW,6]
[AMS,6]
```

5.2. Where can I fly to from here?

Here are some more examples of queries that explore different questions along the lines of "Where can I fly to from here?".

```
// 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')

[IN,BOM,EWR]
[IN,DEL,SFO]
[IN,DEL,EWR]
[IN,DEL,JFK]
[IN,DEL,ORD]
[IN,DEL,YVR]
[IN,DEL,YYZ]
```

Which cities in Europe can I fly to from Ft; Lauderdale in Florida?

```
// 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')

London
Paris
Oslo
Stockholm
Copenhagen
```

Where can I fly to 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')

LHR     FCO
CDG     MAD
FRA     MUC
GIG     DUB
GRU
```

Where in the United States can I fly from to any one of the five airports in the London area in the UK?

```
// 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')

[LHR,AUS]       [LHR,SEA]       [LHR,EWR]
[LHR,ATL]       [LHR,RDU]       [LHR,DEN]
[LHR,BWI]       [LHR,SJC]       [LHR,DTW]
[LHR,BOS]       [LHR,SFO]       [LHR,PHL]
[LHR,IAD]       [LHR,LAX]       [LGW,SFO]
[LHR,DFW]       [LHR,JFK]       [LGW,LAS]
[LHR,MSP]       [LHR,IAH]       [LGW,TPA]
[LHR,MIA]       [LHR,LAS]       [LGW,FLL]
[LHR,PHX]       [LHR,CLT]       [LGW,MCO]
[LHR,ORD]       [LHR,SAN]       [LGW,JFK]
```

Where in New York state can I fly to from any of the airports in Texas?

```
// 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')

[AUS,JFK]       [IAH,EWR]
[AUS,EWR]       [SAT,EWR]
[DFW,EWR]       [SAT,JFK]
[DFW,JFK]       [HOU,JFK]
[DFW,LGA]       [HOU,LGA]
[IAH,JFK]       [HOU,EWR]
[IAH,LGA]
```

Which cities in Mexico can I fly to from Denver?

```groovy
// Where in Mexico can I fly to from Denver?
g.V().has('code','DEN').out().has('country','MX').values('city')

Puerto Vallarta
San Josa del Cabo
Cozumel
Mexico City
Cancun
```

Which cities in Europe can I fly to from Delhi in India?

```groovy
// 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
')

Istanbul
Helsinki
Paris
Frankfurt
London
Birmingham
Vienna
Zurich
Amsterdam
Moscow
Brussels
Munich
Rome
```

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.

```groovy
g.V().has('city',within('London','Munich','Paris')).aggregate('a').out()  \
     .where(within('a')).path().by('code')

[LHR,MUC]   [CDG,LHR]   [MUC,STN]   [STN,MUC]
[LHR,ORY]   [CDG,LGW]   [MUC,LHR]   [ORY,LCY]
[LHR,CDG]   [CDG,MUC]   [MUC,LGW]   [ORY,MUC]
[LGW,MUC]   [CDG,LCY]   [MUC,CDG]   [ORY,LHR]
[LGW,CDG]   [MUC,LTN]   [LCY,CDG]   [LTN,CDG]
[CDG,LTN]   [MUC,ORY]   [LCY,ORY]   [LTN,MUC]
```

5.3. More analysis of distances between airports

In this section you will find some more queries that examine distances between airports.

This query returns a nice list of all the routes from Austin (AUS) along with their distances. The results are sorted in ascending order by distance. For ease of reading I broke the results into four columns.

```groovy
// Distances of all routes from AUS along with destination IATA CODE
g.V().has('code','AUS').outE().order().by('dist',incr).inV().path().by('code').by('dist')

[AUS,142,IAH]      [AUS,755,GDL]      [AUS,1080,LAS]      [AUS,1430,PHL]
[AUS,152,HOU]      [AUS,755,BNA]      [AUS,1080,SLC]      [AUS,1476,SJC]
[AUS,183,DFW]      [AUS,768,DEN]      [AUS,1110,FLL]      [AUS,1493,OAK]
[AUS,189,DAL]      [AUS,809,ATL]      [AUS,1140,DTW]      [AUS,1500,SFO]
[AUS,274,HRL]      [AUS,866,PHX]      [AUS,1160,SAN]      [AUS,1500,EWR]
[AUS,341,LBB]      [AUS,922,CUN]      [AUS,1173,CLE]      [AUS,1520,JFK]
[AUS,444,MSY]      [AUS,925,TPA]      [AUS,1220,LGB]      [AUS,1690,BOS]
[AUS,527,ELP]      [AUS,972,MDW]      [AUS,1230,LAX]      [AUS,1712,PDX]
[AUS,558,MEM]      [AUS,973,ORD]      [AUS,1294,IAD]      [AUS,1768,SEA]
[AUS,618,ABQ]      [AUS,994,MCO]      [AUS,1313,DCA]      [AUS,4901,LHR]
[AUS,722,STL]      [AUS,1030,CLT]     [AUS,1339,BWI]      [AUS,5294,FRA]
[AUS,748,MEX]      [AUS,1040,MSP]     [AUS,1357,YYZ]
```

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 node, the edge and the destination node respectively. Also note that only two were specified but three values are returned. This works because *by* is processed in a round robbin fashion if there are more values than *by* modulators.

```groovy
// 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')

[DFW,8105,HKG]    [DFW,8022,DXB]
[DFW,6951,PEK]    [DFW,6822,ICN]
[DFW,7332,PVG]    [DFW,5228,GIG]
[DFW,4905,AMS]    [DFW,5119,GRU]
[DFW,4950,MAD]    [DFW,5299,EZE]
[DFW,4736,LHR]    [DFW,4884,SCL]
[DFW,4933,CDG]    [DFW,7914,DOH]
[DFW,5127,FRA]    [DFW,8053,AUH]
[DFW,6410,NRT]    [DFW,5015,DUS]
[DFW,8574,SYD]
```

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 I 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.

```groovy
// 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')

[LGW, 5287, MalT]              [LGW, 4380, Calgary]
[LGW, 4618, Varadero]          [LGW, 5987, Cape Town]
[LGW, 5147, Tianjin]           [LGW, 4680, Kingston]
[LGW, 5303, Chongqing]         [LGW, 4953, Cancun]
[LGW, 4410, Ft. Lauderdale]    [LGW, 5399, Colombo]
[LGW, 5463, Los Angeles]       [LGW, 4197, Bridgetown]
[LGW, 4341, Orlando]           [LGW, 4076, St. George]
[LGW, 5374, San Francisco]     [LGW, 4408, Port of Spain]
[LGW, 4416, Tampa]             [LGW, 4699, Montego Bay]
[LGW, 5236, Las Vegas]         [LGW, 4283, Punta Cana]
[LGW, 5364, Oakland]           [LGW, 5419, San Jose]
[LGW, 4731, Vancouver]         [LGW, 4662, Havana]
[LGW, 5982, Hong Kong]         [LGW, 6053, Port Louis]
[LGW, 5070, Beijing]           [LGW, 4222, Vieux Fort]
```

This 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 of all 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 end 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 I 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 I show *sort* being used here but in most cases we can re-write our query to use *order*. You will find several examples of *order* being used throughout this document. Order is introduced in the "Sorting things" section.

```
// 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}


[e:4736, c:LHR]     [e:6410, c:NRT]
[e:4884, c:SCL]     [e:6822, c:ICN]
[e:4905, c:AMS]     [e:6951, c:PEK]
[e:4933, c:CDG]     [e:7332, c:PVG]
[e:4950, c:MAD]     [e:7914, c:DOH]
[e:5015, c:DUS]     [e:8022, c:DXB]
[e:5119, c:GRU]     [e:8053, c:AUH]
[e:5127, c:FRA]     [e:8105, c:HKG]
[e:5228, c:GIG]     [e:8574, c:SYD]
[e:5299, c:EZE]
```

5.3.1. Finding routes longer than 8,000 miles

This next set of queries show 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 of all let's just find all routes longer than 8,000 miles. This will includes routes in both directions between airport pairs.

```
// All outes longer than 8,000 miles
g.V().as('src').outE('route').has('dist',gt(8000)).inV().as('dest').select('src','dest').b
y('code')


[src:ATL,dest:JNB]   [src:SIN,dest:SFO]
[src:DFW,dest:SYD]   [src:DXB,dest:DFW]
[src:DFW,dest:DXB]   [src:DXB,dest:IAH]
[src:DFW,dest:HKG]   [src:DXB,dest:LAX]
[src:DFW,dest:AUH]   [src:DXB,dest:SFO]
[src:IAH,dest:DXB]   [src:DXB,dest:AKL]
[src:IAH,dest:DOH]   [src:HKG,dest:DFW]
[src:JFK,dest:HKG]   [src:HKG,dest:JFK]
[src:LAX,dest:DXB]   [src:HKG,dest:EWR]
[src:LAX,dest:DOH]   [src:AKL,dest:DXB]
[src:LAX,dest:AUH]   [src:DOH,dest:IAH]
[src:LAX,dest:JED]   [src:DOH,dest:LAX]
[src:LAX,dest:RUH]   [src:JNB,dest:ATL]
[src:SFO,dest:SIN]   [src:AUH,dest:DFW]
[src:SFO,dest:DXB]   [src:AUH,dest:LAX]
[src:SFO,dest:AUH]   [src:AUH,dest:SFO]
[src:EWR,dest:HKG]   [src:JED,dest:LAX]
[src:SYD,dest:DFW]   [src:RUH,dest:LAX]
```

Now let's improve the query by including the distance of each route in the query results.

```groovy
// 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')
```

```
[src:ATL,e:8434,dest:JNB]     [src:SIN,e:8433,dest:SFO]
[src:DFW,e:8574,dest:SYD]     [src:DXB,e:8022,dest:DFW]
[src:DFW,e:8022,dest:DXB]     [src:DXB,e:8150,dest:IAH]
[src:DFW,e:8105,dest:HKG]     [src:DXB,e:8321,dest:LAX]
[src:DFW,e:8053,dest:AUH]     [src:DXB,e:8085,dest:SFO]
[src:IAH,e:8150,dest:DXB]     [src:DXB,e:8818,dest:AKL]
[src:IAH,e:8030,dest:DOH]     [src:HKG,e:8105,dest:DFW]
[src:JFK,e:8054,dest:HKG]     [src:HKG,e:8054,dest:JFK]
[src:LAX,e:8321,dest:DXB]     [src:HKG,e:8047,dest:EWR]
[src:LAX,e:8287,dest:DOH]     [src:AKL,e:8818,dest:DXB]
[src:LAX,e:8372,dest:AUH]     [src:DOH,e:8030,dest:IAH]
[src:LAX,e:8314,dest:JED]     [src:DOH,e:8287,dest:LAX]
[src:LAX,e:8246,dest:RUH]     [src:JNB,e:8434,dest:ATL]
[src:SFO,e:8433,dest:SIN]     [src:AUH,e:8053,dest:DFW]
[src:SFO,e:8085,dest:DXB]     [src:AUH,e:8372,dest:LAX]
[src:SFO,e:8139,dest:AUH]     [src:AUH,e:8139,dest:SFO]
[src:EWR,e:8047,dest:HKG]     [src:JED,e:8314,dest:LAX]
[src:SYD,e:8574,dest:DFW]     [src:RUH,e:8246,dest:LAX]
```

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.

```groovy
// Note that this also changes the way the result is returned
g.V().outE().has('dist',gt(8000)).inV().path().by('code').by('dist')
```

```
[ATL,8434,JNB]   [LAX,8287,DOH]   [SIN,8433,SFO]   [AKL,8818,DXB]
[DFW,8574,SYD]   [LAX,8372,AUH]   [DXB,8022,DFW]   [DOH,8030,IAH]
[DFW,8022,DXB]   [LAX,8314,JED]   [DXB,8150,IAH]   [DOH,8287,LAX]
[DFW,8105,HKG]   [LAX,8246,RUH]   [DXB,8321,LAX]   [JNB,8434,ATL]
[DFW,8053,AUH]   [SFO,8433,SIN]   [DXB,8085,SFO]   [AUH,8053,DFW]
[IAH,8150,DXB]   [SFO,8085,DXB]   [DXB,8818,AKL]   [AUH,8372,LAX]
[IAH,8030,DOH]   [SFO,8139,AUH]   [HKG,8105,DFW]   [AUH,8139,SFO]
[JFK,8054,HKG]   [EWR,8047,HKG]   [HKG,8054,JFK]   [JED,8314,LAX]
[LAX,8321,DXB]   [SYD,8574,DFW]   [HKG,8047,EWR]   [RUH,8246,LAX]
```

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.

```groovy
// 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')

[ATL,8434,JNB]
[DFW,8574,SYD]
[DFW,8022,DXB]
[DFW,8105,HKG]
[LAX,8246,RUH]
[SFO,8433,SIN]
[EWR,8047,HKG]
[DXB,8150,IAH]
[DXB,8321,LAX]
[DXB,8085,SFO]
[HKG,8054,JFK]
[AKL,8818,DXB]
[DOH,8030,IAH]
[DOH,8287,LAX]
[AUH,8053,DFW]
[AUH,8372,LAX]
[AUH,8139,SFO]
[JED,8314,LAX]
```

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.

```groovy
// As above but sorted by route lengths
g.V().as('a').outE().has('dist',gt(8000)).order().by('dist',decr).inV().as('b') \
     .filter(select('a','b').by('code').where('a', lt('b'))).path().by('code').by('dist')

[AKL,8818,DXB]
[DFW,8574,SYD]
[ATL,8434,JNB]
[SFO,8433,SIN]
[AUH,8372,LAX]
[DXB,8321,LAX]
[JED,8314,LAX]
[DOH,8287,LAX]
[LAX,8246,RUH]
[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]
```

TinkerPop 3.2.3 added additional capability to the *where* step that allows it to be followed by a *by* modulator. This allows us, should we so desire, to simplify our query a bit more as follows.

```groovy
//Query changed to take advantage of the where().by() construct
g.V().as('s').outE().has('dist',gt(8000)).order().by('dist',decr).inV().as('f')
                            .where('f',lt('s')).by('code').path().by('code').by('dist')
[DXB,8818,AKL]
[SYD,8574,DFW]
[JNB,8434,ATL]
[SIN,8433,SFO]
[LAX,8372,AUH]
[LAX,8321,DXB]
[LAX,8314,JED]
[LAX,8287,DOH]
[RUH,8246,LAX]
[IAH,8150,DXB]
[SFO,8139,AUH]
[HKG,8105,DFW]
[SFO,8085,DXB]
[JFK,8054,HKG]
[DFW,8053,AUH]
[HKG,8047,EWR]
[IAH,8030,DOH]
[DXB,8022,DFW]
```

As an interesting sidenote, 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. So for example, in the prior query the first result was *[DXB,8818,AKL]*. As you will see below, our first result is now *[AKL,8818,DXB]*.

```groovy
g.V().as('s').outE().has('dist',gt(8000)).order().by('dist',decr).inV().as('f') \
                    .where('f',gt('s')).by('code').path().by('code').by('dist')
```

```
[AKL,8818,DXB]
[DFW,8574,SYD]
[ATL,8434,JNB]
[SFO,8433,SIN]
[AUH,8372,LAX]
[DXB,8321,LAX]
[JED,8314,LAX]
[DOH,8287,LAX]
[LAX,8246,RUH]
[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]
```

5.3.2. Combining *aggregate*, *union* and *filter* to compute distances.

This next query is similar to the one we looked above at in the "Where can I fly to from here?" section. It uses *aggregate*, *union*, *filter* and *where* being used together 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.

```groovy
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 biger 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.

```
2168462
8950
```

5.3.3. 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* or in more recent versions of TinkerPop we can use *path* and *by*. I prefer the latter technique.

```groovy
// Distance between the AUS and MEX airports
g.V().has('code','AUS').outE().as('e').inV().has('code','MEX').select('e').values('dist')

748

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. If you remove the call to *count* it will list some of the routes.

```groovy
// Routes Between 100 and 200 miles in length
g.V().as('src').outE().has('dist',within(100..200)).inV().as('dest').
      path().by('code').by('dist').count()

3029
```

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* some of the routes will be returned.

```groovy
// 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()

583
```

Lastly, this query returns a list of all the routes from San Antonio along with their distances. Some of the results are shown.

```groovy
// 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').count()
```

```
[SAT,872,ATL]       [SAT,1140,MIA]
[SAT,820,BNA]       [SAT,698,MEX]
[SAT,1410,BWI]      [SAT,1097,MSP]
[SAT,248,DFW]       [SAT,1093,CLT]
[SAT,1360,IAD]      [SAT,1040,ORD]
[SAT,190,IAH]       [SAT,931,CUN]
[SAT,1580,JFK]      [SAT,841,PHX]
[SAT,1210,LAX]      [SAT,624,MEM]
[SAT,1038,MCO]      [SAT,1772,SEA]
[SAT,1423,YYZ]      [SAT,707,MCI]
```

5.3.4. 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.

```groovy
// 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 I will have to travel in each case. That is the topic of the next section!

```
[AUS,1476,SJC,5352,LHR]       [AUS,183,DFW,4736,LHR]
[AUS,1500,SFO,5350,LHR]       [AUS,1339,BWI,3622,LHR]
[AUS,1768,SEA,4783,LHR]       [AUS,1500,EWR,3453,LHR]
[AUS,866,PHX,5255,LHR]        [AUS,1690,BOS,3254,LHR]
[AUS,973,ORD,3939,LHR]        [AUS,768,DEN,4655,LHR]
[AUS,1040,MSP,4001,LHR]       [AUS,1080,LAS,5213,LHR]
[AUS,1230,LAX,5439,LHR]       [AUS,809,ATL,4198,LHR]
[AUS,1520,JFK,3440,LHR]       [AUS,1160,SAN,5469,LHR]
[AUS,1140,DTW,3753,LHR]       [AUS,1357,YYZ,3544,LHR]
[AUS,142,IAH,4820,LHR]        [AUS,5294,FRA,406,LHR]
[AUS,1430,PHL,3533,LHR]       [AUS,748,MEX,5529,LHR]
[AUS,1294,IAD,3665,LHR]       [AUS,1030,CLT,3980,LHR]
```

5.3.5. 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 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. It allows us to define a place where we can put things as the graph traversal proceeds. You 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 your query will return. Take a look at the following query:

```groovy
// 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(incr).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 on line 2 by only keeping destinations that are *LHR*. On line 3 we take the values that are stored in our sack and sort them in ascending order. Finally on line 4 we process the paths that we have taken. Note that the sack value is included as part of the path. Running the query will produce the following results:

```groovy
[AUS, 1140, DTW, 3753, LHR, 4893]
[AUS, 1357, YYZ, 3544, LHR, 4901]
[AUS, 973, ORD, 3939, LHR, 4912]
[AUS, 183, DFW, 4736, LHR, 4919]
[AUS, 1690, BOS, 3254, LHR, 4944]
[AUS, 1500, EWR, 3453, LHR, 4953]
[AUS, 1294, IAD, 3665, LHR, 4959]
[AUS, 1520, JFK, 3440, LHR, 4960]
[AUS, 1339, BWI, 3622, LHR, 4961]
[AUS, 142, IAH, 4820, LHR, 4962]
```

NOTE

In Tinkerpop 3.3 the syntax of *sack* was changed. Where previously you would write something like *sack(sum,*'runways'*)* you are now required to write *sack(sum).by(*'runways'*)*. The prior format was already deprecated in Tinkerpop 3.2.5 but is now fully removed in Tinkerpop 3.3.

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.

```groovy
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(incr).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 produces.

```groovy
AUS --> DTW --> LHR 4893 miles
AUS --> YYZ --> LHR 4901 miles
AUS --> ORD --> LHR 4912 miles
AUS --> DFW --> LHR 4919 miles
AUS --> BOS --> LHR 4944 miles
AUS --> EWR --> LHR 4953 miles
AUS --> IAD --> LHR 4959 miles
AUS --> JFK --> LHR 4960 miles
AUS --> BWI --> LHR 4961 miles
AUS --> IAH --> LHR 4962 miles
```

5.3.6. 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.

```groovy
// 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())

[ORD:226, DFW:216, FRA:254, CDG:253, PEK:232, AMS:257, ATL:232, MUC:219, IST:259, DME:206,
DXB:225]
```

The prevous 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 overkill for the task (as demonstrated by the much simpler query above) it does show how you can use sacks in powerful ways to store data as your query iterates.

```groovy
// The same query but done using the new sack() step in TinkerPop 3
// 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()

[ATL:232, DFW:216, ORD:226, CDG:253, FRA:254, DXB:225, PEK:232, AMS:257, MUC:219, DME:206,
IST:259]
```

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.

```groovy
g.V().hasLabel('airport').where(out().count().is(gt(200))).
                          group().by('code').by(outE().count()).
                          order(local).by(values)

[DME:213,DFW:221,DXB:229,ORD:232,PEK:232,ATL:232,MUC:237,CDG:260,FRA:266,AMS:269,IST:270]
```

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.

```groovy
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:269,ATL:232,CDG:260,DFW:221,DME:213,DXB:229,FRA:266,IST:270,MUC:237,ORD:232,PEK:232]
```

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 or 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 number of runways for each individual airport so we need to add all of those to get a single grand total.

```
GROOVY
g.V().has('code','AUS').sack(assign).by(constant(0)).
      out().sack(sum).by('runways').sack().sum()

212
```

5.3.7. 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 Janus Graph have some useful geographic coordinate and shape support built in but TinkerGraph does not. Moreover, GraphML does not offer any specific geospatial support so to keeps things simple and flexible, the air routes data set does not assume any specific back end capabilities. This said, even having coordinates provided as basic floating point numbers offers us a chance to do some interesting geospatial queries. So far in this document we have not made us of the latitude and longitude coordinates that are stored for each airport so in this section we will experiment a bit with some queries that do. This first query just returns the coordinates for London Heathrow.

```
GROOVY
// 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.

```
GROOVY
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

```groovy
// 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:[-0.0064389999902457]]
[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.

```groovy
// 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. This type of thing can be done using the Geo shape classes provided by Janus Graph but given we are not at that part of the document yet this is the next best way!

```groovy
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 document, 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*.

```groovy
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.

```groovy
[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" 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.

```groovy
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.

```groovy
// 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!.

```groovy
[AUS,PDX,LAS]
[AUS,ABQ,LAS]
[AUS,LBB,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,ELP,LAS]
[AUS,OAK,LAS]
```

We will revisit the topic of performing geospatial queries in the "The Janus Graph geospatial API" section where we look at some additional capabilities that Janus Graph offers to us in this area.

To finish up examples of how the geospatial and region information stored in the graph can be used to generate interesting results, let's look at one last query that uses the *region* property that is present on all of the airport vertices in the graph.

The query below finds the DFW airport then looks at all routes from there to airports within the United States. Next we group those airports by their region code and airport code. Finally we select just a few states and unfold the results to make them a bit more readable.

```groovy
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 results from running the query. Each of the states we selected is listed along with an array of airport codes in those states that you can fly to fro DFW.

```groovy
US-CA=[LAX, SFO, SJC, SAN, SNA, OAK, ONT, PSP, SMF, FAT, SBA]
US-TX=[AUS, IAH, SAT, HOU, ELP, LBB, MAF, CRP, ABI, ACT, CLL,
       BPT, AMA, BRO, GGG, GRK, LRD, MFE, SJT, SPS, TYR]
US-FL=[FLL, MCO, MIA, PBI, TPA, RSW, TLH, JAX, PNS, VPS]
US-CO=[DEN, COS, DRO, GJT, EGE, HDN, ASE, GUC, MTJ]
US-IL=[ORD, PIA, BMI, CMI, MLI, SPI]
```

5.4. 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 aplhabetic order.

```
g.V().hasId(between(1,201)).values('city').order().fold()
```
GROOVY

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, Bankok, 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. Johns, 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]
```
GROOVY

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
```
GROOVY

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 I 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 setup a named place we can *store* things into later in

our query and we can also give the store a type. In this case I 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.

```groovy
g.withSideEffect("x", [] as Set).V().hasId(between(1,201)).
  values('city').store('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 of the duplicate names are now gone.

```groovy
[Abu Dhabi, Addis Ababa, Albuquerque, Alicante, Alice Springs, Amsterdam, Anchorage,
Athens, Atlanta, Auckland, Austin, Ayers Rock, Baltimore, Bankok, 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. Johns, 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 less names back this time let's count them again.

```groovy
g.withSideEffect("x", [] as Set).V().hasId(between(1,201)).
  values('city').store('x').cap('x').unfold().count()

186
```

As I 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*.

```groovy
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.

```groovy
g.withSideEffect("x", [] as Set).V().hasId(3).as('a').values('city').store('x').
   select('a').out().as('b').values('city').store('x').
   select('b').out().values('city').store('x').cap('x').unfold().count()

804
```

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.

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

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 that were allocated for each node by looking at the *valueMap*.

```groovy
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.

```groovy
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 node A, find all of the edges that go out from node A and also all of 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 but ignore edges that connect to vertices that are not also connected to A. In simple terms we want a query that will return all of 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. I 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.

```groovy
1: g.withSideEffect('x', [] as Set).
2:     V(0L).store('x').
3:     bothE().store('x').
4:     otherV().store('x').
5:     aggregate('tgtlist').
6:     bothE().as('ref').otherV().where(within('tgtlist')).
7:     select('ref').store('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 'store' it into our set 'x'.
3: Store all of the edges connected to 'V(0)' into our set.
4: Store 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: Store the edges we found in 'x' and finally return the set as the overall result of 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.

```groovy
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.

5.5. Using *emit* to return results during a traversal

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 ending condition!

```groovy
g.V().has('code','SAF').repeat(out()).emit().path().by('code').limit(10)

[SAF,DFW]
[SAF,LAX]
[SAF,PHX]
[SAF,DEN]
[SAF,DFW,ATL]
[SAF,DFW,ANC]
[SAF,DFW,AUS]
[SAF,DFW,BNA]
[SAF,DFW,BOS]
[SAF,DFW,BWI]
```

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 less hops. First of all take a look at the query below that does not use an *emit* and the results that it generates.

```groovy
g.V(3).repeat(out()).times(3).has('code','MIA').limit(5).path().by('code')

[AUS,YYZ,MUC,MIA]
[AUS,YYZ,MAN,MIA]
[AUS,YYZ,YUL,MIA]
[AUS,YYZ,SVO,MIA]
[AUS,YYZ,GRU,MIA]
```

As you can see, only routes that took three full hops were returned. Now let's change the query to use an *emit*. This time you can think of the query as saying "at most three hops".

```groovy
g.V(3).repeat(out()).emit().times(3).has('code','MIA').limit(5).path().by('code')

[AUS,MIA]
[AUS,YYZ,MIA]
[AUS,LHR,MIA]
[AUS,FRA,MIA]
[AUS,MEX,MIA]
```

So 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.

You will see more examples of *emit* being used in the next section.

5.6. Modelling 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 node 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.

```groovy
graph=TinkerGraph.open()
g=graph.traversal()
g.addV(label,'root','data',9).as('root').
  addV(label,'node','data',5).as('b').
  addV(label,'node','data',2).as('c').
  addV(label,'node','data',11).as('d').
  addV(label,'node','data',15).as('e').
  addV(label,'node','data',10).as('f').
  addV(label,'node','data',1).as('g').
  addV(label,'node','data',8).as('h').
  addV(label,'node','data',22).as('i').
  addV(label,'node','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.

```groovy
// 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.

```
GROOVY
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 node the *repeat* visits. Note that this does not include the value from the root node.

```
GROOVY
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 node's value.

```
GROOVY
g.V().hasLabel('root').repeat(out('left')).
                       until(__.not(out('left'))).path().by('data')

[9,5,2,1]
```

We can see all of the possible paths through the tree by running the following query.

```
GROOVY
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 node values into trees" section. We could use what we discussed there to create a Tree object from our Binary Tree graph as follows.

```groovy
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.

```groovy
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.

```groovy
println t

[9:[5:[2:[1:[:]], 8:[:]], 11:[10:[:], 15:[22:[16:[:]]]]]]
```

We can use the *getObjectsAtDepth* method to further investigate the tree structure.

```groovy
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
```

5.7. Using *map* to produce a concatenated result string

In the example below we use *map* to build a string containing the airport code concatenated with the city the airport is in for airports in England.

```groovy
g.V().has('airport','region','GB-ENG').limit(10).
  map{it.get().value('code')+" "+it.get().value('city')}
```

Here are the results of running the query.

```groovy
LHR London
LGW London
MAN Manchester
LCY London
STN London
EMA Nottingham
LPL Liverpool
LTN London
SOU Southampton
LBA Leeds
```

5.8. 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) node 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 vertex (node) we are currently at and then moving to the vertex at the other end of that edge. Notice how to get the effect that we want that the edge is picked inside of a *local* step.

```groovy
// Random walk with five hops
g.V().has('code','AUS').repeat(local(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,992,SFB,828,MDT,592,ORD,234,DTW,500,LGA]

[AUS,957,CVG,374,ATL,1890,SAN,2276,DCA,204,PIT]

[AUS,1209,PIT,1399,CUN,941,BJX,729,IAH,1384,BZN]

[AUS,748,MEX,1252,PHX,5255,LHR,2487,LXR,492,JED]

[AUS,722,STL,717,DCA,893,RSW,1103,HPN,563,CLT]
```
GROOVY

5.9. 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 I was still cleaning up the data, I frequently ran into problems with parallel edges getting included in the graph by mistake. I realized I could use Gremlin to help me detect these error cases.

Initially I tried something very basic, that still required quite a bit of manual reading of the output. I 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(valueDecr).next().values().max()
```
GROOVY

If the answer came back greater than one I 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(valueDecr)]
```
GROOVY

As I said this was a very manual and time consuming process. I clearly needed a better query. Given what we know about *groupCount* I realized I could write an arbitrary query to tell me how many times every single route in the graph exists. However, given there are over 43,000 routes I was not going to be able to check that manually. So as is often the best way with Gremlin I built up my query in stages. First of all I wrote the query to count the occurrence of all routes. I 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 of the results. Far from ideal!

```groovy
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 ...
```

I then added a filter to only select any routes that occurred more than once. Note that I had to use *unfold* before I applied the *filter* to turn the map back into a stream of values that could be filtered.

```groovy
g.V().as("a").out().as("b").groupCount().by(select("a","b")).unfold().
                            filter(select(values).is(gt(1)))
```

Next I added one more step to tell me which routes contained the error.

```groovy
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 I ran into was that I 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 I ran the above query I got the following output which told me exactly which route I needed to correct. This is clearly a much more useful query than the prior ones. I could happily have stopped at this point but I wanted to see if I could improve the query some more.

```groovy
[a:v[49],b:v[12]]
```

I was still not totally happy with my query as I really wanted it to give me back the airport codes. So I added a few more tweaks to do the *groupCount* using the airport codes. I have also left the *select(keys)* off the end of this query as I think it aids understanding to see what is returned before that step is performed.

```groovy
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 I want to add just a couple of additional steps to demonstrate other possible refinements that you may find useful in other circumstances.

```groovy
{a=LHR, b=JFK}=2
```

If we don't want the *=2* part returned we can just select the keys part.

```groovy
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.

```groovy
[a:LHR,b:JFK]
```

This is almost exactly what I wanted but we can add one more tiny step to clean up the output by just selecting the values from the map returned.

```groovy
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.

```groovy
[LHR,JFK]
```

TIP

Note how I built up my Gremlin query in stages to solve this problem. I 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.

5.10. 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. I have documented here the learning steps I went through to get what I thought was a very 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 this (below) might seem like the way obvious and easy to do it, because of the way that max() is implemented it will not work. Currently max() and min() cause the prior paths that have been taken in a traverasl to be lost. There is a defect open against TinkerPop for this that we created but until that defect gets resolved, we need to explore other ways of achieving our desired result.

```groovy
g.E().hasLabel('route').as('e').values('dist').max().select('e')
```

First of all we could do this (below) The down side 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.

```groovy
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.

```groovy
r=g.E().hasLabel('route').as('e').order().by('dist',
decr).limit(1).select('e').id().next()
g.E(r).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!

```groovy
g.E().hasLabel("route").order().by("dist", decr).store("d").by("dist").
  filter(values("dist").as("cd").select("d").by(limit(local, 1)).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.

```groovy
g.E().hasLabel("route").order().by("dist",decr).store("d").by("dist").\
  filter(values("dist").as("cd").select("d").by(limit(local, 1)).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 I will leave that as an exercise for the reader!

```groovy
[from:DXB,to:AKL,dist:8818]
[from:AKL,to:DXB,dist:8818]
```

5.11. Miscellaneous other queries

These queries need to be moved to other sections or deleted.

THIS SECTION IS YET TO BE FULLY TIDIED UP

Work in progress

5.11.1. Work in progress

Using a mathematical expression inside of *is*

```
GROOVY
g.V().hasLabel('airport').where(out().count().is(gt(500/2))).values('code')

CDG
FRA
AMS
IST
```

```
GROOVY
a = 500

g.V().hasLabel('airport').where(out().count().is(gt(a/2))).values('code')

CDG
FRA
AMS
IST
```

```
GROOVY
g.V().hasLabel('airport').where(out().count().is(gt(a/2))).
      project('apt','routes').by('code').by(out().count())

[apt:CDG,routes:262]
[apt:FRA,routes:266]
[apt:AMS,routes:269]
[apt:IST,routes:270]
```

```
GROOVY
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')
```

```
GROOVY
a=g.V().has('region','US-OK').values('code').withIndex().toList()
[OKC,0]
[TUL,1]
[LAW,2]
[SWO,3]
```

```groovy
g.V().has('code','DFW').as('dfw').out().as('to').
      filter(select('dfw','to').by('region').where('dfw',eq('to'))).values('code')

g.V(0L).as('v').values('c').as('val').select('v').properties().as('p').properties('c').
  value().as('val2').select('p').filter(select('val','val2').where('val',eq('val2')))

g.V().as("a").properties().has("c").where(eq("a")).by("c")
```

```groovy
g.V().has('code','AUS').out().order().by('code').
                        values('code','icao').fold()

[ABQ,KABQ,ATL,KATL,BKG,KBBG,BNA,KBNA,BOS,KBOS,BWI,KBWI,CLE,KCLE,CLT,KCLT,CUN,MMUN,CVG,KCVG
,DAL,KDAL,DCA,KDCA,DEN,KDEN,DFW,KDFW,DTW,KDTW,ELP,KELP,EWR,KEWR,FLL,KFLL,FRA,EDDF,GDL,MMGL
,HOU,KHOU,HRL,KHRL,IAD,KIAD,IAH,KIAH,IND,KIND,JFK,KJFK,LAS,KLAS,LAX,KLAX,LBB,KLBB,LGB,KLGB
,LHR,EGLL,MCI,KMCI,MCO,KMCO,MDW,KMDW,MEM,KMEM,MEX,MMMX,MIA,KMIA,MSP,KMSP,MSY,KMSY,OAK,KOAK
,ORD,KORD,PDX,KPDX,PHL,KPHL,PHX,KPHX,PIE,KPIE,PIT,KPIT,PNS,KPNS,RDU,KRDU,SAN,KSAN,SEA,KSEA
,SFB,KSFB,SFO,KSFO,SJC,KSJC,SLC,KSLC,SNA,KSNA,STL,KSTL,TPA,KTPA,VPS,KVPS,YYZ,CYYZ]
```

```groovy
// Using next to iterate through vertices
t=g.V().hasLabel('airport');[]
t.next(10)
t.next(10)
t.next(10)
```

```groovy
a = g.V().hasLabel('airport').next(5)
v[1]
v[2]
v[3]
v[4]
v[5]
```

```groovy
g.V().sack(assign).by('code').sack()
```

```groovy
// Experiments with timing and indexes
clock(1000) {g.V().has('longest',gt(12000))}
0.008413653

graph.createIndex('longest',Vertex.class)
clock(1000) {g.V().has('longest',gt(12000))}
0.003952521

graph.getIndexedKeys(Vertex.class)
graph.dropIndex('longest',Vertex.class)
```

```groovy
g.V(3).repeat(out()).until(loops().is(2)).count()

5894

g.V(3).repeat(out()).times(2).count()

5894
```

```groovy
// Experiments with how many route permutations are there based on number of hops.
g.V().has('code','AUS').as('a').repeat(out().where(neq('a'))).times(1).simplePath().has('code','LAX').path().by('code').count()
1
g.V().has('code','AUS').as('a').repeat(out().where(neq('a'))).times(2).simplePath().has('code','LAX').path().by('code').count()
43
g.V().has('code','AUS').as('a').repeat(out().where(neq('a'))).times(3).simplePath().has('code','LAX').path().by('code').count()
2351
g.V().has('code','AUS').as('a').repeat(out().where(neq('a'))).times(4).simplePath().has('code','LAX').path().by('code').count()
122329

// From Austin and back again
g.V().has('code','AUS').out().out().cyclicPath().path().by('code')

// Ugly query! Look at every property of every node and see if it contains "R."
g.V().as('a').values().filter{it.toString().contains('R.')}.select('a').dedup().valueMap()
```

5.11.2. Experiments with Page Rank

This is a placeholder.

```groovy
// Page rank based on outgoing routes
g = graph.traversal().withComputer()
g.V().hasLabel('airport').pageRank().by(outE('route')).by('r').order().by('r',decr).valueMap('code','r').limit(10)
[r:[15.61824172032977],code:[ATL]]
[r:[14.922712714789586],code:[DFW]]
[r:[14.637892542166075],code:[IST]]
[r:[14.544774102601247],code:[ORD]]
[r:[13.82927112415382],code:[DEN]]
[r:[13.794192182285459],code:[DME]]
[r:[13.46824937434635],code:[AMS]]
[r:[13.375467729773533],code:[CDG]]
[r:[13.154068946539626],code:[PEK]]
[r:[13.047424689334612],code:[FRA]]
```

6. MOVING BEYOND THE CONSOLE AND TINKERGRAPH

This section is still under construction!

Most of the examples we have looked at so far were produced using the Gremlin console and the TinkerGraph in-memory graph all running on a single machine. However, there are many ways to deploy and interact with a graph while still using Gremlin and optionally the Gremlin Console, and many of them go beyond doing everything on a single machine. The Apache TinkerPop package for example includes a component called Gremlin Server. Gremlin Server allows you to host a graph database locally or remotely and talk to it over HTTP or WebSockets. The Gremlin console supports accessing graphs both locally or remotely or you can access them using your own code or other tools or even command line utilities such as *curl*.

For a production environment it is likely you will use a technology such as Janus Graph backed by something like HBase or Cassandra and an indexing technology such as Solr or Elastic Search. In these cases the way you work with and manage the graph and the way that query results are retuned will vary.

In some cases data returned will be in the form of a GraphSON (JSON) in others it might be as variables within a program. There are also ways to work with graphs using Gremlin from a Python Notebook. You might setup your own on-premise graph system or you might use a hosted service like IBM Graph. It's quite possible you might still wish to connect to a remote graph using Gremlin Server via the Gremlin Console but you just as likely could use *curl* or some other HTTP/REST type of technology. So that is a long winded way of saying that once you move beyond the basics and head towards putting a system into production, there are a lot of options to consider.

In this section you will a selection of examples from these more sophisticated environments. The purpose of this document thus far has been to teach Gremlin using the Gremlin Console and TinkerGraph as our learning environment. However, it would be remiss to end our discussion without at least touching on some of these other environments, how you might configure them and why you might use them.

6.1. Working with TinkerGraph from Java Application

So far in this document 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 Gremlin Console is just one to the tools you will need to have available in your toolbox. It is very likely, if not certain, that you will want to write stand alone applications that can work with a graph. There are a number of different language bindings currently available for TinkerPop 3. One of the most widely used is the Java API for TinkerPop.

As discussed at earlier, when building a commercial application, you may need capabilities such as ACID transactions 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. As a first step towards writing stand alone applications that use different graph database back ends, lets 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 http://tinkerpop.apache.org/javadocs/current/full

The 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.

When using the Gremlin Console your environment is pre-configured for you so you do not have to import any classes or interfaces. However, as soon as you move to the domain of the stand alone Java application you will need to start importing the relevant classes that your application builds upon.

By way of a reasonable start, here are some imports that will enable us to do a number of Gremlin tasks from a Java application. 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 of a *repeat* step). If you prefer you can statically import the "__" class it which will make explicit use of "__" in your code unnecessary except where you are faced with reserved word conflicts as discussed earlier in this document.

```java
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 *GraphTest*, 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 that is hidden from you when you are working within the Gremlin Console. 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 node and print it. The *toString* method provided by the *Map* should give us some useful output.

```java
public class GraphTest
{
  public static void main(String[] args)
  {
    TinkerGraph tg = TinkerGraph.open() ;

    try
    {
      tg.io(IoCore.graphml()).readGraph("./air-routes.graphml");
    }
    catch( IOException e )
    {
      System.out.println("File not found");
      System.exit(0);
    }
    GraphTraversalSource g = tg.traversal();
    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 setup 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 the purpose of our experiments here, simple use of the *classpath* will suffice.

The following lines of Bash shell script will setup 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. Later when we start using Janus graph you will see that rather than TinkerGraph we will need to adjust these settings to point to the Janus Graph JAR files.

```
# 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 GraphTest.java
```

Assuming everything we have coded so far compiled OK, then we can run it using the same *classpath* that we created previously. the following output.

```
# Run
java -cp $CP GraphTest
```

The output we get back should look something like the following.

```
JAVA
{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.

```java
    List city = (List)(aus.get("city"));
    System.out.println("The AUS airport is in " + city.get(0));

The AUS airport is in Austin
```

```java
aus.forEach( (k,v) -> System.out.println("Key: " + k + ": Value: " + v));
```

```java
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]
```

```java
 Long n = g.V().has("code","DFW").out().count().next();
    System.out.println("There are " + n + " routes from Dallas");
```

```java
    List fromAus = (g.V().has("code","AUS").out().values("code").toList());
    System.out.println(fromAus);
```

```java
[LHR, FRA, MEX, PHX, PIT, RDU, PDX, SEA, CLT, SFO, CUN, SJC, MEM, JFK, LAX, MCO, YYZ, MIA,
MSP, ORD, DAL, STL, LAS, ABQ, DEN, MDW, MSY, LBB, EWR, HRL, HOU, GDL, ELP, PNS, MCI, TPA,
CVG, SAN, IND, LGB, SNA, SLC, PIE, SFB, BKG, CLE, VPS, OAK, PHL, DTW, DFW, FLL, IAD, IAH,
ATL, BNA, BOS, BWI, DCA]
```

```java
    List <Path> lhrToUsa = g.V().has("code","LHR").outE().inV().
                                has("country","US").limit(5).
                                path().by("code").by("dist").toList();

    lhrToUsa.forEach((k) -> System.out.println(k));
```

```
[LHR, 3860, RDU]
[LHR, 4783, SEA]
[LHR, 3939, ORD]
[LHR, 5255, PHX]
[LHR, 5350, SFO]
```

JAVA

```
    ArrayList <Path> routes = new ArrayList<>();
    g.V().has("code","SAT").out().path().by("icao").fill(routes);
    System.out.println(routes);
```

JAVA

```
[[KSAT, CYYZ], [KSAT, KCLT], [KSAT, KELP], [KSAT, MMMX], [KSAT, KEWR], [KSAT, KHOU],
[KSAT, KMIA], [KSAT, KMSP], [KSAT, KORD], [KSAT, KSEA], [KSAT, KSFO], [KSAT, KSAN], [KSAT,
KLAS], [KSAT, KDEN], [KSAT, KSTL], [KSAT, KHRL], [KSAT, KMDW], [KSAT, MMGL], [KSAT, MMMY],
[KSAT, MMTO], [KSAT, KSFB], [KSAT, KJFK], [KSAT, KLAX], [KSAT, KMCO], [KSAT, KPHL], [KSAT,
KMEM], [KSAT, KDTW], [KSAT, MMUN], [KSAT, KATL], [KSAT, KDAL], [KSAT, KBNA], [KSAT, KMCI],
[KSAT, KBWI], [KSAT, KDFW], [KSAT, KIAD], [KSAT, KIAH]]
```

JAVA

```
    Vertex v = g.V().has("code","FRA").next();
    Set <String> s = v.keys();
    for (String k : s)
    {
      System.out.println( k + "\t: " + v.property(k).value());
    }
```

JAVA

```
country : DE
code    : FRA
longest : 13123
city    : Frankfurt
elev    : 364
icao    : EDDF
lon     : 8.54312992096
type    : airport
region  : DE-HE
runways : 4
lat     : 50.0264015198
desc    : Frankfurt am Main
```

JAVA

```
List eng = g.V().has("code","AUS").repeat(__.out()).times(2).
              has("region","GB-ENG").dedup().values("code").toList();

System.out.println(eng);

[LGW, LHR, MAN, LBA, NCL, LCY, BHX, BRS, STN]
```

6.2. Working with TinkerGraph from a Groovy application

Earlier in this document, 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 stand alone Groovy application to work with a TinkerGraph.

In this section we will rewrite the application we coded in Java above in 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 of the TinkerPop 3 classes that our program will use.

```groovy
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.

```groovy
println "Gremlin version is ${Gremlin.version()}"

def tg = TinkerGraph.open()
```

We are now ready to create a new TinkerGraph instance 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. Assuming we did not get an exception we can go ahead and create our graph traversal object

```groovy
println "Loading the air-routes graph...\n"

try
{
  tg.io(IoCore.graphml()).readGraph("/Users/kelvin/krl/code/graph/air-routes.graphml");
}
catch (IOException e)
{
  println "Could not load the graph file"
  System.exit(0);
}

def g = tg.traversal()

def aus = g.V().has('code','AUS').valueMap().next()

println aus
```

Note that just as when we were looking at the 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.

6.2.1. Compiling our Groovy application

In the Java example, included the CLASSPATH on the *javac* invokation 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 GraphTest.groovy
```

6.2.2. Running our Groovy application

Assuming everything compiled cleanly we can now run our Groovy application.

```
groovy GraphTest

Gremlin version is 3.2.4
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]]
```

6.2.3. Adding to our Groovy application

```
def city = aus['city']
println "\nThe AUS airport is in ${city[0]}\n"

aus.each {println "${it.key} : ${it.value[0]}"}
```

```
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
```

GROOVY

```
def n = g.V().has("code","DFW").out().count().next()
println "\nThere are  ${n} routes from Dallas"

There are  221 routes from Dallas
```

GROOVY

```
def fromAus = g.V().has("code","AUS").out().values("code").toList()
println "\nHere are the places you can fly to from Austin\n"
println fromAus

[YYZ, LHR, FRA, MEX, PIT, PDX, CLT, CUN, MEM, CVG, IND, MCI, DAL, STL, ABQ, MDW, LBB, HRL,
GDL, PNS, VPS, SFB, 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, MSY, EWR, HOU,
ELP, CLE, OAK, PHL, DTW]
```

GROOVY

```
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}
```

```
From LHR to airports in the USA (only 10 shown)

[LHR, 4896, PDX]
[LHR, 3980, CLT]
[LHR, 4198, ATL]
[LHR, 4901, AUS]
[LHR, 3254, BOS]
[LHR, 3622, BWI]
[LHR, 4736, DFW]
[LHR, 3665, IAD]
[LHR, 4820, IAH]
[LHR, 3440, JFK]
```

GROOVY

```groovy
def routes = []
g.V().has("code","SAT").out().path().by("icao").limit(10).fill(routes);
println "\nRoutes from San Antonio (only 10 shown)\n"
routes.each {println it}
```

GROOVY

```
Routes from San Antonio (only 10 shown)

[KSAT, KHRL]
[KSAT, MMMY]
[KSAT, MMGL]
[KSAT, KATL]
[KSAT, KSFB]
[KSAT, KBNA]
[KSAT, MMTO]
[KSAT, KBWI]
[KSAT, KDFW]
[KSAT, KIAD]
```

GROOVY

```groovy
def v = g.V().has('code','FRA').next()
println "\nKeys found in the FRA vertex"
println v.keys()
```

GROOVY

```
Keys found in the FRA vertex

[country, code, longest, city, elev, icao, lon, type, region, runways, lat, desc]
```

GROOVY

```groovy
def eng = g.V().has("code","AUS").repeat(__.out()).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"
```

```groovy
Airports in England reachable with no more than one stop from AUS

[LHR, LGW, MAN, LBA, NCL, LCY, BRS, BHX, STN]
```

6.3. Introducing Janus Graph

So far we have been using the TinkerGraph graph that is included with Apache TinkerPop in our examples. Once you move beyond learning about Gremlin and its related technologies and moving towards a production deployment, you will need a graph store that provides capabilities such as reliable persistence, the ability to define schemas and support for ACID transactions. The Janus Graph project, which began in 2016 as an open source fork of the popular Titan graph database, is hosted by the Linux Foundation and provides these advanced capabilities.

Janus Graph can run on a laptop, which is useful for learning and experimenting, but it is designed to handle very large graphs stored on distributed clusters. It can handle graphs containing billions of vertices and edges. As we shall discuss, Janus graph is designed to work with a variety of persistent storage options including Apache Cassandra and Apache HBase as well as indexing technology such as Apache Solr and Apache Elastic Search.

Here are some useful Janus Graph resources

Runtime download (JAR files and more)

http://janusgraph.org/

Documentation

http://docs.janusgraph.org/latest/.

API Documentation

https://javadoc.io/doc/org.janusgraph/janusgraph-core/0.1.1

In the following sections we will take an in depth look at Janus Graph and other technologies that, when combined, provide a way to build and deploy a massively scalable graph database solution. We will start by quickly looking at how to install Janus Graph and access it from the Gremlin Console before getting into more advanced topics including how to create and manage both schemas and indexes and how to use the transactional capabilities provided by Janus Graph.

6.3.1. Installing Janus Graph

To be written

6.3.2. Using Janus Graph from the Gremlin Console

To be written

6.3.3. Using Janus Graph with the *inmemory* option

For almost all production use cases, you will be using Janus Graph along with a persistent back end store such as Apache Cassandra or Apache HBase. However while experimenting with Janus graph it is incredibly useful to be able to get up and running quickly without having to worry about configuring all of the back end storage components. This is made possible by the *inmemory* option that Janus Graph provides. This essentially allows us to use Janus Graph in the same way as we have been using TinkerGraph with all of our graph data stored in the memory of the computer. The one big difference however is that Janus Graph, even while using the *inmemory* storage model, allows us to experiment with features that TinkerGraph does not offer, such as schemas and transactions. We will get into those topics a bit later on. First, let's create an instance of Janus Graph from the Gremlin console that uses the *inmemory* storage model.

Creating a Janus Graph instance is very similar to the way we created a TinkerGraph instance earlier in the document. The only difference is that we use the *JanusGraphFactory* to create the graph and in this case we specify *inmemory* as the only parameter to tell Janus Graph that we want to use the all in memory storage model. We create our graph traversal object *g* in just the same way as before.

```groovy
graph = JanusGraphFactory.open('inmemory')
g = graph.traversal()
```

Now that we have a graph instance created, just like with TinkerGraph, we can query which features the graph supports. Earlier in the document in the "Introducing TinkerGraph" section we looked at the features offered by TinkerGraph. If we compare those features to what Janus Graph offers we can spot some key differences.

We can get the feature set back by calling the *features* method as shown below. The first thing that stands out is that the various features that involve transactions are now set to *true* indicating that Janus Graph supports transactions. We will take a look at how to use these transactional capabilities in the next section. Note that *Persistence* still shows as *false* as we are using the *inmemory* mode. Another thing to note is that, unlike with TinkerGraph, *UserSuppliedIds* is set to false, indicating that JanusGraph will create its own ID values and ignore any that we provide. The list is formatted in two columns to aid readability.

```
graph.features()

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

Now that we have an empty instance of an *inmemory* Janus Graph we can use it from the Gremlin Console just as we did with TinkerGraph in our prior examples. Notice that the ID values that Janus Graph generates look quite different (as in they don't start at zero) from what we might expect from TinkerGraph.

```groovy
g.addV(label,'person','name','Kelvin')
v[1466464]

g.V().has('name','Kelvin')
v[1466464]
```

Before we experiment too much more with Janus Graph there are three important subjects we need to discuss. One is transactions, another is defining a schema for our vertices, edges and properties and the third is the Janus Graph management API. We will cover each of these key subjects in the following sections.

6.3.4. Janus Graph transactions

To be written

```groovy
// Start a new transaction
xyz = graph.addVertex()

// Add a property
xyz.property("name", "XYZ")

// Commit the transaction
graph.tx().commit()

g.V().has('name','XYZ')
v[4344]
```

```groovy
// Start a new transaction
xyz = graph.addVertex()

// Add a property
xyz.property("name", "XYZ")

// Commit the transaction
graph.tx().rollback()

// Nothing will be returned
g.V().has('code','XYZ')
```

6.3.5. Loading air-routes into a Janus Graph instance

We can use the same basic steps to load the air-routes graph into a Janus Graph instance that we used with Tinkergraph. Note that after loading the graph from the XML file we then call *commit* to finalize the transaction.

```groovy
graph.io(graphml()).readGraph('air-routes.graphml')
graph.tx().commit()
```

Note that as we have not defined a schema before loading the air-routes graph that Janus Graph will create vertices, edges and properties using default types and settings. A bit later we will look at creating a schema for air-routes and then loading it again to see the differences. It is strongly recommended to create the schema before loading the data but lets examine a few things before we discuss how to do that.

Unlike TinkerGraph, Janus Graph does not, by default, guarantee to respect user provided node and edge ID values. Instead it creates its own ID values as vertices and edges are added to the graph. You may have noticed from earlier in the document or from the air-routes.graphml file if you happened to look in there, that the ID provided for Austin in the GraphML markup is 3. However, having loaded air-routes into Janus Graph if we query the ID for the Austin node we can see that it is no longer 3. There is a setting that can be changed to force Janus Graph to honor user provided ID values but it is not recommended this be used as it will disable some other useful Janus Graph features. If you are interested in learning more about this option this please refer to the Janus Graph documentation.

```groovy
g.V().has('code','AUS').id()

4240
```

Having the graph system allocate its own ID values is not a big problem as we can always query the graph to get the ID but it is a reminder that you should not get into the habit of relying on any user provided ID values as you work with graphs.

If necessary, as we discussed earlier in the document, we can always store important ID values in a variable for later use.

```groovy
ausid = g.V().has('code','AUS').id().next()

g.V(ausid).values('city','desc','region').join(',')

Austin Bergstrom International Airport,Austin,US-TX
```

6.3.6. The Janus Graph management API

Janus graph includes a management API that is made available via the ManagementSystem class. You can use the management API to perform various important functions that include querying metadata about the graph, defining the edge, vertex and property schema types and creating and updating the index.

You can create an instance of the ManagementSystem object using the *openManagement* method call.

```groovy
mgmt = graph.openManagement()
```

The example below uses the Management API to get a list of all the vertex labels currently defined by in graph.

```groovy
mgmt.getVertexLabels()

version
airport
country
continent
```

This query finds all of the currently defined edge labels.

```groovy
mgmt.getRelationTypes(EdgeLabel.class)

route
contains
```

This query will find all of the currently defined property keys. Note that this list will include both vertex and edge property key names

```groovy
mgmt.getRelationTypes(PropertyKey.class)

dist
code
type
desc
country
longest
city
elev
icao
lon
region
runways
lat
```

We can query the cardinality of a property.

```groovy
mgmt.getPropertyKey('code').cardinality()

SINGLE
```

Note that as we have not so far defined a schema for the air-routes graph. if we query the dataType for any of the already loaded properties we will get back *Object.class* and by default that is what Janus will use in the absence of a schema having been defined.

```groovy
mgmt.getPropertyKey('code').dataType()

Object.class
```

We can also test for the existence of a label definition in the graph.

```groovy
mgmt.containsEdgeLabel('route')

true

mgmt.containsEdgeLabel('travels')

false
```

6.3.7. Defining a Janus Graph schema for air-routes

You are not required to define the types and labels of your edges, vertices and properties ahead of time but it is strongly recommended that you do so. If you do not define anything and load the air routes data for example, it will work fine but Janus Graph will make assumptions about various things. One thing it will do is default the type of all property keys to Java's *Object.class* which is not ideal if you want the graph to help you enforce stricter type checking. Also, without a schema being defined, Janus Graph will default the usage constraint or *multiplicity* setting on all edges to *MULTI*. We will explain what that means in a minute but in essence it means there is no restriction by default on how many edges with the same label that can exist between two vertices.

You can use the Management API do define your schema. You can add additional property types at any time but once defined you cannot change their types. The only thing you can do once they have been created is to change the names of the keys.

A best practice when working with Janus Graph is to define your labels and property types before you load any data into the graph. As the graph grows if you find you need to add additional property types or labels you are allowed to do that.

Using the management API you can define the labels that will be used by vertices and edges. These values must be unique across the graph. You can also define the type and cardinality (SINGLE, LIST or SET) of each property key and for edges you can specify the allowed usage of edges for any given label (MULTI, MANY2ONE, ONE2MANY, ONE2ONE or SIMPLE). Property key names must also be unique across the graph.

Before we can define a schema for our edge labels we need to understand what each option allows and decide on the best fit for each of our edge types.

The multiplicity options provide the following constraints:

MULTI

- This is the default option if no multiplicity has been defined for an edge with a given label. This setting permits multiple edges of the same label between any pair of vertices. The air-routes graph uses a multiplicity of MULTI for the *routes* edges between airports.

SIMPLE

- This setting permits at most one edge of a given label between any pair of vertices. In the air-routes graph this setting is used for the edges between a continent and an airport as an airport cannot be in more than one continent. The same is used for the edges between airports and countries.

MANY2ONE

- This setting permits at most one outgoing edge of a given label name from any vertex in the graph but places no constraint on the number of incoming edges with this label.

ONE2MANY

- This setting permits at most one incoming edge of a given label to any vertex in the graph but places no constraint on the number of outgoing edges

ONE2ONE

- This setting permits at most one incoming and one outgoing edge of a given label to and from any vertex in the graph.

Let's look at how we can use the Janus Graph Management API to specify the multiplicty for the *route* and *contains* edges used by the air-routes graph.

```groovy
// Define edge labels and usage
mgmt = graph.openManagement()
mgmt.makeEdgeLabel('route').multiplicity(MULTI).make()
mgmt.makeEdgeLabel('contains').multiplicity(SIMPLE).make()
mgmt.commit()
```

To be written

```groovy
// Define vertex labels
mgmt = graph.openManagement()
mgmt.makeVertexLabel('version').make()
mgmt.makeVertexLabel('airport').make()
mgmt.makeVertexLabel('country').make()
mgmt.makeVertexLabel('continent').make()
mgmt.commit()
```

To be written

```groovy
// Define vertex property keys
mgmt = graph.openManagement()
mgmt.makePropertyKey('code').dataType(String.class).cardinality(Cardinality.SINGLE).make()
mgmt.makePropertyKey('icao').dataType(String.class).cardinality(Cardinality.SINGLE).make()
mgmt.makePropertyKey('type').dataType(String.class).cardinality(Cardinality.SINGLE).make()
mgmt.makePropertyKey('city').dataType(String.class).cardinality(Cardinality.SINGLE).make()
mgmt.makePropertyKey('country').dataType(String.class).cardinality(Cardinality.SINGLE).mak
e()
mgmt.makePropertyKey('region').dataType(String.class).cardinality(Cardinality.SINGLE).make
()
mgmt.makePropertyKey('desc').dataType(String.class).cardinality(Cardinality.SINGLE).make()
mgmt.makePropertyKey('runways').dataType(Integer.class).cardinality(Cardinality.SINGLE).ma
ke()
mgmt.makePropertyKey('elev').dataType(Integer.class).cardinality(Cardinality.SINGLE).make(
)
mgmt.makePropertyKey('lat').dataType(Double.class).cardinality(Cardinality.SINGLE).make()
mgmt.makePropertyKey('lon').dataType(Double.class).cardinality(Cardinality.SINGLE).make()
mgmt.commit()
```

To be written

```groovy
// Define edge property keys
mgmt = graph.openManagement()
mgmt.makePropertyKey('dist').dataType(Integer.class).cardinality(Cardinality.SINGLE).make(
)
mgmt.commit()
```

To be written

```groovy
// Load the air routes graph
graph.io(graphml()).readGraph('air-routes.graphml')
graph.tx().commit()
```

To be written

```groovy
// Look at the properties
mgmt = graph.openManagement()
types = mgmt.getRelationTypes(PropertyKey.class)
types.each{println "$it\t: " +
                   mgmt.getPropertyKey("$it").dataType() +
                   " " + mgmt.getPropertyKey("$it").cardinality()}

mgmt.commit()
```

To be written

```
lat     : class java.lang.Double SINGLE
lon     : class java.lang.Double SINGLE
dist    : class java.lang.Integer SINGLE
longest : class java.lang.Object SINGLE
code    : class java.lang.String SINGLE
icao    : class java.lang.String SINGLE
type    : class java.lang.String SINGLE
city    : class java.lang.String SINGLE
country : class java.lang.String SINGLE
region  : class java.lang.String SINGLE
desc    : class java.lang.String SINGLE
runways : class java.lang.Integer SINGLE
elev    : class java.lang.Integer SINGLE
```
GROOVY

6.3.8. Creating a property with cardinality LIST

To be written

```
mgmt = graph.openManagement()
maker = mgmt.makePropertyKey('mylist')
maker.dataType(String.class)
maker.cardinality(LIST)
maker.make()
mgmt.commit()

mgmt = graph.openManagement()
mgmt.makePropertyKey('anotherlist').dataType(String.class).cardinality(LIST).make()

mgmt = graph.openManagement()
mgmt.getPropertyKey('mylist').cardinality()

LIST

mgmt.commit()

n = g.addV('mylist','one','mylist','two').next()

v[3043568]

g.V(n).property(list,'mylist','one')
g.V(n).valueMap()

// Duplicates allowed
[mylist:[one,one,two]]

graph.tx().commit()
```
GROOVY

6.3.9. Creating a property with cardinality SET

To be written

```groovy
mgmt = graph.openManagement()
mgmt.makePropertyKey('numbers').dataType(Integer.class).cardinality(SET).make()
mgmt.getPropertyKey('numbers').cardinality()

SET

mgmt.commit()

n = g.addV('numbers',1,'numbers',2,'numbers',3).next()

v[2846792]

g.V(n).valueMap()

[numbers:[1,2,3]]

g.V(n).property(set,'numbers',2)
g.V(n).valueMap()

// Duplicates not allowed
[numbers:[1,2,3]]

g.V(n).property(set,'numbers',4)
g.V(n).valueMap()

[numbers:[1,2,3,4]]

graph.tx().commit()
```

6.3.10. The Janus Graph geospatial API

To be written

```groovy
lon = g.V().has('code','LHR').values('lon').next()
lat = g.V().has('code','LHR').values('lat').next()

boundary = Geoshape.circle(lat,lon,100)

g.V().hasLabel('airport').
    where(map{a=it.get().value('lat');
              b=it.get().value('lon');
              Geoshape.point(a,b).within(boundary)}.is(true)).
              valueMap('code','lat','lon')
```

```
[code:[LGW],lon:[-0.190277993679047],lat:[51.1481018066406]]
[code:[BZZ],lon:[-1.58361995220184],lat:[51.749964]]
[code:[STN],lon:[0.234999999404],lat:[51.8849983215]]
[code:[LTN],lon:[-0.368333011865616],lat:[51.874698638916]]
[code:[SOU],lon:[-1.35679996013641],lat:[50.9502983093262]]
[code:[LHR],lon:[-0.461941003799],lat:[51.4706001282]]
[code:[LCY],lon:[0.055278],lat:[51.505278]]
[code:[SEN],lon:[0.695555984973907],lat:[51.57139968872O7]]
```

GROOVY

6.4. Choosing a persistent storage technology for Janus Graph

To be written

6.4.1. Berkley DB

To be written

6.4.2. Apache Cassandra

To be written

6.4.3. Apache HBase

To be written

6.5. Using an external index with Janus Graph

To be written

6.5.1. Apache Elastic Search

To be written

6.5.2. Apache Solr

To be written

6.5.3. Using the Management API to control the index

```
mgmt = graph.openManagement()
mgmt.buildIndex('code',Vertex.class)
```

GROOVY

```
vindex = mgmt.getGraphIndexes(Vertex.class)

mgmt.updateIndex(vindx.get(0), SchemaAction.REINDEX).get()

mgmt.commit()
```

GROOVY

To be written

6.6. Using Gremlin Server

To be written

6.6.1. Connecting the Gremlin Console to a remote graph

To be written

6.6.2. Using Janus Graph and Gremlin from a Java program

To be written

6.6.3. Connecting to a Gremlin Server from the command line

To be written

6.6.4. Examples of the JSON returned from a Gremlin Server

```groovy
g.V().has('code','CDG')
```


```groovy
[{'id': 40972336,
  'label': 'airport',
  'properties': {'city': [{'id': 'oeadi-oe6gg-745', 'value': 'Paris'}],
   'code': [{'id': 'oe9l2-oe6gg-sl', 'value': 'CDG'}],
   'country': [{'id': 'oe96u-oe6gg-5j9', 'value': 'FR'}],
   'desc': [{'id': 'oedja-oe6gg-3yd', 'value': 'Paris Charles de Gaulle'}],
   'elev': [{'id': 'oearq-oe6gg-7wl', 'value': 392}],
   'icao': [{'id': 'oeb5y-oe6gg-8p1', 'value': 'LFPG'}],
   'lat': [{'id': 'oed52-oe6gg-but', 'value': 49.0127983093}],
   'lon': [{'id': 'oebk6-oe6gg-9hh', 'value': 2.54999995232}],
   'longest': [{'id': 'oe9za-oe6gg-6bp', 'value': 13829}],
   'region': [{'id': 'oeccm-oe6gg-a9x', 'value': 'FR-J'}],
   'runways': [{'id': 'oecqu-oe6gg-b2d', 'value': 4}],
   'type': [{'id': 'oebye-oe6gg-35x', 'value': 'airport'}]},
  'type': 'vertex'}]
```


```groovy
g.V().has('type','airport').or(has('region','US-TX'),has('region','US-
OK')).values('code').order()
```

```JSON
{"requestId":"8602665d-e1c0-488f-b8a1-3343c3a1c0b6"
,"status":{"message":""
          ,"code":200
          ,"attributes":{}}
,"result":{"data":["ABI","ACT","AMA","AUS","BPT","BRO"
                  ,"CLL","CRP","DAL","DFW","ELP","GGG"
                  ,"GRK","HOU","HRL","IAH","LAW","LBB"
                  ,"LRD","MAF","MFE","OKC","SAT","SJT"
                  ,"SPS","SWO","TUL","TYR","VCT"]
          ,"meta":{}
}
```

6.6.5. Tweaking queries to make the JSON returned easier to work with

Below is a query that we have seen used earlier in this document. It finds all routes longer than 8,000 miles and returns the airport pairs and the distance between them.

```GROOVY
g.V().as('a').outE().has('dist',gt(8000)).order().by('dist',decr).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 get back results that have been to a degree *pretty printed* by the Console as shown below.

```GROOVY
[AKL,9025,DOH]    [DXB,8150,IAH]
[AKL,8818,DXB]    [AUH,8139,SFO]
[DFW,8574,SYD]    [DFW,8105,HKG]
[ATL,8434,JNB]    [DXB,8085,SFO]
[SFO,8433,SIN]    [HKG,8054,JFK]
[AUH,8372,LAX]    [AUH,8053,DFW]
[DXB,8321,LAX]    [EWR,8047,HKG]
[JED,8314,LAX]    [DOH,8030,IAH]
[DOH,8287,LAX]    [DFW,8022,DXB]
[LAX,8246,RUH]
```

However, if you were to use a system that returns the full JSON response, for example when using Gremlin Server or talking to a hosted graph like IBM Graph, and not get the benefit of the *pretty printing* that the Gremlin Console does using TinkerGraph you will get back something that looks a lot like this from the exact same query as the one we used above.

```
[{'labels': [['a'], [], ['b']], 'objects': ['AKL', 9025, 'DOH']},
 {'labels': [['a'], [], ['b']], 'objects': ['AKL', 8818, 'DXB']},
 {'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': ['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']}]
```

What is being returned is useful in some cases, for examples 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 (probably using a JSON helper class) 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',decr).inV().as('b') \
     .filter(select('a','b').by('code').where('a', lt('b'))) \
     .path().by('code').by('dist').toList().toString()
```

When we run the query, we now get back something that looks a lot more like what we got back when working with our local TinkerGraph and is certainly a bit easier to process in your application. 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.

```
['[[AKL, 9025, DOH], [AKL, 8818, DXB], [DFW, 8574, SYD], [ATL, 8434, JNB],
[SFO, 8433, SIN], [AUH, 8372, LAX], [DXB, 8321, LAX], [JED, 8314, LAX], [DOH,
8287, LAX], [LAX, 8246, RUH], [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]]']
```

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 I could not just do a simple *split(",")* as that would not have returned what I wanted. To make the split work, I replaced every occurence of *]*, in the string with *]x* and then did the split using *split("x")*. Here is the modified query.

```groovy
g.V().as('a').outE().has('dist',gt(8000)).order().by('dist',decr).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.

```groovy
[' [AKL, 9025, DOH]',
 ' [AKL, 8818, DXB]',
 ' [DFW, 8574, SYD]',
 ' [ATL, 8434, JNB]',
 ' [SFO, 8433, SIN]',
 ' [AUH, 8372, LAX]',
 ' [DXB, 8321, LAX]',
 ' [JED, 8314, LAX]',
 ' [DOH, 8287, LAX]',
 ' [LAX, 8246, RUH]',
 ' [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.

6.7. Using Gremlin within a Python notebook

To be written

6.8. Putting it all into a Docker Container

To be written

7. COMMON GRAPH SERIALIZATION FORMATS

To be written

7.1. Comma Separated Values (CSV)

This section is still under construction!

There are a number of ways that a graph can be stored using CSV files. There is no single preferred format that I am 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 of the node data and the other will contain all of the edge data.

7.1.1. 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 I have not included every property (or indeed every airport) in this example. Notice how each node has a unique ID assigned. This is important as when we define the edges we will need the node 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. I did not include an edge ID as we typically let the graph system assign those. For completeness I 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 clearly 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.

I often find myself writing Ruby or Groovy scripts that can generate CSV or GraphML files so that a graph system can ingest them. In some cases I 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.

I 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.

7.1.2. Adjacency matrix format

The examples shown above of how a CSV file can be used to store data about vertices and edges presents 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 node 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 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
```

7.1.3. 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 of the other vertices that this node 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 node and its properties are listed first, followed by all of the other vertices it connects to and the properties for those edges.

Some graph database systems actually store their graphs to disk using a variation of this format. Janus Graph in fact uses a system a lot like this when storing node and edge data to its persistent store.

7.1.4. 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 very simple 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.

```groovy
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]
```

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.

```groovy
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
```

7.1.5. Exporting relational database data to a graph using CSV format

To be written

7.2. GraphML

To be written

```xml
<?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>
```

7.3. GraphSON

To be written

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

g.addV(label,'airport','code','AUS').as('aus').
  addV(label,'airport','code','DFW').as('dfw').
  addV(label,'airport','code','LAX').as('lax').
  addV(label,'airport','code','JFK').as('jfk').
  addV(label,'airport','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')
```

7.3.1. Adjacency list format GraphSON

To be written

```json
{"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}]}, ... }
```

```JSON
{
    "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"
        }]
    }
}
```


```JSON
{"id":197,"label":"airport","inE":{"contains":[{"id":46566,"outV":3378},
{"id":49931,"outV":3608}],"route":[{"id":9524,"outV":55,"properties":{"dist":520}},
{"id":9753,"outV":57,"properties":{"dist":903}},{"id":22158,"outV":231,"properties":
{"dist":1036}}]},"outE":{"route":[{"id":20448,"inV":231,"properties":{"dist":1036}},
{"id":20446,"inV":55,"properties":{"dist":520}},{"id":20447,"inV":57,"properties":
{"dist":903}}]},"properties":{"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":153.091003418}],"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"}]}}
```


7.3.2. Wrapped adjacency list format GraphSON

To be written

JSON


```
{
    "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"
            }]
        }
    }]
}
```

7.4. Other formats you may encounter

To be written

- GML
- GEXF
- etc

8. FURTHER READING

Apache TinkerPop Getting Started guide

http://tinkerpop.apache.org/docs/current/tutorials/getting-started/

Official Apache TinkerPop home page and Gremlin downloads

http://tinkerpop.apache.org/

Current Apache TinkerPop documention

http://tinkerpop.apache.org/docs/current/reference/

Apache TinkerPop JavaDoc API reference material

http://tinkerpop.apache.org/javadocs/current/core/

http://tinkerpop.apache.org/javadocs/current/full/

Useful Gremlin "recipies"

http://tinkerpop.apache.org/docs/current/recipes/

Public mailing list for Gremlin users

https://groups.google.com/forum/#!forum/gremlin-users

Places where you can experiment with graphs in a browser

http://www.graphstory.com/

http://www.graphenedb.com/

http://www.gremlinbin.com

Janus Graph home page

http://janusgraph.org/

Janus Graph overview documentation

http://docs.janusgraph.org/latest/

Janus Graph API documentation

https://javadoc.io/doc/org.janusgraph/janusgraph-core/0.1.1

Public mailing list for Janus Graph users

https://groups.google.com/forum/#!forum/janusgraph-users

Version 266
Last updated 2017-10-17 16:35:03 CDT