O'REILLY®

Graph-Powered Analytics and Machine Learning with TigerGraph

Driving Business Outcomes with Connected Data

Early Release
Raw & Unedited

Sponsored by TigerGraph

Victor Lee,
Phuc Kien Nguyen
& Xinyu Chang

Graph-Powered Analytics and Machine Learning with TigerGraph

Driving Business Outcomes with Connected Data

With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

Victor Lee, PhD, Phuc Kien Nguyen, and Xinyu Chang

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

Graph-Powered Analytics and Machine Learning with TigerGraph
by Victor Lee, Phuc Kien Nguyen, and Xinyu Chang

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Rebecca Novack
Development Editor: Gary O'Brien
Production Editor: Jonathon Owen
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Kate Dullea

Revision History for the Early Release

2021-04-12: First Release
2021-09-29: Second Release
2022-02-22: Third Release
2022-04-21: Fourth Release
2022-05-19: Fifth Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781098106652* for release details.

978-1-098-10658-4

[LSI]

Table of Contents

1. Connect and Explore Data. 9
Graph Structure 10
Graph Terminology 10
Graph Schemas 15
Traversing a Graph 17
Hops and Distance 17
Breadth and Depth 18
Graph Modeling 19
Schema Options and Tradeoffs 19
Transforming Tables in a Graph 23
Model Evolution 25
Graph Power 26
Connecting the Dots 26
The 360 View 27
Looking Deep for More Insight 28
Seeing and Finding Patterns 30
Matching and Merging 31
Weighing and Predicting 33
Chapter Summary 34

2. See Your Customers and Business Better: 360 Graphs 37
Case 1: Tracing and Analyzing Customer Journeys 38
Solution: Customer 360 + Journey Graph 38
Implementing the C360 + Journey Graph: A GraphStudio Tutorial 41
Create a TigerGraph Cloud Account 41
Get and Install the Customer 360 Starter Kit 42
An Overview of GraphStudio 44
Design a Graph Schema 45

Data Loading 48
Queries and Analytics 49
Case 2: Analyzing Drug Adverse Reactions 58
Solution: Drug Interaction 360 Graph 59
Implementation 59
Graph Schema 60
Queries and Analytics 61
Chapter Summary 67

3. Studying Startup Investments. 69
Goal: Find promising startups 70
Solution: A Startup Investment Graph 70
Implementing A Startup Investment Graph and Queries 72
The Crunchbase Starter Kit 72
Graph Schema 72
Queries and Analytics 74
Chapter Summary 91

4. Detecting Fraud and Money Laundering Patterns. 93
Goal: Detect Money Laundering 94
Solution: Modeling Financial Crimes as Network Patterns 94
Implementing Financial Crime Pattern Searches 95
The Fraud and Money Laundering Detection Starter Kit 95
Graph Schema 95
Queries and Analytics 97
Chapter Summary 109

5. Graph-Powered Machine Learning Methods. 111
Unsupervised Learning with Graph Algorithms 113
Finding Communities 114
Finding Similar Things 117
Finding Frequent Patterns 127
Summary 128
Extracting Graph Features 129
Domain-Independent Features 129
Domain-Dependent Features 137
Graph Embeddings: A Whole New World 139
Summary 150
Graph Neural Networks 151
Graph Convolutional Networks 151
GraphSAGE 156
Summary 158

Comparing Graph Machine Learning Approaches 159
Use Cases for Machine Learning Tasks 159
Graph-based Learning Methods for Machine Learning Tasks 160
Graph Neural Networks: Summary and Uses 160
Chapter Summary 161

6. Entity Resolution Revisited. 163
Goal: Identify Real-World Users and Their Tastes 164
Solution Design 165
Implementation 167
Starter Kit 167
Graph Model 168
Data Loading 171
Queries and Analytics 171
Method 1: Jaccard Similarity 173
Method 2: Scoring Exact and Approximate Matches 185
Chapter Summary 193

CHAPTER 1

Connect and Explore Data

> **A Note for Early Release Readers**
>
> With Early Release ebooks, you get books in their earliest form—the authors' raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.
>
> This will be the 2nd chapter of the final book. Please note that the GitHub repo will be made active later on.

In Chapter 1, we showed the potential of graph analytics and machine learning applied to human and business endeavors, and we proposed to present the details in three stages: the power of connected data, the power of graph analytics, and the power of graph machine learning. In this chapter, we will take a deep dive into the first stage: the power of connected data.

Before we delve into the power of connected data, we need to lay some groundwork. We start by introducing the concepts and nomenclature of the graph data model. If you are already familiar with graphs, you may want to skim this section to check that we're on the same page with regard to terminology. Besides graphs themselves, we'll cover the important concepts of a graph schema and traversing a graph. Traversal is how we search for data and connections in a graph.

And along the way we talk about the differences between graph and relational databases and how we can ask questions and solve problems with graph analytics that would not be feasible in a relational database.

From that foundational understanding of what is a graph, we move on to present examples of the *power* of a graph by illustrating six ways that graph data provides you with more insight and more analytical capability than tabular data.

After completing this chapter, you should be able to:

1. Use the standard terminology for describing graphs
2. Know the difference between a graph schema and a graph instance
3. Create a basic graph model or schema from scratch or from a relational database model
4. Apply the "traversal" metaphor for searching and exploring graph data
5. Understand six ways that graph data empowers your knowledge and analytics
6. State the entity resolution problem and show how graphs resolve this problem

Graph Structure

In Chapter 1, we introduced you to the basic idea of a graph. In this section, we are going to go deeper. First we will establish the terminology that we will be using for the rest of this book. Then we will talk more about the idea of a graph schema, which is the key to having a plan and awareness of your data's structure.

Graph Terminology

Suppose you're organizing data about movies, actors, and directors. Maybe you work for Netflix or one of the other streaming services, or maybe you're just a fan.

Let's start with one movie, Star Wars: A New Hope, its three main actors and its director. If you were building this in a relational database, you could record this information in a single table, but the table would grow quickly and rapidly become unwieldy. How would we even record details about a movie, the fact that 50 actors appeared in it, and the details of each of those actor's careers, all in one table?

Best practice for the design of relational databases would suggest putting actors, movies and directors each into a separate table, but that would mean also adding in cross-reference tables to handle the many-to-many relationships between actors and movies and between movies and directors.

So in total you'd need five tables just to represent this example in a relational database, as in Figure 1-1.

Separating different types of things into different tables is the right answer for organizing the data, but to see how one record relates to another we have to rejoin the data. A query asking which actors worked with which directors would involve building a temporary table in memory called a join table that includes all possible combinations of rows across all the tables you've called which satisfy the conditions of the query. Join tables are expensive in terms of memory and processor time.

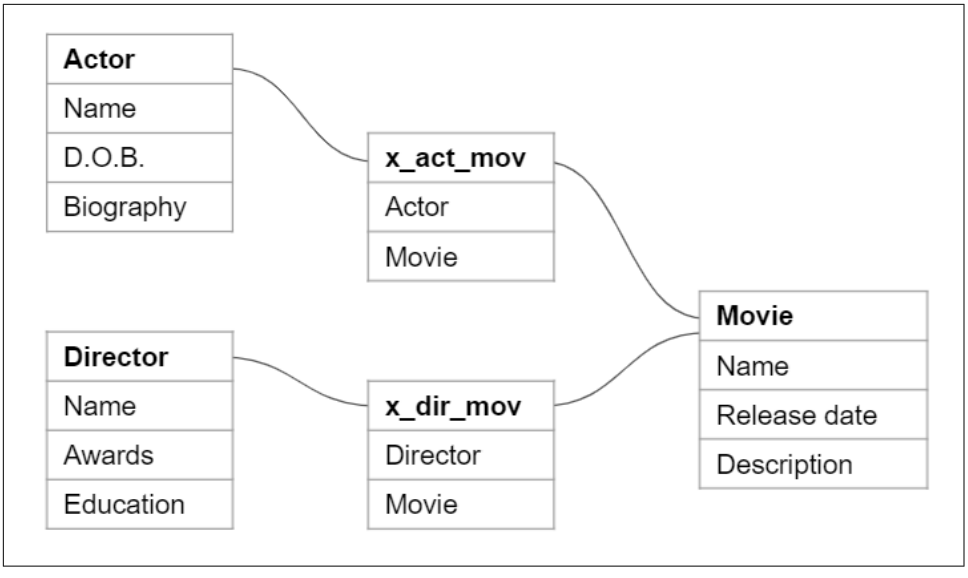


Figure 1-1. Diagram of relational tables for a simple movie database

Actor	Movie	Director
Mark Hamill	Star Wars	George Lucas
Carrie Fisher	Star Wars	George Lucas
Harrison Ford	Star Wars	George Lucas

Figure 1-2. Temporary table created from relational database query showing how three actors are linked to George Lucas via the movie Star Wars

As we can see from Figure 1-2, there is a lot of redundant data in this table join. For very large or complex databases, you would want to think of ways to structure the data and your queries to optimize the join tables.

However, if we compare that to the graph approach, as shown in Figure 1-3, one thing becomes immediately clear: The difference between a table and graph is that a graph can directly show how one data element is related to another. That is, the relationships between the data points are built into the database and don't have to be constructed at run-time. So one of the key differences between a graph and relational database is that in a graph database, the relationships between data points are *explicit*.

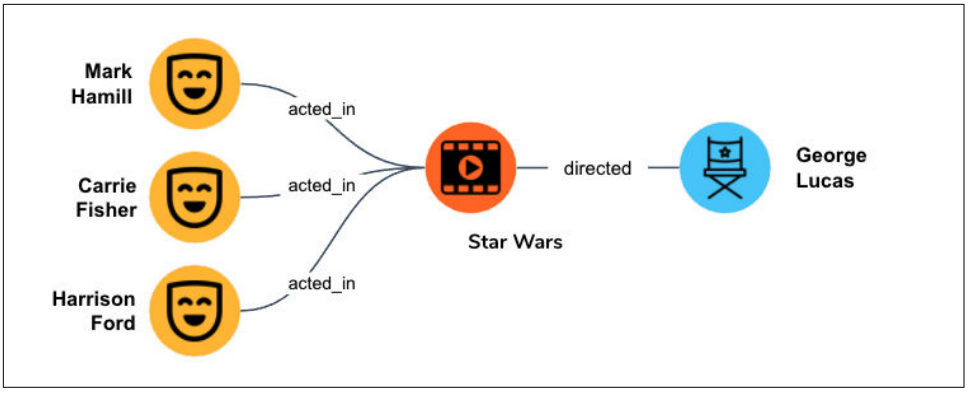


Figure 1-3. Graph showing our basic information about Star Wars

Each actor, movie, and director is called a *node* or a *vertex* (plural: *vertices*). Vertices represent things, physical or abstract. In our example, the graph has five vertices which describe the relationships between the vertices. The connections between vertices are called *edges*. Edges are also considered data elements. This graph has four edges: three for actors showing how they are related to a movie (`acted_in`) and one for a director showing their relationship to a movie (`directed`). In its simplest form, a *graph* is a collection of vertices and edges. We will use the general term *object* to refer to either a vertex or an edge.

With this graph, we can answer a basic question: what actors have worked with the director George Lucas? Starting from George Lucas, we look at the movies he directed which include Star Wars and then we look at the actors in that movie which include Mark Hamill, Carrie Fisher and Harrison Ford.

It can be useful or even necessary to distinguish the direction of an edge. In a graph database, an edge can be *directed* or *undirected*. A *directed edge* has a specific directionality, going from a source vertex to a target vertex. We draw directed edges as arrows.

By adding a directed edge, we can also show hierarchy, that is, *The Empire Strikes Back* was the sequel to *Star Wars* (Figure 1-4).

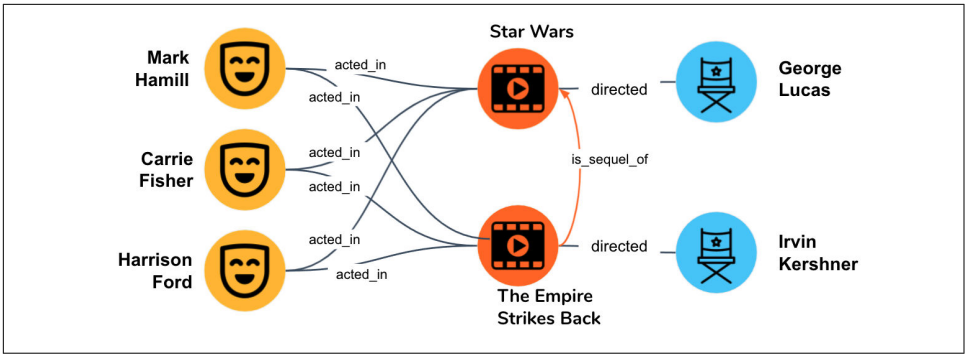


Figure 1-4. Multi-movie graph with a directed edge. This shows how we begin to build up the database with additional movies and production personnel. Note the directed edge, is_sequel_of, which provides the context to show that Empire was the sequel to Star Wars and not vice versa.

To do more useful work with a graph, however, we will want to add more details about each vertex or edge, such as an actor's birthdate or a movie's genre.

This book describes property graphs. A *property graph* is a graph where each vertex and each edge can have properties which provide the details about individual elements. If we look again at relational databases, properties are like the columns in a table. Properties are what make graphs truly useful. They add richness and context to data which enable us to develop more nuanced queries to extract just the data that we need. Figure 1-5 shows the Star Wars graph with some added features.

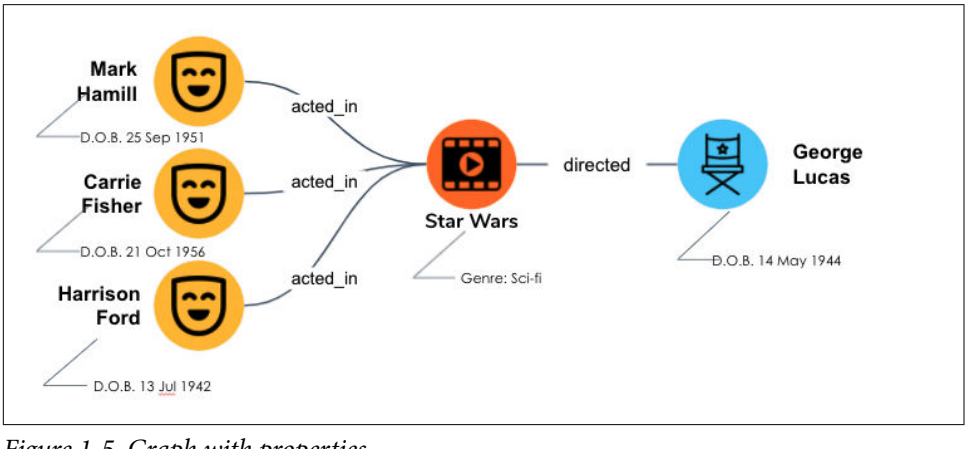


Figure 1-5. Graph with properties

Graphs offer us another choice for modeling properties. Instead of treating genre as a property of movies, we could make each genre a separate vertex. Why do this? When the property is categorical, then we expect lots of other vertices to have the same

property value (e.g., there are lots of sci-fi movies). All the sci-fi movies will link to the Sci-fi vertex, making it incredibly easy to search them or to collect statistics about them, such as, what was the top grossing sci-fi movie? All the non-sci-fi movies have already been filtered out for you. Either way, the additional data allows us to refine our queries to find just the information we need.

In our movie database example, we might want to create a new type of vertex called `Character` so we can show who played what role.

Figure 1-6 shows our Star Wars graph with the addition of `Character` vertices. The interesting thing about Darth Vader, of course, is that he was played by two people: David Prowse (in costume) and James Earl Jones (voice). Fortunately our database can represent this reality with a minimum of modification.

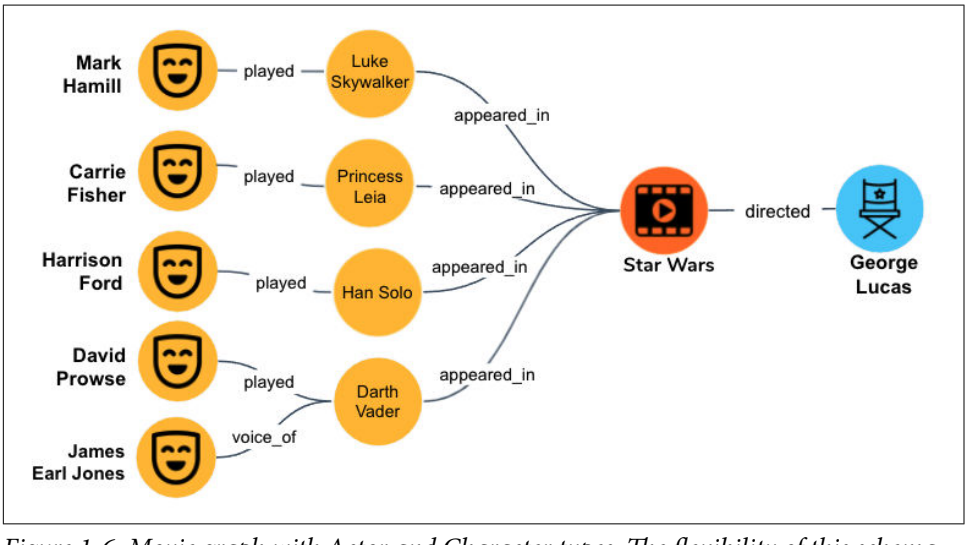


Figure 1-6. Movie graph with Actor and Character types. The flexibility of this schema enables us to easily show two actors portraying one character.

What else can we do with this graph? Well, it's flexible enough to allow us to add just about every person who was involved in the production of this movie from the director and actors to make-up artists, special effects artists, key grip and even best boy. Everyone who contributed to a movie could be linked using an edge called `worked_on` and an edge property called `role` which could include `director`, `actor`, `voice actor`, `camera operator`, `key grip` and so on.

If we then built up our database to include thousands of movies and everyone who had worked on them, we could use graph algorithms to answer questions like, which actors do certain directors like to work with most? With a graph database you can answer less obvious questions like who are the specialists in science fiction special

effects, or which lighting technicians do certain directors like to work with most? Interesting questions for companies that sell graphics software or lighting equipment.

With a graph database, you can connect to multiple data sources, extract just the data you need as vertices and run queries against the combined dataset. If you had access to a database of lighting equipment used on various movie projects, you could connect that to your movie database and use a graph query to ask which lighting technicians have experience with what equipment.

Table 1-1 summarizes the essential graph terminology we have introduced.

Table 1-1. Glossary of essential graph terminology

graph	A collection of vertices, edges, and properties used to represent connected data and support semantic queries.
vertex[a]	A graph object used to represent an object or thing. Plural: vertices.
edge	A graph object which links two vertices, often used to represent a relationship between two objects or things.
property	A variable associated with a vertex or edge, often used to describe it.
schema	A database plan comprising vertex and edge types and associated properties which will define the structure of the data.
directed edge / undirected edge	A directed edge represents a relationship with a clear semantic direction, from a source vertex to a destination vertex. An undirected edge represents a relationship in which no direction is implied.

[a] Another commonly used alternative name is node. It is a matter of personal preference. It's been proposed that the upcoming ISO standard query language for property graphs accept either VERTEX or NODE.

Graph Schemas

In the previous section, we intentionally started with a very simple graph and then added complexity, not only by adding more vertices, edges, and properties, but also by adding new *types* of vertices and edges. To model and manage a graph well, especially in a business setting, it's essential to plan out your data types and properties.

We call this plan a graph *schema*, or graph *data model*, analogous to the schema or entity-relationship model for a relational database. It defines the types of vertices and edges that our graph will contain as well as the properties associated with these objects.

You could make a graph without a schema by just adding arbitrary vertices and edges, but you'd quickly find it difficult to work with and difficult to make sense of. Also, if you wanted to search the data for all the movies, for example, it would be extremely helpful to know that they are all in fact referred to as "movie" and not "film" or "motion picture"!

It's also helpful to settle on a standard set of properties for each object type. If we know all movie vertices have the same core set of properties, such as title, genre, and release date, then we can easily and confidently perform analysis on those properties.

Figure 1-7 shows a possible schema for a movie graph database. It systematically handles several of the data complexities that arose as we talked about adding more and more movies to the database.

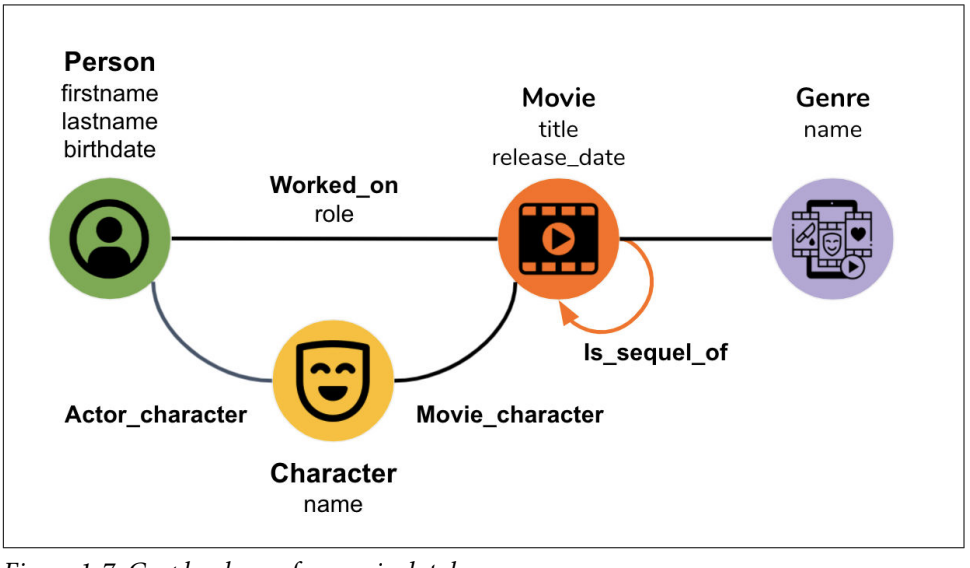


Figure 1-7. Graph schema for movie database

Let's run through the features of the schema:

- A `Person` vertex type represents a real-world person, such as *George Lucas*.
- The `Worked_on` edge type connects a `Person` to a `Movie`. It has a property to describe the person's role: *director, producer, actor, gaffer*, etc. By having the role as a property, we can support as many roles as we want with only one vertex type for persons and one edge type for working on a film. If a person had multiple roles, then the graph can have multiple edges[1]. Schemas only show one of each type of object.
- The `Character` vertex type is separate from the `Person` vertex type. One `Person` could portray multiple `Characters` (Tyler Perry in the *Madea* films), or multiple

1 Some graph databases would handle multiple roles by having a single Worked_on edge whose role property accepts a list of roles.

Persons could portray one `Character` (David Prowse, James Earl Jones, and Sebastian Shaw as Darth Vader in *The Return of the Jedi*).

- The `Movie` vertex type is straightforward.
- `Is_sequel_of` is a directed edge type, telling us that the source `Movie` is the sequel of the destination `Movie`.
- As noted before, we chose to model the genre of a movie as a vertex type instead of as a property, to make it easier to filter and analyze movies by genre.

The key to understanding schemas is that having a consistent set of object types makes your data easier to interpret.

Traversing a Graph

Traversing a graph is the fundamental metaphor for how a graph is searched and how the data is gathered and analyzed. Imagine the graph as a set of interconnecting stepping stone paths, where each stepping stone represents a vertex. There are one or more agents who are accessing the graph. To read or write a vertex, an agent must be standing on its stepping stone. From there, the agent may step or traverse across an edge to a neighboring stone/vertex. From its new location, the agent can then take another step. Remember: if two vertices are directly connected, it means there is a relationship between them, so traversing is following the chain of relationships.

Hops and Distance

Traversing one edge is also called making a *hop*. An analogy to traversing a graph is moving on a game board, like the one shown in Figure 1-8. A graph is an exotic game board, and you traverse the graph as you would move across the game board.

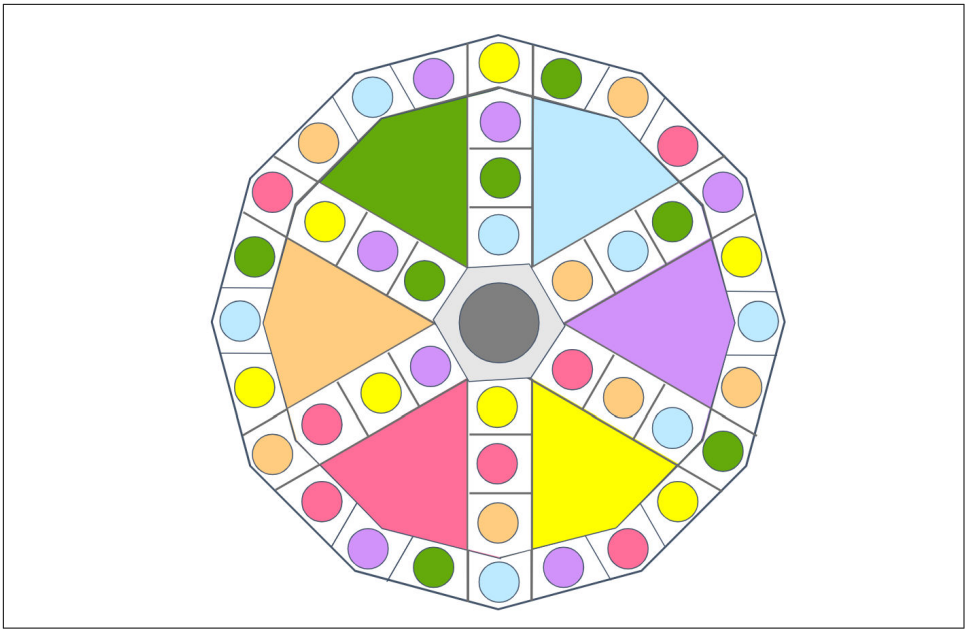


Figure 1-8. Traversing a graph is like moving on a game board

In many board games, when it is your turn, you roll a die to determine how many steps or hops to take. In other games, you may traverse the board until you reach a space of a certain type. This is exactly like traversing a graph in search of a particular vertex type.

Graph hops and distance come up in other real-world situations. You may have heard of "six degrees of separation." This refers to the belief that everyone in the U.S. is connected to everyone else through at most six hops of relationship. Or, if you use the LinkedIn business network app, you have probably seen that when you look at a person's profile, LinkedIn will tell you if they are connected to you directly (one hop), through two hops, or through three hops.

Breadth and Depth

There are two basic approaches to systematically traversing a graph to conduct a search. Breadth-first search (BFS) means visit each of your direct neighbors before continuing the search to the next level of neighbors, the next level, and so on. Graph databases with parallel processing can accelerate BFS by having multiple traversals take place at the same time.

Depth-First Search (DFS) means follow a single chain of connections as far as you can, before backtracking to try other paths. Both BFS and DFS will result in eventually visiting every vertex, unless you stop because you have found what you sought.

Graph Modeling

Now you know what is a graph and what is a graph schema. But how do you come up with a good graph model?

Start by asking yourself these questions:

- What are the key objects or entities that I care about?
- What are the key relationships that I care about?
- What are the key properties of entities you want to filter on?

Schema Options and Tradeoffs

As we have seen, good graph schema design represents data and relationships in a natural way that allows us to traverse vertices and edges as if they were real-world objects. As with any collection of real-world things, there are many ways we could organize our collection to optimize searching and extracting what we need.

In designing a graph database, two considerations that will influence the design are the format of our input data and our query use cases. And as we will see in this section, a key tradeoff is whether we want to optimize our schema to use less memory or make queries run faster.

Vertex, edge or property?

If you are converting tabular data into a graph, the natural thing seems to be to convert each table to a vertex type, and each table column in a vertex property. In fact, a column could map to a vertex, an edge, a property of a vertex or a property of an edge.

Entities and abstract concepts generally map to vertices, and you could think of this as a noun. Relationships generally map to edges and you can think of them as verbs. Descriptors are analogous to adjectives and adverbs and can map to vertex or edge properties depending on the context and your query use case.

At first glance it would appear that storing object attributes as close to the object as possible -- ie, as properties -- would deliver the most optimal solution. However, consider a use case in which you need to optimize your search for product color. Color is a quality that would usually be expected to be found as a property of a vertex, but then searching for blue objects would necessitate looking at every vertex.

In a graph, you can create a search index by defining a vertex type called `color` and linking the `color` vertex and the `product` vertex via an undirected edge. Then to find all *blue* objects, you simply start from the `color` vertex *blue* and find all linked `prod`

uct vertices. This speeds up query performance with the tradeoff being greater complexity and higher memory usage.

Edge direction

Earlier we introduced the concept of directionality in edges and noted that you can, in your design schema, define an edge type as directed or undirected. In this section we'll discuss the benefits and tradeoffs of each type. We'll also discuss a hybrid option available in the TigerGraph database.

This is so useful you might think you could use it all the time, but with all things computational, there are benefits and tradeoffs in your choice of edge type.

Undirected edge
: Links any two vertices of defined type with no directionality implied. The benefit is they are easy to work with when creating links and easy to traverse in either direction. For example, if users and email addresses are both vertex types, you can use an undirected edge to find someone's email but also find all the users who use that same email address, something you can't do with a directed edge.

  The tradeoff with an undirected edge is it does not give you contextual information such as hierarchy. If you have an enterprise graph and want to find the parent company, for example, you can't do this with undirected edges because there is no hierarchy. In this case you would need to use a directed edge.

Directed edge
: Represents a relationship with a clear semantic direction, from a source vertex to a destination vertex. The benefit to a directed edge is it gives you more contextual information. It is likely to be more efficient for the database to store and handle than an undirected edge. The tradeoff, however, is you can't trace backward should you need to.

Directed edge paired with a reverse directed edge
: You can have the benefits of directional semantics and traversing in either direction if you define two directed edge types, one for each direction. For example, to implement a family tree, you could define a `child_of` edge type to traverse down the tree and a `parent_of` edge type to traverse up the tree. The tradeoff, though, is you have to maintain two edge types: every time you insert or modify one edge, you need to insert or modify its partner. The TigerGraph database makes this easier by allowing you to define the two types together and to write data ingestion jobs that handle the two together.

As you can see, your choice of edge type will be influenced by the types of queries you need to run balanced against operational overheads such as memory, speed and coding.

If the source vertex and destination vertex types are different, such as `Person` and `Product`, you can usually settle for an undirected edge and let the vertex types provide the directional context. It's when the two vertex types are the same and you care about direction that you must use a directed edge.

Granularity of edge type

How many different edge types do you need and how can you optimize your use of edge types? In theory, you could have one edge type -- undirected -- that linked every type of vertex in your schema. The benefit would be simplicity -- only one edge type to remember! -- but the tradeoffs would be the number of edge properties you would need for context and slower query performance.

At the other extreme, you could have a different edge type for each type of relationship. For instance, in a social network, you could have separate edge types for `coworker`, `friend`, `parent_of`, `child_of`, and so on. This would be very efficient to traverse if you were looking for just one type of relationship, such as professional networks. The tradeoff is the need to define new edge types to represent new types of relationships and a loss of abstraction -- ie, an increase in complexity -- in your code.

Modeling Interaction Events

In many applications, we want to track interactions between entities, such as a financial transaction where one financial account transfers funds to another account. You might think of representing the transaction (transferring funds) as an edge between two `Account` vertices. If you have multiple occurrences, will you have multiple edges? While it seems easy to conceive of this (Figure 1-9), in the realms of both mathematical theory and real-world databases, this is not so straightforward.

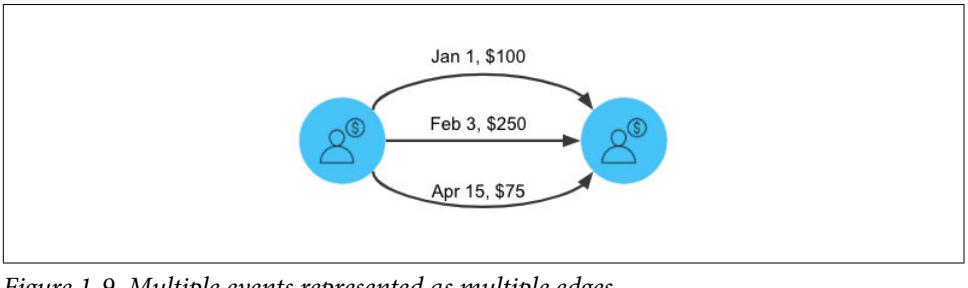


Figure 1-9. Multiple events represented as multiple edges

In mathematics, having multiple edges between a given pair of vertices goes beyond the definition of ordinary graphs into multi-edges and multigraphs. Due to this complexity, not all graph databases support this, or if they do, they don't have a convenient way to refer to a specific edge in the group. Another way to handle this is to model each interaction event as a vertex, and use edges to connect the event to the

participants (Figure 1-10[a]). Modeling an event as a vertex provides the greatest flexibility for linking it to other vertices and for designing analytics. A third way is to create a single edge between the two entities and aggregate all the transactions into an edge property (Figure 1-10[b]).

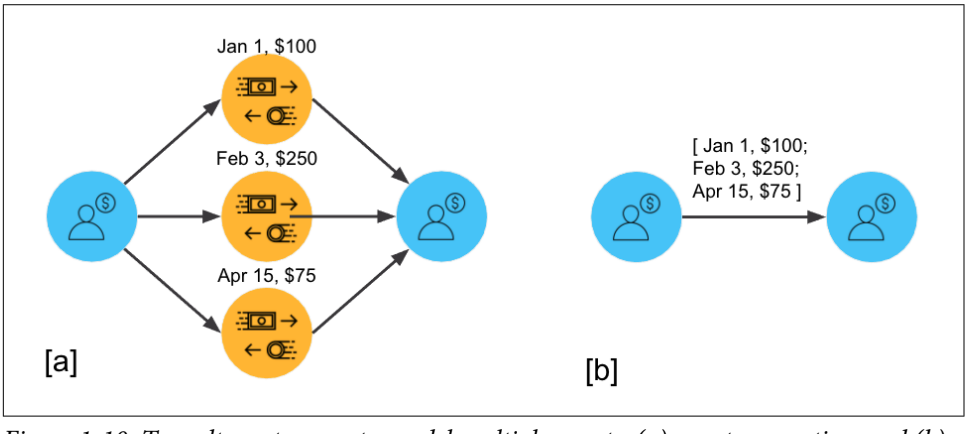


Figure 1-10. Two alternate ways to model multiple events: (a) events as vertices and (b) a single event edge with a property that contains a list of occurrences.

Table 1-2 summarizes the pros and cons of each approach. The simplest model is not always your best choice, because application requirements and database performance issues may be more important.

Table 1-2. Comparing options for modeling multiple occurrences of an interaction

Model	Benefit	Tradeoff
Multiple edges	Simple model	Database support is not universal
Vertex linked to related vertices	Filtering on vertex properties Ease of analytics including community and similarity of events Advanced search tree integration	Uses more memory Takes more steps to traverse
Single edge with property recording details of occurrences	Less memory usage Fewer steps to traverse between users	Searching on transactions less efficient Slower update/insert of the property

Adjusting your design schema based on use case

Suppose you are creating a graph database to track events in an IT network. We'll assume you would need these vertex types: event, server, IP, event type, user and device. But what relationships would you want to analyze and what edges would you need? The design would depend on what you wanted to focus on, and your schema could be event-centered or user-centered.

For the event-centered schema (Figure 1-11[a]), the key benefit is that all related data is just one hop away from the event vertex. This makes it straightforward to find

communities of events, find servers that processed the most events of a given type, and find the servers that were visited by any given IP. The tradeoff is that from a user perspective, the user is two hops away from a device or IP vertex.

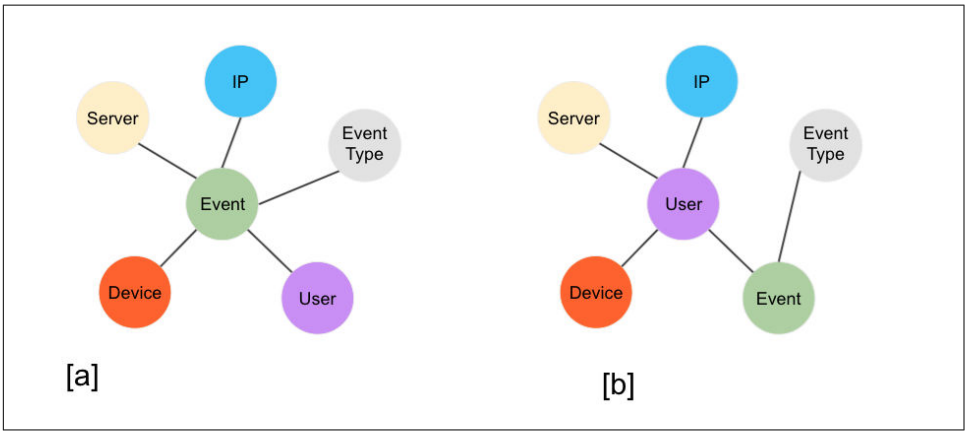


Figure 1-11. Two options for arranging the same vertex types: [a] event-centered, and [b] user-centered.

We can fix this by making our schema user-centered at the expense of separating events from IPs and servers by two hops and event types are separated from devices, servers and IPs by three hops (Figure 1-11[b]). However, these disadvantages might be worth the tradeoff of being able to do useful user-centered analysis such as detecting blacklisted users, finding whitelisted users that are similar to blacklisted users, and finding the paths between two users.

Transforming Tables in a Graph

You won't always create graph databases from scratch. Often, you'll be taking data that is already stored in tables and then moving or copying the data into a graph. But how should you reorganize the data into a graph?

Migrating data from a relational database into a graph database is a matter of mapping the tables and columns onto a graph database schema. To map data from a relational database to a graph database, we create a one-to-one correspondence between columns and graph objects. Table 1-3 outlines a simple example of mapping data from a relational database to a graph database for bank transaction data.

Table 1-3. Example of mapping tables in a relational database to vertices, edges and properties in a graph database

Source: Relational database	Destination: Graph database
Table: `Customers` – multiple columns including customer_id, first_name, last_name, DOB	Vertex type: `Customer` -- with corresponding properties of customer_id, first_name, last_name, DOB

Table: `Banks` – columns bank_id, bank_name, routing_code, address	Vertex type: `Bank` - properties bank_name, routing_code, address
Table: `Accounts` – columns bank_id, customer_id,	Vertex type: `Account` - properties bank_id, customer_id
Table: `Transactions` – columns source_account, destination_account, amount	Vertex type: `Transaction` - properties source_account, destination_account, amount OR Directed edge: `transaction` - properties source_account, destination_account, amount

The graph schema would be as shown in Figure 1-12.

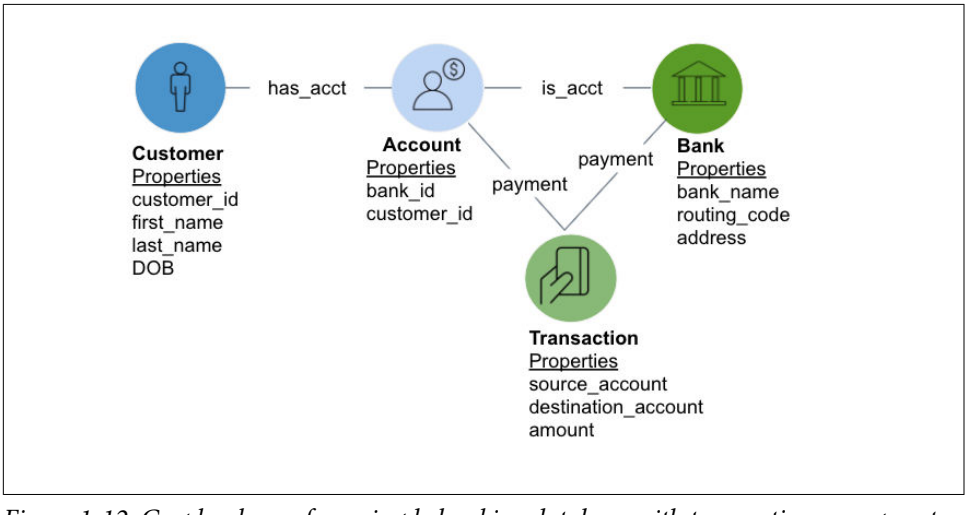


Figure 1-12. Graph schema for a simple banking database with transactions as separate vertices.

One of the key decisions in creating your data schema is deciding which columns need to be mapped to their own vertices. For instance, people are generally key to understanding any real-life situation -- whether they be customers, employees or others -- so they would generally map to their own vertices.

In theory, every column in a relational database could become a vertex in your schema, but this is unnecessary and would quickly become unwieldy. In the same way that you have to think about structuring a relational database, optimizing a graph database is about understanding the real-world structure of your data and how you intend to use it.

In a graph database, the key columns from your relational database become vertices and the contextual or supporting data become properties of those vertices. Edges generally map to foreign keys and cross reference tables.

Some graph databases have tools that facilitate the importing of tables and mapping of foreign keys to vertex and edge IDs.

As with a relational database, a well-structured graph database eliminates redundant or repetitive data. This not only ensures efficient use of computing resources but, perhaps more importantly, ensures the consistency of your data by ensuring that it doesn't exist in different forms in different locations.

Optimizing mapping choices

Simple mapping of columns to vertices and vertex properties works, but it may not take advantage of the richness of connections available in a graph, and in reality it is often necessary to adjust mapping choices based on differing search use cases.

For instance, in a graph database for a contacts database, mobile number and email address are properties of an individual person and are generally represented as properties of that vertex.

However, if you were trying to use a banking application to detect fraud, you might want to separate email addresses and telephone numbers out as separate vertices because they are useful in linking people and financial transactions.

It is not uncommon for information from multiple tables to map to one vertex or edge type. This is especially common when the data is coming from multiple sources, each of which provides a different perspective on the same real-world entities. Likewise, one table can map to more than one vertex and edge type.

Model Evolution

Most likely, your data is going to evolve over time, and you will need to adjust the schema to take account of new business structures and external factors. That's why schemas are designed to be flexible, to allow the system to be adapted over time without having to start from scratch.

If we look at the banking sector, for instance, financial institutions are constantly moving into new markets, either through geographical expansion or introducing new types of products.

As a simple example, let's assume we have a bank that's always operated in a single country. The country of origin for all its customers is therefore implicit. However, moving into a second country would require updating the database to include country data. One could either add a country property to every vertex type for which it was relevant or create a new vertex type called country and create vertices for each country in which the bank operated.

With a flexible schema, the schema can be updated by adding the new vertex type and then linking customer vertices to the new country vertex.

Although this is a simple example, it shows how modeling data can be an evolutionary process. You can start with an initial model, perhaps one that closely resembles a prior relational database model. After you use your graph database for a while, you may learn that some model changes would serve your needs better. Two common changes are converting a vertex property into an independent vertex type and adding additional edge types.

Adapting a graph to evolving data can be simple. Adding a property, a vertex type, or an edge type is easy. Connecting two different data sets is easy, as long as we know how they relate. We can add edges to connect related entities, and we can even merge entities from two sources which represent the same real-world entity.

Graph Power

We've now seen how to build a graph, but the most important question that needs to be answered is *why* build a graph. What are the advantages? What can a graph do for you that other data structures don't do as well? We call graph technology's collected capabilities and advantages *graph power*.

What follows are the key facets of graph power. We humbly admit that this is neither a complete nor the best possible list. We suspect that others have presented lists that are more complete and more precise in a mathematical sense. Our goal, however, is not to present theory but to make a very human connection: to take the ideas that resonate with us and to share them with you, so that you will understand and experience graph power on your own.

Connecting the Dots

A graph forms an actionable body of knowledge.

As we've seen, connecting the dots is graph power at its most fundamental level. Whether we are linking actors and directors to movies or financial transactions to suspected fraudsters, a graph lets you describe the relationship between one entity and another across multiple hops.

The power of graph comes from being able to describe a network of connections, detect patterns, and extract intelligence from those patterns. While individual vertices may not contain the intelligence we are looking for, taken together, we may discover patterns in the relationships between multiple vertices that reveal new information.

With this knowledge we can begin to infer and predict from the data, like a detective joining the dots in a murder investigation.

In every detective story, the investigator gathers a set of facts, possibilities, hints, and suspicions. But these isolated bits and pieces are not the answer. The detective's magic is to stitch these pieces together into a hidden truth. They might use the pattern of

known or suspected connections to predict relationships which they had not been given.

When the detective has solved the mystery, they can show a sequence or network of connections that connect the suspect to the crime, along with the means, opportunity, and motive. They can likewise show that a sufficiently robust sequence of connections does not exist for any other suspect.

Did those detectives know they were doing graph analytics? Probably not, but we all do it every day in different aspects of our lives. Whether that's work, family or our network of friends, we are constantly connecting the dots to understand connections between people and people, people and things, people and ideas, and so on.

The power of graph as a data paradigm is that it closely parallels this process, making the use of graph more intuitive.

The 360 View

A 360 graph view eliminates blind spots.

Organizations of all sizes bemoan their data silos. Each department expects the other to yield up their data on demand while at the same time failing to appreciate their own inability to be open on the same basis. The problem is that business processes and the systems that we have to support them actively work against this open sharing of data.

For instance, two departments may use two different data management systems. Although both may store their data in a relational database, the data schema for each is so alien to the other that there is little hope of linking the two to enable sharing.

The problem may not be obvious if you look at it at the micro scale. If for instance, you are compiling a record for customer X, an analyst with knowledge of the two systems in which customer data is stored will be able to easily extract the data from both, manually merge or reconcile the two records, and present a customer report. The problem comes when you want to replicate this a hundred thousand or a million times over.

And it's only by sharing the data in a holistic, integrated way that a business would be able to remove the blinders that prevent it from seeing the whole picture.

The term Customer 360 describes a data architecture in which customer data from multiple sources and domains is brought together into a single data set so that you have a comprehensive and holistic view of each customer.

Working with a relational database, the most obvious solution would be to merge these two departmental databases into one. Many businesses have tried grand data integration projects, but they usually end in tears because while merging data yields

considerable benefits, there are also considerable tradeoffs to be made that result in the loss of contextual nuance and functionality. Let's face it, there's usually a reason why the creators of a certain software package chose to construct their data schema in that particular way, and attempting to force it to conform to the schema of another system, or a new hybrid schema, will break at least one of the systems.

Graph allows you to connect databases in a natural, intuitive way without disturbing the original tables. Start by granting the graph application access to each database and then create a graph schema that links the data points from each database in a logical way. The graph database maps the relationships between the data points and does the analytical heavy lifting, leaving the source databases to carry on with what they were doing before.

If you want to see your full surroundings, you need a view that looks out across every angle – all 360 degrees. If you want to understand your full business or operational circumstances, you need data relationships across all the data you know is out there.

This is something we will look at in more depth in Chapter 3 where we will demonstrate a use case involving Customer Journey.

We have seen in the previous two points how to set up the data, and now in the next four points, we will look at how to extract meaningful intelligence from it.

Looking Deep for More Insight

Searching deep in a graph reveals vast amounts of connected information.

The "six degrees of separation" experiment conducted in the 1960s by Stanley Milgram demonstrated that just by following personal connections (and knowing that the target person is in Boston), randomly selected persons in Omaha, Nebraska could reach the mystery person through no more than six person-to-person connections.

Since then, more rigorous experiments have shown that many graphs are so-called small world graphs, meaning that the source vertex can reach millions and even billions of other vertices in a very small number of hops.

This vast reach in only a few hops occurs not only in social graphs but also in knowledge graphs. The ability to access this much information, and to understand how those facts relate to one another, is surely a super power.

Suppose you have a graph which has two types of vertices: persons and areas of expertise, like the one in Figure 1-13. The graph shows *who* do you know well and *what* do you know well. Each person's direct connections represent what is in their own head.

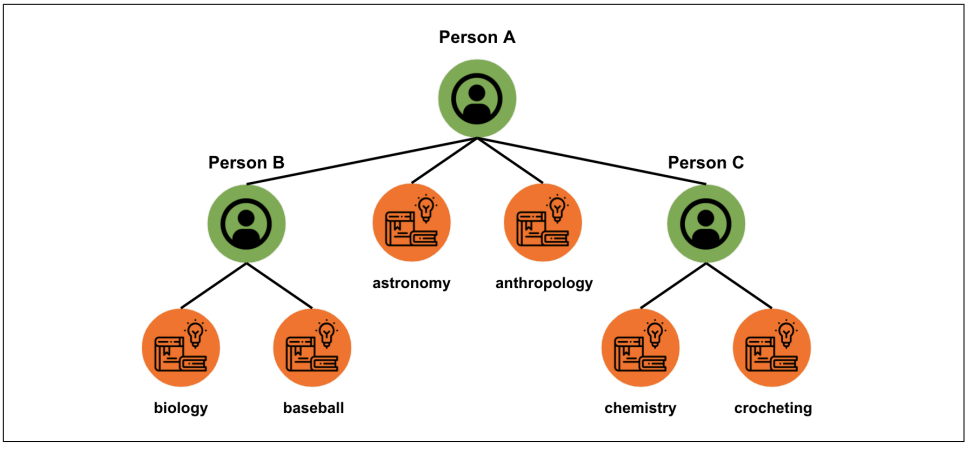


Figure 1-13. A graph showing who knows who and what each of them are experts in.

From this we can readily see that A is an expert in two topics, astronomy and anthropology, but by traversing one additional hop to ask B and C what do they know, A has access to four more specialties.

Now, suppose each person has 10 areas of expertise and 100 personal connections. Consider how many people and how many areas of expertise are reached by your friends' friends. There are 100 x 100 = 10,000 personal connections each with 10 areas of expertise. Chances are that that is not 10,000 unique persons – you and your friends know some of the same people. Nevertheless, with each hop in a graph, you are exposed to an exponentially larger quantity of information. Looking for the answer to a question? Want to do analytics? Want to understand the big picture? Ask around and you'll find someone who knows someone who knows.

We talk about "looking deeper" all the time, but in graph it means something particular. It is a structured way of searching for information and understanding how those facts are related. Looking deeper includes breadth-based search to consider what is accessible to you from your current position. It then traverses to some of those neighboring vertices to gain depth and see what is accessible from those new positions. Whether it's for a fraud investigation or to optimize decision making, looking deeper in a graph uncovers facts and connections that would otherwise be unknown.

As we saw in "Connecting the Dots", one relationship on its own may be unremarkable, and there may be little if any information in a given vertex to reveal bad intentions, but thousands or even millions of vertices and edges considered in aggregate can begin to reveal new insights which in turn leads to actionable intelligence.

Seeing and Finding Patterns

Graphs present a new perspective, revealing hidden data patterns which are easy to interpret.

As we have seen, a graph is a set of vertices and edges, but within the set of vertices and relationships, we can begin to detect patterns.

A graph pattern is a small connected set of vertices and edges which can be used as a template for searching for groups of vertices and edges which have a similar configuration.

The most basic graph pattern is the data triplet: vertex → edge → vertex. The data triplet is sometimes thought of as a semantic relationship because it is related to the grammar of language and can be read as "subject → verb → object", e.g, Bob → owns → boat.

We can also use graph patterns to describe higher-level objects or relationships that we have in mind. For instance, depending on your schema, a person could be linked to a number of vertices containing personal data such as address, telephone, and email. Although they are separate vertices, they are all related to that one person. Another example is a wash sale, which is the combination of two securities trades: selling a security at a loss, and then purchasing the same or substantially similar security within 30 days.

Patterns come in different shapes. The simplest pattern, which we have looked at already, is the linear relationship between two vertices across a series of hops. The other common pattern is the star shape: many edges and vertices radiating from a central vertex.

A pattern can be Y-shaped, a pattern you would see when two vertices come together on a third vertex which is then related to a fourth vertex. We can also have circular or recursive patterns and many more.

In contrast to relational databases, graph data is easy to visualize, and graph data patterns are easy to interpret.

A well-designed graph gives names to the vertex and edge types that reflect their meaning. When done right, you can almost look at a connected sequence of vertices and edges and read the names like a sentence. For example, consider Figure 2-15 which shows Items purchased by Persons.

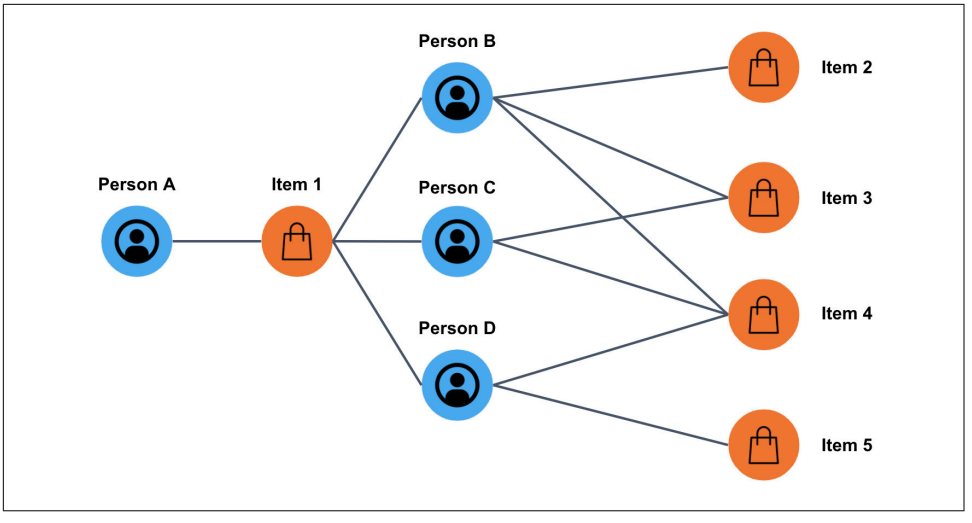


Figure 1-14. Fig 2-14. People who bought Product A also bought these products.

Starting from the left, we see that Person A (you) bought Item 1. Moving to the right, we then see another group of persons B, C, and D who also bought item 1. Finally we see some more items that were purchased by these persons. So, we can say, "You bought Item 1. Other persons who bought Item 1 also bought Items 2, 3, 4, and 5." Sound familiar?

A closer analysis reveals that Item 4 was the most popular item, purchased by all three shoppers in your co-purchaser group. Item 3 was next most popular (purchased by two), and Items 2 and 5 were the least popular. With this information, we can refine our recommendations.

Many retailers use graph analytics for their recommendation analytics, and they often go deeper yet, classifying purchases by other customer properties such as gender, age, location, and time of year. One could even make recommendations based on time of day if we saw that customers were, for instance, more likely to purchase luxury items in the evening and make more pragmatic purchases in the morning.

If we also analyze the sequence of purchases, we can also work out some highly personal information about customers. One large retailer was famously able to tell which customers were pregnant and when they were due simply by focusing on the purchases of 25 products. They were then able to send them targeted promotional offers to coincide with the birth of their child.

Matching and Merging

Graph is the most intuitive and efficient data structure for matching and merging records.

As we discussed earlier, organizations want to have a 360-degree view of their data, but a big obstacle to this is data ambiguity. An example of data ambiguity is having multiple versions of customer data, and the challenges of deduplicating data are well known to many organizations.

Duplication is sometimes caused by the proliferation of enterprise systems which split your customer view across many databases. For instance, if you have customer records in a number of databases -- such as Salesforce, a customer service database, an order processing system and an accounting package -- the view of that customer is divided across those systems.

To create a joined up view of your customers, you need to query each database and join together the records for each customer.

However, it's not always that easy because customers can end up being registered in your databases under different reference IDs. Names can be spelled differently. Personal information (surname, phone number, email address, etc.) can change. How do you match together the correct records?

Entity resolution matches records based on properties that are assumed to be unique to the entities that are being represented. In the case of person records, this might be email addresses and telephone numbers, but it could also be aggregates of properties – for instance, we can take name, date of birth and place of birth together as a unique identifier because what are the chances of those three things being the same for any two people in the world?

Entity resolution is challenging across relational databases because in order to compare entities, you need to be comparing like with like. If you are working with a single table, you can say that similar values in similar columns indicate a match, allowing you to resolve two entities into one, but across multiple tables, the columns may not match. You may also have to construct elaborate table joins to include cross-referenced data in the analysis.

By comparison, entity resolution in a graph is easy. Similar entities share similar neighborhoods, which allows us to resolve them using similarity algorithms such as cosine similarity and Jaccard similarity.

In entity resolution, we actually do two things:

- Find matches – compare attributes and look for indicators of similarity. Give the match a confidence score.
- Merge matching records – using the confidence score, use one of several strategies to merge the records.

When it comes to merging records, we have a few options including:

- Copy the data from record B to record A, redirect the edges that pointed to B to point to A, and delete B.
- Create a special link called "same_as" between records A and B.
- Create a new record, C, copy the data from A and B, redirect the links from A and B to link to C, and finally create "same_as" edges pointing from vertex C to vertices A and B.

Which is better? The second is quicker to execute because there is only one step involved – adding an edge – but a graph query can execute the first and third options just as well. In terms of outcomes, which option is better depends on your search use case – for instance, do you prioritize richness of data or search efficiency? It might also depend on the degree of matching and merging you expect to do in your database.

We will demonstrate and discuss entity resolution with a walkthrough example in a later chapter.

Weighing and Predicting

Graphs with weighted relationships let us easily model and analyze complex cost structures.

As we've shown, graphs are a powerful tool for analyzing relationships, but one thing to consider is that relationships don't have to be binary, on or off, black or white. Edges, representing the relationships between vertices, can be weighted to indicate the strength of the relationship, such as distance, cost or probability.

If we weight the edges, path analysis then becomes a matter of not just tracing the links between nodes but also doing computational work such as aggregating their values.

However, weighted edges make graph analysis more complex in other ways, too. In addition to the computational work, finding shortest paths in a graph with weighted edges is algorithmically harder than in an unweighted graph. Even after you've found a path to a vertex, you cannot be certain that it is the shortest path. If the edge weights are always positive, then you have to keep trying until you have considered every in-edge to the vertex, and if edge weights can be negative, then it gets harder yet because you must consider all possible paths.

Then again, edge weighting does not always make for a significant increase in work. In the PageRank algorithm, which computes the influence of each vertex on all other vertices, edge weighting makes little difference except that the influence that a vertex receives from a referring neighbor is multiplied by the edge weight, which adds a minimal computational overhead to the algorithm.

There are many problems that can be solved with edge weighting. Anything to do with maps, for instance, lends itself to edge weighting. You can have multiple weights per edge. Considering the map example, these could include constant weights such as distance and speed limits and variable weights such as current travel times to take account of traffic conditions.

We could use a graph of airline routes and prices to work out the optimal journey for a passenger based not only on their itinerary but also their budget constraints. Are they looking for the fastest journey regardless of price or are they willing to accept a longer journey, perhaps with more stops, in exchange for a lower price? In both cases you might use the same algorithm, shortest path, but prioritize different edge weights.

With access to the right data, we could even work out the probability of having a successful journey. For instance, what is the probability of our flight departing and arriving on time? For a single hop, we might accept an 80% chance that the flight wouldn't be more than an hour late, but for a two hop trip, where the chance for the second hop not being late was 85%, the combined risk of being delayed would be 68%.

Likewise, we could look at a supply chain model and ask, what are the chances of a severe delay in the production of our finished product? If we assume that there are six steps and the reliability of each step is 99%, then the combined reliability is about 94% -- in other words, there is a 6% chance that something will go wrong. We can model that across hundreds of interconnecting processes and use a shortest path algorithm to find the 'safest' route that satisfies a range of conditions.

Chapter Summary

In this chapter, we have looked at graph structure and how we can use a graph database to represent data as a series of data nodes and links. In graphs, we call these vertices and edges, and they enable us to not only represent data in an intuitive way – and query it more efficiently – but also use powerful graph functions and algorithms to traverse the data and extract meaningful intelligence.

Property graphs are graphs in which every vertex and edge – which we collectively refer to as objects – can hold properties which describe that object. One property of an edge is direction, and we discuss the benefits and tradeoffs of different directed edge types in indicating hierarchy and sequence.

We looked at what is meant by traversing a graph as well as 'hops' and 'distance'. There are two approaches to traversing a graph: breadth-first search and depth-first search, each with its own benefits and tradeoffs.

We looked at the importance of using a graph schema to define the structure of the database, how a consistent set of object types makes your data easier to interpret and how it can closely relate to the real world.

Careful consideration was given to different approaches to the design, in particular the search use case and how mapping the columns of a relational database to a graph database can impact query time and the complexity of your coding.

A key step in implementing a graph database is mapping columns in a relational database to a graph because a common use case for graph is building relationships between disparate databases. One of the decisions you have to make is which columns to map to their own objects and which to include as properties of other objects.

We looked at the evolution of databases over time and why a flexible schema is essential to ensuring your database remains up to date.

In the design of a database schema, whether that be for a relational or graph database, there are benefits and tradeoffs to be made, and we looked at a few of those including the choice of whether to map a column to an object or make it the property of an object. We also considered the choice of edge directionality and the granularity of edge types.

There are also tradeoffs to be made in recording multiple events between the same two entities and tracking events in an IT network.

Finally, we looked at what we mean by graph power including the essential question, why use graph in the first place? We looked at some general use cases including:

1. Connecting the Dots – how a graph forms an actionable body of knowledge
2. The 360 View – how a 360 graph view eliminates blind spots
3. Looking Deep for More Insight – how deep graph search reveals vast amounts of connected information
4. Seeing and Finding Patterns – how graphs present a new perspective, revealing hidden data patterns which are easy to interpret
5. Matching and Merging – why graph is the most intuitive and efficient data structure for matching and merging records
6. Weighing and Predicting – how graphs with weighted relationships let us easily model and analyze complex cost structures

As stated at the beginning of this chapter, you should now be able to:

- Use the standard terminology for describing graphs
- Know the difference between a graph schema and a graph instance
- Create a basic graph model or schema from scratch or from a relational database model
- Apply the "traversal" metaphor for searching and exploring graph data
- List three ways that graph data empowers your knowledge and analytics

- State the entity resolution problem and show how graphs resolve this problem

CHAPTER 2

See Your Customers and Business Better: 360 Graphs

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 3rd chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

This chapter will employ some real-world use cases to illustrate two of the six graph powers that we discussed in the previous change: connecting the dots and the 360 view. The 360 view offered by graphs help enterprises and agencies see their data more comprehensively, which in turn enables better analytics. In the first use case, we build a Customer 360 graph to enable a company to track and understand pre-sales customer journeys. In the second case, we build a Drug Interaction 360 graph so researchers can develop safer drug therapies.

After completing this chapter, you should be able to:

- Define the term C360 and explain its value proposition.
- Know how to model and analyze customer journeys in a graph.
- Know how to use graph analytics to count and filter properties and relationships.

- Set up and run a TigerGraph Cloud Starter Kit using GraphStudio.
- Read and understand basic GSQL queries.

Case 1: Tracing and Analyzing Customer Journeys

A business is nothing without sales. Selling, whether to consumers (B2C) or to other businesses (B2B) has become not only an art, but also a science. Businesses analyze every stage of the interactions with a prospect (a potential customer) from beginning to end, hopefully resulting in a sale. According to Gartner, worldwide spending on customer relationship management (CRM) software increased by 15.6% in 2018 to reach $48.2 billion in 2020[1]. Salesforce has established itself as the market leader in CRM software, with approximately a 20% market share[2].

A key way to think about the process of selling is to consider the prospective customer's experience as a series of events over time. How and when did someone engage with the business and its wares? Mapping out the interactions with a sales prospect is known as *tracing the customer's journey*.

The customer journey model is an essential tool for sales and marketing. First, it takes the customer's point of view, and customers are the ultimate decision makers. Second, by realizing that the customer may need to move through stages, the business can map out what it believes will be attractive journeys that will secure many successful business deals. Third, when looking at individual journeys, we can see how far they have progressed, whether a journey is stalled, slow, or has changed course. Fourth, by analyzing the collected set of journeys, businesses can see patterns and trends and compare them to their targeted behavior. Are users in fact following the journey that was designed? Do particular engagements succeed in moving prospects forward?

There is a need for an effective and scalable data system that can collect the mixed types of data in a customer journey, and support the analysis of both individual and aggregated journeys.

Solution: Customer 360 + Journey Graph

CRMs would seem to offer the solution, but they have not fully met business' needs for customer journey analysis. Designed for data to be either entered manually or ingested digitally, they record and present data primarily in tabular form. A column

1 "CRM Market Share - Salesforce Bright Future in 2020", nix-united.com

2 "Market share of CRM leading vendors worldwide 2016-2020", Statista.com

in one table can correspond to a column in another table, just as in a relational database.

Tables can record activities by a person and index them by time. The challenge comes from the fact that each type of engagement (watching a video, attending a demonstration, downloading trial software) are all different, with different characteristics. Storing mixed data like this in one table doesn't work well, so data must be spread across multiple tables and connected with numerous joins.

Tracing and analyzing customer journeys is a natural fit for graphs. A journey is a time sequence of events, as shown in Figure 2-1. Graphs have no problems with connecting mixed (heterogeneous) data from assorted events and interactions. The journeys of all prospective customers can be stored in one graph. Individual journeys will have similarities and intersections with one another, as persons attend the same events or engage in similar activities.

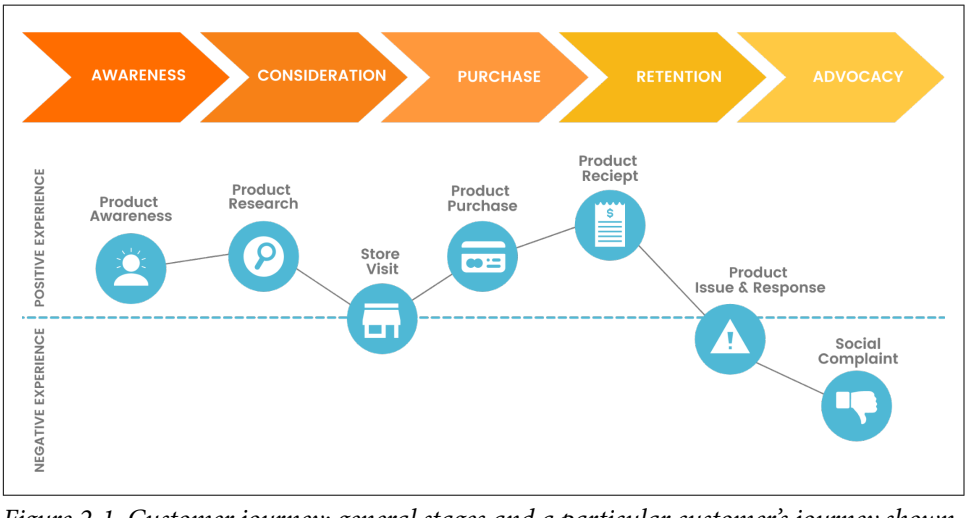


Figure 2-1. Customer journey: general stages and a particular customer's journey shown as a graph

Businesses not only want to map out customer journeys, but they also want to make them more successful: to increase customer satisfaction, to increase the percentage of journeys that end in a sale, to increase the value of sales, and to shorten the journeys. To do this, businesses need to understand the context of each customer and their decisions. This is where a 360 view helps. The 360 view is one of the unique powers of graphs which we discussed in the previous chapter.

Customer 360 (C360) is a comprehensive view of a customer (or any entity of interest) created by integrating data from multiple sources. Like customer journeys, Customer 360 is a great fit for graphs and graph analytics. A graph can support an unlimited number of relationships between one vertex (a customer) and other enti-

ties. These entities can describe not just the journey (a cold call, webinar, brochure, product demonstration, or website interaction) but also the context of the customer (current and past job titles, tenures, employers, locations, skills, interests, and education). A good 360 database will also include information about employers and industries (size, initiatives, news, etc.).

Figure 2-2. A Customer 360 graph gathers and connects information about an individual from multiple areas of interaction, to form a holistic view.

With the combination of 360° data and journey analysis, businesses are able to clearly see what is happening in the sales process, at the individual and aggregate levels, to see the context of these actions, to see where improvement is desired, to assess the impact of efforts at sales improvement.

Our proposed solution is to develop a data model which makes it easy to examine and analyze customer journeys. The data model should also incorporate data described and related to customers, to produce a Customer 360 view. The model should support queries about what events a customer journey does or doesn't contain, as well as the timing of such events.

Implementing the C360 + Journey Graph: A GraphStudio Tutorial

The implementation of a C360 and customer journey graph we present below is available as a TigerGraph Cloud Starter Kit. Don't worry if this is your first time using TigerGraph Cloud. We'll show you how to sign up for a free account and to deploy a free Starter Kit. Alternatively, if you have TigerGraph installed on your own machine, we'll tell you how to import the starter kit into your system.

Then we'll simultaneously walk you through the design of the C360 graph and GraphStudio in general.

Figure 2-3 maps out the two paths towards setting up a starter kit. In the sections below, we'll first tell you how to create a TigerGraph Cloud account. Then we'll walk you through the steps for getting and loading a Starter Kit, first for TigerGraph Cloud users and then for TigerGraph on-premises users.

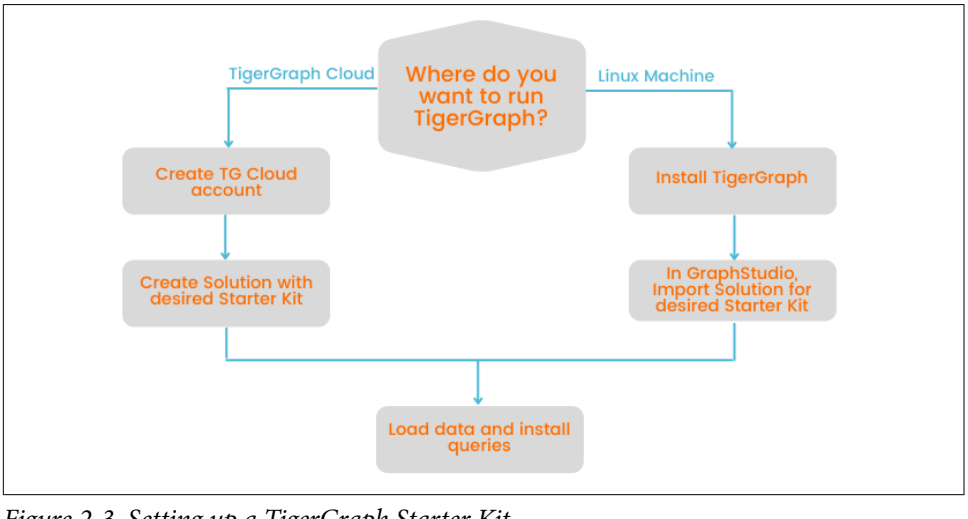


Figure 2-3. Setting up a TigerGraph Starter Kit

Create a TigerGraph Cloud Account

If this is your first time using TigerGraph Cloud, you need to set up an account. It's simple and free.

1. In a web browser, go to tgcloud.io.
2. Click the Login / Register button, and then click Sign Up.
3. You can open a new account using your existing credentials for Google or LinkedIn, or you can create a new username and password.
4. You should now be on the Dashboard page of TigerGraph Cloud (Figure 2-4). You should see some free credits in your account, but you won't need that if you stick with the Free tier when you provision a database.

That's it. You now have a TigerGraph Cloud account. In the next section, we'll tell you how to create a TigerGraph Cloud database, with your choice of starter kit.

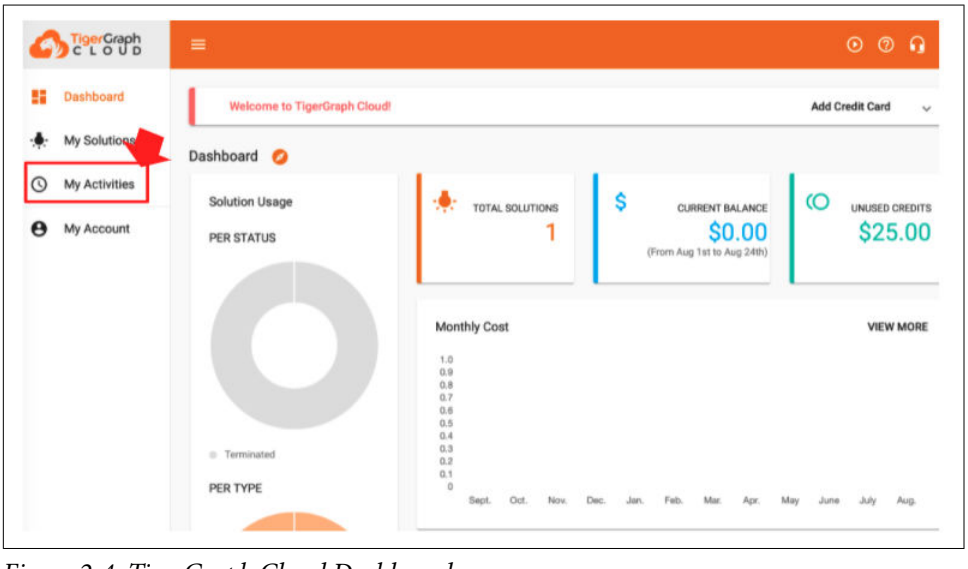


Figure 2-4. TigerGraph Cloud Dashboard

Get and Install the Customer 360 Starter Kit

We are going to use the starter kit called *Customer 360 - Attribution and Engagement Graph*. If you are a TigerGraph Cloud user, you can get the Starter Kit as part of a new database deployment. If you are running TigerGraph on your own computer, you can download the starter kit files from TigerGraph and then upload them into your TigerGraph instance.

The next two sections go over the details for these two options.

Deploy a Cloud Solution with a Starter Kit

On your TigerGraph Cloud control panel, the Solutions page is where you will see a list of all the TigerGraph Cloud databases you have created, except for the ones you

have archived. Each cloud database in TigerGraph Cloud is referred to as a *solution*. To create a new cloud solution, follow these steps:

1. In TigerGraph Cloud, click My Solutions on the left-hand side menu.
2. Click the Create Solution button at the upper right.
3. Select the Starter Kit you want. In this case, it's Customer 360 - Attribution and Engagement Graph. Scroll down and click Next.
4. Select the cloud platform, instance size, region, disk size, and cluster configuration that you want. On some platforms, the smallest instance and disk size is offered for free. This is enough to handle a few gigabytes of data, enough for any of the starter kits. Click Next.
5. The last page is for naming your solution. Remember your password! Click Next.
6. Review your selected configuration, make corrections if needed, and then click the Submit button. It will take a few minutes for your database to be created and deployed.
7. Continue to the Load Data and Install Queries section below.

Import the Starter Kit into your TigerGraph Instance

If you have TigerGraph software installed on your own machine, follow these steps to get a starter kit.

Importing a GraphStudio Solution will delete your existing database. If you wish to save your current design, perform a GraphStudio Export Solution and also gsql backup.

1. Go to www.tigergraph.com/starterkits.
2. Find *Customer 360 - Attribution and Engagement Graph*.
3. Download *Data Set* and the solution package corresponding to your version of the TigerGraph platform.
4. Start your TigerGraph instance. Go to the GraphStudio home page.
5. Click *Import An Existing Solution*, as highlighted in Figure 2-5, and select the solution package which you downloaded.
6. Continue to the Load Data and Install Queries section below.

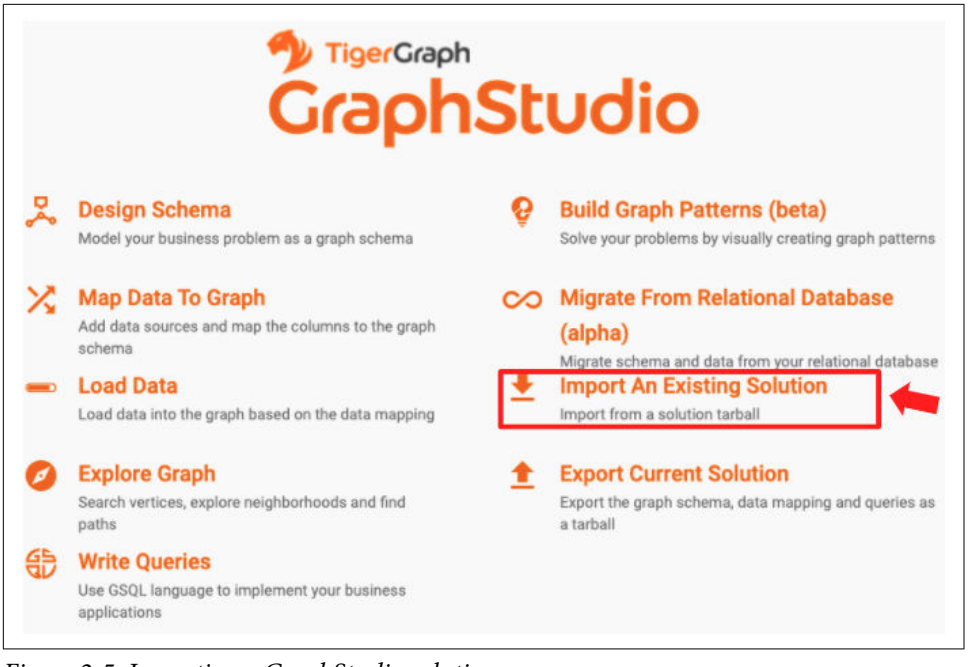


Figure 2-5. Importing a GraphStudio solution

Load Data and Install Queries for a Starter Kit

There are three additional steps needed to complete the installation of a starter kit. If you know GraphStudio and just want to know how to install a Starter Kit, then follow these steps:

1. Go to the Design Schema page. Switch from the Global view to the local graph view.
2. Go to the Load Data page. Wait about 5 seconds until the Load Data button on the left end of the menu becomes active. Click the button and wait for the data to finish loading. You can track the loading progress in the timeline display at the lower right.
3. Go to the Write Queries page. Above the list of queries, click the Install All Queries button and wait for the installation to complete.

An Overview of GraphStudio

TigerGraph's GraphStudio is a complete graph solution development kit, covering every stage in the process from developing a graph model to running queries. It is organized as a series of views or pages, each one for a different task in the development process.

Because this is our first time through GraphStudio together, we are going to walk through all five stages: Design Schema, Map Data to Graph, Load Data, Explore Graph, and Write Queries. At each stage we will both explain the general purpose of the page as well as guide you through details of the specific starter kit we are working with. In future chapters, we will skip over most of the generalities and only talk about the starter kit details.

If we were beginning with an empty database, we would need to do additional design work, such as creating a graph model. Having a starter kit lets you skip most of this and get right to exploring and querying an example dataset.

Design a Graph Schema

The Starter Kit is preloaded with a graph model based on commonly used data objects in Salesforce and similar customer relationship management (CRM) software. The name of the graph in this starter kit is *MyGraph*. When you start GraphStudio, you are initially at the global graph level. You are not yet working on a particular graph. In a TigerGraph database, the global level is used to define data types that are potentially available to all users and all graphs. See the section labeled Global types in Figure 2-6. A database can then host one more graphs. A graph can contain local types, and it can include some or all of the global types. See graphs G1 and G2 in the figure.

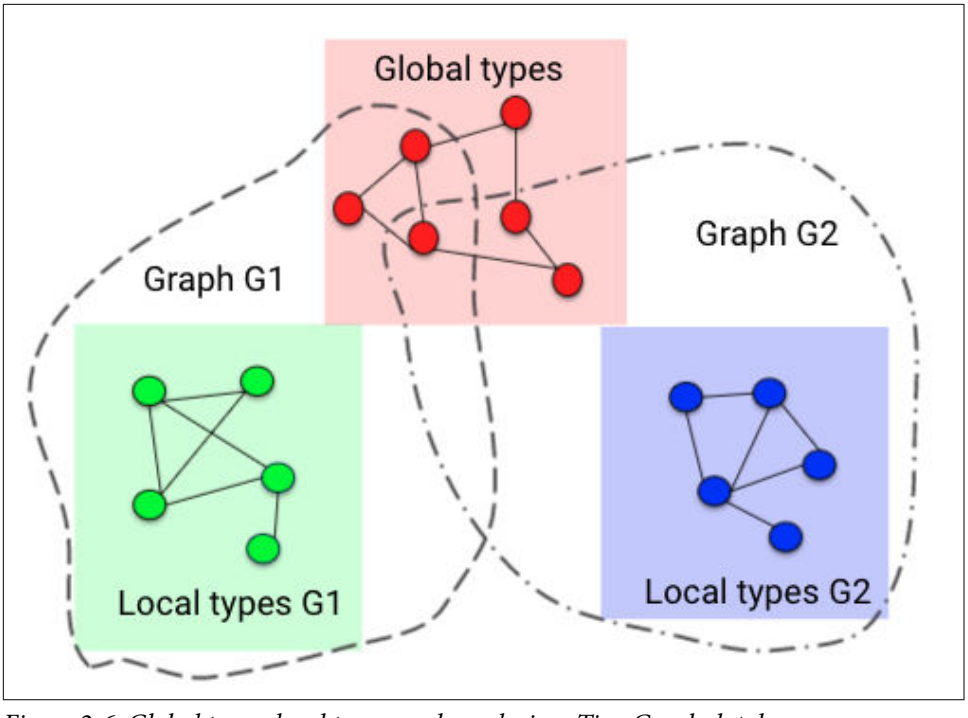


Figure 2-6. Global types, local types, and graphs in a TigerGraph database

To work on a graph, you need to select the graph, which moves you from the global level to the local graph level. To switch to a local graph, click on the circular icon in the upper left corner (step 1 in Figure 2-8). A dropdown menu will appear, showing you the available graphs and letting you create a new graph. Click on MyGraph (step 2). Just below that, click on Design Schema to be sure we're starting at the right place.

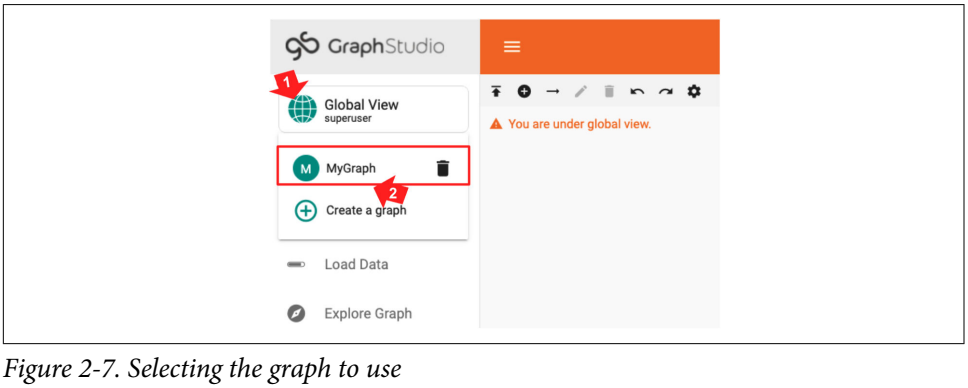


Figure 2-7. Selecting the graph to use

You should now see a graph model or schema like the one in Figure 2-8 in the main display panel.

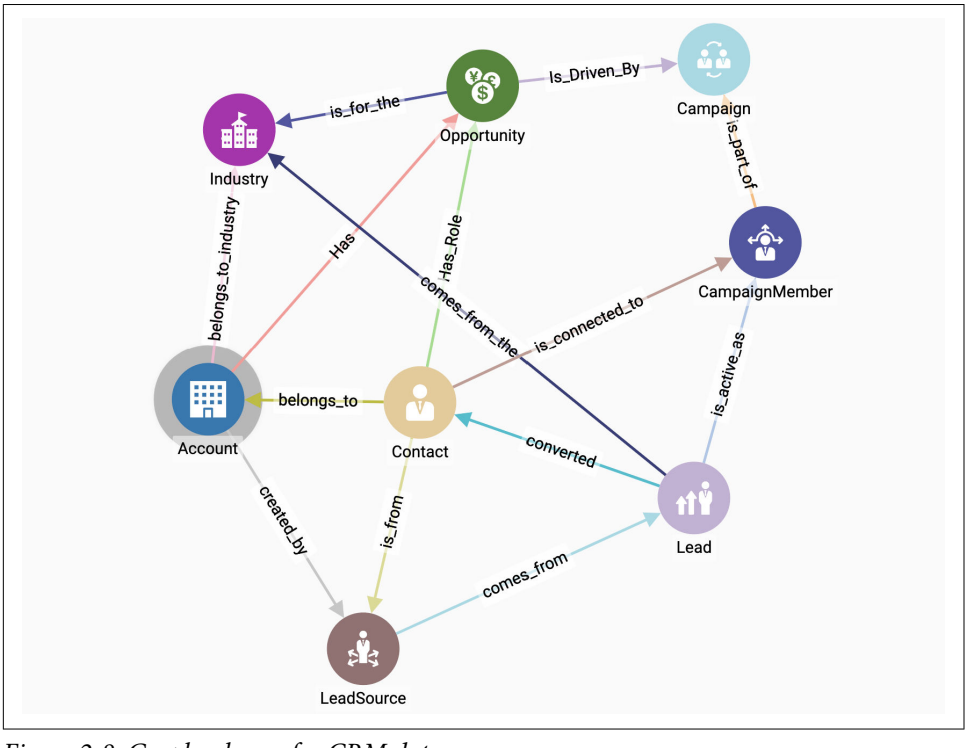


Figure 2-8. Graph schema for CRM data

A graph schema defines the *types* of data objects to be stored in the database. If the schema is depicted visually, then each data type is shown once. This schema has eight vertex types and 14 edge types.

The central vertex type is Contact, which is a prospective buyer of the product. However, a Contact is not just any prospective buyer, which reflects the fact that a person buying a B2B product on behalf of a company is not making a spur-of-the-moment decision. Instead the person transitions through stages of the buying process. We call the person's flow through the buying process the *customer journey*.

One real-world person might show up more than once in the database. If the vendor conducts a marketing Campaign, then persons who respond to the campaign show up as CampaignMembers. Also, if a third party, a LeadSource, provides contact information about potential buyers, then the potential buyers show up as Leads. Salespersons engage with a Lead to see if there is a realistic possibility of a sale. If there is, then the Lead's information is copied to a new vertex type called a Contact. This Con-

tact and their source Lead represent the same physical person but at different stages of the customer journey.

Table 3-1 contains descriptions of all eight vertex types. In some cases, the description of one vertex type talks about how it is related to another vertex type. For example, an Account is "an organization that a Contact belongs to." Looking at Figure 2-6, you can see an edge type called belongs_to between Account and Contact. There are 13 other edge types in the figure. The edge types have descriptive names, so if you understand the vertex types, you should be able to figure out the meaning of the edges.

Table 2-1. Vertex types in the Salesforce Customer 360 graph model

Vertex Type	Description
Account	an organization that a Contact belongs to
Campaign	a marketing initiative intended to generate Leads
CampaignMember	a person who responds to a Campaign
Contact	a Lead who is now associated with a sales Opportunity
Industry	a business sector of an Account
Lead	a person who is a potential buyer of the product but is not yet associated with an Opportunity
LeadSource	a channel through which a Lead finds out about the product
Opportunity	a potential sales transaction, characterized by a monetary amount

Data Loading

In TigerGraph Starter Kits, the data is included, but it is not yet loaded into the database. To load the data, switch to the Load Data page (step 1 of Figure 2-9), wait a few seconds until the Load button in the upper left of the main panel becomes active, and then click it (step 2). You can watch the progress of the loading in the real-time chart at the right (not shown). Loading the 34K vertices and 105K edges should take two minutes on the TGCloud free instances; faster on the paid instances.

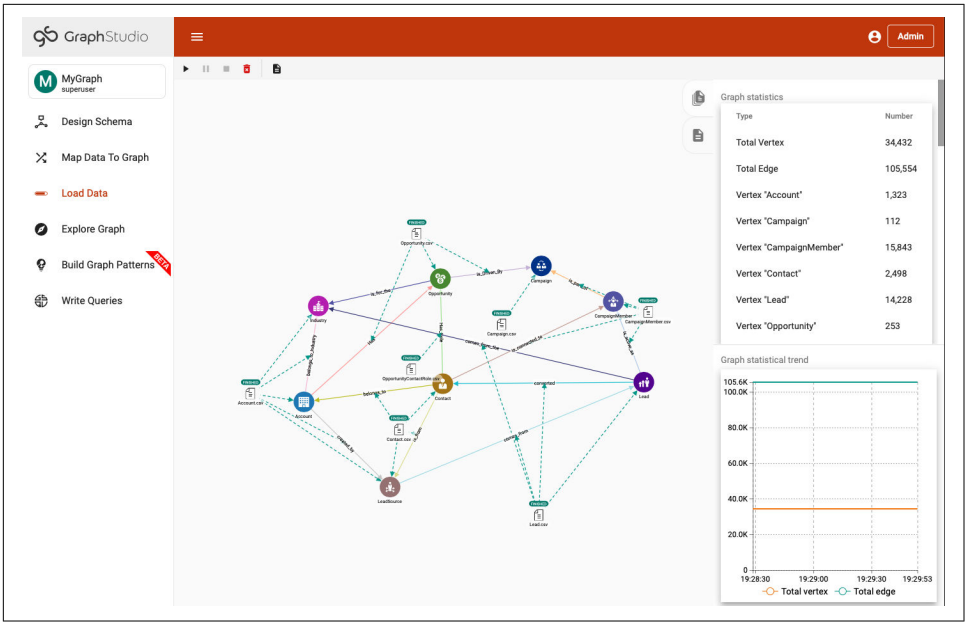


Figure 2-9. Loading data in a Starter Kit

Queries and Analytics

We will analyze the graph and run graph algorithms by composing and executing queries in GSQL, TigerGraph's graph query language. When you first deploy a new Starter Kit, you need to install the queries. Switch to the Write Queries page (step 1 of Figure 2-10). Then click the Install All icon at the top right of the list of queries (step 2).

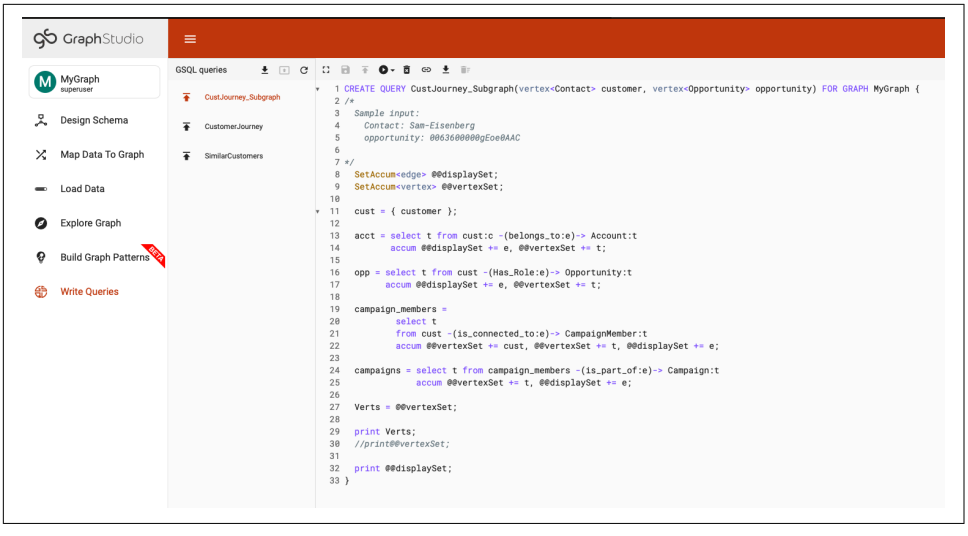


Figure 2-10. Installing queries

For our Customer 360 use case, we will discuss three queries.

1. **Customer Journey Subgraph**: This query generates a subgraph that gives us a holistic view of the customer journey. It contains vertices that refer to the interactions that the customer has made on which of the campaigns. The query starts with a given customer of type **Contact** vertex. From there, it continues to select elements of an **Account**, **Opportunity**, and a **CampaignMember** that the customer had interacted with. Furthermore, for each of the **CampaignMember** elements a **Campaign** is selected as well. Finally, the customer journey is returned as a subgraph.
2. **Customer Journey:** This query finds all the **CampaignMember** elements that the customer has interacted with during a time period. The query starts with a given **Contact** and filters out all the **CampaignMember** elements that have been in touch with the **Contact** between a start-time and end-time. Unlike the first query, we don't return a subgraph with the connections between the **Contact** and **CampaignMember**. Here we only return the **Contact**, **CampaignMember**, and **Campaign** vertices.
3. **Similar Customers**: This query returns similar customers from a given **Contact**, so that we can reach out to potential customers of which we know that will have a higher chance of conversion. For this query we use the Jaccard Similarity to calculate the similarity between a given **Contact** and other **Contact** who share a similar **Campaign**.Then the contacts with the top highest similarity score will be returned.

For each of the three queries, we'll give a high level explanation, directions for operations to perform in TigerGraph's GraphStudio, what to expect as a result, and a closer look at some of the GSQL code in the queries.

Customer Journey Subgraph

The CustomerJourney_Subgraph query takes one argument: a customer who is a natural person of type **Contact**. First, we select all the Account identities which belongs_to the given Contact. Then we find all the **Opportunity** vertices connected to the **Contact**. . In addition, the **Contact** vertex also has a connection to a **CampaignMember**, who is a natural person that is part of a **Campaign**.

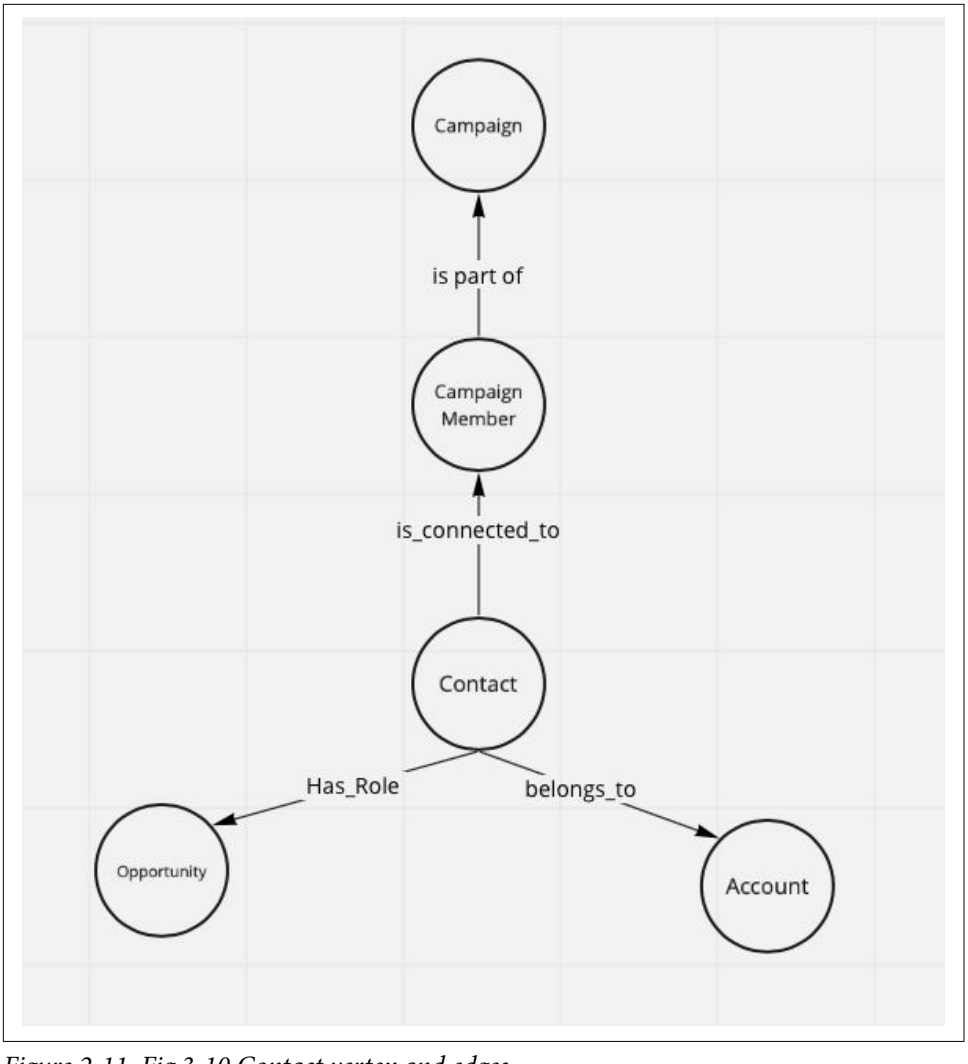


Figure 2-11. Fig 3-10 Contact vertex and edges

DO: Run the GSQL query CustomerJourney_Subgraph by selecting the query name from the list (step 1 in Figure 3-11) and then clicking the Run icon above the code panel (step 2). This query has one input parameter: the 'Contact' field lets us fill in a name of the customer. If you look at the code window (Figure 3-11), you will see a comment that suggests an example value for Contact: Sam-Eisenberg.

Use your mouse to copy and paste values from the code window into the query parameter input boxes.

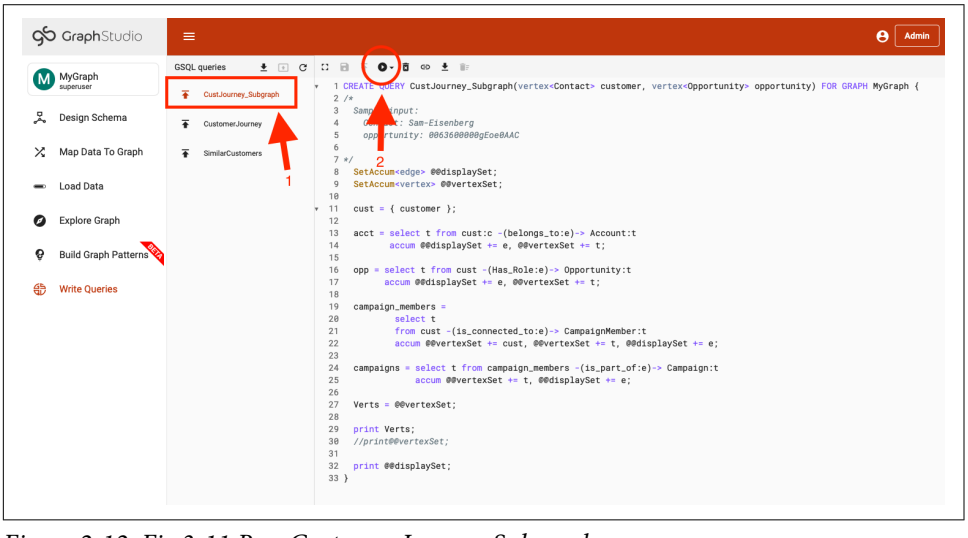


Figure 2-12. Fig 3-11 Run Customer Journey Subgraph

In this section we will look at how the GSQL query CustomerJourney_Subgraph works. Refer to the code block that follows. In the first line, we define the name of the query and its input parameters. .To find a customer subgraph, we need one parameter which is a vertex of type **Contact**. Lines 6 and 7 define two accumulators which will gather the edges and vertices of our subgraph, called *displaySet* and *vertexSet* respectively. Next, in line 9 we set *cust* to the customer that the user has placed in the input field when running the query, which is of type **Contact**. Line 11 selects each **Account** that the customer belongs to. Then in line 15 we add the resulting **Account** vertices and the edges to vertexSet and displaySet. In lines 14 and 15 we do something similar, however here we select the vertices and edges where the customer has a role in creating an opportunity. In lines 17 to 21 we find the **CampaignMember** vertices that are connected to the customer and update the *vertexSet* and displaySet again with the results. In lines 23 to 26, we start from the campaign_members we selected in the previous step ("FROM campaign_member"), then we find the vertices and edges of each **CampaignMember** that is part of a **Campaign**, and update the sets *displaySet* and *vertexSet* again with the results. Lines 30 to 33 plot the subgraph by printing the *vertexSet* and *displaySet*.

```
CREATE QUERY CustJourney_Subgraph(VERTEX<Contact> customer) {
/*
  Sample input:
```

```
    Contact: Sam-Eisenberg
*/
    SetAccum<EDGE> @@displaySet;
    SetAccum<VERTEX> @@vertexSet;
    cust = { customer };

    acct = SELECT t FROM cust:c -(belongs_to:e)- Account:t
    ACCUM @@displaySet += e, @@vertexSet += t;

    opp = SELECT t FROM cust -(Has_Role:e)- Opportunity:t
    ACCUM @@displaySet += e, @@vertexSet += t;

    campaign_members =
        SELECT t
        FROM cust -(is_connected_to:e)- CampaignMember:t
        ACCUM @@vertexSet += cust, @@vertexSet += t,
            @@displaySet += e;

    campaigns =
        SELECT t
        FROM campaign_members -(is_part_of:e)- Campaign:t
        ACCUM @@vertexSet += t, @@displaySet += e;

    Verts = @@vertexSet;

    PRINT Verts;
    //print@@vertexSet;
    PRINT @@displaySet;
}
```

Customer Journey

The CustomerJourney query shows all **CampaignMember** and **Account** vertices that have a relationship with the customer during a given time period. Here we are not interested in a snapshot of how the subgraph of our customer journey looks at a certain point in time, but we are more interested in collecting all the **CampaignMember** and **Account** that were related to the customer during a time period. Let's take a look at the GSQL implementation.

This GSQL query has four attributes (line 1 and 2). The first attribute is a vertex of type **Contact** and represents the customer that we are interested in. The second attribute is a set of type String which we will use to keep track of all campaigns. The third and fourth attributes are of *datetime* types, and we use these attributes to determine the time window in which our query should be executed. First we select the **Account** to which our target customer belongs (line 11 and 14). Then in lines 17 to 26 we select all **CampaignMember** that are connected to the customer within the given time window. During this selection, we will build three lists to profile the date: what are the campaign types, campaign names, and campaign descriptions (lines 27 to 29).

```
CREATE QUERY CustomerJourney(VERTEX<Contact> customer, SET<string> campaignTypes, DATETIME startTime,
/*
  Sample input:
    Contact: Sam-Eisenberg
    startTime: 2018-06-01
    endTime: 2018-10-01
*/

    SumAccum<STRING> @camType, @camName, @camDesc;
    Customer = { customer };
    PRINT Customer;
    Company = SELECT t FROM Customer -(belongs_to)- Account:t;
    PRINT Company;
    campaign_mem =
        SELECT c
        FROM Customer-(is_connected_to)-> CampaignMember:c
        WHERE c.CreatedDate >= startTime
            AND c.CreatedDate <= endTime;
    campaign =
        SELECT c FROM campaign_mem:c -(is_part_of)- Campaign:t
        WHERE campaignTypes.size() == 0
            OR t.Campaign_Type IN campaignTypes
        ACCUM c.@camType = t.Campaign_Type,
            c.@camName = t.Name,
            c.@camDesc = t.Description;
    PRINT campaign as Campaign;
}
```

Similar Customers

Before we implement similarity measures, we need to determine first for which attribute we want to compute the similarity. In our case, we want to compute the similarity for customers who have been in the same or similar campaign types. To identify similar customers, we use the Jaccard Similarity algorithm. The Jaccard Similarity is not a graph algorithm exclusively applicable for graph-structured data. It is a way to compute the similarity between two sets, based on how many items belong to one set that also belong to the other set, divided over the total number of items in both sets. In the case of a graph, every vertex has a set of neighboring vertices. So, graph-based Jaccard Similarity measures the overlap between the neighbor set of one vertex with the neighbor set of another vertex. In other words, how many common neighbors are there, relative to the total number of neighbors?

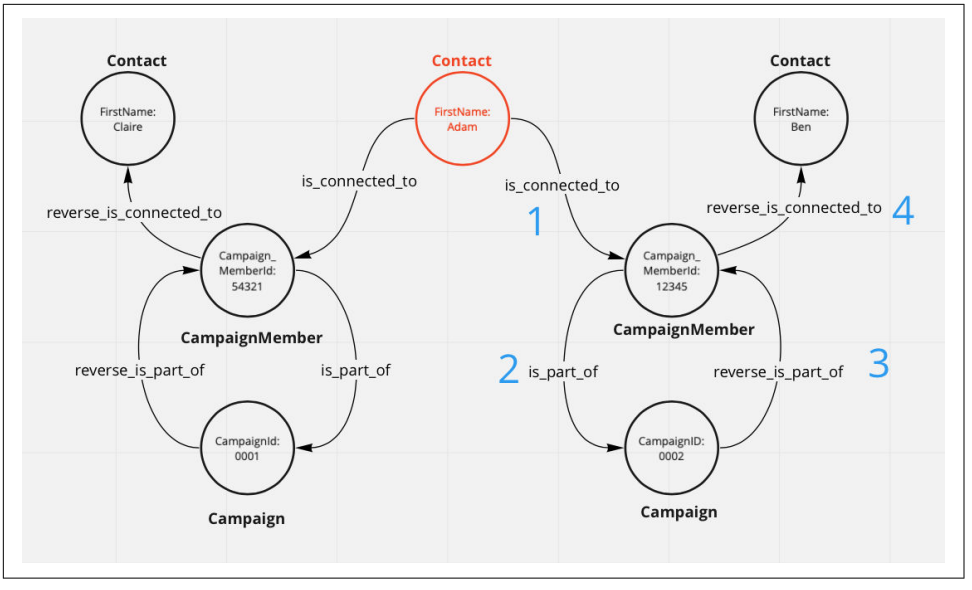


Figure 2-13. Fig. 3-12 Overview of steps to take for selecting the similar customers to calculate Jaccard Similarity score.

Our situation is a little more complicated, because we want to assess the similarity of associations to campaigns; however, Campaigns are two hops away from Contacts rather than being directly connected. Furthermore, we allow the user to filter by which types of campaigns to count.

Let's walk through the GSQL code for the SimilarCustomers query (shown in the following code block) to get a better understanding of how to implement graph-based analytics. This query accepts three parameters (line 1). The first parameter **sourceCustomer** is a vertex of type **Contact** and represents the customer of whom we want to find similar customers. The second parameter is the set of campaign types (strings) that the user wants to consider. The third parameter is an integer value to determine how many similar customers we want to return.

In line 19 we declare three *accumulators*. Accumulators, a unique feature of the GSQL language, are data objects which have a special operation called *accumulate*, indicated by the += operator. The exact meaning of += varies depending on the accumulator type, but it always is used to accept additional input data to update the accumulator's external value. Accumulators can accept multiple asynchronous accumulate operations, thus they are ideal for concurrent/parallel processing. The @@ prefix indicates a global accumulator. A @ prefix indicates a set of local (also known as vertex-attached) accumulators. Local means each vertex in the query has its own independent instance of this accumulator. For example, **@interact_size** is a local accumulator of type **SumAccum<INT>**. **SumAccum<INT>** is typically used to count. With

set_size_A, we keep track of the number of nodes in the first set, and with *set_size_B*, we do that for the second set. In lines 39 to 41, we count the out-degree of the *sourceCustomer* node with an edge type *is_connected_to*. Then we traverse from sourceCustomer over all those edges to find each **CampaignMember** related to our target customer (line 44 to 46). We can see this in step 1 of figure 3-12. In step 2 we continue, from each **CampaignMember**, we travel further over the edges with type *is_part_of* to find all **Campaign**, then we filter all **Campaign** nodes on the types that we have defined in *campaignType* (line 49 to 55). In step 3 we identify all **Contact** for the second set B that share the same **Campaign**, we traverse back from those **Campaign** nodes to the **CampaignMember** using the edge type *reverse_is part_of* (line 58 to 60). In step 4 we repeat this with **CampaignMember** to arrive in **Contact** nodes using edge type *reverse_is_connected_to* in the final SELECT block (line 71 to 73). For every **Contact** in set B we accumulate the *intersection_size* of the shared **Campaign**, and we then compute the out-degree to set the *set_size_B* (line 76).

Now we can calculate the Jaccard Similarity as follows. We take the intersection_size and multiply this with 1.0 to convert the integer value into a float number (line 77). Then we divide it over the sum of set_size_A and set_size_B and subtract the intersection_size from it; this latter step is required, or else we would end up counting the shared **Campaign** nodes twice (line 78 and 79). In line 80 we order the resulting similarity score from highest to lowest, and we only take the *topK* result to print it (line 81 to 84).

```
CREATE QUERY SimilarContacts(VERTEX<Contact> source_contact,
  SET<STRING> campaign_types, INT top_k = 5) {
/*
  Calculates the Jaccard similarities between a given customer (or Contact)
  and other customers (or Contacts) who share similar Campaigns.
  Outputs the top_k Contacts with the highest similarity scores.

  SAMPLE INPUT:
    Contact: Sam-Eisenberg
    campaignTypes: Webinar, Demo Signup / Trial
    topK: 5
  This query is more complex than a standard Jaccard similarity algorithm because
  there are 2 hops from a Contact to a Campaign (with CampaignMember in between)
  and because we only count the given types of Campaigns.

  Jaccard similarity = intersect_size / (size_A + size_B - intersect_size)
*/
    SumAccum<INT> @intersect_size, @@set_size_A, @set_size_B;
    SumAccum<FLOAT> @similarity;
    A = {source_customer};
    A = SELECT s
        FROM A:s
        ACCUM @@set_size_A += s.outdegree("is_connected_to");
    // From set A (Contact), traverse 'is_connected_to' edges to CampaignMembers
    CampaignMembersSet =
```

```
        SELECT t
        FROM A:s -(is_connected_to:e)- CampaignMember:t;
    // From CampaignMembersSet, traverse 'is_part_of' edges to Campaigns, for all
    // desired campaign_types (e.g. Webinar, Website Direct, Demo Signup/Trial)
    CampaignSet =
        SELECT t
        FROM CampaignMembersSet:s -(is_part_of:e)- Campaign:t
        WHERE campaign_types.size() == 0 OR (t.Campaign_Type IN campaign_types);
    // From CampaignSet, traverse 'reverse_is_part_of' edges back to all
    // CampaignMembers
    CampaignMembersSet =
        SELECT t
        FROM CampaignSet:s -(reverse_is_part_of:e)- CampaignMember:t;
    // From CampaignMemberSet, traverse 'reverse_is_connected_to' edges back to
    // Contacts (set B). For each Contact in set B, accumulate the intersection
    // size of the shared Campaigns, and compute its Jaccard Similarity score as
    // intersection_size / (size_A + size_B - intersection_size)
    B = SELECT t
        FROM CampaignMembersSet:s -(reverse_is_connected_to:e)- Contact:t
        WHERE t != source_customer
        ACCUM t.@intersect_size += 1,
            t.@set_size_B = t.outdegree("is_connected_to")
        POST-ACCUM t.@similarity = t.@intersect_size*1.0/
                            (@@set_size_A + t.@set_size_B - t.@intersect_size)
        ORDER BY t.@similarity DESC
        LIMIT top_k;
    PRINT B[B.FirstName, B.LastName, B.@similarity];
}
```

Case 2: Analyzing Drug Adverse Reactions

In our second use case, we seek to analyze the adverse reactions to drug treatments.

Today's healthcare system covers 30% of the world's data volume, and its compound annual growth is projected to be 36% by 2025[3]. This data collection ranges from external sources such as the US Food & Drug Administration (FDA) and National Databases Medical Associations to privately-owned datasets from health insurance companies. Organizations mine these data for valuable insights to create targeted content and engagement campaigns, improve health insurance plans, and develop medicines. Developing better medical therapies is our focus for this use case.

When developing medicines, it is vital to have clear insight into the composition of drugs, how they interact with each other and what side effects they might cause. Therefore, the FDA requires every drug manufacturer to administrate how their drugs are being used with other drugs and report on any adverse reaction.

3 "The Healthcare Data Explosion", www.rbccm.com

Analysts and researchers want to find relationships between various drugs, patients who use them, and the possible side effects. These tasks become challenging as analysts need to guess how those relationships establish. Do doctors prescribe the same drug to people in a particular postal district, or are their assessments mainly built upon patients who went to the same college? When a patient reports an adverse reaction from a given drug, other patients might also be in danger, given their drug interaction history. Without a view of how these drug interactions occur and to whom the drugs prescriptions are given, research in this field becomes challenging, and it could threaten public health when vital links between drugs and side effects are overlooked.

Solution: Drug Interaction 360 Graph

The growing amount of healthcare data brings challenges in combining external and internal data sources at a large scale and presenting this in a meaningful way. The applications in this domain require an approach that can not only handle this large amount of data but also find hidden patterns between the various data sources.

Graph databases are an ideal data platform for the discovery and analysis of drug interactions. With a graph database, we can form a 360 view of key entities and connect the dots to expose all possible correlations between patients, the drug interactions that have incurred, and the manufacturers of those drugs.

In contrast, relational and NoSQL databases store data in separate tables and rely on the analyst's domain expertise to choose which tables to join, each join an expensive operation. Analysts would build a solution around the correlations that they foresee, limiting their vision. For example, analysts can only pay attention to drug interactions that they are aware of and generate side effects. When a series of interactions occur that they have not seen before and lead to a reaction, they will miss this in their research scope.

Implementation

To illustrate a drug interaction 360 graph, we will use the TigerGraph Cloud starter kit called *Healthcare Graph (Drug Interaction/FAERS)*. To follow along, see the earlier instructions for how to deploy a TigerGraph Cloud starter kit. Load the data and install the queries.

The data we are using for this use case is publicly available from the US FDA. It contains quarterly data from the FDA's Adverse Event Reporting System (FAERS), including demographic and administrative information on the drug, patient outcome, and reaction from case reports. The FDA releases the data as seven tables. Their documentation includes an entity-relationship diagram which is suggestive of a two-hub 360 graph. However, investigating this data using relational database techni-

ques would require creating many join tables. With graph databases, we can traverse these relationships much easier.

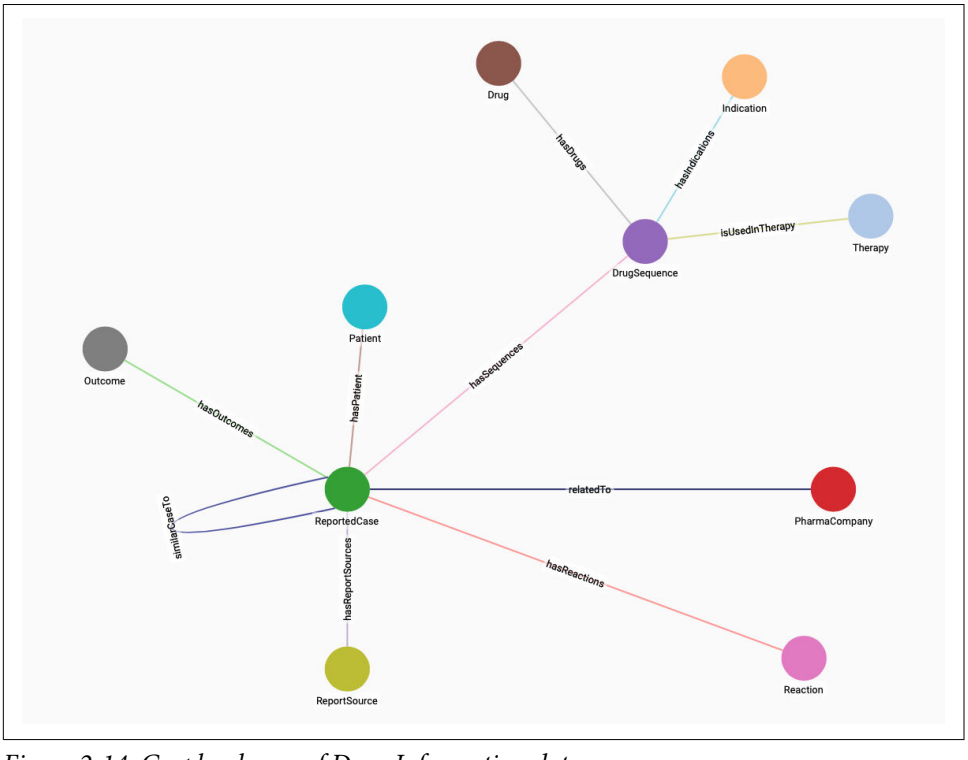


Figure 2-14. Graph schema of Drug Information data

Graph Schema

To improve the visibility and analysis of the data, we propose to transform the seven tables to ten vertex types and ten edge types. We split the **Drug** table into **Drug** and **DrugSequence** vertex types, and we split the **Demographic** table into **ReportedCase**, **Patient**, and **PharmaCompany** tables. These splits give us the agility to shift our focus as needed and to more easily see the interplay between different factors. For every **ReportedCase**, we can find information about the patient, the drug manufacturer, the patient's reaction, the source of the report, the outcome, and the various drugs the patient has been taking. For every **DrugSequence** we can find the related drug, the indication, and the patient's therapy.

Table 3-2 describes the ten vertex types. The starter kit contains data from one calendar quarter. In total, there are 1.87M vertices and 3.35M edges.

Table 2-2. Vertex types in the Drug Information model

Vertex Type	Description	Instances
DrugSequence	A sequence of Drug elements	689,312
Drug	A drug that is part of a DrugSequence	40,622
Indication	An indication (medical condition) that can be treated with a DrugSequence	422,145
Therapy	A therapy where the DrugSequence is used	268,244
ReportedCase	A reported case of side effects	211,414
Patient	A person who reported a case	211,414
Outcome	A result after assessment of a ReportedCase	7
ReportSource	A source from where the ReportedCase report is initiated	9
Reaction	A reaction from a ReportedCase	9,791
PharmaCompany	A pharmaceutical company that manufactures a Drug	7,740

Queries and Analytics

The drug interaction starter kit comes with three queries as examples of drug interaction analysis. From these examples, a studious analyst can see how to construct other queries and even more complex queries.

1. **Find similar reported cases based on reactions:** It is important to be aware that different disorders can exhibit similar symptoms. This query starts from a given reported case and calculates its similarity to other cases, based on the similarity of reactions of the patients. Then, it returns the top scoring similar cases.
2. **Most reported drugs for a company:** Pharmaceutical companies want to know which of their drugs are receiving the most reports of adverse reactions. Government regulatory agencies might also want to know this. This query performs that calculation for them.
3. **Top side effects for top drugs:** Pharmaceuticals and regulators not only want to know which drugs are being reported, but also what are the top side effects. This query selects the topmost **Drug** type for a given **Company** and counts how many times each **Reaction** for that drug is reported.

Find Similar Reported Cases

Things that have similar characteristics are similar, but how exactly do we measure this? You must decide what characteristics matter, and how to assess the strength of similarity. In a graph, an entity's characteristics include not only its attributes but also its relationships. Looking at Figure 2-14, you can see that *ReportCase* is surrounded by relationships to six other vertex types which are all potential similarity factors. It also has an edge type *similarCaseTo* where the results of similarity calculations can be stored.

The query *jaccard_nbor_reaction* implements relationship-based similarity scoring. The query's first argument *source* is a *ReportedCase* of interest. The *etype* argument tells the query what type of relationships to consider. The *topK* argument determines how many reported cases the query returns, and *sampSize* invokes sampling if each instance of a Reaction (or other characteristic) has more than this threshold number of related cases.

Once we specify which characteristics to consider, we still need to pick a formula for measuring similarity. This query uses Jaccard similarity, the most commonly used measure when the property is categorical rather than numeric. Reactions aren't numeric, so Jaccard is a good choice. Given two sets A and B, *Jaccard*(A, B) is defined as

(intersection of A and B) / (size of A + size of B - intersection of A and B)

In line 14, we initialize *Start* with the *source* vertex. Then in lines 15 to 17, we count the out-degree of this source vertex for all *etype* edges and store this in *@@set_size_A*. In lines 19 to 20, we collect *Neighbors* by traversing from *Start* over every *etype* edge.

The expression Start:s-(etype:e)-:t represents a traversal pattern in the graph. This particular pattern means

- begin with a member of the set Start, which
- connects to an edge of eType type, which
- connects to any target vertex.

The expression also defines three aliases for the three parts of the pattern: *s, e*, and *t*. The result of the FROM clause is a set of tuples (*s,e,t*) that satisfy the pattern. The alias *t* represents a member of the set of target vertices. These aliases are local; they can only be used within this SELECT block. They are unrelated to the aliases in other SELECT blocks.

If eType is *reactionTo*, then *Neighbors* will comprise all the *Reactions* of the given source *ReportedCase*. Then in lines 22 to 25, we build a set of *ReportedCases* by traversing from the *Neighbors* across *eType* edges again. If the out-degree of a neighbor is greater than *sampSize* we traverse only a sample of the connecting edges. Line 25 excludes the source vertex from the selection.

This pattern (traverse to neighbors, traverse back along the same edge type, exclude the starting vertex) is a common technique to find entities that have something in common with the starting entity. It is the graph-based technique for collaborative filtering recommendation.

Lines 26 to 32 compute the Jaccard similarity score between the source vertex and each member of *Others*. Given two sets A and B, *Jaccard*(A, B) is defined as

(intersection of A and B) / (size of A + size of B - intersection of A and B)

The efficient GSQL implementation is a little subtle. We will not go into line-by-line detail, but we point out two paradigms:

1. In our case, the sets are composed of neighbors of A and B. We do not start from sets A and B and then compute their intersection. We start from A, got to its neighbors, then got to *their* neighbors. This finds all B such that *intersection*(A,B) is not empty.
2. We use distributed processing to perform operations on multiple members of a set concurrently. The ACCUM and POST-ACCUM clauses in GSQL are implicit FOREACH loops, specifying what to do for each member of the iteration sets. The order of iteration is unspecified. The TigerGraph compute engine may operate on multiple iterations concurrency.

An ACCUM clause acts like a FOREACH loop on each set of connected vertices and edges that satisfy the preceding FROM/SAMPLE/WHERE clauses, that is, on each pattern tuple. In this SELECT block, *s* refers to a member of Neighbors, which is a Reaction, and *t* refers to a ReportedCase having that Reaction. A POST-ACCUM clause is another FOREACH loop, but it can only operate on one vertex alias (e.g., either *s* or *t*).

In lines 31 and 32 we order the *Others* vertices by descending similarity score and then prune the set to include only the *topK* vertices. Finally, we print all the vertices in *Others* and the *tSize* value.

```
CREATE QUERY jaccard_nbor_reaction(VERTEX source, STRING etype
  ="hasReactions", INT top_k=100, INT sampSize=100) FOR GRAPH faers {
  //example: ReportedCase=100640876
/*
Calculates the Jaccard Similarity between a given vertex and every other vertex. A simplified version
https://github.com/tigergraph/gsql-graph-algorithms
*/
    SumAccum<INT> @intersection_size, @@set_size_A, @set_size_B;
    SumAccum<FLOAT> @similarity;
    SumAccum<INT> @@tSize;

    Start (ANY) = {source};
    Start = SELECT s
        FROM Start:s
        ACCUM @@set_size_A += s.outdegree(etype);
    Neighbors = SELECT t
        FROM Start:s-(etype:e)-:t;
    Others = SELECT t
        FROM Neighbors:s -(:e)- :t
```

```
        SAMPLE sampSize EDGE when s.outdegree(etype) > sampSize
        WHERE t != source
        ACCUM t.@intersection_size += 1,
            t.@set_size_B = t.outdegree(etype)
        POST-ACCUM t.@similarity = t.@intersection_size*1.0/
                  (@@set_size_A + t.@set_size_B - t.@intersection_size),
            @@tSize += 1
        ORDER BY t.@similarity DESC
        LIMIT top_k;
    PRINT Others;
    PRINT Others.size();
}
```

Most reported drug for a company

The *mostReportedDrugForCompany* query takes three parameters (line 1 and 2). The first parameter *companyName* selects the company for which we want to find the top-most reported drugs. The second parameter *k* determines how many drug types we wish to return. Lastly, the third parameter filters *DrugSequence* elements with the given *role* value.

Reflect on the words "most *reported drug* for a *company*." We can logically conclude that the query must traverse *ReportedCase*, *Drug*, and *PharmaCompany* vertices. Take a look back at Figure 2-14 to see how these vertex types are connected:

Drug – DrugSequence – ReportCase – PharmaCompany

There are three hops from *Drug* to *PharmaCompany*; our query will perform its work in three stages.

In stage 1 (lines 12 to 15), we find all the *ReportedCase* vertices that relate to the company given as an input parameter. Drilling down: Line 12 builds a vertex set that contains all the *Company* vertices, because GSQL requires that a graph traversal begin with a vertex set. Lines 13 and 14 select all the *ReportedCase* vertices that link to a *Company* vertex, as long as that company's name matches the *company_name* argument.

In stage 2 (lines 18 to 20), we then traverse from the selected *ReportCase* vertices from stage 1 to their associated *DrugSequence* vertices. Line 20 filters the *DrugSequence* set to only include those whose role matches the query's *role* argument.

In stage 3 (lines 23 to 27), we connect the *DrugSequence* vertices selected in stage 2 with their associated *Drug* vertices. Of course, we need to do more than just find the drugs. We count how many cases feature a particular drug (line 25), then sort the drugs by decreasing count (line 26), and select the *k* most frequently mentioned drugs (line 27).

```
CREATE QUERY mostReportedDrugsForCompany_v2(STRING
company_name="PFIZER",
```

```
  INT k=5, STRING role="PS") FOR GRAPH faers {
    // Possible values for role: PS, SS, I, C
    // PS = primary suspect drug, SS = secondary suspect drug
    // C = concomitant, I = interacting

    # Keep count of how many times each drug is mentioned.
    SumAccum<INT> @numCases;
    # 1. Find all cases where the given pharma company is the 'mfr_sndr'
    Company = {PharmaCompany.*};
    Cases = SELECT c
        FROM Company:s -(relatedTo:e)- ReportedCase:c
        WHERE s.mfr_sndr == company_name;

    #. 2. Find all drug sequences for the selected cases.
    DrugSeqs = SELECT ds
        FROM Cases:c -(hasSequences:e)- DrugSequence:ds
        WHERE (role == "" OR ds.role_cod == role);

    # 3. Count occurences of each drug mentioned in each drug sequence.
    TopDrugs = SELECT d
        FROM DrugSeqs:ds -(hasDrugs:e)-> Drug:d
        ACCUM d.@numCases += 1
        ORDER BY d.@numCases DESC
        LIMIT k;

    PRINT TopDrugs;
}
```

Top side effects for top drugs

The query topSideEffectsForTopDrugs returns the top side effect for the most reported *Drug* of a given *Company.* Like the previous query, it also wants to find the most reported drug of a company, but it does additional work to count the side effects. Its parameter list looks the same as that of mostReportedDrugsForCompany_v2; however, here *k* refers to not only the topmost reported drugs but also the topmost frequent side effects.

As we did for the previous query, let's look at the name and description of the query to understand what vertex and edge types we must traverse. We can see that we need to include *ReportedCase*, *Drug*, and *PharmaCompany*, as well as *Reaction* (side effect). The sets up a Y-shaped graph traversal pattern:

Drug – DrugSequence – ReportCase – PharmaCompany

\– Reaction

This query has five stages. Stages 1, 3, and 4 of this query are the same or are slightly enhanced versions of stages 1, 2, and 3 in the mostReportedDrugsForCompany_v2 query.

Stage 1 is the same as Stage 1 of mostReportedDrugsForCompany_v2: find all the *ReportedCase* vertices that relate to the company given as an input parameter.

Stage 2 is new: Now that we have a set of *ReportedCase* vertices, we can count their associated Reactions. We traverse all the *ReportedCase – Reaction* edges (line 24) and then add each reaction type *r.pt* of a case c to a string list attached to that case c (line 25).

In Stage 3, we then traverse from the selected *ReportedCase* vertices from stage 1 to their associated *DrugSequence* vertices. Lines 29 and 30 perform the traversal, and Line 31 filters the *DrugSequence* set to only include those whose role matches the query's *role* argument. Line 32 copies the list of reactions attached to *ReportedCase* vertices to their associated *DrugSequences*. This last step is a GSQL technique to move data to where we need it.

In Stage 4, we connect the *DrugSequence* vertices selected in stage 2 with their associated *Drug* vertices. Besides counting the number of cases for a drug (line 38), we also count the occurrences of each *Reaction* (lines 39 and 40).

Finally, in stage 5, we take only the top *k* side effects. We do this by counting each *reaction* in *tally,* sorting them in descending order, and returning the top ones (lines 45 to 51).

```
CREATE QUERY topSideEffectsForTopDrugs(STRING company_name="PFIZER",
  INT k=5, STRING role="PS") FOR GRAPH faers {
    // Possible values for role: PS, SS, I, C
    // PS = primary suspect drug, SS = secondary suspect drug
    // C = concomitant, I = interacting
    # Define a heap which sorts the reaction map (below) by count.
    TYPEDEF TUPLE<STRING name, INT cnt> tally;
    HeapAccum<tally>(k, cnt DESC) @topReactions;

    # Keep count of how many times each reaction or drug is mentioned.
    ListAccum<STRING> @reactionList;
    SumAccum<INT> @numCases;
    MapAccum<STRING, INT> @reactionTally;
    # 1. Find all cases where the given pharma company is the 'mfr_sndr'
    Company = {PharmaCompany.*};
    Cases = SELECT c
        FROM Company:s -(relatedTo:e)- ReportedCase:c
        WHERE s.mfr_sndr == company_name;
    # 2. For each case, attach a list of its reactions.
    Tally = SELECT r
        FROM Cases:c -(hasReactions:e)- Reaction:r
        ACCUM c.@reactionList += r.pt;

    # 3. Find all drug sequences for the selected cases, and transfer
    #    the reaction list to the drug sequence.
    DrugSeqs = SELECT ds
        FROM Cases:c -(hasSequences:e)- DrugSequence:ds
```

```
        WHERE (role == "" OR ds.role_cod == role)
        ACCUM ds.@reactionList = c.@reactionList;

    # 4. Count occurences of each drug mentioned in each drug sequence.
    #    Also count the occurences of each reaction.
    TopDrugs = SELECT d
        FROM DrugSeqs:ds -(hasDrugs:e)- Drug:d
        ACCUM d.@numCases += 1,
            FOREACH reaction in ds.@reactionList DO
                d.@reactionTally += (reaction -> 1)
            END
        ORDER BY d.@numCases DESC
        LIMIT k;

    # 5. Find only the Top K side effects for each selected Drug.
    TopDrugs = SELECT d
         FROM TopDrugs:d
         ACCUM
             FOREACH (reaction, cnt) IN d.@reactionTally DO
                 d.@topReactions += tally(reaction,cnt)
             END
         ORDER BY d.@numCases DESC;

    PRINT TopDrugs[TopDrugs.prod_ai, TopDrugs.@numCases,
                   TopDrugs.@topReactions];
}
```

Chapter Summary

In this chapter we delved into two use cases to demonstrate the power of graphs to help users see the relationships in their data more clearly and completely. In the first use case, we defined what is a customer journey and described how sales groups benefit from recording and analyzing them. We then showed how a Customer 360 graph provides a powerful and flexible way to integrate customer data, which can then be represented as customer journeys. In the second use case, we showed how a 360 graph can be used to show all the possible interactions and correlations among drugs used for medical treatment. Such analysis is vital for detecting and then taking action about adverse side effects.

We introduced TigerGraph Starter Kits – demonstration databases and queries, preinstalled on TigerGraph Cloud instances, that show the basics of a variety of different use cases. We walked through the process of obtaining and installing a Customer 360 starter kit. At the same time, we walked you through the first several steps of using GraphStudio, TigerGraph's graphical user interface.

We also introduced you to GSQL, the procedural SQL-like graph query language used by the TigerGraph graph database. Readers who know SQL and a conventional pro-

gramming language should be able to learn GSQL question. We pointed out these characteristics of GSQL:

- A GSQL query is a procedure which can have input parameters. Output is accomplished by using PRINT statements.
- The lang
- Graph traverse is accomplished by SELECT blocks which are modeled after SELECT blocks in SQL.
- Know how to model and analyze customer journeys in a graph.
- Know how to use graph analytics to count and filter properties and relationships.
- Set up and run a TigerGraph Cloud Starter Kit using GraphStudio.
- Read and understand basic GSQL queries.

CHAPTER 3
Studying Startup Investments

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 4th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In this chapter, we will dive into the world of startup investments. This real-world use case shows us how three of the six graph powers help us to reveal high potential investment opportunities. The first graph power, connecting the dots, allows us to view how various actors in the investment landscape are connected. The second graph power, looking deep, offers investors a method to include connected information about those actors in our analysis. The third graph power, weighing and predicting, enables us to utilize past funding events and investment portfolios to predict the success rate of future investments.

After completing this chapter, you should be able to:

- Understand how graph data empowers connecting the dots, looking deep, and weighing and predicting drives value proposition.
- Know how to model and analyze investment opportunities.
- Know how to traverse multi-hop relationships to filter deeper connected information.

- Read and understand more advanced GSQL queries.

Goal: Find promising startups

Investing in startups is an exciting and lucrative way of building wealth. Investors poured over $156 billion into US startups in 2020. Those startups generated over $290 billion of liquidity[1]. However, nine out of ten startups will fail, and with only 40% becoming profitable, it becomes a challenge to bet on the right horse. [2]

Startups start with a founding team consisting of one or a few personnel. Over time, as they go through different development stages, their product improves, and the team grows. To fund these developments, they need money from investors. From the perspective of investment, one way to identify which startup is a proper candidate to finance is by looking at the composition of the startup team and its organization. Startups that have the right people at the right places in their organization tend to have a higher chance of success. So startups led by founders with a positive track record of building up companies are more likely to succeed in other companies. Another way to assess the investment opportunity is by looking at the startup's investors. Investors with a high return on their investment portfolio show that they can see the potential of startups in the early stages and help them grow into a more profitable business.

Investing in startups is a risky and complex assessment requiring understanding the product and market that it tries to take on and the people and organization that drive it. Investors need to have an overview of relationships between these aspects that help support the analysis of a startup's potential.

Solution: A Startup Investment Graph

Data to support the assessment of investments is mainly unstructured, because they are collected from different sources. One example of such a source is the Crunchbase dataset. This dataset contains information on investment rounds, founders, companies, investors, and investment portfolios. However, the dataset is in raw format, meaning that the data is not structured to answer the questions we have on the entities related to the startup for investment purposes. Data about the startup and the entities contributing to the current state are hidden from us unless we query for them explicitly. With graphs, we can form a schema centered around the target startup that we want to investigate and view the impact of other entities on the startup.

1 "In 2020, VCs invested $428M into US-based startups every day", techcrunch.com

2 "STARTUP STATISTICS – The Numbers You Need to Know", smallbiztrends.com

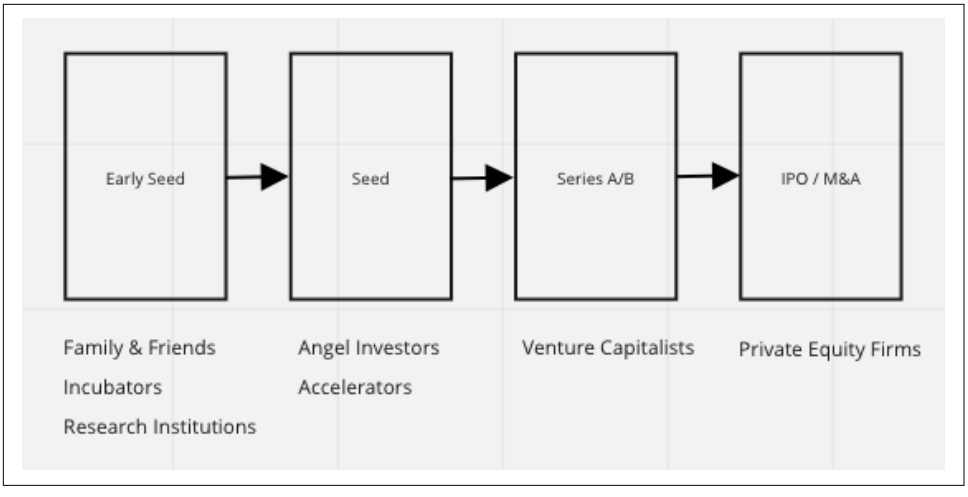


Figure 3-1. Startup funding stages and types of investors per stage

Investing in startups occurs in a series of funding events, as shown in Figure 3-1. Startups typically want to raise more money from a more extensive mixture of investors in every later funding stage. Knowing the timing and sequence of events throughout these funding stages is essential to validate successful investment interactions. Graphs can provide a complete overview of an investment network by searching for multi-hop chains of events. By doing this, we can connect angel investors and venture capitalists through different funding stages and expose their investment portfolio's success rate over time.

Traditional relational database queries provide us with a snapshot of an event and the state of each entity at a single point in time. However, when assessing investment portfolios, we need to understand the relationships between investors and the companies they have invested in and how these relationships have evolved. Graphs solve this by showing the investment portfolio as a series of events using multi-hop queries. A hop is a query from one node directly to another, whereas a multi-hop query repeats this hop to take additional connected nodes into the query. These multi-hops series return the investment events of an investor and the parties they have interacted with during those events.

For example, we want to know what startups are colleagues of a successful investor investing in now. This insight allows us to utilize successful investors' expertise and network based on their past investments. A multi-hop query can realize this by first selecting one or more successful investors. We might already have some in mind, or we could find them by counting the number of successful investors per investor; that would be one hop. The second hop selects all financial organizations where the investors work. The third hop query selects colleagues at those financial organiza-

tions, and the fourth hop selects other funding events where those colleagues participate.

Implementing A Startup Investment Graph and Queries

TigerGraph provides the startup investment graph for our use case as a cloud Starter Kit. In the previous chapter, we learned how you can set up a free account and deploy a Starter Kit. Once you've done this, we will use the Starter Kit to design the startup investment network and explain how you can derive investment opportunities using four GSQL queries.

The Crunchbase Starter Kit

Use your TigerGraph Cloud account that you created in chapter 2 to deploy a new cloud solution and select *Enterprise Knowledge Graph (Crunchbase*) as the Starter Kit. Once this Starter Kit is installed, you can load the data following the *Load Data and Install Queries* steps *for a Starter Kit* in chapter 2.

Graph Schema

The Starter Kit includes data from investments into startups in 2013 collected by Crunchbase. It has more than 575K vertices and over 664K edges, with ten vertex types and 24 edge types. In Figure 3-2, we show the graph schema of this Starter Kit. We can immediately see that Company is a vertex type that acts as a hub because it connects to many other vertex types.

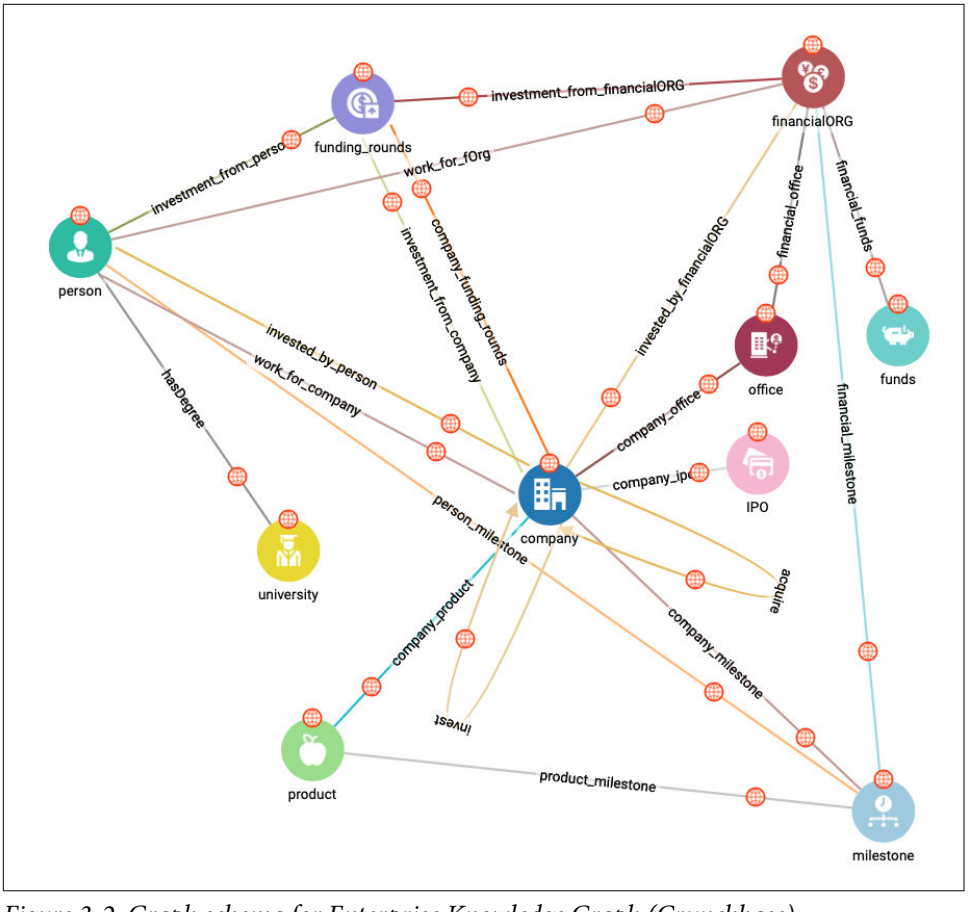


Figure 3-2. Graph schema for Enterprise Knowledge Graph (Crunchbase)

Furthermore**,** there are two types of self edges. A **Company** can **acquire** another **Company**, and a **Company** can also **invest** in another **Company**. A **Person** type vertex, on the other hand, does not have self edges, which means that a social connection always goes through another vertex type such as **University**, **FinancialORG**, **Funding_Rounds**, or **Company**. For example, if a **Person** works for a company, this type of relationship is indicated with the edge type **work_for_company**.

In Table 3-1 we describe the ten vertex types. From the description, we can see that many vertex types are related to **Company**. Some of them even have multiple relationship types that connect to **Company**. For example, a **Person** can invest in a **Company**, but it can also work for a **Company**.

Table 3-1. Vertex types in the Crunchbase Starter Kit

Vertex Type	Description

company	A company
funding_rounds	An investment event where a Company invest or receive funds
person	A natural person who works for a Company or invested in a Company
university	A university institution
financialORG	A financial institution that invest in a Company
funds	A financial investment
office	A physical office of a Company
IPO	An initial public offering of a Company
product	A product or service of a Company
milestone	A milestone that a Company has accomplished

Queries and Analytics

Let's look at the queries in the Enterprise Knowledge Graph (Crunchbase) Starter Kit. There are four queries in this Starter Kit. Each query is designed to answer questions that a potential investor or employer might ask.

Key Role Discovery
: This query finds all the persons with a key role at a given **Company** and its parent companies. A key role for a **Person** is defined as serving as a founder, CEO, CTO, director, or executive for the **Company** where they work.

Investor Successful Exits
: Given a certain investor, this query finds the startups that had a successful exit within a certain number of years after the investor invested. A successful exit is when a company has an IPO or is acquired by another company. The visual output of the query is the graph of the given investor with all its relationships with **IPO** and acquired **Company** elements. An investor could be any element of type **Person**, **FinancialORG,** or **Company**.

Top Startups Based on Board
: This query ranks startups based on the number of times that a current board member working for a top investment firm (**FinancialOrg**) was also a board member of a previous startup that had a successful exit. Investment firms are ranked by the amount of funds they invested in the past N years. Board members are scored according to their number of successful exits. In addition, the query filters output startups which are beyond a certain funding round stage.

Top Startups Based on Leader
: This query ranks startups based on the number of times one of its founders previously worked at another **Company**, during an early stage of that company, and which then went on to have a successful exit. The search is filtered to look only at a given industry sector.

Key Role Discovery

This query has two arguments. The first argument company_name is our target Company for which we want to find the persons who played key roles either there or at a parent Company. The second argument k determines how many hops from our starting company_name we will search for parent companies. This query fits very naturally with a graph model because of the k hops parameter. Figure 3-3 shows part of the graph traverse for two hops. Starting from company Com A, we could find connections to a parent company Com B and two key persons, Ben and Adam. We then look to see if Com B has key persons or has another parent company.

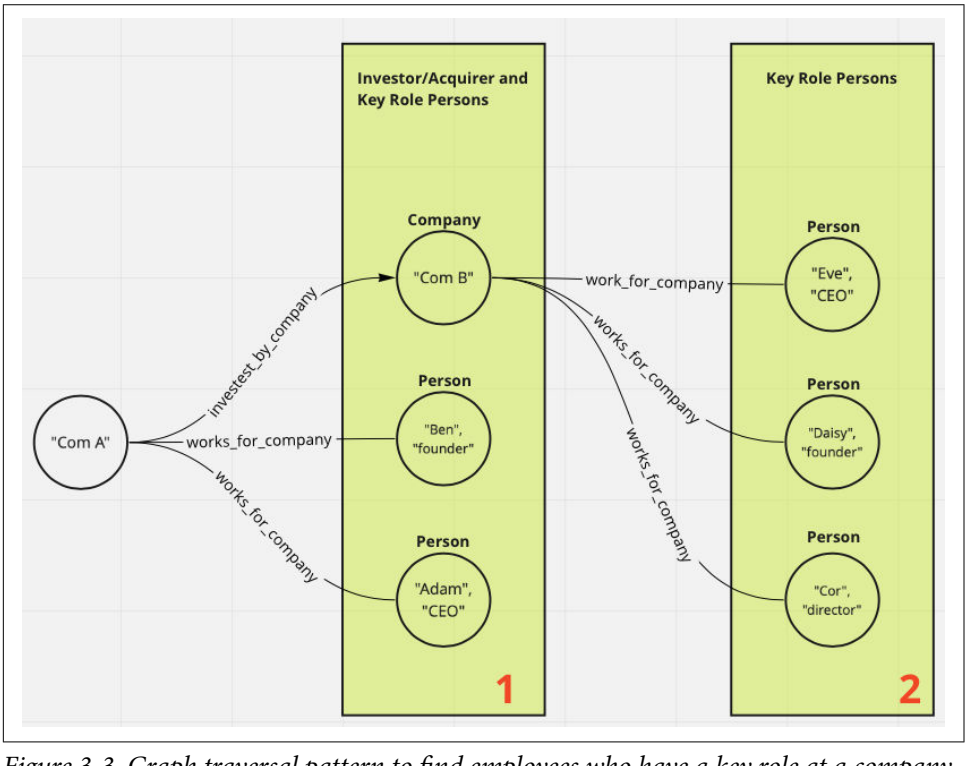


Figure 3-3. Graph traversal pattern to find employees who have a key role at a company and its parent companies

We'll now walk you through the GSQL implementation. In your starter kit, look for the query called *key_role_discovery*. Select it so you can see the code.

First, we declare some accumulators in which to gather our output objects, @@output_vertices and @@output_edges. We also declare visited to mark vertices which the query has encountered already, to avoid double-counting or searching in circles. In this dataset, if a time variable does not have a genuine value, it is set to code 0, which

translates to January 1, 1970. We declare TNULL as a more descriptive name for this situation.

```
OrAccum @visited;
SetAccum<VERTEX> @@output_vertices;
SetAccum<EDGE> @@output_edges;
DATETIME TNULL = to_datetime("1970-01-01 00:00:00");
```

Next we select all the company elements whose name attribute matches the input parameter company_name. The functions lower(trim()) remove any leading or trailing blank spaces and convert all the letters to lowercase, so that differences in capitalization won't matter. Each vertex whose name matches is added to the @@output_vertices set and is also marked as @visited.

```
Linked_companies (ANY) = SELECT tgt
    FROM company:tgt
    WHERE lower(trim(tgt.name)) == lower(trim(company_name))
    ACCUM @@output_vertices += tgt
    POST-ACCUM tgt.@visited = TRUE;
```

Now we start a WHILE loop to look for key persons and parent companies up to k levels deep. At each iteration, select all company elements that have an invested_by_company, acquired_by, or work_for_company edge to a vertex type company or person. This is a good example of the importance of selecting descriptive names for your vertices and edges.

```
WHILE TRUE LIMIT k DO
    Linked_companies = SELECT tgt
        FROM Linked_companies:s
            - ((invested_by_company> | acquired_by> | work_for_company):e)
            - (company | person):tgt
```

There is more to this SELECT block. Its WHERE clause performs additional filtering of the selected companies and persons. First, to make sure we are traversing company-to-person edges in the correct direction, we require that the source vertex (using the alias s) is a company. We also require that we haven't visited the target vertex before (NOT tgt.@visited). Then, if the edge type is work_for_company, the job title must contain "founder", "CEO", "CTO", "[b]oard [of] directors", or "executive".

```
        WHERE s.type == "company" AND tgt.@visited == FALSE AND
          (e.type == "work_for_company" AND
            (e.title LIKE "%founder%" OR e.title LIKE "%Founder%" OR
             e.title LIKE "%CEO%" OR e.title LIKE "% ceo%" OR
             e.title LIKE "%CTO%" OR e.title LIKE "% cto%" OR
             ((e.title LIKE "%oard%irectors%" OR e.title LIKE "%xecutive%")
               AND datetime_diff(e.end_at, TNULL) == 0))
            ) OR
             e.type != "work_for_company"
```

After selecting the desired vertices and edges, we add the vertices and edges to our accumulators @@output_vertices and @@output_edges, and we mark the vertices as visited.

Finally, we display the selected companies and persons with their interconnecting edges, both graphically and as JSON data. The line Results = {@@output_vertices} is due to a quirk of GSQL: for efficiency, accumulators containing vertices store only their ID values. To get all the information about the vertices, we copy them into an ordinary vertex set.[5] Finally, we display the selected companies and persons with their interconnecting edges, both graphically and as JSON data. The line Results = {@@output_vertices} is due to a quirk of GSQL: for efficiency, accumulators containing vertices store only their ID values. To get all the information about the vertices, we copy them into an ordinary vertex set.

```
IF @@output_vertices.size() != 0 THEN
    Results = {@@output_vertices}; // conversion to output more that just id
    PRINT Results;
    PRINT @@output_edges;
ELSE
    PRINT "No parties with key relations to the company found within ", k,
        " steps" AS msg;
```

In Figure 3-4, we show the output when company_name = LuckyCal and k = 3. While some of the company and person names are not in the dataset, we can guess that the company in the center is Facebook, because the Founder is Mark Zuckerberg.

Figure 3-4. Key Role Discovery when company_name = LuckyCal and k = 3

Investor Successful Exits

This query finds the achievement of a given investor, where achievement is measured by the number of investments that lead to IPOs and acquisitions. It takes three arguments. investor_name is the name of our target investor of whom we want to know the achievements, investor_type is the type of investor, which could be **company**, **person** or **financialOrg**. We use year to evaluate how many exits the investor has in a certain period. We can answer this query by using the following graph traversal pat-

tern as illustrated in Figure 3-5. Start from the selected investor vertex (investor_name).

1. Hop to the funding rounds which the investor participated in.
2. Hop to the companies funded by these rounds.
3. Hop to the exit events (acquired_by or company_ipo edges).

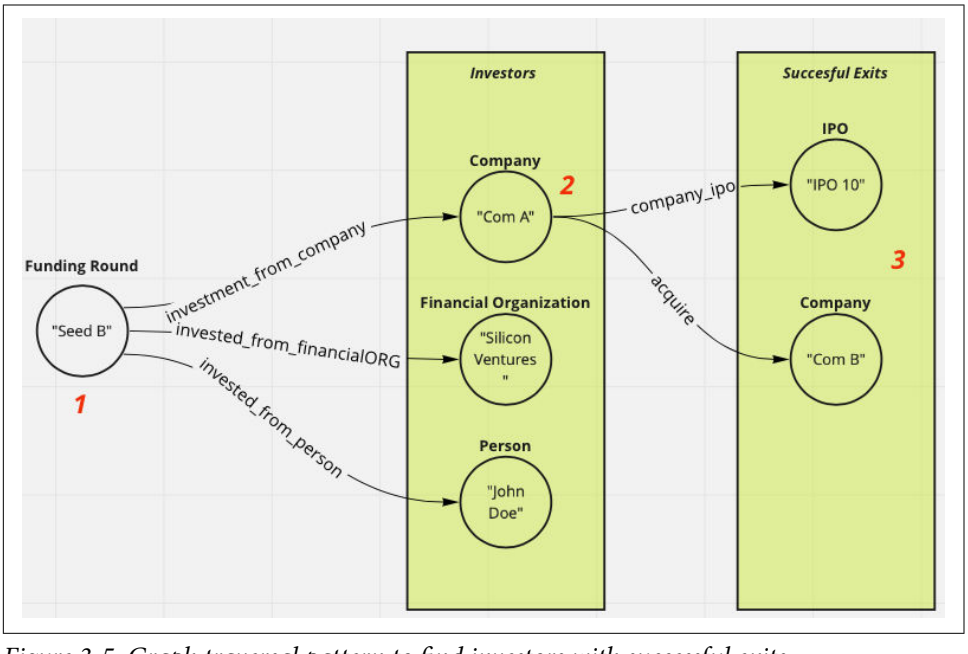


Figure 3-5. Graph traversal pattern to find investors with successful exits

We'll walk you through key parts of the GSQL code for the *investor_successful_exits* query.

We start by declaring several variables. We want to show the paths from investor to successful exits. As we traverse through the graph, parent_vertex_set and parent_edge_set act like breadcrumbs. At each newly visited vertex, we use them to record how we got there. After we reach the end, we used these accumulators to find our way backward. This time, we gather all the vertices and edges on these paths into the global accumulators result_vertex_set and result_edge_set.

```
SetAccum<VERTEX> @parent_vertex_set;
SetAccum<EDGE> @parent_edge_set;
SetAccum<VERTEX> @@result_vertex_set;
SetAccum<EDGE> @@result_edge_set;
```

Next we create the Start set of vertices, using a CASE statement and the investor_type parameter to select the right type of investors.

```
Start (ANY) = {};
CASE lower(trim(investor_type))
    WHEN "person"    THEN Start = {person.*};
    WHEN "company"    THEN Start = {company.*};
    WHEN "financialorg"  THEN Start = {financialORG.*};
END;
```

We complete the preliminaries by finding the individual investor who has investor_name. If the investor is a **person**, we check the attribute called fullname; otherwise we check the attribute called name.

```
Investor (ANY) = SELECT inv
    FROM Start :inv
    WHERE ( inv.type == "person"
         AND lower(trim(inv.fullname)) == lower(trim(investor_name))
     ) OR lower(trim(inv.name)) == lower(trim(investor_name));
```

We could have searched for the individual in one step, combining steps 2 and 3. However, that would have required scanning the whole dataset, or at least all three types of investors. By doing step 2 as a separate step, we filter down to one vertex type without scanning the graph at all.

Now we begin our graph hops. Here we find the funding rounds in which the investor participates. We start by selecting all the **funding_rounds** linked to the investor. At each selected funding_rounds vertex, store the identity of the vertex and edge traversed to arrive there. The destination or target vertices of this hop are stored in a variable called Funding_rounds (We capitalize the names of vertex sets.)

```
Funding_rounds = SELECT tgt
    FROM Investor :s - ((investment_from_company | investment_from_person |
        investment_from_financialORG) :e) - funding_rounds :tgt
    ACCUM
        tgt.@parent_vertex_set += s,
        tgt.@parent_edge_set += e;
```

Now we take another hop from the selected funding rounds to the companies they funded. An investor can invest in a company at more than one funding round. For example, in Figure 3-6, we see that Ted Leonsis invested in Revolution Money in both rounds B and C. An investor's success should be judged from the time of their first investment. Each **funding_rounds** vertex sends it funded_at time to a MinAccum @min_invested_time which remembers the minimum value that it is given.

```
Invested_companies = SELECT tgt
    FROM Funding_rounds :s - ((company_funding_rounds) :e) - company :tgt
    ACCUM
```

```
        tgt.@parent_vertex_set += s,
        tgt.@parent_edge_set += e,
        tgt.@min_invested_time += s.funded_at;
```

Finally, from each company that received investment funding, we look to see if it had a successful investment within the required time window. A company_ipo or acquired_by edge indicates an exit. If it was an IPO, we check that the IPO date (the public_at attribute) is later than the investment date but not more than years later. An analogous check is performed on the acquired_at attribute if it was an acquisition event.

```
IPO_acquired_companies = SELECT tgt
    FROM Invested_companies :s - ((company_ipo | acquired_by>) :e) - :tgt
    ACCUM
        tgt.@parent_vertex_set += s,
        tgt.@parent_edge_set += e,
        // See if IPO occured within `years` after Investor's investment
        IF (e.type == "company_ipo"
            AND datetime_diff(tgt.public_at, s.@min_invested_time) > 0
            AND datetime_diff(
            tgt.public_at, s.@min_invested_time) <= years * SECS_PER_YR)
        // See if Acquisition occured within `years` of investment
        OR (e.type == "acquired_by"
            AND datetime_diff(e.acquired_at, s.@min_invested_time) > 0
            AND datetime_diff(
            e.acquired_at, s.@min_invested_time) <= years * SECS_PER_YR)
        THEN @@result_vertex_set += tgt
        END;
```

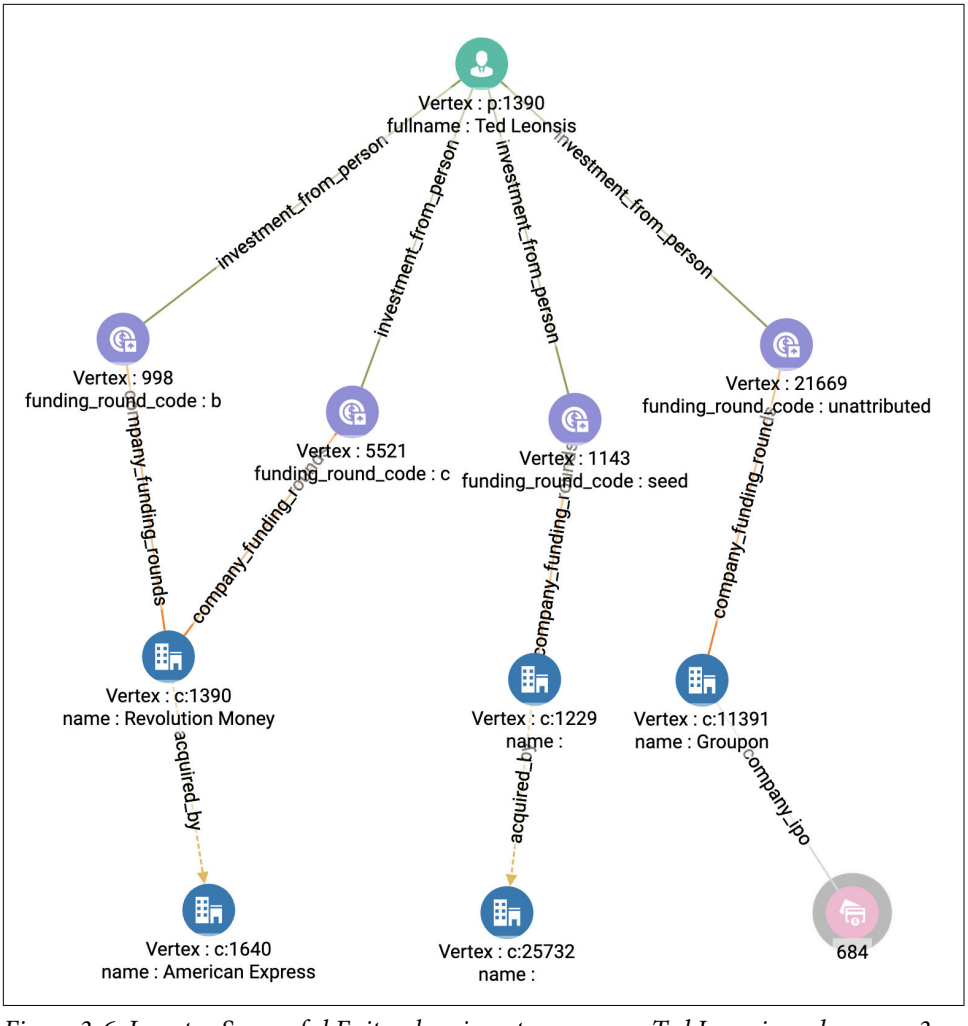


Figure 3-6. Investor Successful Exits when investor_name = Ted Leonsis and years = 3

If we only wanted to know how many successful exits our investor had, or the company details of those exits, we would be done. However, it's interesting to graphically show the paths from investor → funding → company → exit, as in Figure 3-6. To gather that information, we traverse from the exit vertices backward to the investor, using the breadcrumbs (parent_vertex_set and parent_edge_set) that we set previously.

```
Children = {@@result_vertex_set};
PRINT Children.size() as Num_Successful_Exits;
WHILE(Children.size() > 0) DO
    Start = SELECT s
        FROM Children :s
```

```
        ACCUM
            @@parents += s.@parent_vertex_set,
            @@result_edge_set += s.@parent_edge_set;
    @@result_vertex_set += @@parents;
    Children = {@@parents};
    @@parents.clear();
END;
```

Top Startups Based On Board

This query adds some complexity by adding in two forms of ranking: top performing investment companies and top performing leaders at those investment companies. It starts by identifying the **financialORG** entities which have raised the most money in recent years. Then, we rank **persons** at those **financialORGs** according to the number of times they were on the board of a startup **company** and guided it to a successful exit. Then we display any pre-exit **companies** that currently have one of these successful executives as a current board member.

The *top_startups_based_on_board* query have four input parameters:

- k_orgs is the number of top financial institutions we want to include in our selection scope.
- num_persons is the number of top board members to select.
- max_funding_round filters the list of promising startups to exclude those that have received investment funding at a later stage than max_funding_round.
- past_n_years restricts the selection until a specific year.

We can implement this query according to the following steps, most of which correspond to a graph hop. These steps are illustrated in Figure 3-7.

1. Compute how much **funding_rounds** investment each **financialORG** made in the past N years [Hop 1].
2. Rank the **financialORGs** by the investment amount and take the top k_orgs.
3. Find **persons** who work for a top k **financialORG** (from Step 2) [Hop 2].
4. Find companies at which those **persons** (from Step 3) served as a board member [Hop 3].
5. Rank those **persons** (from Step 3) by the number of times they were on the board of a **company** (from Step 4) before its successful exit [Hop 4].
6. Find pre-exit **companies** that have a top board member **person** (from Step 5). Filter these startups (from Step) by the funding round cutoff [Hop 5].

The *top_startups_based_on_board* declares several accumulators and other variables to assist with this computation. There are also two interesting data preparation steps.

One stores some currency exchange rates in a lookup table. Another makes a list of all the funding round codes @@allowed_funding_rounds up to our max_cut-off_round.

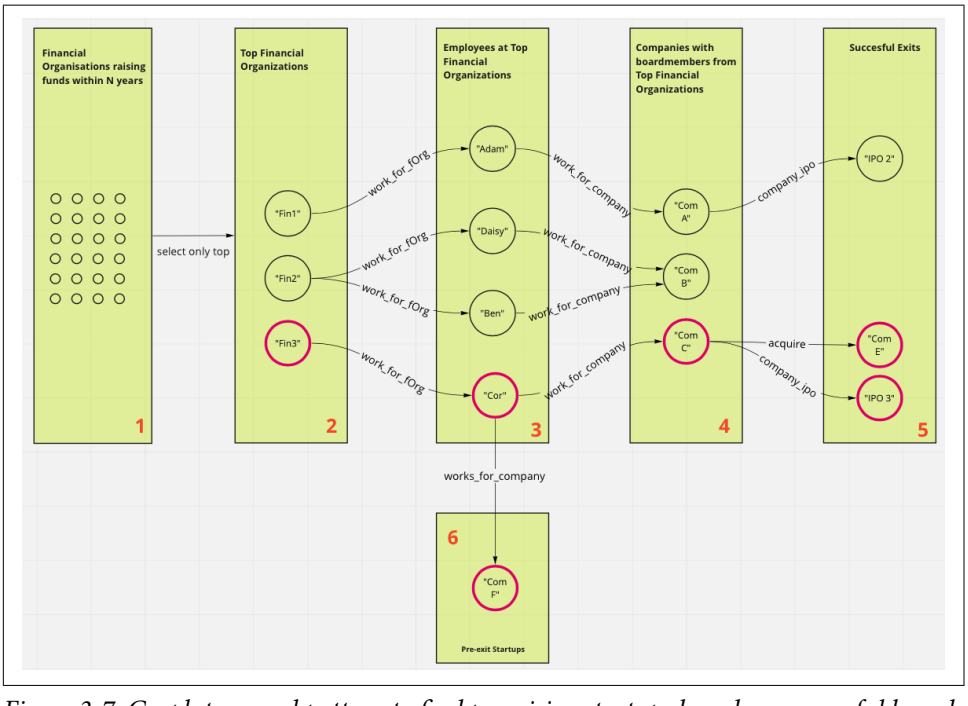


Figure 3-7. Graph traversal pattern to find promising startups based on successful board members from top financial organizations

Our first graph hop is also a data preparation step. Our Crunchbase graph schema stores the IPO or acquisition date of a company on an edge. Copy that data so that it is also available with the companies themselves.

```
Comp = SELECT c
    FROM (company) :c - ((company_ipo|acquired_by>): e) - (IPO|company): x
    ACCUM
        CASE WHEN
          e.type == "company_ipo" AND datetime_diff(x.public_at, T0) != 0
        THEN
          c.@t_exit += x.public_at
        END,
        CASE WHEN
          e.type == "acquired_by" AND datetime_diff(e.acquired_at,T0) != 0
        THEN
          c.@t_exit += e.acquired_at
        END;
```

In the next hop, we connect **financialORGs** with their investment **funds** in order to tally the investments of the past_n_years and then take the top k financialORGs. To take the top k, GSQL offers ORDER BY and LIMIT clauses, just as in SQL

```
Top_orgs = SELECT org
    FROM (financialORG): org - (financial_funds: e) - funds: f
    WHERE datetime_diff(TIME_END, f.funded_at) <= past_n_years*SECS_PER_YR
    ACCUM org.@amount +=
        (f.raised_amount / @@currency2USD.get(f.raised_currency_code)),
        f.@visited = TRUE
    ORDER BY org.@amount DESC
    LIMIT k_orgs;
```

Advanced GSQL users may sometimes choose to use HeapAccum instead of ORDER BY/LIMIT because incremental sorting uses less computer memory than sorting all of the active data at once.

Select all employees (**person** who **work_for_fOrg**) at these top financial organizations (the Top_org vertex set from the previous step).

```
Persons_at_top_orgs = SELECT p
    FROM Top_orgs: o - (work_for_fOrg: e) - person: p;
```

From these Persons_at_top_orgs, find the companies they worked and which satisfy all of these conditions:

- their job title included “Board”,
- the company has had an exit (c.@t_exit.size() != 0),
- the person has a valid work start date (datetime_diff(w.start_at, T0) != 0)
- and the company’s exit occurred after the board member joined.

```
Top_board_members = SELECT p
    FROM Persons_at_top_orgs: p - (work_for_company :w) - company: c
    WHERE (w.title LIKE "%Board%" OR w.title LIKE "%board%")
        AND c.@t_exit.size() != 0 AND datetime_diff(w.start_at, T0) != 0
        AND datetime_diff(c.@t_exit.get(0), w.start_at) > 0
    ACCUM
        @@comp_set += c,
        c.@board_set += p,
        p.@amount += 1
    ORDER BY p.@amount DESC
    LIMIT num_persons;
```

After finding the persons that satisfy all these conditions, we build a list of companies (@@comp_set) so that we exclude them when looking for new investment opportuni-

ties. We also have each company record its key board member (c@board_set), and we tally the successful exits of each key person (p.@amount += 1). Finally, we take the most prolific board members (ORDER BY/LIMIT).

Find all pre-exit **company** entities that have a top_board_member, from our list in the previous step. Exclude companies from the previous step.

```
Top_startups = SELECT c
    FROM Top_board_members: s - (work_for_company: w) - company: c
    WHERE (w.title LIKE "%Board%" OR w.title LIKE "%board%")
        AND w.start_at != T0
        AND c.status == "operating" AND c NOT IN @@comp_set;
```

Finally, include only those pre-exit companies whose **funding_rounds** have been early enough to satisfy the max_cutoff_round limit..

```
Top_early_startups = SELECT r
    FROM Top_startups: s - (company_funding_rounds: e) - funding_rounds: r
    ACCUM
        s.@visited += TRUE,
        IF @allowed_funding_rounds.contains(r.funding_round_code) THEN
            r.@visited = TRUE
        ELSE
            s.@early += FALSE
        END;
```

The remainder of the query is used to trace back from the top board members to display the companies they worked for and their successful exits.

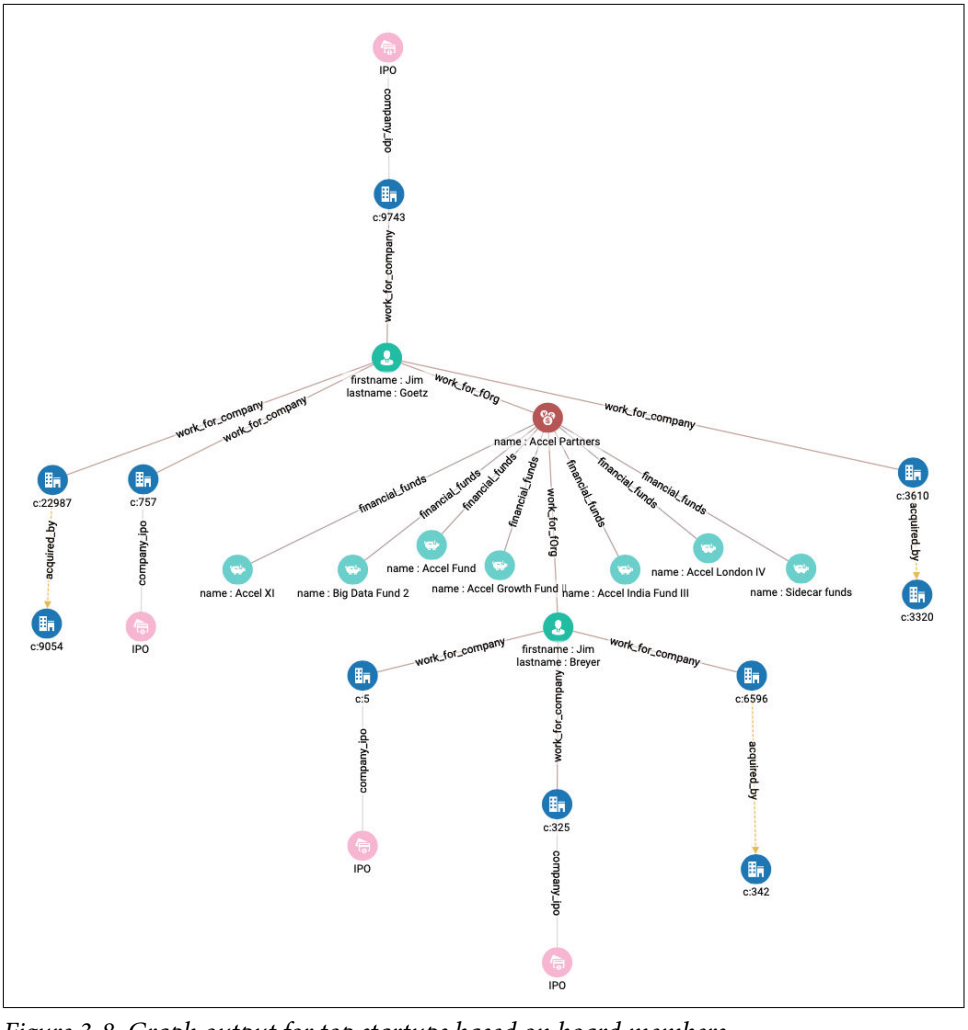


Figure 3-8. Graph output for top startups based on board members

Figure 3-8 shows the results when we set k_orgs = 10, num_persons = 2, max_funding_round = b, and past_n_years = 10. The two key board members Jim Goetz and Jim Breyer, who both work from Accel Partners. Goetz has had four successful exits while Breyer has had three.

Top Startups Based On Leader

Our last query in this starter kit is similar to the previous one, except that rather than looking for top board members, we are looking for founders. This query takes three arguments: *max_funding_round* is the funding round cut-off, meaning that we only select startups whose investment rounds have been no later than *max_funding_round*.

Argument *return_size* is the number of top startups we want to retrieve from our query, and *sector* is the industry sector we want to filter out the result.

Figure 3-9 illustrates how we construct this query as a series of graph hops.

1. Find all companies that have IPOed or been Acquired [Hop 1].
2. Find employees who contributed to the companies in Step 1 [Hop 2].
3. Find startups whose founder also was a key employee from Step 2 [Hop 3]. Filter the startups based on the cutoff round and sector.
4. Find companies whose founders have the most successful connections.

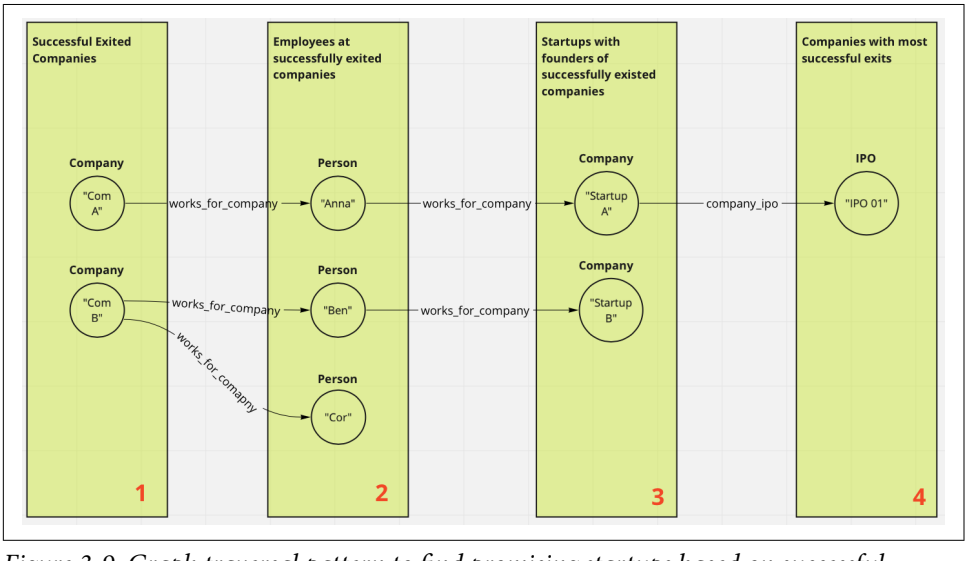


Figure 3-9. Graph traversal pattern to find promising startups based on successful founders

This query introduces some data structures that we haven't seen before. First, define a Tuple and a HeapAccum. A HeapAccum manages a sorted list of tuples up to a maximum number of items. This particular HeapAccum will maintain a list of company vertices sorted by some score. The heap can contain up to return_size companies.

```
TYPEDEF TUPLE<VERTEX<company> company, INT score> Score_Results;
HeapAccum<Score_Results>(return_size, score DESC) @@top_score_results_heap;
```

We also define two nested MapAccums. A map is like a lookup table. Looking at the structural definition of @@person_company_leave_date_map, this means that for a given person, we record when that person left a given company. For @@person_company_employment_map, we record the employment relationship between a person and a company.

```
    // Declare map to store when an employee left which company <1>
    MapAccum<VERTEX<person>,
        MapAccum<VERTEX<company>, DATETIME>> @@person_company_leave_date_map;
    MapAccum<VERTEX<person>,
        MapAccum<VERTEX<company>, EDGE>> @@person_company_employment_map;
```

Now we find all the companies with an IPO or which have been acquired by another company. For clearer code, we have a code that focuses only on IPO, another that focuses on acquisitions, and then we merge the two sets of companies. For the IPOs, we traverse from **IPO** vertices to **company** vertices. We check that the IPO has a valid public_at attribute. Once selected, we tag each **company** with the path back to the **IPO** vertex and with the public_at date. We tag the company as no longer in startup phase.

```
    IPO_companies = SELECT c
        FROM IPO :i - (company_ipo :e) - company :c
        //Filter out companies with null acquisition time (not yet acquired)
        WHERE datetime_diff(i.public_at, TNULL) != 0
        ACCUM
            c.@parent_vertex_set += i,
            c.@parent_edge_set += e,
            c.@min_public_date = i.public_at,
            c.@is_still_startup += FALSE;
```

A similar code block finds the Acquired_companies. The edge type is different (acquire instead of company_ipo), and the effective data attribute is different (acquired_at instead of public_at).

We then join the output sets from these two blocks:

```
    IPO_acquired_companies = IPO_companies UNION Acquired_companies;
```

Next we select all the persons who have worked for a successfully exited company before the exit event. For each such person, we store their relevant information into the nested maps that we described earlier. Notice the -> operator used to specify a map's key -> value pair.

```
Startup_employees = SELECT p
     FROM IPO_acquired_companies :c - (work_for_company :e) - person :p
      WHERE datetime_diff(e.start_at, TNULL) != 0
        AND datetime_diff(e.end_at, TNULL) != 0
        AND datetime_diff(e.start_at, c.@min_public_date) < 0
      ACCUM
            @@person_company_employment_map += (p -> (c -> e)),
            @@person_company_leave_date_map += (p -> (c -> e.end_at));
```

Now find the startups where these successful-exit employees are currently a founder, filtered by industry. The checks for the startup status and founder status are performed in the WHERE clause.

```
Startups_from_employees = SELECT c
    FROM Startup_employees :p - (work_for_company :e) - company :c
    WHERE c.@is_still_startup AND c.@has_input_funding_round
        AND c.status != "acquired"
        AND c.status != "ipo"
        AND e.title LIKE "%ounder%"
        AND lower(trim(c.category_code)) == lower(trim(sector))
        AND datetime_diff(e.start_at, TNULL) != 0
        AND datetime_diff(e.end_at, TNULL) != 0
```

After selecting these startups, we tally the founders' past successes.

```
    ACCUM
    // Tally the founder:past-success relationships per new company
        FOREACH (past_company, leave_date)
        IN @@person_company_leave_date_map.get(p) DO
            IF datetime_diff(e.start_at, leave_date) > 0 THEN
                p.@parent_edge_set +=
                   @@person_company_employment_map.get(p).get(past_company),
                p.@company_list += past_company,
                c.@parent_vertex_set += p,
                c.@parent_edge_set += e,
                c.@sum_ipo_acquire += 1
            END
        END
    HAVING c.@sum_ipo_acquire > 0;
```

Select companies where the founders have the most relationships with successfully exited companies. We use the HeapAccum we described previously to rank the companies based on the tally of successful exits of its founder(s).

```
Top_companies = SELECT c
    FROM Startups_from_employees :c
    ACCUM @@top_score_results_heap += Score_Results(c, c.@sum_ipo_acquire);
PRINT @@top_score_results_heap;
FOREACH item IN @@top_score_results_heap DO
    @@output_vertex_set += item.company;
END;
```

Figure 3-10 shows the results when the input arguments are max_funding_round = c, return_size = 5, and sector = software. The five selected startups are listed on the right. Looking at the second company from the top, we read from right to left: Packet Trap Networks is selected because founder Steve Goodman was a Founder/CEO of Lasso Logic which was acquired by SonicWALL.

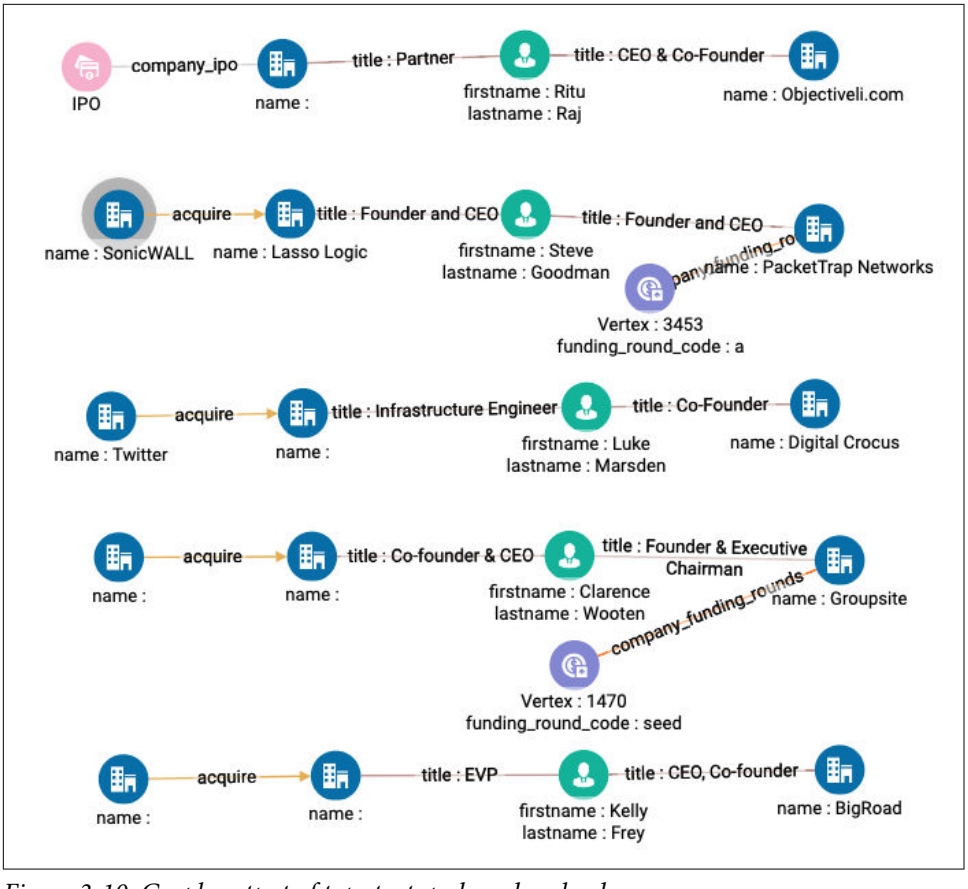


Figure 3-10. Graph output of top startups based on leader

Chapter Summary

In the chapter we have seen how we can use graph analytics to answer important questions and gain valuable insight about startup investments. Looking at the graph schema for Crunchbase data, we see that such data is highly interconnected. In the case of investment advice, we often look to past performance as an indicator of possible future results. So, we look for one pattern (success in the past) and see if there is potential for a repeat of that pattern. This type of pattern search or similarity search is typical of graph analytics.

This chapter also gave us an opportunity to learn more about writing and reading GSQL queries, through some practical examples. The queries demonstrated several techniques, such as

- using a WHILE loop to search multiple levels deep.

- tagging vertices with a boolean accumulator to mark that it has been visited
- during multi-step traversal, tagging vertices with a parent_vertex and a parent_edge to serve as breadcrumbs, so we can recover our paths later.]
- The ORDER BY and LIMIT clauses in a SELECT block can be used to find the top ranked vertices, similar to selecting the top ranked records in SQL.

CHAPTER 4

Detecting Fraud and Money Laundering Patterns

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 5th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

In this chapter, we take on the serious problem of fraud and money laundering. Fraud is typically conducted by one or more parties as a multi-step process. Sometimes, the only way to distinguish fraud or money laundering from legitimate activity is to detect a characteristic or unusual pattern of activity. Modeling the activity and relationships with a graph enables us to detect suspicious activity by searching for those patterns along with checking for their frequency.

After completing this chapter, you should be able to:

- Describe coordinated activity among multiple parties in terms of a graph pattern.
- Use a multi-hop or iterated single-hop graph traversal to perform a deep search.
- Describe bidirectional search and its advantage.
- Understanding the use of timestamps to find a time sequence.

Goal: Detect Money Laundering

Financial institutions are responsible for averting criminal money flows through the economic infrastructure. According to The Financial Action Task Force (FATF), illicit funds amount to 3.6% of global GDP[1]. A well-known criminal activity is money laundering. Money laundering is the process of disguising the illegal origin of the money. According to the FATF 2.7% of the global GDP is laundered per year. As part of the banking license, banks are obligated to investigate their clients' payment behavior and report any suspicious activities.

Banks have built a wide range of applications and procedures into their daily operations to identify and detect money laundering. Broadly speaking, these financial crime detection efforts can fall into two areas. The first area of investigation, Know Your Customer (KYC), looks into the client profile. Similar to what we've seen in Chapter 3 with the Customer 360 use case, analysts need to conduct client due diligence. This client risk assessment can happen at multiple stages of the client lifecycle, such as during new client take over (NCTO) or during a periodic review. The second area of investigation,Transaction Monitoring, mainly focuses on identifying criminal behavior through bank transactions. Here analysts try to identify unusual payment patterns between sender and beneficiaries. Although these two investigation areas often overlap from a bank operational perspective and on a risk management level, we will mainly focus on transaction monitoring in this chapter.

Transaction monitoring involves thorough investigations into entities that show suspicious payment behavior. Analysts start these investigations from an entity that has been flagged by business rules and moves from there to other related instances to explore high-risk interactions. Thus analysts do not know the complete picture of how the money flows and lack visibility on where the flagged entity is in the entire money trail. To gain this visibility, they have to query step-by-step the next payment interaction to build up a complete picture of the payment network. Therefore analysts need an approach that helps them retrieve a set of consecutive payments and the parties involved in those payments.

Solution: Modeling Financial Crimes as Network Patterns

Traditional transaction monitoring relies on rule-based systems where fixed risk indicators are triggered if it applies to a client's payment behavior. Such a risk indicator could be when, for example, clients received 15,000 USD cash in their account and immediately sent that money to several third-party accounts. This kind of activity is known as layering in the context of anti-money laundering (AML). It indicates a sus-

1 "What is Money Laundering?", fatf-gafi.org

picious activity because it revolves around a large amount of cash, and that money moves to several third parties, making it harder to backtrack its origin. There are two major problems with relying on rule-based risk indicators. First, the analyst is still required to do an in-depth follow-up investigation on flagged clients, which involves querying consecutive payments between different clients. Second, rule-based risk indicators are set up based on analysts' known patterns. Thus they only cover cases from historical records, making it challenging to identify risks from unforeseen patterns.

When modeling this problem as a network, it becomes easier to identify high-risk patterns because we can visualize the money flow directly from the graph data model. Doing so shows us how the money moves in a network and which parties are involved in those payment interactions. This graph approach solves the first problem because the graph pattern search will discover the consecutive payments for the analyst. It also solves the second problem because the network will expose all the relationships between involved parties, including those that the analysts do not explicitly query.

Implementing Financial Crime Pattern Searches

TigerGraph provides a graph for detecting money laundering as a cloud Starter Kit. Follow the installation steps from chapter 2 to install the Starter Kit. After the installation we will use the Starter Kit to design our money laundering network and explore how we can detect suspicious payment interactions on this network.

The Fraud and Money Laundering Detection Starter Kit

UsingTigerGraph Cloud, deploy a new cloud solution and select *Fraud and Money Laundering* as the Starter Kit. Once this Starter Kit is installed, you can load the data following the *Load Data and Install Queries* steps *for a Starter Kit* in chapter 2.

Graph Schema

The Fraud and Money Laundering Detection Starter Kit contains over 4.3M vertices and 7M edges, with a schema that has four vertex types and five edge types. Figure 4-1 shows the graph schema of this Starter Kit.

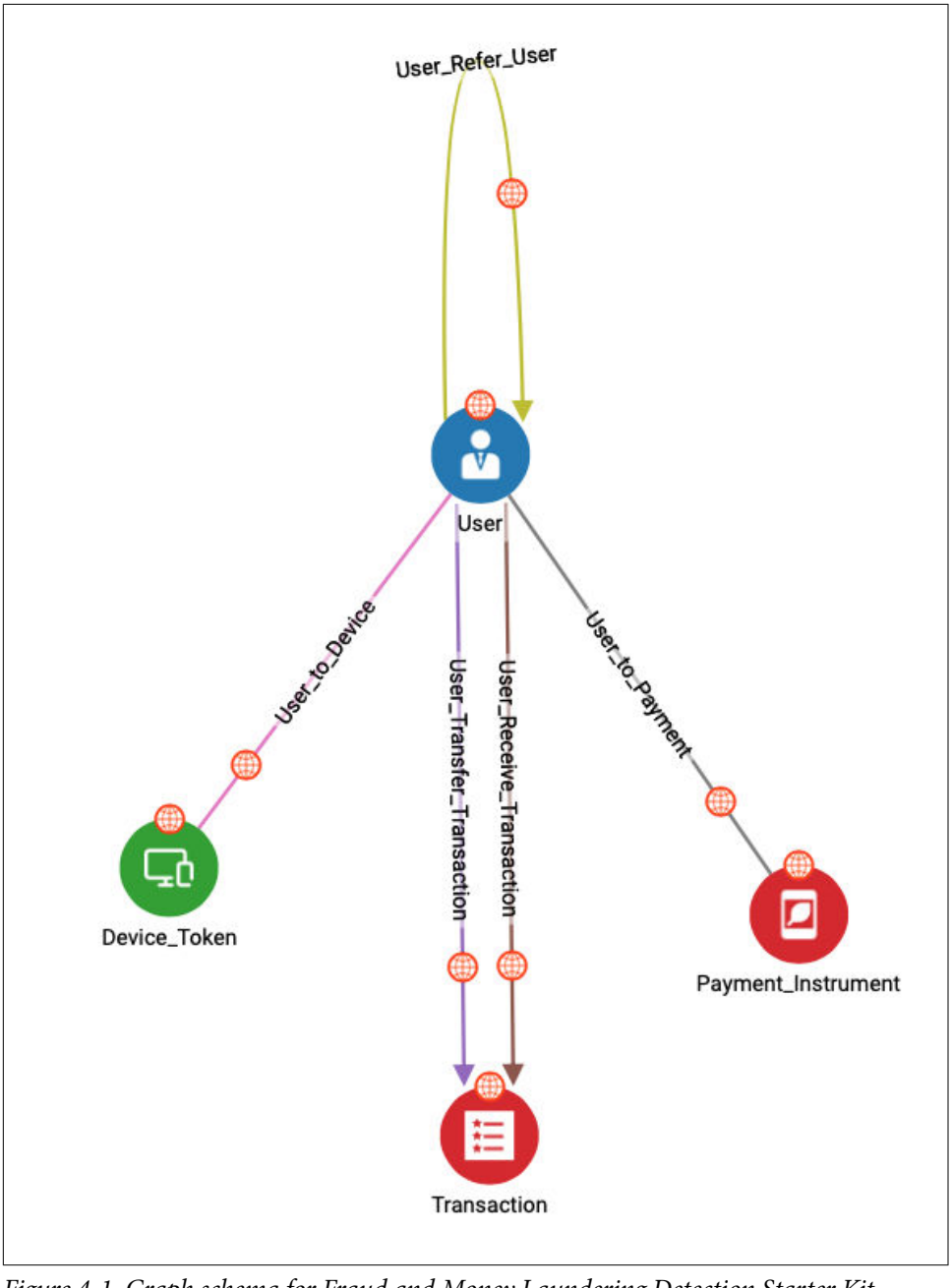


Figure 4-1. Graph schema for Fraud and Money Laundering Detection Starter Kit

In Table 4-1 we describe the four vertex types. A **User** has a central role in a payment interaction, where it can receive and send payments. **Transaction** is the payment

itself. **Device_Token** is a unique ID number that refers to the device used for the payment, and **Payment_Instrument** refers to the type of instrument used for the payment.

Table 4-1. Vertex types in Fraud and Money Laundering Detection Starter Kit

Vertex Type	Description
User	A person who is involved in a payment
Transaction	A payment
Device_Token	A unique ID number used to carry out the Transaction
Payment_Instrument	An instrument to execute the payment

There are two types of relationships between **User** and **Transaction**. A **User** can receive a transaction, denoted with **User_Receive_Transaction**, or a **User** can send a transaction, marked with **User_Transfer_Transaction**. A **User** can refer another **User**, which is indicated by **User_Refer_User**. The edge type **User_to_Payment** links a **User** to a **Payment_Instrument** (check, cash, warrant, etc.) used to carry out a transaction. Finally, the **User_to_Device** edge type connects a **User** to the **Device_Token** used when making an electronic payment.

Queries and Analytics

The queries included in this starter kit showcase how graphs can help analysts detect high-risk payment behavior to combat fraud and money laundering. We'll first give a high-level description of the pattern that each query looks for and how this pertains to transaction fraud or money laundering. We then go into depth for three of them, to give you a better idea of how they are implemented.

1. **Circle Detection:** This query detects when money moves in a circular flow. It selects all the **Transaction** elements which form a chain from an input **User** and then return to that **User**. If the amount of money that comes back is close to the amount that went out, then this may indicate money laundering.
2. **Invited User Behavior:** This query looks for suspicious patterns of **User_Refer_User** behavior, which may indicate that a User is collaborating with other parties to collect referral bonuses. It looks at the number of referrals within two hops of a source User, and at the number of transactions these users have conducted.
3. **Repeated User:** This query discovers if there is a connection among **User** elements that send money to the same receiver. It starts with the input **User** who receives money and selects all other **User** elements that send the money to that input **User**. Then it checks if there is a path between those senders using **Device_Token**, **Payment_Instrument**, and **User**.

4. **Same Receiver Sender:** This query detects if a **User** uses a fake account to send money to itself. Given an input **Transaction**, this query returns true if the receiver and sender can be linked to each other by **Device_Token** and **Payment_Instrument**.
5. **Transferred Amount:** This query looks within a given time window for the total amount of funds transferred out from the Users who are connected within a few hops of a source User. While not directly suspicious, a high volume of funds could help to build the case for AML layering.
6. **Multi Transaction**: This query showcases the payments between two networks of **User** elements. Starting with an input **Transaction**, the first group is a network of **User** elements related to the sending party. The second group is a network of **User** elements from the receiving party. Then visualize all the money flow between the **User** elements of these two groups.

We now take a closer look at Invited user Behavior, Multi Transaction, and Circle Detection.

Invited User Behavior

This pattern assumes that a User can earn a referral bonus for referring a new User to the bank. This query contains a two-hop traversal implementation. We start our traversal from a given input_user. The first hop selects all the invited User elements that are invited by this input_user. Then, with the second hop, we collect all the User elements that the first order invitees invite. We then aggregate the transaction amount of those invitees. The input_user is a fraudulent User if the amount of money directly being transferred is high, while the aggregated money from the second-order invitees is low or zero. The intuition behind this is that input_user has many fake referrals that fuel itself with referral bonuses so that it can send a large number of transactions.

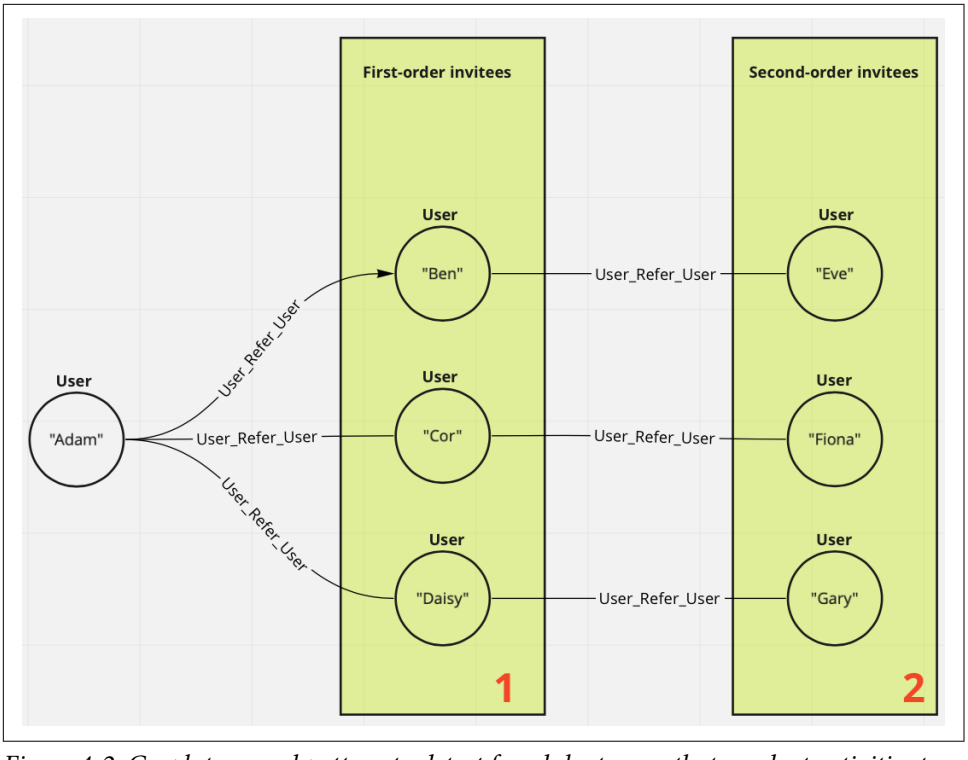


Figure 4-2. Graph traversal pattern to detect fraudulent users that conduct activities to earn referral bonus

First we declare some accumulator variables to store our aggregated data.

```
SumAccum<INT> @@num_invited_persons;
SumAccum<FLOAT> @@total_amount_sent;
SetAccum<EDGE> @@edges_to_display;
```

The variable @@num_invited_persons stores the number of second-hop invitees. We aggregate the amount of all transactions from the one-hop invitees and store that in @@total_amount_sent. With @@edges_to_display we store all the edges (User_Ref_User) between the input **User** and a referred **User**, so that the visualization system knows to display them.

Then, we find the one-hop invitees referred by the source **User**. We save each edge between the Start user and an invitee in @@display_edge_set.

```
Start = {input_user};

First_invitees = SELECT t
    FROM Start:s - (User_Refer_User>:e) - :t
    ACCUM @@edges_to_display += e;
```

In the FROM clause, we don't need to specify what type of vertices we are targeting, because the edge type (User_Refer_User) only permits one type of target vertex (User)..

Next, we add up the amount of money that these first-order invitees have sent out. Each **Transaction** has an attribute called amount.

```
Trans = SELECT t
    FROM First_invitees:s - (User_Transfer_Transaction>:e) - :t
    ACCUM
        @@total_amount_sent += t.amount,
        @@edges_to_display += e;
```

Finally, we get the additional invitees referred by first-hop invitees. This search looks very much like the first hop, with two additional steps:

We check that we are not hopping back to the source **User**.

We count the number of second-order invitees.

```
Second_invitees = SELECT t
    FROM First_invitees:s - (User_Refer_User>:e) - :t
    WHERE t != input_user
    ACCUM @@edges_to_display += e
    POST-ACCUM (t) @@num_invited_persons += 1;
```

Multi Transaction

Analysts motivate that criminals transfer money between two networks. The following query exposes this intuition. Given any input transaction, the first network consists of related accounts from the sender of that transaction, and the second network consists of associated accounts from the receiving party. Then we look for payment activities between all parties from those two networks.

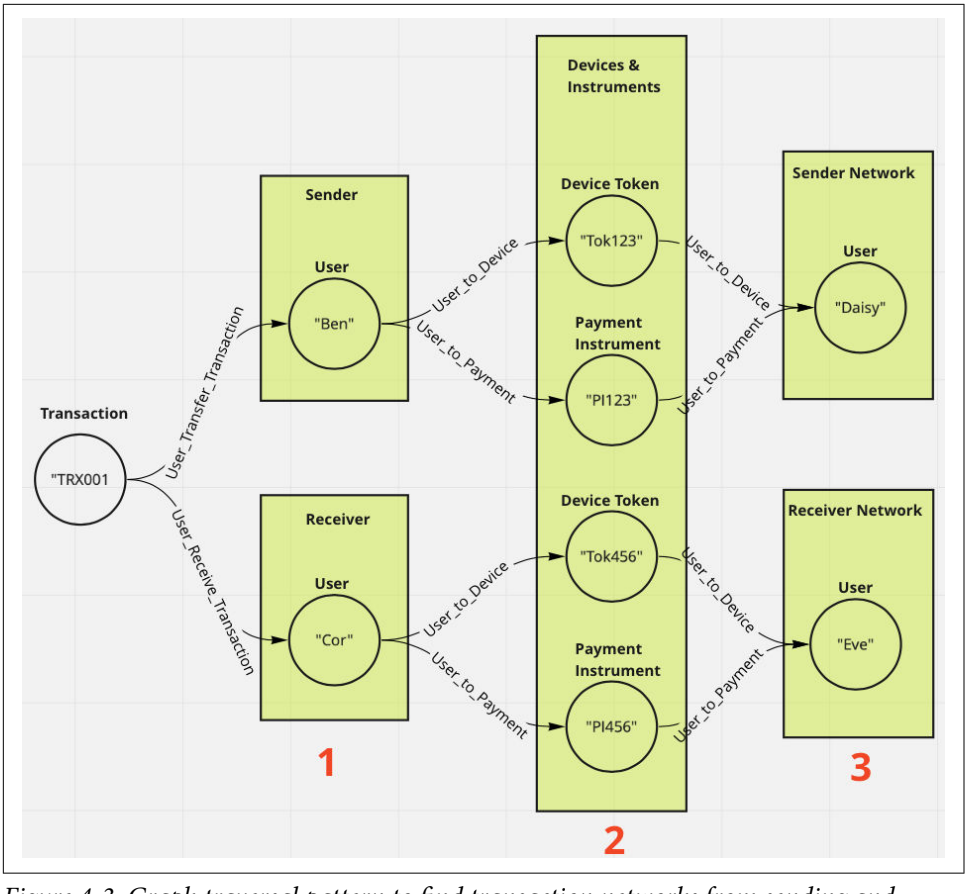


Figure 4-3. Graph traversal pattern to find transaction networks from sending and receiving parties

We start with selecting the sender and receiver **User** elements for a given **Transaction** by traversing **User_Transfer_Transaction** or **User_Receive_Transaction** edge types.

```
Start (ANY) = {source_transaction};
Start = SELECT t FROM
    Start:s-((<User_Receive_Transaction|<User_Transfer_Transaction):e)-User:t
```

In the FROM clause, we are traversing from a **Transaction** (source_transaction) to **User** elements, which is the reverse direction of the **User_Receive_Transaction** and **User_Transfer_Transaction** edges. That is why the direction arrows point to the left and are on the left side of the edge type names. Alternatively, if those edges have reverse edge types defined, we could use their reverse edges instead (and use right-facing arrows.)

We use cases to determine if the **User** is a receiving or sending party of the **Transaction**. If a **User** connects to a **Transaction** via **User_Receive_Transaction,** we set @from__receiver to true and add that **User** to the @@receiver_set. In other cases, the **User** is a sending party of the **Transaction**, so we set @from_sender to true and add this **User** to @@sender_set.

```
    CASE WHEN e.type == "User_Receive_Transaction" THEN
      t.@from_receiver += true,
      @@receiver_set += t
  ELSE
      t.@from_sender += true,
      @@sender_set += t
```

Now that we know the sender and receiver, we find **User** elements that belong to the receiving or sending party. That is, we traverse over **User_to_Device** or User_to_Payment edges and add **User** elements to either the @@sender_set or @@receiver_set if they exist within four hops (WHILE Start.size() > 0 LIMIT 4 DO). Since it takes two hops to make a transaction (Sender - Transaction - Receiver), 4 hops equals a chain of two transactions.

```
    WHILE Start.size() > 0 LIMIT 4 DO
        Start = SELECT t FROM Start:s - ((User_to_Device|User_to_Payment):e) - :t
            WHERE t.@from_receiver == false AND t.@from_sender == false
            ACCUM
                t.@from_receiver += s.@from_receiver,
                t.@from_sender += s.@from_sender,
                @@edgeSet += e
            POST-ACCUM (t)
                CASE WHEN t.type == "User" AND t.@from_sender == true THEN
                    @@sender_set += t
                WHEN t.@from_receiver == true THEN
                    @@receiver_set += t
```

If we end up at a **User** vertex type and that **User** is a sending party, we add that **User** to @@sender_set. If t.@from_receiver is true, then the **User** belongs to the receiving party, and we add that **User** to @@receiver_set.

After forming the sending and receiving groups, we now look for transactions other than the source transaction which connect the sender and receiver groups. First we find transactions adjacent to the receiver set:

```
    Start = {@@receiver_set};
    Rec_T = SELECT t FROM
        Start:s - ((User_Receive_Transaction>|User_Transfer_Transaction>):e) - :t
        ….
```

Then we find transactions adjacent to the sender set.

```
    Start = {@@sender)set};
    Send_T = SELECT t FROM
```

```
Start:s - ((User_Receive_Transaction>|User_Transfer_Transaction>):e) - :t
    WHERE t != transaction
    ACCUM
        t.@from_sender += s.@from_sender,
        @@edge_set += e
    HAVING t.@from_receiver AND t.@from_sender;
```

The HAVING clause checks whether a transaction is considered part of the receiving group and the sending group.

Circle Detection

The essence of money laundering is to transfer money so that it becomes a challenge to backtrace its origin. Criminals have several routing schemas to mask the source of their illicit money. A popular transfer schema is one where the money is transferred via various intermediaries to return eventually to one of the senders. In this case, the money traverses in a circular pattern. With graphs, we can detect such circular patterns easier than with traditional databases because we can hop repeatedly from one transaction to the next until a transaction arrives at the originator. As we explained in Chapter 2, graph hops are computationally much cheaper than table joins in a relational database.

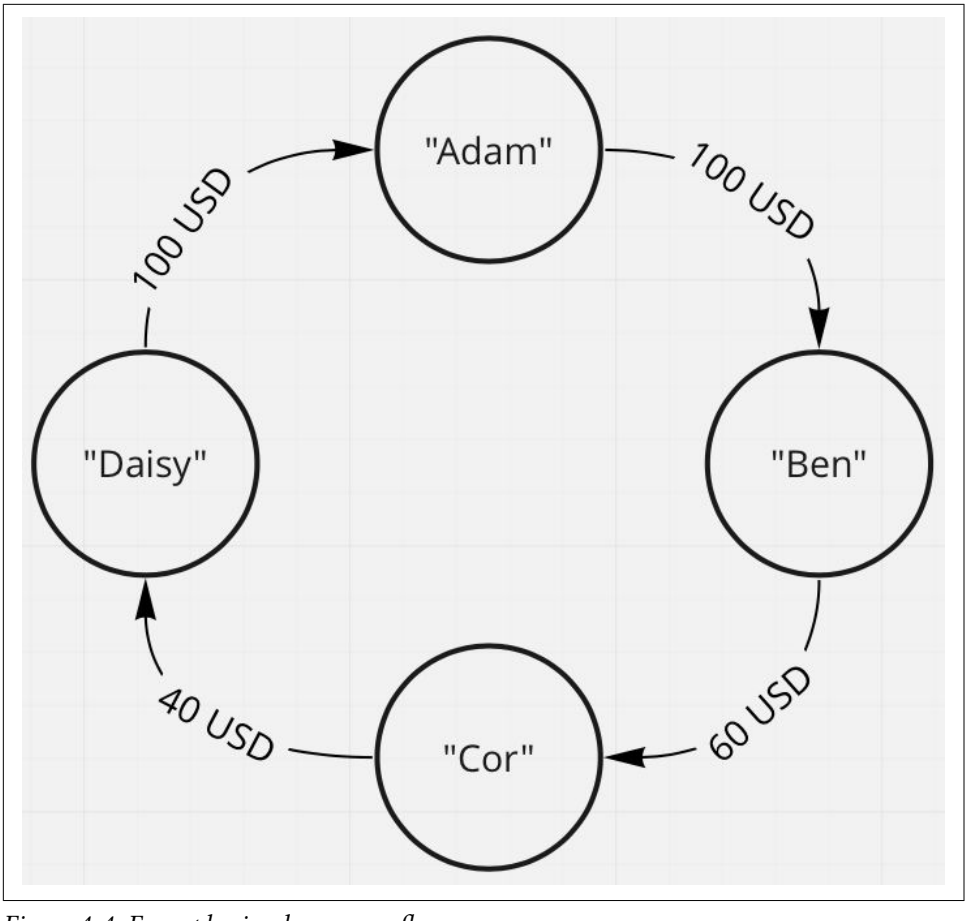


Figure 4-4. Example circular money flow

In Figure 4-4 we see such a circular money flow. In this example, Adam is the originator and sends 100 USD to Ben. Ben sends 60 USD to Cor, and she sends 40 USD to Daisy, who in her turn, sends 100 USD back to Adam. We show in this example that Ben, Cor, and Daisy do not send the same amount of money they have received to the next person in the chain. Criminals do this to add another layer of noise by making the starting amount branch out into different chunks across various intermediaries, making it harder to find out who the originator is and what amount is being laundered.

The query *circle_detection* finds all the circular transaction chains starting from a given User (source_id) which have up to a maximum number of transactions per circle (max_transactions). Since there are two hops per transaction (Sender → Transaction → Recipient), the circles can have up to twice as many hops. To be a valid circle,

the sequence of transactions in a circle must move forward in time. For example, for this to be a valid circle:

```
source_id → txn_A → txn_B → txn_C → source_id
```

Then txn_A.ts < txn_B.ts < txn_C.ts, where ts is the timestamp of a transaction.

Because there are so many possible paths to check, the query's implementation employs a couple of performance and filtering techniques. The first one is bidirectional search. Bidirectional search searches forward from the starting point while simultaneously searching backward from the ending point. It is faster to conduct two half-length searches than one full-length search. When the two searches intersect, you have a complete path.

The second technique is filtering out implausible times. The query starts with a preparatory step; one of its purposes is to set some boundary times.

```
Seed = {source_id};
Seed = SELECT src
    FROM Seed:src - ((User_Transfer_Transaction>|User_Receive_Transaction>):e)
                  - Transaction:tgt
    ACCUM
        CASE WHEN
          e.type == "User_Transfer_Transaction"
        THEN
          @@min_src_send_time += tgt.ts
        ELSE
          @@max_src_receive_time += tgt.ts
        END
    …
    HAVING @@max_src_receive_time >= @@min_src_send_time;
```

Starting from source_id, make one step both forward (User_Transfer_Transaction) and backward (User_Receive_Transaction). Find the earliest time of any transaction sent by source_id (@@min_src_send_time) and the latest time of any transaction received by source_id (@@max_src_receive_time). Check to make sure that @@max_src_receive_time >= @@min_src_send_time. These global limits will also be used later to check the plausibility of other transactions which are candidates for a circular path.

Then we begin Phase 1 of the search. Starting from source_id, step forward two hops (equals one transaction). Using Figure 4-4 as an example, this would step from Adam to Ben. Also traverse two hops backward (Adam to Daisy). Iterate this combination of steps, moving forward (or backward) in time until each direction has stepped halfway around a maximum size circle. Table 5-2 shows the paths that would be traversed if we consider the graph of Figure 4-4.

Table 4-2. Forward and reverse paths, using the graph of Figure 4-4

Iteration	1	2	3
Forward	Adam→Ben	Ben→Cor	Cor→Daisy
Reverse	Adam→Daisy	Daisy→Cor	Cor→Ben

The code snippet below shows a simplified version of one iteration of the forward traversal. For brevity, the checking of timing and step constraints has been omitted.

```
Fwd_set = SELECT tgt
    FROM Fwd_set:src - (User_Transfer_Transaction>:e) - Transaction:tgt
    WHERE tgt.ts >= @@min_src_send_time
        AND src.@min_fwd_dist < GSQL_INT_MAX
        AND tgt.@min_fwd_dist == GSQL_INT_MAX
    ACCUM tgt.@min_fwd_dist += src.@min_fwd_dist + 1
    … // POST-ACCUM clause to check time and step constraints
    ;
Fwd_set = SELECT tgt
    FROM Fwd_set:src - (<User_Receive_Transaction:e) - User:tgt
    WHERE src.@min_fwd_dist < GSQL_INT_MAX
        AND tgt.@min_fwd_dist == GSQL_INT_MAX
    ACCUM tgt.@min_fwd_dist += src.@min_fwd_dist + 1
    … // POST-ACCUM clause to check time and step constraints
    HAVING tgt != source_id;
```

Looking at Table 5-2, we see that after the second iteration, a forward path and a reverse path have met at a common point, Cor. We have a circle! But wait. What if the Ben→Cor timestamp is later than the Cor→Daisy timestamp? If so, then it's not a *valid* circle.

In Phase 2 of the query, we discover and validate circular paths by doing the following : For the forward search, continue traversing forward but only along paths which were previously traversed in the reverse direction and which move forward in time. In our example, if max_transactions = 2 so that Phase 1 got as far as Ben→Cor, then Phase 2 could continue on to Cor→Daisy, but only because we had already traversed Daisy→Cor in Phase 1 and only if the timestamps continue to increase.

```
Fwd_set = SELECT tgt
    FROM Fwd_set:src - (User_Transfer_Transaction>:e) - Transaction:tgt
    // tgt must have been touched in the reverse search above
    WHERE tgt.@min_rev_dist < GSQL_INT_MAX
        AND tgt.ts >= @@min_src_send_time
        AND src.@min_fwd_dist < GSQL_INT_MAX
        AND tgt.@min_fwd_dist == GSQL_INT_MAX
    ACCUM tgt.@min_fwd_dist += src.@min_fwd_dist + 1
    POST-ACCUM
        CASE WHEN
          tgt.@min_fwd_dist < GSQL_INT_MAX
          AND tgt.@min_rev_dist < GSQL_INT_MAX
          AND tgt.@min_fwd_dist + tgt.@min_rev_dist
```

```
            <= 2 * STEP_HIGH_LIMIT
        THEN
          tgt.@is_valid = TRUE
        END;
Fwd_set = SELECT tgt
    FROM Fwd_set:src - (<User_Receive_Transaction:e) - User:tgt
    //tgt must have been touched in the reverse search above
    WHERE tgt.@min_rev_dist < GSQL_INT_MAX
        AND src.@min_fwd_dist < GSQL_INT_MAX
        AND tgt.@min_fwd_dist == GSQL_INT_MAX
    ACCUM tgt.@min_fwd_dist += src.@min_fwd_dist + 1
    POST-ACCUM
        CASE WHEN
          tgt.@min_fwd_dist < GSQL_INT_MAX
          AND tgt.@min_rev_dist < GSQL_INT_MAX
          AND tgt.@min_fwd_dist + tgt.@min_rev_dist
           <= 2 * STEP_HIGH_LIMIT
        THEN
          tgt.@is_valid = TRUE
        END
    HAVING tgt != source_id;
```

After Phase 2, we have found our circles. There is a Phase 3 which traverses the circles and marks the vertices and edges so that they can be displayed. Figures 5-5 and 5-6 show example results from circle detection, for maximum circle sizes of 4, 5, and 6 transactions. As the circle size limit increases, more circles are found.

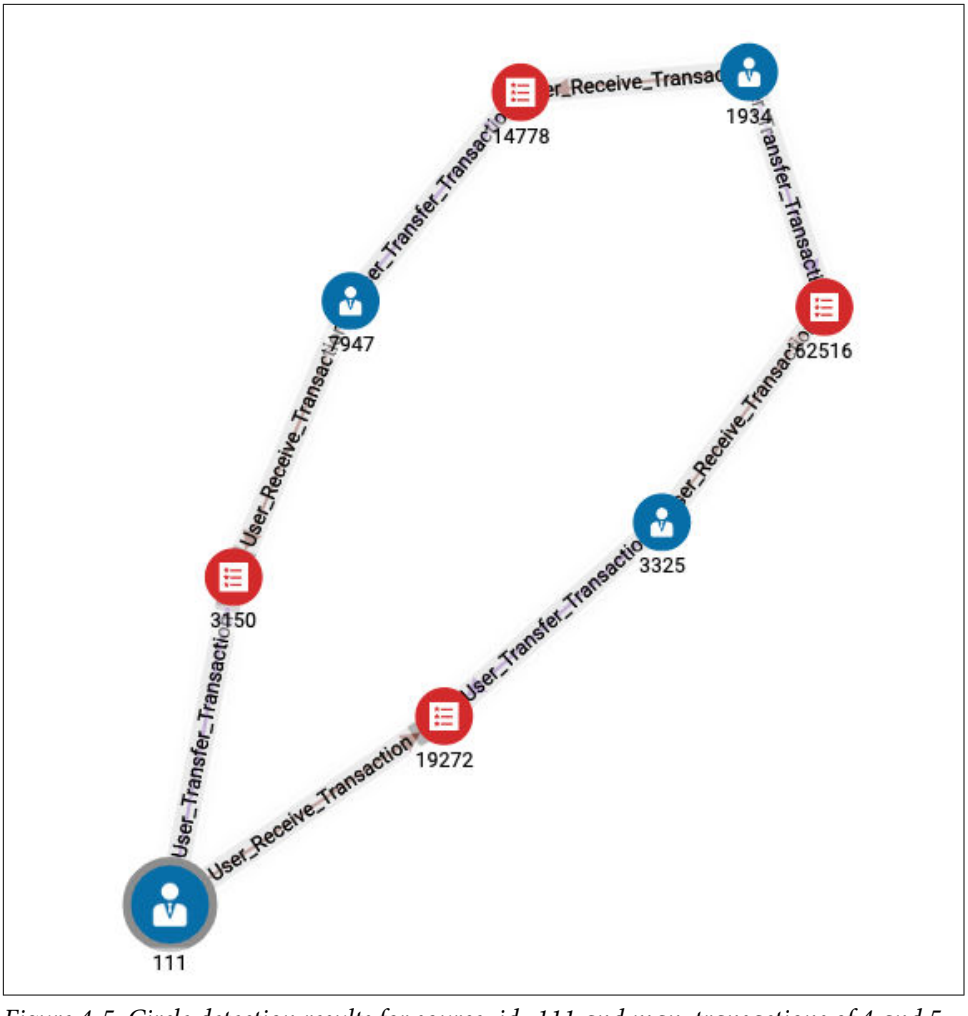


Figure 4-5. Circle detection results for source_id=111 and max_transactions of 4 and 5, respectively.

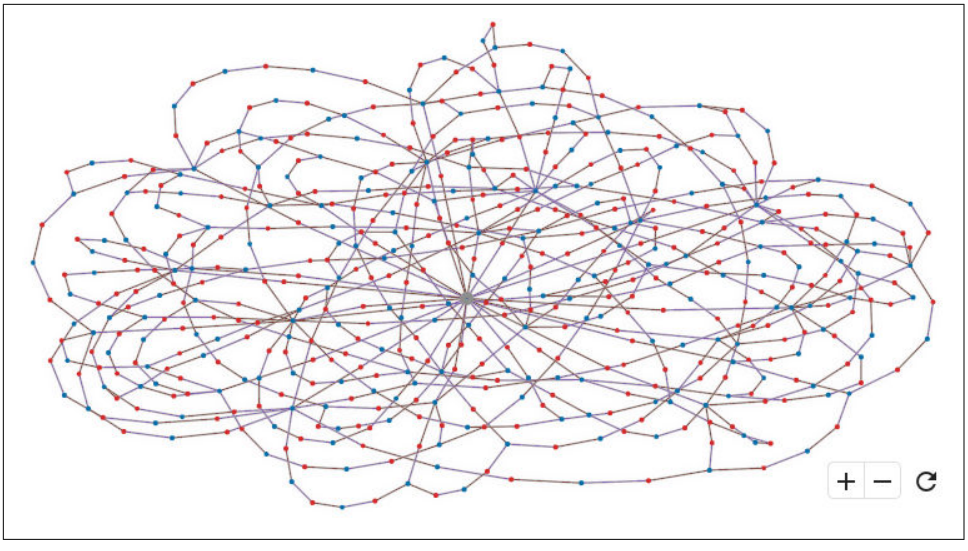


Figure 4-6. Circle detection results for source_id=111 and max_transactions=6

Chapter Summary

Financial fraud is a serious and costly problem that most businesses and all financial institutions must face. Better and faster techniques to detect and stop fraud are needed. We showed that graph data modeling and graph queries are a powerful way to detect suspicious patterns of activity that would have otherwise gone unnoticed. Graph modeling makes it easy to address three key phases for searching for patterns: describing the search, performing the search, and examining the results.

We showed that even with a simple graph schema, queries can be aimed at detecting a variety of different behaviors, such as a high concentration of referral bonuses without a corresponding increase in business, a concentration of transactions from one community to another community, and a high concentration and high dollar value of circular transaction sequences.

In the next chapter, we will offer a systematic approach to analyzing graphs. In particular, we will delve into the rewarding world of graph measures and graph algorithms.

CHAPTER 5

Graph-Powered Machine Learning Methods

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 10th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

After completing this chapter, you should be able to:

- List three basic ways that graph data and analytics can improve machine learning
- Point out which graph algorithms have proved valuable for unsupervised learning
- Extract graph features to enrich your training data for supervised machine learning
- Describe how neural networks have been been extended to learn on graphs
- Provide use cases and examples to illustrate graph-powered machine learning
- Choose which types of graph-powered machine learning are right for you

We now begin the third theme of our book: Learn. That is, we're going to get serious about the core of machine learning: model training. Let's review the machine learning pipeline that we introduced in Chapter 1. For convenience, we've shown it here again,

as Figure 5-1. In the first part of this book, which explored the Connect theme. That theme fits the first two stages of the pipeline: data acquisition and data preparation. Graph databases make it easy to pull data from multiple sources into one connected database and to perform entity resolution.

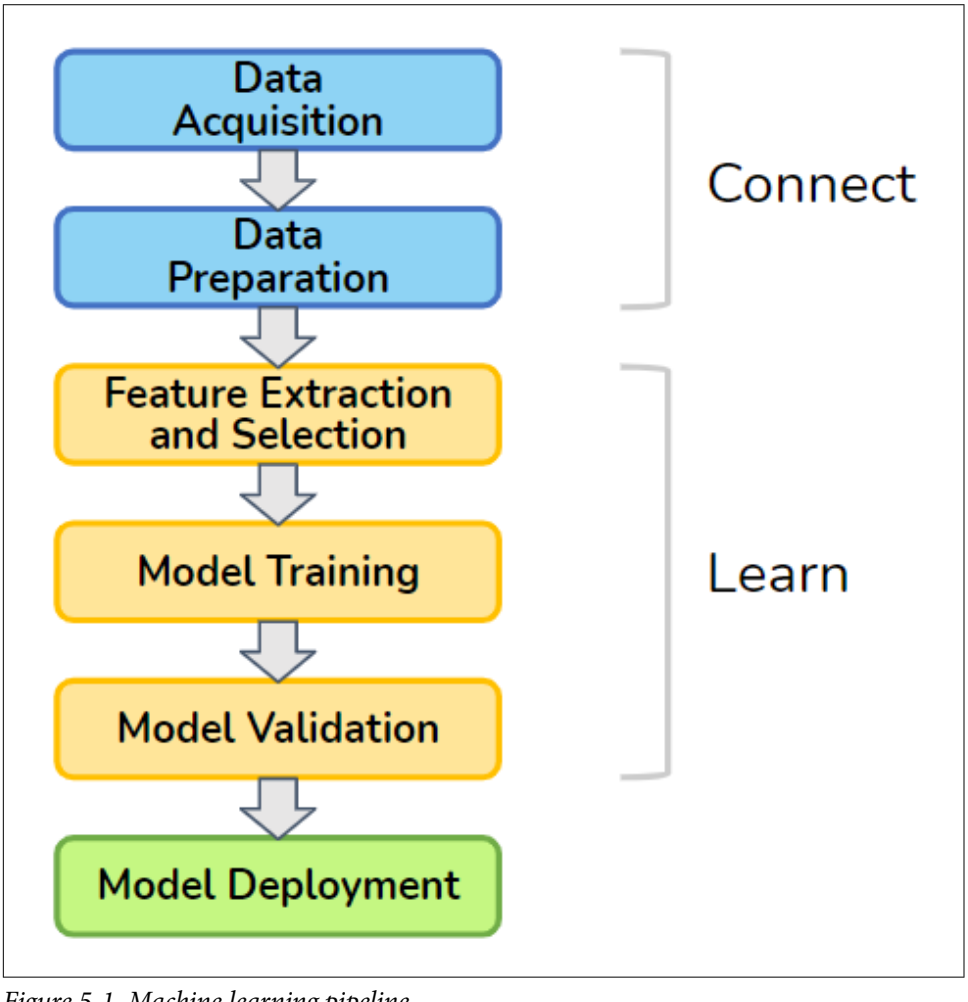


Figure 5-1. Machine learning pipeline

In this chapter, we'll show how graphs enhance the central stages in the pipeline: feature extraction and all-important model training. Features are simply the characteristics or properties of your data entities, like the age of a person or the color of a sweater. Graphs offer a whole new realm of features which are based on how an entity connects to other entities. The addition of these unique graph-oriented features provides machine learning with better raw materials with which to build its models.

This chapter has four sections. The first three sections each describe a different way that graphs enhance machine learning. First, we will start with unsupervised learning, as it is similar to the techniques we discussed in the Analyze section of the book. Second, we will turn to graph feature extraction for supervised and unsupervised machine learning. Third, we culminate with model training directly on graphs, for both supervised and unsupervised approaches. This includes techniques for clustering, embedding, and neural networks. Because they work directly on graphs, some graph databases are now available to offer in-database machine learning. The fourth section reviews the various methods in order to compare them and to help you decide which approaches will meet your needs.

Unsupervised Learning with Graph Algorithms

Unsupervised learning is the forgotten sibling of supervised learning and reinforcement learning, who together form the three major branches of machine learning. If you want your AI system to learn how to do a task, to classify things according to your categories, or to make predictions, you want to use supervised learning and/or reinforcement learning. Unsupervised learning, however, has the great advantage of being self-sufficient and ready-to-go. Unlike supervised learning, you don't need to already know the right answer for some cases. Unlike reinforcement learning, you don't have to be patient and forgiving as you stumble through learning by public trial and error. Unsupervised learning simply takes the data you have and reports what it learned

Think of each unsupervised learning technique as a specialist who is confident in their own ability and specialty. We call in the specialist, *they* tell us something that we hadn't noticed before, and we thank them for their contribution. An unsupervised learning algorithm can look at your network of customers and sales and tell you what are your actual market segments, which may not fit simple ideas of age and income. An unsupervised learning algorithm can point out customer behavior that is an outlier, far from normal, by determining "normal" from your data, not from your preconceptions. For example, an outlier can point out which customers are likely to churn (choose a competitor's product or service).

The first way we will learn from graph data is by applying graph algorithms to discover patterns or characteristics of our data. Earlier in the book, in the Analyze section, we took our first detailed look at graph algorithms. We'll now look at additional families of graph algorithms which go deeper. The line between Analyze and Learn can be fuzzy. What characterizes these new algorithms is that they look for or deal with patterns. They could all be classified as pattern discovery or data mining algorithms for graphs.

Finding Communities

When people are allowed to form their own social bonds, they will naturally divide into groups. No doubt you observed this, in school, in work, in recreation, and in online social networks. We form or join communities. However, graphs can depict any type of relationship, so it can be valuable to know about other types of communities in your data. Use cases include:

- Finding a set of highly interconnected transactions among a set of parties / institutions / locations / computer networks which are out of the ordinary: is this a possible financial crime?
- Finding pieces of software / regulation / infrastructure which are highly connected and interdependent. It is good that they are connected, as a sign of popularity, value, and synergy? Or it is a concerning sign that some components are unable to stand on their own as expected?
- In the sciences (ecology, cell biology, chemistry), finding a set of highly interacting species / proteins is evidence of (inter)dependence and can suggest explanations for such dependence.

Community detection can be used in connection with other algorithms. For example, first use a PageRank or centrality algorithm to find an entity of interest (e.g., a known or suspected criminal). Then find the community to which they belong. You could also swap the order. Find communities first, then count how many parties of note are within that community. Again, whether that is healthy or not depends on the context. Algorithms are your tools, but you are the craftsperson.

What is a Community?

In network science, a *community* is a set of vertices that have a higher density of connections within the group than to outside the group. Specifically, we look at the *edge density* of the community, which is the number of edges between members divided by the number of vertices. A significant change in edge density provides a natural boundary between who is in and who is not. Network scientists have defined several different classes of communities based on how densely they are interconnected. Higher edge density implies a more resilient community.

The following three types of communities span the range from weakest density to strongest density and everything in between.

Connected Component
: You are in if you have a direct connection to one or more members of the community. Requiring only one connection, this is the weakest possible density. In graph theory, if you simply say "component," it refers to a connected community plus the edges that connect them.

K-Core
: You are in if you have a direct connection to k or more members of the community, where *k* is a positive integer.

Clique
: You are in if you have a direct connection to every other member of the community. This is the strongest possible density. Note that clique has a specific meaning in network science and graph theory.

A connected component is a k-core where $k = 1$. A clique containing c vertices is a k-core where $k = (c - 1)$, the largest possible value for k.

Figure 5-2 illustrates examples of these three definitions of community. On the left, there are three separate connected components, represented by the three different shadings. In the center, there are two separate k-cores for k = 2. The lower community, the rectangle with diagonal connections, would also satisfy $k = 3$ because each member vertex connects to three others. On the right, there are two separate cliques.

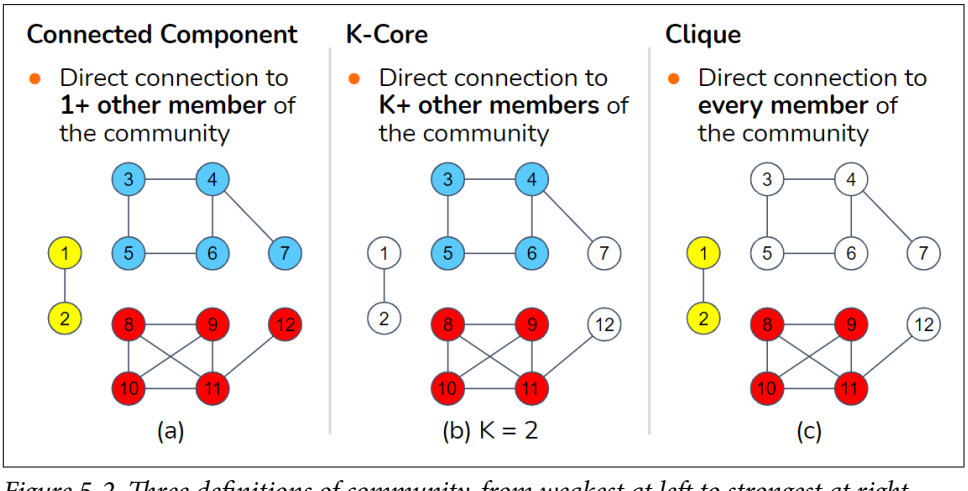


Figure 5-2. Three definitions of community, from weakest at left to strongest at right

Modularity-Based Communities

In many applications, however, we don't have a specific value of k in mind. In the spirit of unsupervised learning, we want the data itself to tell us what is dense enough. To address this, network scientists came up with a measure called *modularity* which looks at relative density, comparing the density within communities versus the density of connections between communities. This is like looking at the density of streets

within cities versus the road density between cities. Now imagine the city boundaries are unknown; you just see the roads. If you propose a set of city boundaries, modularity will rate how good a job you did of maximizing the goal of "dense inside; not dense outside." Modularity is a scoring system, not an algorithm. A modularity-based community algorithm finds the community boundaries which produce the highest modularity score.

The Louvain algorithm, named after the University of Louvain, is perhaps the fastest algorithm for finding community boundaries with a very high modularity. It works hierarchically by forming small local communities, then substituting each community with a single meta-vertex, and repeating by forming local communities of meta-vertices.

To measure modularity (Q), we first partition the vertices into a set of communities so that every vertex belongs to one community. Then, considering each edge as one case, we calculate some totals and averages:

Q =[actual fraction of edges that fall within a community]
minus [expected fraction of edges if edges were distributed at random]

"Expected" is used in the statistical sense: If you flip a coin many times, you expect 50/50 odds of heads vs. tails. Modularity can handle weighted edges by using weights instead of simple counts:

Q = [average weight of edges that fall within a community]
minus [expected weight of edges if edges were distributed at random]

Note that the average is taken over the total number of edges. Edges which run from one community to another add zero to the numerator and add their weight to the denominator. It is designed this way so that edges which run between communities hurt your average and lower your modularity score.

What do we mean by "distributed at random"? Each vertex v has a certain number of edges which connect to it: the *degree* of a vertex is *d(v) = total number (or total weight)* of *v*'s edges. Imagine that each of these d(v) edges picks a random destination vertex. The bigger *d(v)* is, the more likely that one or more of those random edges will make a connection to a particular destination vertex. The expected (i.e., statistical average) number of connections between a vertex *v1* and a vertex *v2* can be computed with Equation 5-1.

Equation 5-1. Expected number of connections between two vertices based on their degrees

$$E_{rand}[wt(v1, v2)] = \frac{d(v1)d(v2)}{2m}$$

where m is the total number of edges in the graph. For the mathematically inclined, the complete formula is shown in Equation 5-2 below, where wt(i,j) is the weight of the edge between i and j, com(i) is the community ID of vertex i, and δ(a,b) is the Kronecker delta function which equals 1 if a and b are equal, 0 otherwise.

Equation 5-2. Definition of modularity

$$Q = \frac{1}{2m} \sum_{i,j \in G} \left[wt(i,j) - \frac{d(i)d(j)}{2m} \right] \delta(comm(i), comm(j))$$

Finding Similar Things

Being in the same community is not necessarily an indicator of similarity. For example, a fraud ring is a community, but it requires different persons to play different roles. Moreover, one community can be composed of different types of entities. For example, a grassroots community centered around a sports team could include persons, businesses selling fan paraphernalia, gathering events, blog posts, etc. If you want to know which entities are similar, you need a different algorithm. Fans who attend the same events and buy the same paraphernalia intuitively seem similar; to be sure, we need a well-defined way of assessing similarity.

Efficient and accurate measurement of similarity is one of the most important tools for businesses because it is what underlies personalized recommendation. We have all seen recommendations from Amazon or other online vendors suggest that we consider a product because "other persons like you bought this." If the vendor's idea of "like you" is too broad, they may make some silly recommendations which annoy you. They want to use several characteristics they know or infer about you and your interests to filter down the possibilities, and then see *what similar customers did*. Similarity is used not only for recommendations but also for following up after detecting a situation of special concern, such as a security vulnerability or a crime of some sort. We found this case. Are there similar cases out there?

Simply finding similar things does not necessarily qualify as machine learning. The first two types of similarity we will look at are important to know, but they aren't really machine learning. These types are graph-based similarity and neighborhood similarity. They lead to the third type, role similarity, which can be considered unsupervised machine learning. Role similarity looks deeply and iteratively in the graph to refine its understanding of the graph's structural patterns.

Similarity values are a prerequisite for at least three other machine learning tasks, clustering, classification, and link prediction. *Clustering* is putting things that are close together into the same group. "Close" does not necessarily mean physically close together; it means…similar. People often confuse communities with clusters. Communities are based on density of connections; clusters are based on similarity of the

entities. *Classification* is deciding which of several predefined categories an item should be placed into. A common meta-approach is to say that an item should be put in the same category as that of similar items which have a known category. If things that look like and walk like it are ducks, then we conclude that it is a duck. *Link prediction* is making a calculated guess that a relationship, which does not currently exist in the dataset, either does exist or will exist in the real world. In many scenarios, entities which have similar neighborhoods (circumstances) are likely to have similar relationships. The relationship could be anything: a person-person relationship, a financial transaction, or a job change. Investigators, actuaries, gamblers, and marketers all seek to do accurate link prediction. As we can see, similarity pops up as a prerequisite for numerous machine learning tasks.

Graph-Based Similarity

What makes two things similar? We usually identify similarities by looking at observable or known properties: color, size, function, etc. A passenger car and a motorcycle are similar because they are both motorized vehicles for one or a few passengers. A motorcycle and a bicycle are similar because they are both two-wheeled vehicles. But, how do we decide whether a motorcycle is more similar to a car or to a bicycle? For that matter, how would we make such decisions for a set of persons, products, medical conditions, or financial transactions? We need to agree upon a system for measuring similarity. But, to take an unsupervised approach, is there some way that we can let the graph itself suggest how to measure similarity?

A graph can give us contextual information to help us decide how to determine similarity. Consider the axiom

> A vertex is characterized by its properties, its relationships, and its neighborhood.

Therefore, if two vertices have similar properties, similar relationships, and similar neighborhoods, then they should be similar. What do we mean by similar neighborhoods? Let's start with a simpler case: the exact same neighborhood.

> Two entities are similar if they connect to the same neighbors.

In Figure 5-3, Person A and Person B have three identical types of edges (purchased, hasAccount, and knows), connecting to the three exact same vertices (Phone model Y, Bank Z, and Person C, respectively). It is not necessary, however, to include all types of relationships or to give them equal weight. If you care about social networks, for example, you can consider only friends. If you care about financial matters, you can consider only financial relationships.

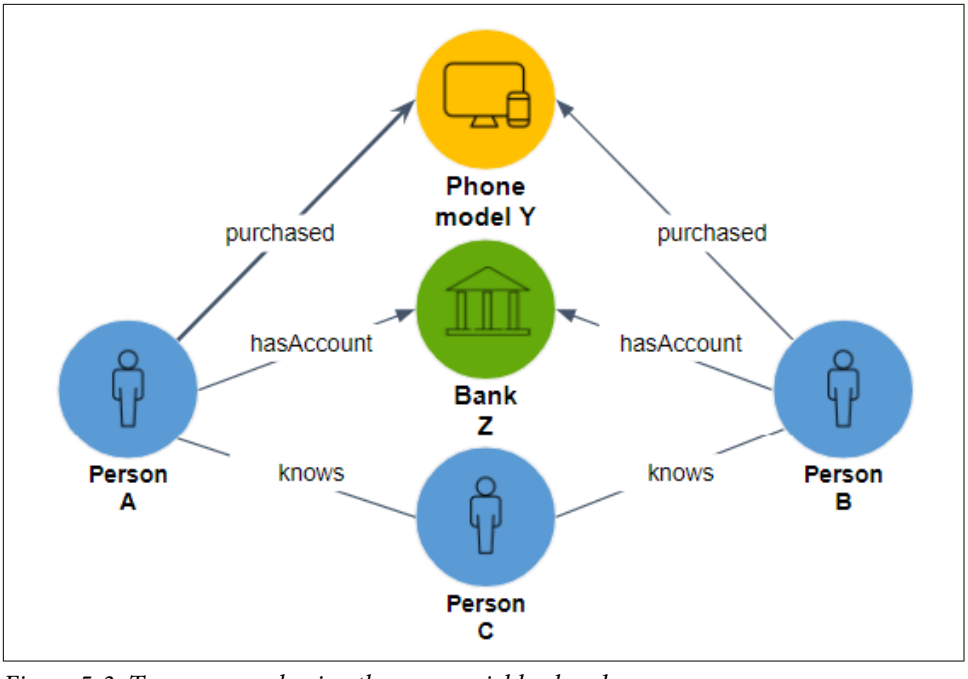


Figure 5-3. Two persons sharing the same neighborhood

To reiterate the difference between community and similarity: All five entities in Figure 5-3 are part of a connected component *community*. However, we would not say that all five are *similar* to one another. The only case of similarity suggested by these relationships is Person A being similar to Person B.

Neighborhood Similarity

It's rare to find two entities which have exactly the same neighborhoods. We'd like a way to measure and rank the degree of neighborhood similarity. The two most common measures for ranking neighborhood similarity are Jaccard similarity and cosine similarity.

Jaccard Similarity. Jaccard similarity measures the relative overlap between two general sets. Suppose you run Bucket List Travel Advisors, and you want to compare your most frequent travelers to one another based on which destinations they have visited. Jaccard similarity would be a good method for you to use; the two sets would be the destinations visited by each of two customers being compared. To formulate Jaccard similarity in general terms, suppose the two sets are N(a), the neighborhood

of vertex a, and N(b), the neighborhood of vertex b. Equation 5-3 gives a precise definition of neighborhood-based Jaccard similarity:

Equation 5-3. Jaccard neighborhood similarity

$$
\begin{aligned}
jaccard(a,b) &= \frac{number\_of\_shared\_neighbors}{size(N(a)) + size(N(b)) - number\_of\_shared\_neighbors)} \\
&= \frac{number\_of\_shared\_neighbors}{number\_of\_unique\_neighbors} \\
&= \frac{|N(a) \cap N(b)|}{|N(a) \cup N(b)|}
\end{aligned}
$$

The maximum possible score is 1, which occurs if a and b have exactly the same neighbors. The minimum score is 0, if they have no neighbors in common.

Consider the following example. Three travelers, A, B, and C, have traveled to the following places as shown in Table 5-1.[1]

Table 5-1. Dataset for Jaccard similarity example

Bucket List's Top 10	A	B	C
Amazon Rainforest, Brazil	✓	✓	
Grand Canyon, USA	✓	✓	✓
Great Wall, China	✓		✓
Machu Picchu, Peru	✓	✓	
Paris, France	✓		✓
Pyramids, Egypt		✓	
Safari, Kenya		✓	
Taj Mahal, India			✓
Uluru, Australia		✓	
Venice, Italy	✓		✓

Using the table's data, the Jaccard similarities for each pair of travelers can be computed.

- A and B have three destinations in common (Amazon, Grand Canyon, Machu Picchu). Collectively they have been to nine destinations. jaccard(A, B) = 3/9 = 0.33.
- B and C have only one destination in common (Grand Canyon). Collectively they have been to ten destinations. jaccard(B, C) = 1/10 = 0.10.

1 The use of a table for our small example may suggest that a graph structure is not needed. We assume you already decided to organize your data as a graph and now are analyzing and learning from that data.

- A and C have three destinations in common (Grand Canyon, Paris, Venice). Collectively they have been to seven destinations. jaccard(A, C) = 3/7 = 0.43.

Among these three, A and C are the most similar. As the proprietor, you might suggest that C visit some of the places that A has visited, such as the Amazon and Machu Picchu. Or you might try to arrange a group tour, inviting both of them to somewhere they both have not been to, such as Uluru, Australia.

Cosine Similarity. Cosine similarity measures the alignment of two sequences of numerical characteristics. The name comes from the geometric interpretation in which the numerical sequence is the entity's coordinates in space. The data points on the grid (the other type of "graph") in Figure 5-4 illustrate this interpretation.

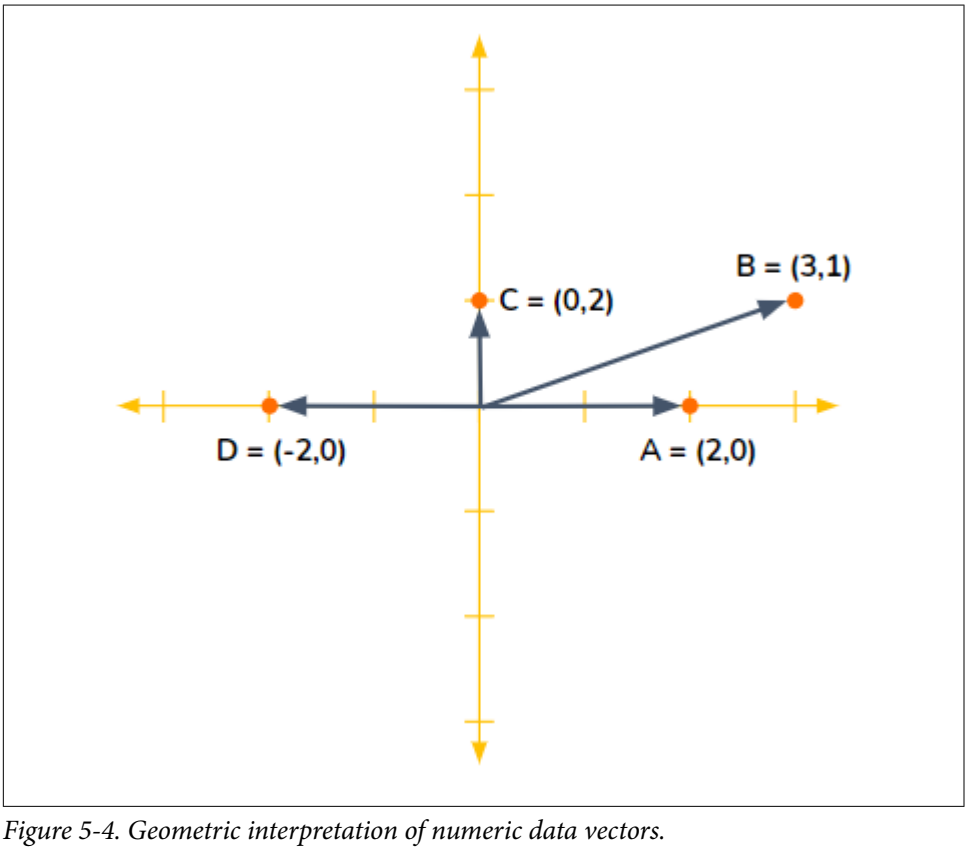


Figure 5-4. Geometric interpretation of numeric data vectors.

Point A represents an entity whose feature vector is (2,0). B's feature vector is (3,1). Now we see why we call a list of property values a "vector." The vectors for A and B are somewhat aligned. The cosine of the angle between them is their similarity score. If two vectors are pointed in exactly the same direction, the angle between them is 0;

the cosine of their angle is 1. cos(A,C) is 0 because A and C are perpendicular; the vectors (2,0) and (0,2) have nothing in common. cos(A,D) is -1 because A and D are pointed in opposite directions. So, cos(x,y) = 1 for two perfectly similar entities, 0 for two perfectly unrelated entities, and -1 for two perfectly anticorrelated entities.

Suppose you have scores across several categories or attributes for a set of entities. You want to cluster the entities, so you need an overall similarity score between entities. The scores could be ratings of individual features of products, employees, accounts, etc. Let's continue the example of Bucket List Travel Advisors. This time, each customer has rated their enjoyment of a destination on a scale of 1 to 10, so we have numerical values, not just yes/no.

Table 5-2. Dataset for cosine similarity example

Bucket List's Top 10	A	B	C
Amazon Rainforest, Brazil	8		
Grand Canyon, USA	10	6	8
Great Wall, China	5		8
Machu Picchu, Peru	8	7	
Paris, France	9		4
Pyramids, Egypt		7	
Safari, Kenya		10	
Taj Mahal, India			10
Uluru, Australia		9	
Venice, Italy	7		10

Here are steps for using this table to compute cosine similarity between pairs of travelers.

1. List all the possible neighbors and define a standard order for the list so we can form vectors. We will use the top-down order in Table 10-1, from Amazon to Venice.
2. If the graph has D possible neighbors, this gives each vertex a vector of length D. For Table 10-2, D = 10. Each element in the vector is either the edge weight, if that vertex is a neighbor, or the null score if it isn't a neighbor.
3. Determining the right null score is important so that your similarity scores mean what you want them to mean. If a 0 means someone absolutely hated a destination, it is wrong to assign a 0 if someone has not visited a destination. A better approach is to normalize the scores. You can either normalize by entity (traveler), by neighbor/feature (destination), or by both. The idea is to replace the empty cells with a default score. You could set the default to be the average destination, or you could set it a little lower than that, because not having visited a place is a weak vote against that place. For simplicity, we won't normalize the scores; we'll

just use 6 as the default rating. Then traveler A's vector is *Wt(A)* = [8, 10, 5, 8, 9, 6, 6, 6, 6, 7].

4. Then, apply the formula in Equation 5-4 for cosine similarity:

Equation 5-4. Cosine neighborhood similarity

$$cosine(a,b) = \frac{Wt(a)\dot{W}(b)}{\| Wt(a) \| \| Wt(b) \|} = \frac{\Sigma_{i=1}^{D} Wt(a)_i Wt(b)_i}{\sqrt{\Sigma_{i=1}^{D} Wt(a)_i^2}\sqrt{\Sigma_{i-1}^{D} Wt(b)_i^2}}$$

Wt(a) and *Wt(b)* are the neighbor connection weight vectors for *a* and *b*, respectively. The numerator goes element by element in the vectors, multiplying the weight from a by the weight from *b*, then adding together these products. The more that the weights align, the larger the sum we get. The denominator is a scaling factor, the Euclidean length of vector *Wt(a)* multiplied by the length of vector *Wt(b)*.

Let's look at one more use case, people who rate movies, to compare how Jaccard and cosine similarity work. In Figure 5-5, we have two persons A and B who have each rated three movies. They have both rated two of the same movies, Black Panther and Frozen.

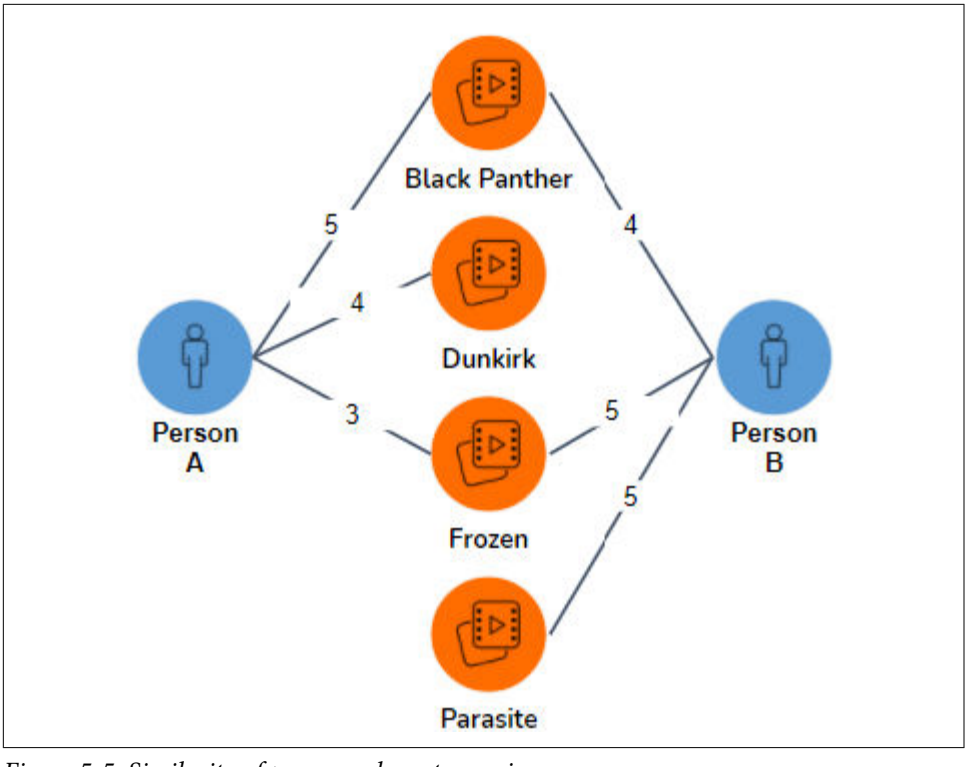


Figure 5-5. Similarity of persons who rate movies

If we only care about what movies the persons have seen and not about the scores, then Jaccard similarity is sufficient and easier to compute. This would also be your choice if scores were not available or if the relationships were not numeric. The Jaccard similarity is (size of overlap) / (size of total set) = 2 / 4 = 0.5. That seems like a middling score, but if there are hundreds or thousands of possible movies they could have seen, then it's a very high score.

If we want to take the movie ratings into account to see how similar are A's and B's taste, we should use cosine similarity. Assuming the null score is 0, then A's neighbor score vector is [5, 4, 3, 0] and B's is [4, 0, 5, 5]. For cosine similarity, the numerator is [5, 4, 3, 0] • [4, 0, 5, 5] = (5)(4)+(4)(0)+(3)(5)+(0)(5) = 35. The denominator = $\sqrt{5^2 + 4^2 + 3^2 + 0^2}\sqrt{4^2 + 0^2 + 5^2 + 5^2} = \sqrt{50}\sqrt{66} = 57.446$. The final result is 0.60927. This again seems like a reasonably good score, not strong similarity, but much better than a random pairing.

Use Jaccard similarity when the features of interest are yes/no or categorical variables.

Use cosine similarity when you have numerical variables. If you have both types, you can use cosine similarity and treat your yes/no variables as having values 1/0

Role Similarity

Earlier we said that if two vertices have similar properties, similar relationships, and similar neighborhoods, then they should be similar. We first examined the situation in which the neighborhoods contained some of the exact same members, but let's look at the more general scenario in which the individual neighbors aren't the *same*, just similar.

Consider a graph of family relationships. One person, Jordan, has two living parents, is married to someone who was born in a different country, and they have three children together. Another person, Kim, also has two living parents, is married to someone born in another country, and together they have four children. None of the neighboring entities (parents, spouses, children) are the same. The number of children is similar but not exactly the same. Nevertheless, Jordan and Kim are similar because they have similar relationships. This is called role similarity.

Moreover, if Jordan's spouse and Kim's spouse are similar, that's even better. If Jordan's children and Kim's children are similar in some way (ages, hobbies, etc.), that's even better. You get the idea. Instead of people, these could be products, components in a supply chain or power distribution network, or financial transactions.

> Two entities have similar roles if they have similar relationships to entities which themselves have similar roles.

This is a recursive definition: A and B are similar if their neighbors are similar. Where does it stop? What is the base case where we can say for certain how much two things are similar?

SimRank. In their 2002 paper, Glen Jeh and Jennifer Widom proposed SimRank which measures similarity by having equal length paths from A and B both reach the same individual; e.g., if Jordan and Kim share a grandparent, that contributes to their SimRank score. The formal definition is shown in Equation 5-5 below.

Equation 5-5. SimRank

$$simrank(a,b) = \frac{C}{|In(a)In(b)|} \sum_{u \in In(a)} \sum_{v \in In(b)} simrank(In(u), In(v))$$

$$simrank(a,a) = 1$$

In(a) and *In*(b) are the sets of in-neighbors of vertices *a* and *b*, respectively, *u* is a member of *In(a)*, *v* is a member of *In(b)*, and *C* is a constant between 0 and 1 to control the rate at which neighbors' influence decreases as distance from a source vertex increases. Lower values of *C* mean a more rapid decrease. SimRank computes a N × N array of similarity scores, one for each possible pair. This differs from cosine and Jaccard similarity which compute a single pair's score on demand. It is necessary to compute the full array of SimRank scores because the value of *SimRank(a,b)* depends of the SimRank score of pairs of their neighbors (e.g., *SimRank(u,v)*), which in turn depends on *their* neighbors, and so on. To compute SimRank, you initialize the array so that S*imRank(a,b)* = 1 if *a* = *b*; otherwise it is 0. Then calculate a revised set of scores by applying Equation 5-5 for each pair (a,b) where the SimRank scores on the right side are from the previous iteration. Note that this is like PageRank's computation except that we have N × N scores instead of just N scores. SimRank's reliance on eventually reaching a shared individual, and through equal length paths, is too rigid for some cases. SimRank does not fully embrace the idea of role similarity. Two individuals in two completely separate graphs, e.g. graphs representing two unrelated families or companies, should be able to show role similarity.

RoleSim. To address this shortcoming, Ruoming Jin et al.[2] introduced RoleSim in 2011. RoleSim starts with the (over)estimate that the similarity between any two entities is the ratio of the sizes of their neighbors. The initial estimate for RoleSim(Jordan, Kim) would be 5/6. RoleSim then uses the current estimated similarity of your neighbors to make an improved guess for the next round. Equation 5-6 has the formal definition of RoleSim.

Equation 5-6. RoleSim

$$rolesim(a,b) = (1-\beta)\max_{M(a,b)} \frac{\sum_{(u,v)\in M(a,b)} rolesim(u,v)}{max(|N(u)|,|N(v)|)} + \beta$$

$$rolesim_0(a,b) = \frac{min(|N(u)|,|N(v)|)}{max(|N(u)|,|N(v)|)}$$

The parameter ? similar to SimRank's C. The main difference is the function M. M(a,b) is a *bipartite matching* between the neighborhoods of *a* and *b*. This is like trying to pair up the three children of Jordan with the four children of Kim. *M(Jordan, Kim)* will consist of three pairs (and one child left out). Moreover, there are 24 possible matchings. For computational purposes (not actual social dynamics), assume that the oldest child of Jordan selects a child of Kim; there are four options. The next child of Jordan picks from the three remaining children of Kim, and the third child of Jor-

2 One of the authors of this book, Victor Lee, is a coauthor of RoleSim.

dan can choose from the two remaining children of Kim. This yields (4)(3)(2) = 24 possibilities. The max term in the equation means that we select the matching that yields the highest sum of RoleSim scores for the three chosen pairs. If you think of RoleSim as a compatibility score, then we are looking for the combination of pairings that gives us the highest total compatibility of partners. You can see that this is more computational work than SimRank, but the resulting scores have nicer properties:

1. There is no requirement that the neighborhoods of a and b every meet.
2. If the neighborhoods of a and b "look" exactly the same, because they have the same size, and each of the neighbors can be paired up so that their neighborhoods look the same, and so on, then their RoleSim score will be a perfect 1. Mathematically, this level of similarity is called *automorphic equivalence*.

This idea of looking at neighbors of neighbors to get a deeper sense of similarity will come up again later in the chapter.

Finding Frequent Patterns

As we've said early on in this book, graphs are wonderful because they make it easy to discover and analyze multi-connection patterns based on connections. In the Analyze section of the book, we talked about finding particular patterns, and we will return to that topic later in this chapter. In the context of unsupervised learning, the goal is to:

> Find any and all patterns which occur frequently.

Computer scientists call this the Frequent Subgraph Mining task, because a pattern of connections is just a subgraph. This task is particularly useful for understanding natural behavior and structure, such as consumer behavior, social structures, biological structures, and even software code structure. However, it also presents a much more difficult problem. "Any and all" patterns in a large graph means a vast number of possible occurrences to check. The saving grace is the threshold parameter T. To be considered frequent, a pattern must occur at least T times. If T is big enough, that could rule out a lot of options early on. The example below will show how T is used to rule out small patterns so that they don't need to be considered as building blocks for more complex patterns.

There any many advanced approaches to try to speed up frequent subgraph mining, but the basic approach is to start with 1-edge patterns, keep the patterns that occur at least T times, and then try to connect those patterns to make bigger patterns:

1. Group all the edges according to their type and the types of their endpoint vertices. For example, Shopper-(bought)-Product is a pattern.
2. Count how many times each pattern occurs.

3. Keep all the frequent patterns (having at least T members) and discard the rest. For example, we keep Shopper-(lives_in)-Florida but eliminate Shopper-(lives_in)-Guam because it is infrequent.
4. Consider every pair of groups that have compatible vertex types (e.g., groups 1 and 2 both have a Shopper vertex), and see how many individual vertices in group 1 are also in group 2. Merge these individual small patterns to make a new group for the larger pattern. For example, we merge the cases where the same person in the frequent pattern Shopper-(bought)-Blender was also in the frequent pattern Shopper-(lives_in)-Florida.
5. Repeat steps 2 and 3 (filtering for frequency) for these newly formed larger patterns.
6. Repeat step 4 using the expanded collection of patterns.
7. Stop when no new frequent patterns have been built.

There is a complication with counting (step 2). The complication is *isomorphism*, how the same set of vertices and edges can fit a template pattern in more than one way. Consider the pattern A-(friend_of)-B. If Jordan is a friend of Kim, which implies that Kim is a friend of Jordan, is that one instance of the pattern or two? Now suppose the pattern is "find pairs of friends A and B who are both friends with a third person C." This forms a triangle. Let's say Jordan, Kim, and Logan form a friendship triangle. There are six possible ways we could assign Jordan, Kim, and Logan to the variables A, B, and C. You need to decide up front whether these types of symmetrical patterns should be counted separately or merged into one instance, and then make sure your counting method is correct.

Summary

Graph algorithms can perform unsupervised machine learning on graph data. Key takeaways from this section are as follows:

- Community detection algorithms find densely interconnected sets of entities.
- Neighborhood similarity algorithms find entities who have similar relationships to others.
- Clustering is grouping by nearness, and similarity is proxy for physical nearness.
- Link prediction uses clues such as neighborhood similarity to predict whether an edge is likely to exist between two vertices.
- Frequent pattern mining algorithms find patterns of connections which occur frequently.

Extracting Graph Features

In the previous section, we showed how you can use graph algorithms to perform unsupervised machine learning. In most of those examples, we analyzed the graph as a whole to discover some characteristics, such as communities or frequent patterns.

In this section, you'll learn how graphs can provide additional and valuable features to describe and help you understand your data. A *graph feature* is a characteristic that is based on the pattern of connections in the graph. A feature can either be local – attributed to the neighborhood of an individual vertex or edge – or global – pertaining to the whole graph or a subgraph. In most cases, we are interested in vertex features – characteristics of the neighborhood around a vertex. That's because vertices usually represent the real world entities which we want to model with machine learning.

When an entity (an instance of a real-world thing) has several features and we arrange those features in a standard order, we call the ordered list a *feature vector*. Some of the methods we'll look at in this section provide individual features; others produce entire sets of features. You can concatenate a vertex's entity properties (the ones that aren't based on connections) with its graph features obtained from one or more of the methods discussed here to make a longer, richer feature vector. We'll also look at a special self-contained feature vector called an *embedding* that summarizes a vertex's entire neighborhood.

These features can provide insight as is, but one of their most powerful uses is to enrich the training data for supervised machine learning. Feature extraction is one of the key phases in a machine learning pipeline (refer back to Figure 5-1). For graphs, this is particularly important because traditional machine learning techniques are designed for vectors, not for graphs. So, in a machine learning pipeline, feature extraction is also where we transform the graph into a different representation.

In the sections that follow, we'll look at three key topics: domain-independent features, domain-dependent features, and the exciting developments in graph embedding.

Domain-Independent Features

If graph features are new to you, the best way to understand them is to look at simple examples that would work for any graph. Because these features can be used regardless of the type of data we are modeling, we say they are domain-independent. Consider the graph in Figure 5-6. We see a network of friendships, and we count occurrences of some simple domain-independent graph features.

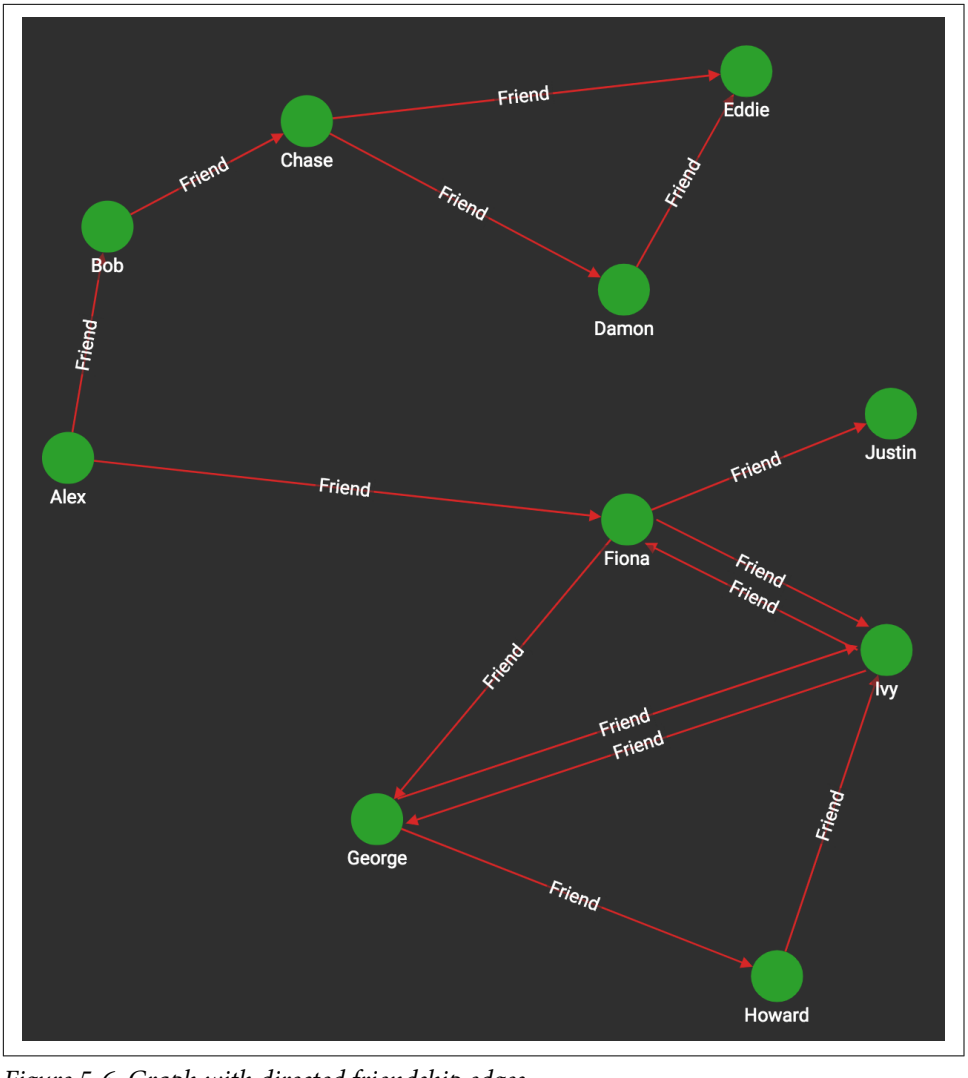


Figure 5-6. Graph with directed friendship edges

Table 5-3 shows the results for four selected vertices (Alex, Chase, Fiona, Justin) and four selected features.

Table 5-3. Examples of domain-independent features from the graph of Figure 5-6

	Number of in-neighbors	Number of out-neighbors	Number of vertices within 2 forward hops	Number of triangles (ignoring direction)
Alex	0	2 (Bob, Fiona)	6 (B, C, F, G, I, J)	0
Chase	1 (Bob)	2 (Damon, Eddie)	2 (D, E)	1 (Chase, Damon, Eddie)

Fiona	2 (Alex, Ivy)	3 (George, Ivy, Justin)	4 (G, I, J, H)	1 (Fiona, George, Ivy)
Justin	1 (Fiona)	0	0	0

You could easily come up with more features, by looking farther than one or two hops[3], by considering generic weight properties of the vertices or edges, and by calculating in more sophisticated ways: computing average, maximum, or other functions. Because these are domain-independent features, we are not thinking about the meaning of "person" or "friend". We could change the object types to "computers" and "sends data to". Domain-independent features, however, may not be the right choice for you if there are many types of edges with very different meanings.

Graphlets

Another option for extracting domain-independent features is to use graphlets[4]. *Graphlets* are small subgraph patterns, which have been systematically defined so that they include every possible configuration up to a given size limit. Figure 5-7 shows all 72 graphlets for subgraphs up to five vertices (or nodes). Note that the figure shows two types of identifiers: shape IDs (G0, G1, G2, etc.) and graphlet IDs (1, 2, 3, etc.). Shape G1 encompasses two different graphlets: graphlet 1, when the reference vertex is on the end of the 3-vertex chain, and graphlet 2, when the reference vertex is in the middle.

Counting the occurrences of every graphlet pattern around a given vertex provides a standardized feature vector which can be compared to any other vertex in any graph. This universal signature lets you cluster and classify entities based on their neighborhood structure, for applications such as predicting the world trade dynamics of a nation[5] or link prediction in dynamic social networks like Facebook.[6]

3 A hop is the generic unit of distance in a graph, from one vertex to its neighbor. We defined this in Chapter 1.

4 Graphlets were first presented by Nataša Pržulj et al. in "Modeling interactome: scale-free or geometric?".

5 "Graphlet-based Characterization of Directed Networks", Sarajlić et al., 2016.

6 "Link Prediction in Dynamic Networks using Graphlet", Rahmat et al., 2016.

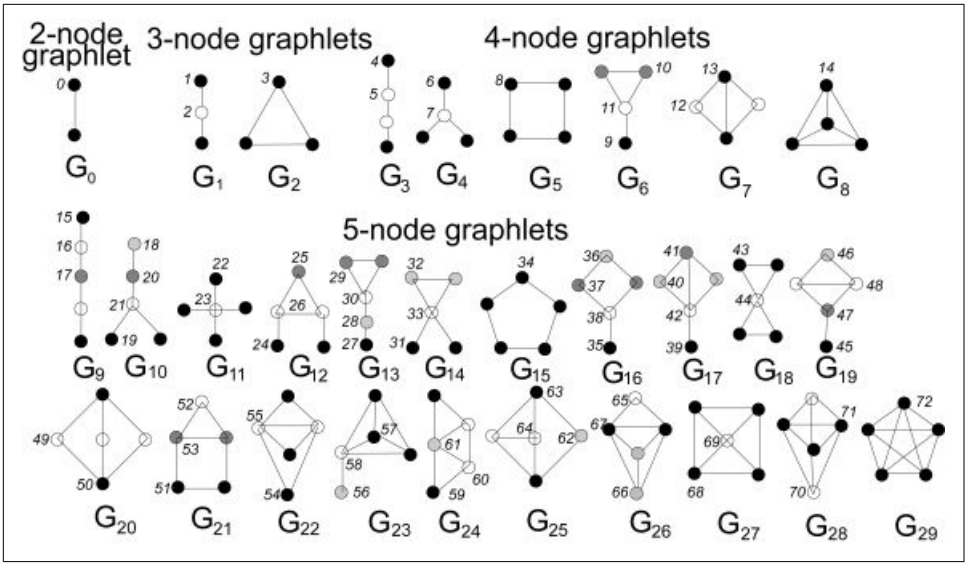


Figure 5-7. Graphlets up to five vertices (or nodes) in size[7]

A key rule for graphlets is that they are the *induced* subgraph for a set of vertices within a graph of interest. Induced means that they include ALL the edges that the base graph has among the selected set of vertices. This rule causes each particular set of vertices to match at most one graphlet pattern. For example, consider the four persons Fiona, George, Howard, and Ivy in Figure 5-6. Which shape and graphlet do they match, if any? It's shape G7, because those four persons form a rectangle with one cross connection. They do not match shape G5, the square because of that cross connection between George and Ivy. While we're talking about that cross connection, look carefully at the two graphlets for shape G7, graphlets 12 and 13. Graphlet 13's source node is located at one end of the cross connection, just as George and Ivy are. This means graphlet 13 is one of their graphlets. Fiona and Howard are at the other corners of the square, which don't have the cross connection. Therefore they have graphlet 12 in their graphlet portfolios.

There is obviously some overlap between the ad hoc features we first talked about (e.g., number of neighbors) and graphlets. Suppose a vertex A has three neighbors B, C, and D, as shown in Figure 5-8. However, we do not know about any other connections. What do we know about its graphlets?

7 From "Uncovering Biological Network Function via Graphlet Degree Signatures" by Tijana Milenković and Nataša Pržulj, licensed under CC BY 3.0

- It exhibits the graphlet 0 pattern three times. Counting the occurrences is important.
- Now consider subgraphs with three vertices.We can define three different subgraphs containing A: (A, B, C), (A, B, D), and (A, C, D). Each of those threesomes satisfies either graphlet 2 or 3. Without knowing about the connections among B, C, and D (the dotted-line edges in the figure), we can be more specific.
- Considering all four vertices, we might be tempted to say they match graphlet 7. Since there might be other connections between B, C, and D, it might actually be a different graphlet. Which one? Graphet 11 if there's one peripheral connection, graphlet 13 if it's two connections, or graphlet 14 if it's all three possible connections.

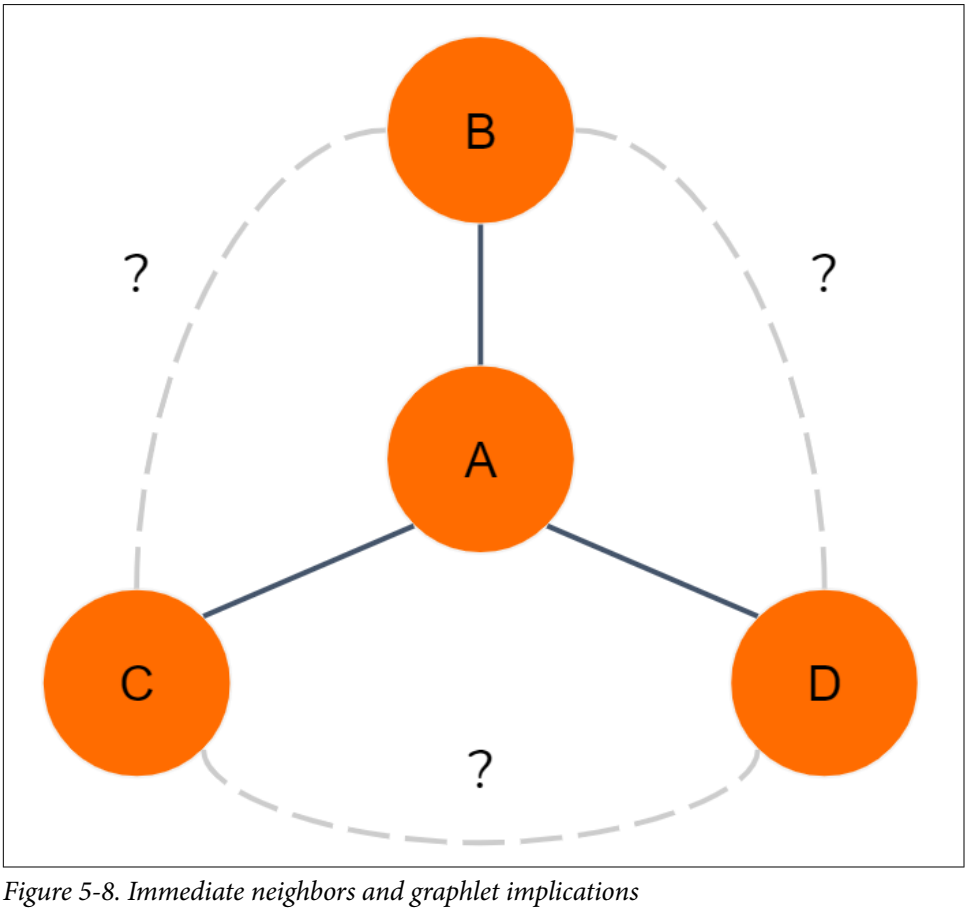


Figure 5-8. Immediate neighbors and graphlet implications

The advantage of graphlets is that they are thorough and methodical. Checking for all graphlets up to 5-node size is equal to considering all the details of the source vertex's

4-hop neighborhood. You could run an automated graphlet counter without spending time and money to design custom feature extraction. The disadvantage of graphlets is that they can require a lot of computational work, and it might be more productive to focus on a more selective set of domain-dependent features. We'll cover these types of features shortly.

Graph Algorithms

Here's a third option for extracting domain-independent graph features: graph algorithms! In particular, the centrality and ranking algorithms which we discussed in the Analyze section of the book work well because they systematically look at everything around a vertex and produce a score for each vertex. Figures 10-9 and 10-10 show the PageRank and closeness centrality[8] scores respectively for the graph presented earlier in Figure 5-7. For example, Alex has a PageRank score of 0.15 while Eddie has a PageRank score of 1. This tells us that Eddie is valued by his peers much more than Alex. Eddie's ranking is due not only to the number of connections, but also to the direction of edges. Howard, who like Eddie has two connections and is at the far end of the rough "C" shape of the graph, has a PageRank score of only 0.49983 because one edge comes in and the other goes out.

8 These algorithms were introduced in the Analyze section of the book.

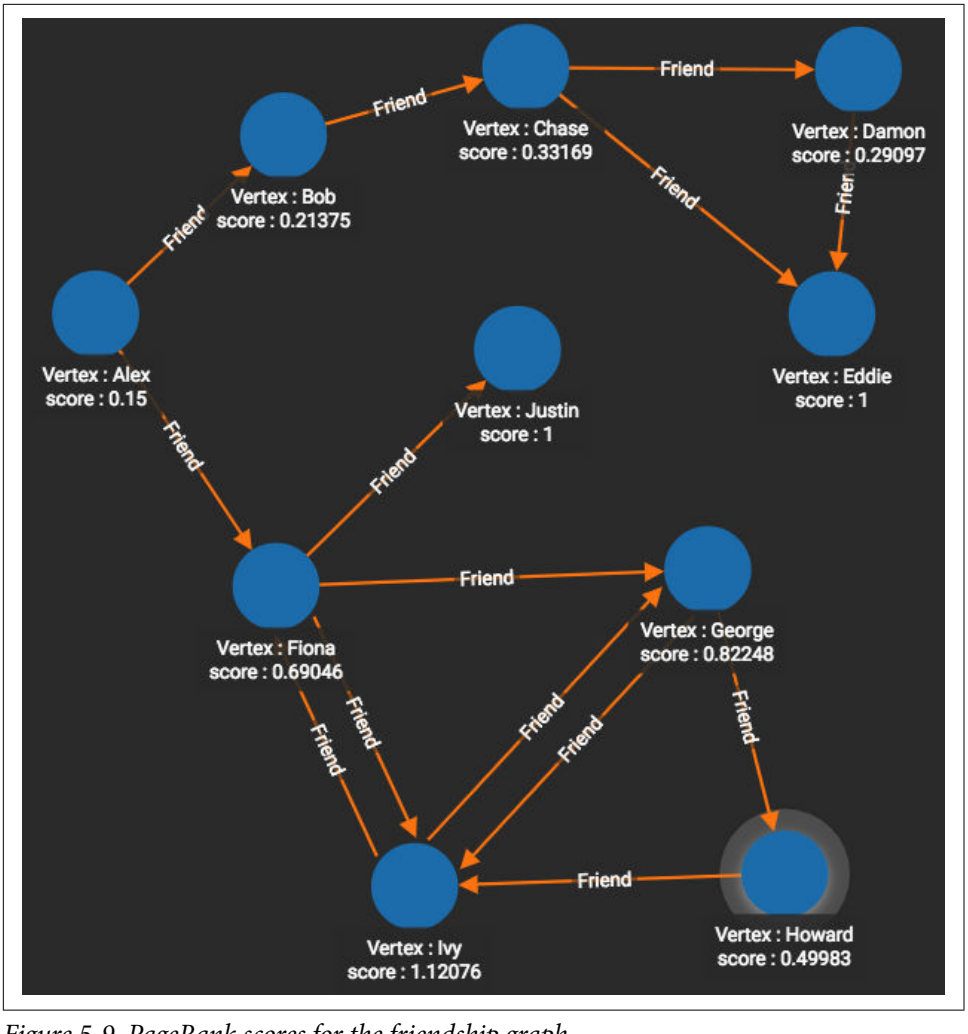


Figure 5-9. PageRank scores for the friendship graph.

The closeness centrality scores in Figure 5-10 tell a completely different story. Alex has a top score of 0.47368 because she is at the middle of the C. Eddie and Howard have scores at or near the bottom, 0.2815 and 0.29032 respectively, because they are at the ends of the C.

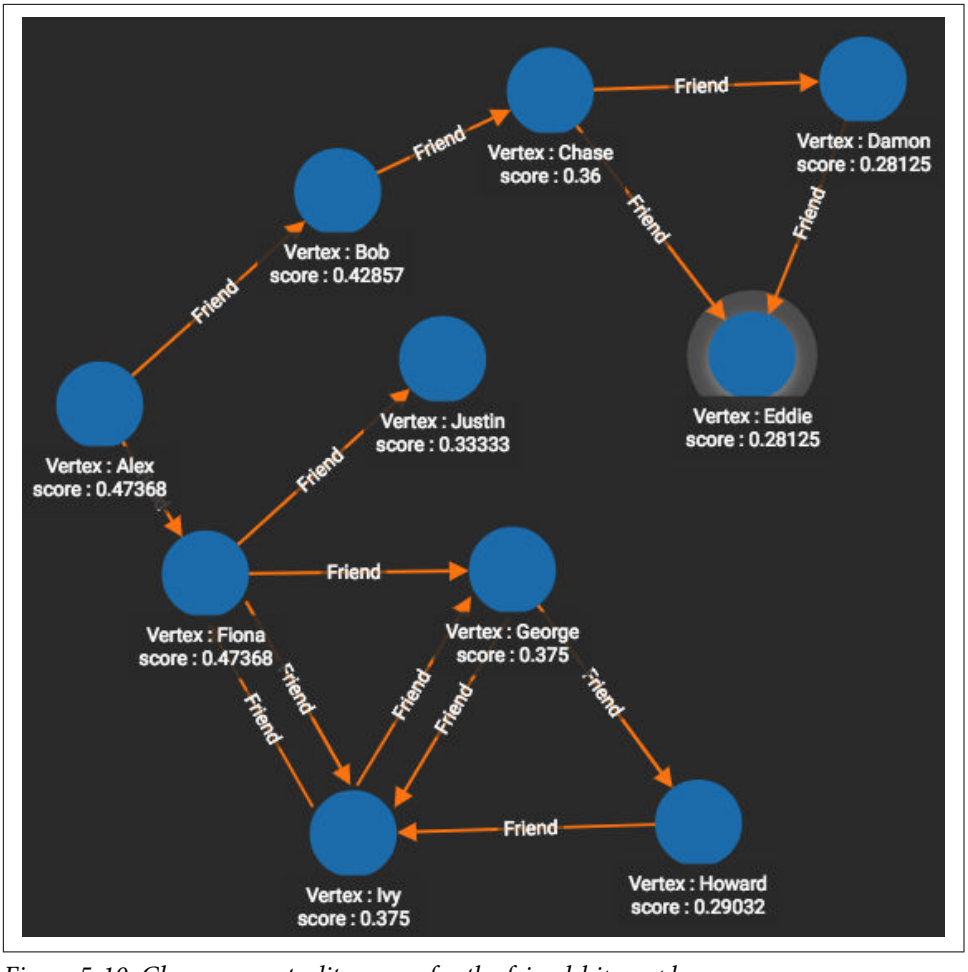


Figure 5-10. Closeness centrality scores for the friendship graph

The main advantage of domain-Independent feature extraction is its universality: generic extraction tools can be designed and optimized in advance, and the tools can be applied immediately on any data. Its unguided approach, however, can make it a blunt instrument.

Domain-independent feature extraction has two main drawbacks. Because it doesn't pay attention to what type of edges and vertices it considers, it can group together occurrences that have the same shape but have radically different meanings. The second drawback is that it can waste resources computing and cataloging features which have no real importance or no logical meaning. Depending on your use case, you may want to focus on a more selective set of domain-dependent features.

Domain-Dependent Features

A little bit of domain knowledge can go a long way towards making your feature extraction smarter and more efficient.

When extracting domain-dependent features, the first thing you want to do is pay attention to the vertex types and edge types in your graph. It's helpful to look at a display of your graph's schema. Some schemas break down information hierarchically into graph paths, such as City-(IN)-State-(IN)-Country or Day-(IN)-Month-(IN)-Year. This is a graph-oriented way of indexing and pre-grouping data according to location or date. This is the case in a graph model for South Korean COVID-19 contact tracing data, shown in Figure 5-11. While City-to-County and Day-to-Year are each 2-hop paths, those paths are simply baseline information and do not hold the significance of a 2-hop path like `Patient-(INFECTED_BY)-Patient-(INFECTED_BY)-Patient`.

You can see how the graphlet approach and other domain-independent approaches can provide confusing results when you have mixed edge types. A simple solution is to take a domain-semi-independent approach by considering only certain vertex types and edge types when looking for features. For example, if looking for graphlet patterns, you might want to ignore the month vertices and their connecting edges. You might still care about the Year of birth of Patients and the (exact) Day on which they traveled, but you don't need the graph to tell you that each Year contains twelve Months.

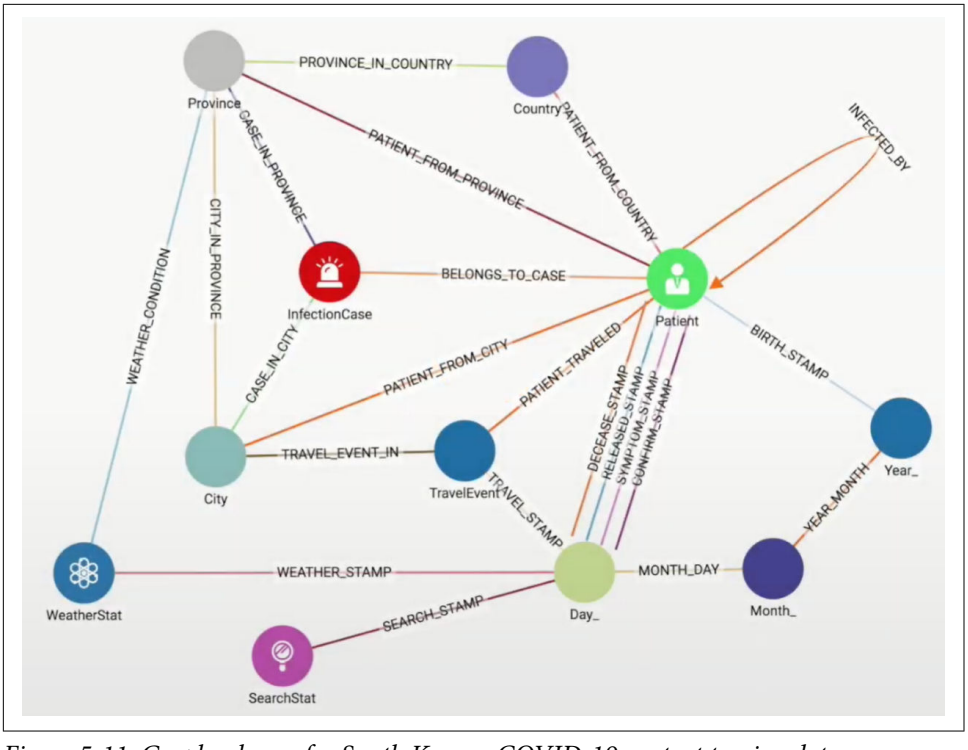


Figure 5-11. Graph schema for South Korean COVID-19 contact tracing data

With this vertex- and edge-type awareness you can refine some of the domain-independent searches. For example, while you can run PageRank on any graph, the scores will only have significance if all the edges have the same or relatively similar meanings. It would not make sense to run PageRank on the entire COVID-19 contact tracing graph. It would make sense, however, to consider only the Patient vertices and INFECTED_BY edges. PageRank would then tell you who was the most influential Patient in terms of causing infection: Patient zero, so to speak.

In this type of scenario, you also want to apply your understanding of the domain to think of small patterns with two or more edges of specific types that indicate something meaningful. For this COVID-19 contact tracing schema, the most important facts are Infection Status (InfectionCase), Who (Patient), Where (City and TravelEvent) and When (Day_). Paths that connect these are important. A possible feature is "number of travel events made by Patient P in March 2020." A more specific feature is "number of Infected Patients in the same city as Patient P in March 2020." That second feature is the type of question we posed in the Analyze section of our book. You'll find examples of vertex and edge type-specific PageRank and domain-dependent pattern queries in the TigerGraph Cloud Starter Kit for COVID-19.

Let's pause for a minute to reflect on your immediate goal for extracting these features. Do you expect these features to directly point out actionable situations, or are you building a collection of patterns to feed into a machine learning system? The machine learning system will then tell you which features matter, to what degree, and in what combination. If you're doing the latter, which is our focus for this chapter, then you don't need to build overly complex features. Instead, focus on building block features. Try to include some that provide a number (e.g., how many travel events) or a choice among several possibilities (e.g., city most visited).

To provide a little more inspiration for using graph-based features, here are some examples of domain-dependent features used in real-world systems to help detect financial fraud:

- How many shortest paths are there between a loan applicant and a blacklisted entity, up to a maximum path length (because very long paths represent negligible risk)?
- How many times has the loan applicant's mailing address, email address, or phone number been used by differently named applicants?
- How many credit card charges has this card made in the last 10 minutes?

While it is easy to see that high values on any of these measures make it more likely that a situation involves financial misbehavior, our goal is to be more precise and more accurate. We want to know what are the threshold values, and what about combinations of factors? Being precise cuts down on false negatives (missing cases of real fraud) and cuts down on false positives (labeling a situation as fraud when it really isn't). False positives are doubly damaging. They hurt the business because they are rejecting an honest business transaction, and they hurt the customer who has been unjustly labeled a crook.

Graph Embeddings: A Whole New World

Our last approach to feature extraction is graph embedding, a hot topic of research and discussion of late. Some authorities may find it unusual that I am classifying graph embedding as a type of feature extraction. Isn't graph embedding a kind of dimensionality reduction? Isn't it representation learning? Isn't it a form of machine learning itself? All of those are true. Let's first define what is a graph embedding.

An *embedding* is a representation of a topological object in a particular system such that the properties we care about are preserved (or approximated well). The last part "the properties we care about are preserved" speaks to why we use embeddings. A well-chosen embedding makes it more convenient to see what we want to see.

Here are several examples to help illustrate the meaning of embeddings:

- The earth is a sphere but we print world maps on flat paper. The representation of the earth on paper is an embedding. There are several different standard representations or embeddings of the earth as a map. Figure 5-12 shows some examples.
- Prior to the late 2010s, when someone said graph embedding, they probably meant something like the earth example. To represent all the connections in a graph without edges touching one another, you often need three or more dimensions. Whenever you see a graph on a flat surface, it's an embedding, as in Figure 5-13. Moreover, unless your data specifies the location of vertices, then even a 3-D representation is an embedding because it's a particular choice about the placement of the vertices. From a theoretical perspective, it takes up to n-1 dimensions to represent a graph with n vertices: huge!
- In natural language processing (NLP), a word embedding is a sequence of scores (i.e., a feature vector) for a given word (see Figure 5-14). There is no natural interpretation of the individual scores, but an ML program sets the scores so that words that tend to occur near each other in training documents have similar embeddings. For example, "machine" and "learning" might have similar embeddings. A word embedding is not convenient for human use, but it is very convenient for computer programs that need a computational way of understanding word similarities and groupings.

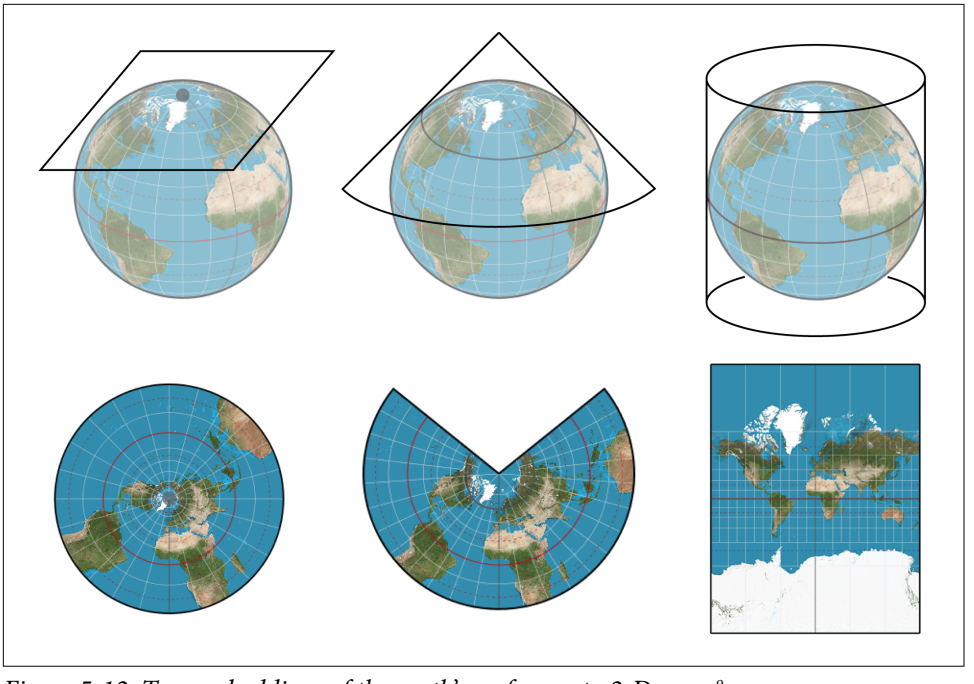


Figure 5-12. Two embeddings of the earth's surface onto 2-D space[9]

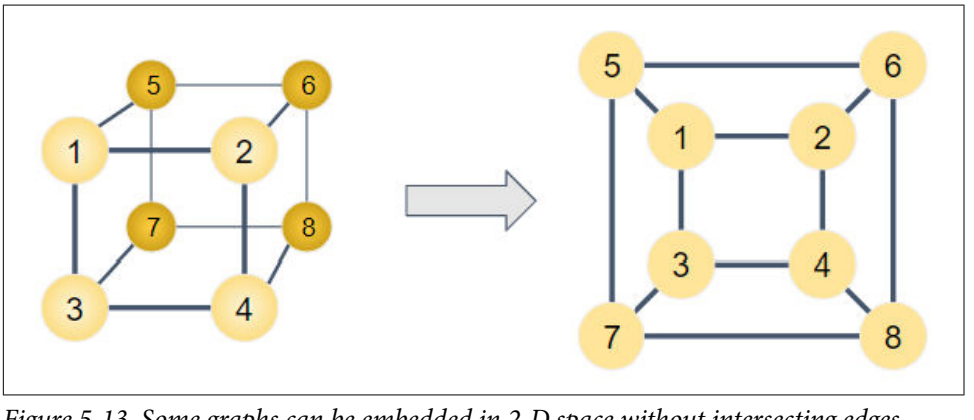


Figure 5-13. Some graphs can be embedded in 2-D space without intersecting edges

9 From Battersby, S. (2017). Map Projections. *The Geographic Information Science & Technology Body of Knowledge* (2nd Quarter 2017 Edition), John P. Wilson (ed.). DOI: 10.22224/gistbok/2017.2.7

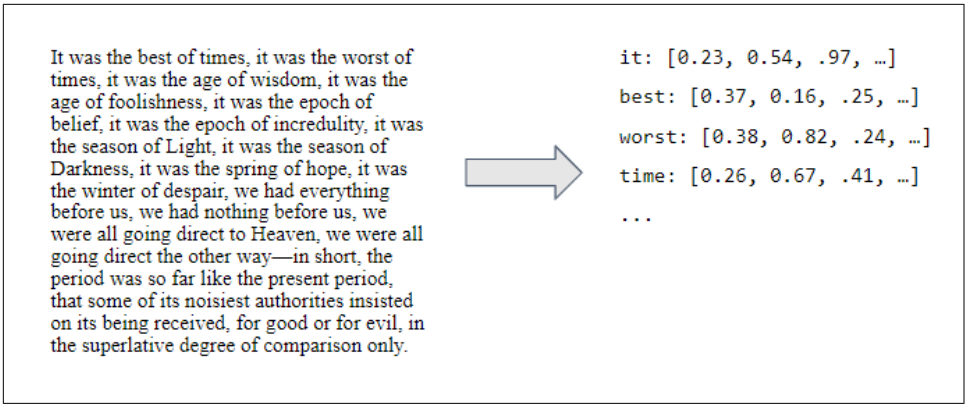
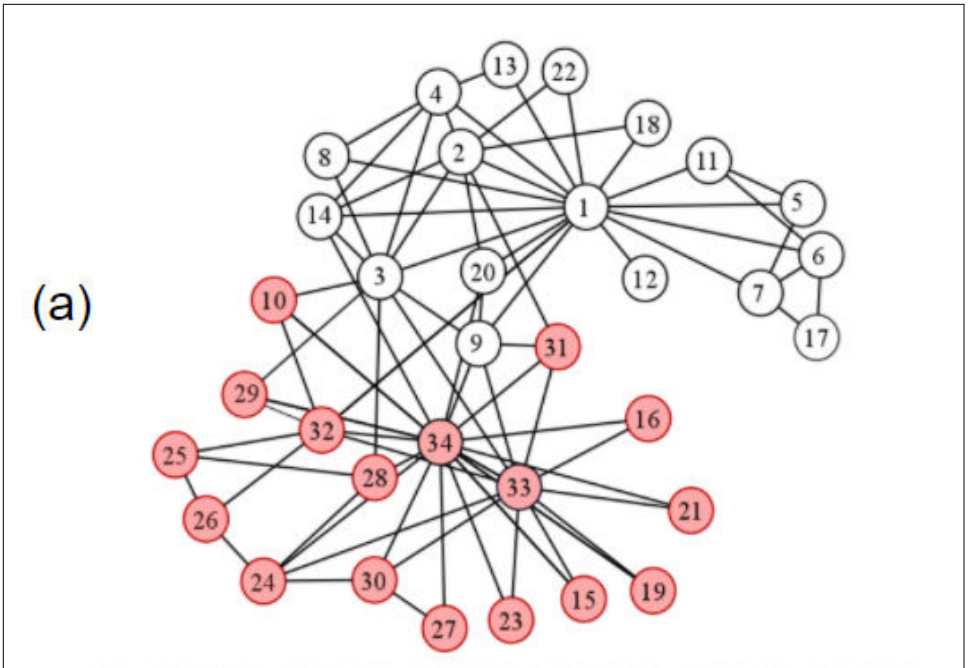


Figure 5-14. Word embedding

In recent years, graph embedding has taken on a new meaning, analogous to word embedding. We compute one or more feature vectors to approximate the graph's neighborhood structure. In fact, when people say graph embedding, they often mean vertex embedding: computing a feature vector for each vertex in the graph. A vertex's embedding tells us something about how it connects to others. We can then use the collection of vertex embeddings to approximate the graph, no longer needing to consider the edges. There are also methods to summarize the whole graph as one embedding. This is useful for comparing one graph to another. In this book, we will focus on vertex embeddings.

Figure 5-15 shows an example of a graph (a) and portion of its vertex embedding (b). The embedding for each vertex (a series of 64 numbers) describes the structure of its neighborhood without directly mentioning any of its neighbors.



(b)

```
34 +0.18332475 +0.31954828 -0.24084991 -0.16343851 -0.14603490 +0.2781425
   +0.25475094 -0.06086616 +0.14059557 -0.2163396  -0.18654257 +0.00942777
   +0.04872169 +0.06530257 +0.04376319 -0.02641196 +0.19182187 +0.0423865
   -0.06893285 +0.27996296 +0.06975124 +0.15666965 +0.08508979 -0.00168657
   -0.25423405 -0.03138730 -0.1018979  +0.15954465 +0.04479356 +0.03317976
   +0.10489599 -0.03110625 +0.05532243 +0.02704849 +0.02639644 +0.02269153
   +0.14549349 -0.24917622 -0.02252658 +0.01116217 -0.02710110 +0.10001109
   +0.31656152 -0.06831823 +0.01232658 -0.24100816 +0.14023747 -0.21998182
   +0.16954944 +0.10862143 +0.14225066 -0.37280235 -0.01532987 -0.37170425
   -0.05201775 +0.20045769 -0.04008826 -0.2185707  -0.01035363 -0.01397756
   +0.01404634 +0.19628309 +0.06172511 +0.38338876
1  +0.2854745  -0.05263712 -0.08923145 -0.06466998 -0.01500806 +0.36512214
   +0.17976986 -0.11145525 +0.22375904 -0.55104405 -0.12485117 +0.30182636
   +0.01254723 -0.46946302 +0.10715616 -0.13087504 -0.05794775 +0.03050438
   -0.12340258 -0.16824125 -0.06328319 -0.02420025 -0.05511753 -0.1426265
   +0.07845344 +0.14444374 +0.10426739 +0.15535158 +0.04042969 +0.16488937
   -0.19482751 -0.19246309 -0.03519640 -0.0498297  -0.04588598 -0.22919762
   +0.02512437 -0.2605723  -0.02214461 -0.2646808  -0.1862748  +0.07729626
   +0.12595864 +0.04708025 +0.04960880 +0.1518577  +0.0948344  -0.19433172
   +0.24654388 +0.16227436 +0.21935917 -0.05113884 +0.06953595 -0.20784278
   +0.50398374 -0.01181463 +0.05175329 -0.04353086+0.06476174  -0.05663567
   +0.08702151 +0.05070712 -0.26178887  0.16224241
```

Figure 5-15. (a) Karate club graph[10] and (b) 64-element embedding for two of its vertices

10 Visualized partitioning of Zachary's Karate Club, licensed under CC BY_SA 4.0

Let's return to the question of classifying graph embeddings. As we will see, the technique of graph embedding requires unsupervised learning, so it can also be considered a form of representation learning. What graph embeddings give us is a set of feature vectors. For a graph with a million vertices, a typical embedding vector would be a few hundred elements long, a lot less than the upper limit of one million dimensions. Therefore, graph embeddings also represent a form of dimensionality reduction. And, we're using graph embeddings to get feature vectors, so they're also a form of feature extraction.

Does any feature vector qualify as an embedding? That depends on whether your selected features are telling you what you want to know. Graphlets come closest to the learned embeddings we are going to examine because of the methodical way they deconstruct neighborhood relationships.

Random Walk-based Embeddings

One of the best known approaches for graph embedding is to use random walks to get a statistical sample of the neighborhood surrounding each vertex v. A random walk is a sequence of connected hops in a graph G. The walk starts at some vertex v. It then picks a random neighbor of v and moves there. It repeats this selection of random neighbors until it is told to stop. In an unbiased walk, there is an equal probability of selecting any of the outgoing edges.

Random walks are great because they are easy to do and gather a lot of information efficiently. All feature extraction methods we have looked at before require following careful rules about how to traverse the graph; graphlets are particularly demanding due to their very precise definitions and distinctions from one another. Random walks are carefree. Just go.

For the example graph in Figure 5-16, suppose we start a random walk at vertex A. There is an equal probability of 1 in 3 that we will next go to vertex B, C, or D. If you start the walk at vertex E, there is a 100% chance that the next step will be to vertex B. There are variations of the random walk rules where there's a chance of staying in place, reversing your last step, jumping back to the start, or jumping to a random vertex.

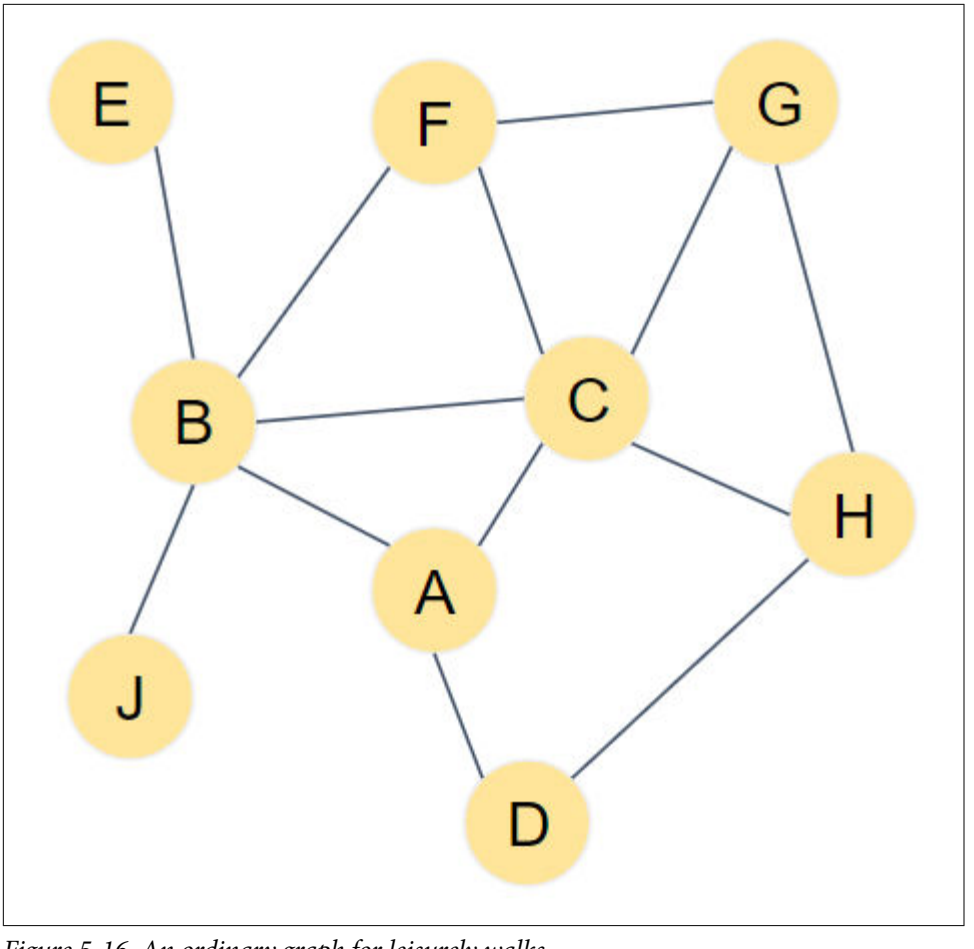


Figure 5-16. An ordinary graph for leisurely walks

Each walk can be recorded as a list of vertices, in the order in which they were visited. A-D-H-G-C is a possible walk. You can think of each walk as a signature. What does the signature tell us? Suppose that walk W1 starts at vertex 5 and then goes to 2. Walk W2 starts at vertex 9 and then goes to 2. Now they are both at 2. From here on, they have exactly the same probabilities for the remainder of their walks. Those individual walks are unlikely to be the same, but if there is a concept of a "typical walk" averaged over a sampling of several walks, then yes, the signatures of 5 and 9 would be similar. All because 5 and 9 share a neighbor 2. Moreover, the "typical walk" of vertex 2 itself would be similar, except offset by one step.

It turns out that these random walks gather neighborhood information much in the same way that SimRank and RoleSim gather theirs. The difference is that those role similarity algorithms considered *all* paths (by considering all neighbors), which is

computationally expensive. SimRank and RoleSim differ from one another in how they aggregate the neighborhood information. Let's take a look at two random-walk based graph embedding algorithms that use a completely different computational method, one borrowed from neural networks.

DeepWalk. The DeepWalk algorithm (Perozzi et al. 2014) collects k random walks of length λ for every vertex in the graph. If you happen to know the word2vec algorithm (Mikolov et al. 2013), the rest is easy. Treat each vertex like a word and each walk like a sentence. Pick a window width w for the *skip-grams* and a length d for your embedding. You will end up with an embedding (latent feature vector) of length d for each vertex. The DeepWalk authors found that k=30 walks, walk length λ=40, with window width w=10 and embedding length d=64 worked well for their test graphs. Your results may vary. Figure 5-17(a) shows an example of a random walk, starting at vertex C and with a length of 16, which is 15 steps or hops from the starting point. The shadings will be explained when we explain skip-grams.

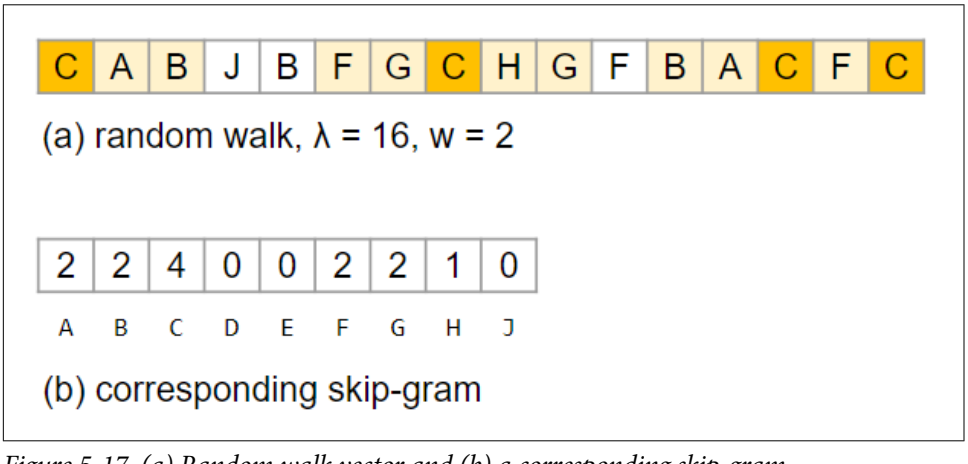


Figure 5-17. (a) Random walk vector and (b) a corresponding skip-gram

We assume you don't know word2vec, so we'll give a high-level explanation, enough so you can appreciate what is happening. This is the conceptual model. The actual algorithm plays a lot of statistical tricks to speed up the work. First, we construct a simple neural network with one hidden layer, as shown in Figure 5-18. The input layer accepts vectors of length n, where n = number of vertices. The hidden layer has length d, the embedding length because it is going to be learning the embedding vectors. The output layer also has length n.

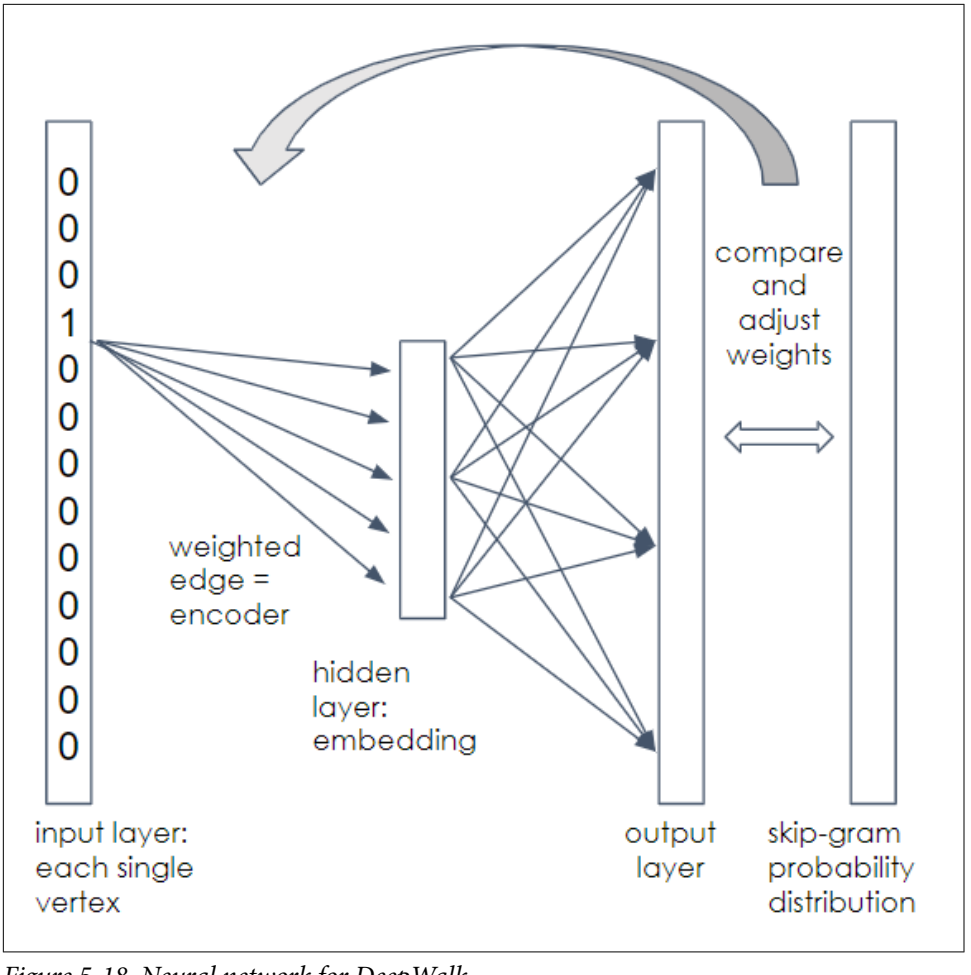


Figure 5-18. Neural network for DeepWalk

Each vertex needs to be assigned a position in the input and output layer, e.g., vertex A is at position 1, vertex B is at position 2, etc. Between the layers are two meshes of n × d connections, from each element in one layer to each element in the next layer. Each edge has a random weight initially, but we will gradually adjust the weights of the first mesh.

Start with one walk for one starting vertex. At the input, we represent the vertex using *one-hot encoding*. The vector element that corresponds to the vertex is set to 1; all other elements are set to 0. We are going to train this neural network to predict the neighborhood of the vertex given at the input.

Applying the weights in the first mesh to our one-hot input, we get a weighted vertex in the hidden layer. This is the current guess for the embedding of the input vertex.

Take the values in the hidden layers and multiply by the weights of the second mesh to get the output layer values. You now have a length-n vector with random weights.

We're going to compare this output vector with a skip-gram representation of the walk. This is where we use the window parameter w. For each vertex in the graph, count how many times it appears within w-steps before or after the input vertex v. We'll skip the normalization process, but your final skip-gram vector expresses the relative likelihood that each of the n vertices is near vertex v in this walk. Now we'll explain the result of Figure 5-17. Vertex C was the starting point for the random walk; we've used dark shading to highlight every time we stepped on vertex C. The light shading shows every step that is within w = 2 steps of vertex C. Then, we form the skip-gram in (b) by counting how many times we set foot on each vertex within the shaded zones. For example, vertex G was stepped on twice, so the skip-gram has 2 in the position for G. This is a long walk on a small graph, so most vertices were stepped on within the windows. For short walks on big graphs, most of the values will be 0.

Our output vector was supposed to be a prediction of this skip-gram. Comparing each position in the two vectors, if the output vector's value is higher than the skip-gram's value, then lower the corresponding weight in the input mesh. If the value was lower, then raise the corresponding weight.

You've processed one walk. Repeat this for one walk of each vertex. Now repeat for a second walk of each vertex, until you've adjusted your weights for k × n walks. You're done! The weights of the first n × d mesh are the length-d embeddings for your n vectors. What, about the second mesh? Strangely, we were never going to directly use the output vectors, so we didn't bother to adjust its weight.

Here is how to interpret and use a vertex embedding:

- For neural networks in general, you can't point to a clear real-world meaning of the individual elements in the latent feature vector.
- Based on how we trained the network, though, we can reason backwards from the skip-grams, which represent the neighbors around a vertex: vertices that have similar neighborhoods should have similar embeddings.
- If you remember the example earlier about two paths that were offset by one step, note that those two paths would have very similar skip-grams. So, vertices that are close to each other should have similar embeddings.

One critique of DeepWalk is that its uniformly random walk is too random. In particular, it may wander far from the source vertex before getting an adequate sample of the neighborhoods closer to the source. One way to address that is to include a probability of resetting the walk by magically teleporting back to the source vertex and then continuing with random steps again, as in Zhou et al. 2021. This is known as "random walk with restart".

Node2vec. Another method which has already gained popularity is node2vec [Grover and Leskovec 2016]. It uses the same skip-gram training process as DeepWalk, but it gives the user two adjustment parameters to control the direction of the walk: go farther (depth), go sideways (breadth), or go back a step. Farther and back seem obvious, but what exactly does sideways mean?

Suppose we start at vertex A of the graph in Figure 5-19. Its neighbors are vertices B, C, and D. Since we are just starting, any of the choices would be moving forward. Let's go to vertex C. For the second step, we can choose from any of vertex C's neighbors: A, B, F, G, or H.

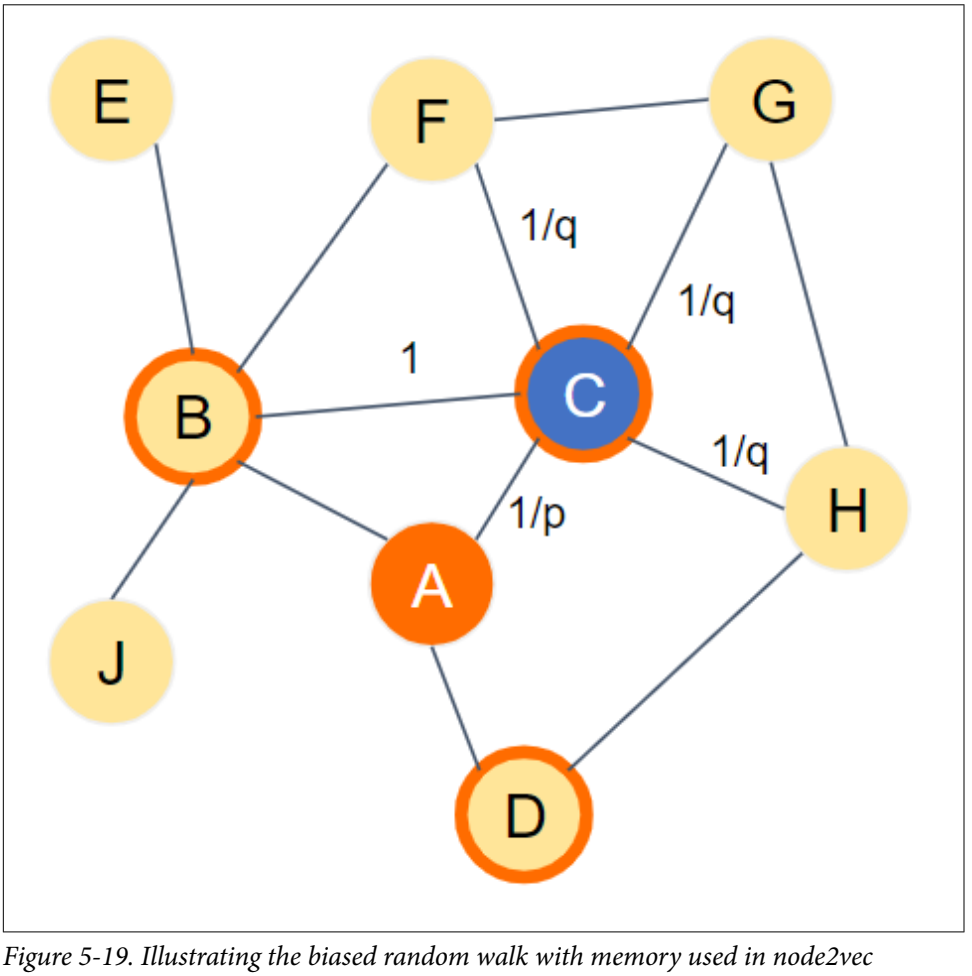


Figure 5-19. Illustrating the biased random walk with memory used in node2vec

If we remember what our choices were in our previous step, we can classify our current set of neighbors into three groups: back, sideways, and forward. We also assign a

weight to each connecting edge, which represents the unnormalized probability of selecting that edge.

Back
: In our example: vertex A. Edge weight = 1/p.

Sideways
: These are the vertices that were available last step and are available this step (e.g., the union of the two sets of vertices. So, they represent a second chance to visit a different neighbor of where you were previously. In our example: vertex B. Edge weight = 1.

Forward
: There are all the vertices that don't return or go sideways. In our example: vertices F, G, and H. Edge weight = 1/q.

If we set p = q = 1, then all choices have equal probability, so we're back to an unbiased random walk. If p < 1, then returning is more likely than going sideways. If q < 1, then each of the forward (depth) options is more likely than sideways (breadth). Returning also keeps the walk closer to home (e.g., similar to breadth-first-search), because if you step back and then step forward randomly, you are trying out the different options in your previous neighborhood.

This ability to tune the walk makes node2vec more flexible than DeepWalk, which results in better models in many cases.

Besides the random walk approach, there are several other techniques for graph embedding, each with their advantages and disadvantages: matrix factorization, edge reconstruction, graph kernel, and generative models. Though already a little dated, *A Comprehensive Survey of Graph Embedding: Problems, Techniques, and Applications*, by Cai et al. provides a thorough overview with several accessible tables which compare the features of different techniques.

Summary

Graphs can provide additional and valuable features to describe and understand your data. Key takeaways from this section are as follows:

- A *graph feature* is a characteristic that is based on the pattern of connections in the graph
- Graphlets and centrality algorithms provide domain-independent features for any graph.
- Applying some domain knowledge to guide your feature extraction leads to more meaningful features.

- Machine learning can produce vertex embeddings which encode vertex similarity and proximity in terms of compact feature vectors.
- Random walks are a simple way to sample a vertex's neighborhood.

Graph Neural Networks

In the popular press, it's not AI unless it uses a neural network, and it's not machine learning unless it uses deep learning. Neural networks were originally designed to emulate how the human brain works, but they evolved to address the capabilities of computers and mathematics. The predominant models assume that your input data is in a matrix or tensor; it's not clear how to present and train the neural network with interconnected vertices. Yet, are there graph-based neural networks? Yes!

Graph neural networks (GNNs) are conventional neural networks with an added twist for graphs. Just as there are several variations of neural networks, there are several variations of GNNs. The simplest way to include the graph itself into the neural network is through convolution.

Graph Convolutional Networks

In mathematics, *convolution* is how two functions affect the result if one acts on the other in a particular way. It is often used to model situations in which one function describes a primary behavior and another function describes a secondary effect. For example, in image processing, convolution takes into account neighboring pixels to improve identification of boundaries and to add artificial blur. In audio processing, convolution is used to both analyze and synthesize room reverberation effects. A convolutional neural network (CNN) is a neural network that includes convolution in the training process. For example, you could use a CNN for facial recognition. The CNN would systematically take into account neighboring pixels, an essential duty when analyzing digital images.

A graph convolutional network (GCN) is a neural network that uses graph traversal as a convolution function during the learning process. While there were some earlier related works, the first model to distill the essence of graph convolution into a simple but powerful neural network was presented in 2017 with "Semi-Supervised Classification with Graph Convolutional Networks" by Thomas Kipf and Max Welling.

For graphs, we want the embedding for each vertex to include information about the relationships to other vertices. We can use the principle of convolution to accomplish this. Figure 5-20 shows a simple convolution function, with a general model at top and a more specific example at bottom.

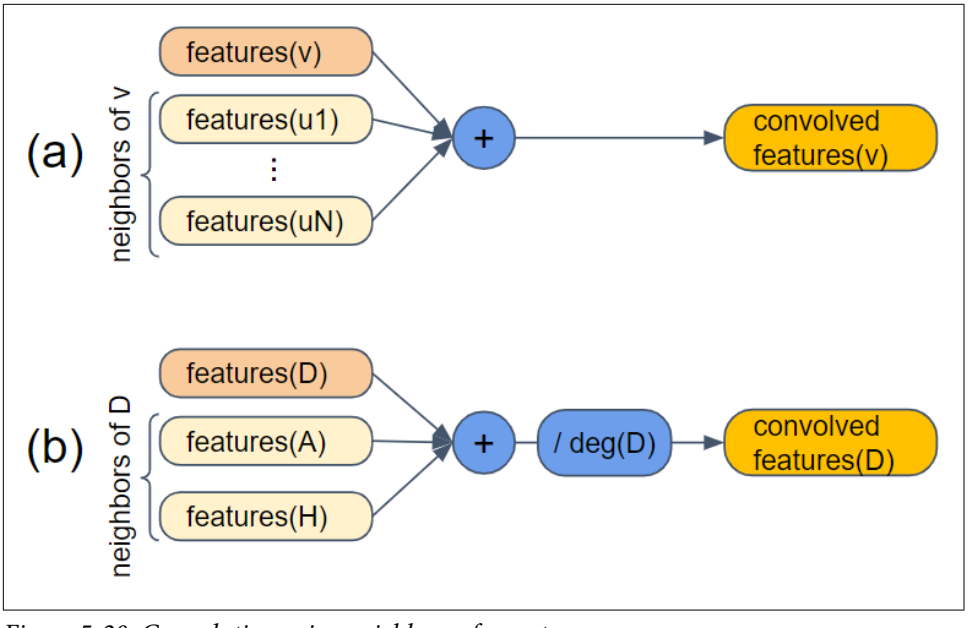


Figure 5-20. Convolution using neighbors of a vertex

In part (a) of the figure, the primary function is features(v): given a vertex v, output its feature vector. The convolution is to combine the features of v with the features of all of the neighbors of v: u1, u2,...,uN. If the features are numeric, we can just add them. The result is the newly convolved features of v. In part (b), we set v = D. Vertex D has two neighbors, A and H. We insert one more step after summing the feature vectors: divide by the degree of the primary vertex. Vertex D has 2 neighbors, so we divide by 2. This regularizes the output values so that they don't keep getting bigger and bigger. (Yes, technically we should divide by deg(v)+1, but the simpler version seems to be good enough.)

Let's do a quick example:

```
features[0](D) = [3, 1 ,4, 1]
features[0](A) = [5, 9, 2, 6]
features[0](H) = [5, 3, 5, 8]
features[1](D) = [6.5, 6.5, 5.5, 7.5]
```

By having neighbors share their feature values, this simple convolution function performs selective information sharing: it determines what is shared (features) and by whom (neighbors). A neural network which uses this convolution function will tend to evolve according to these maxims:

- Vertices which share many neighbors will tend to be similar.

- Vertices which share the same initial feature values will tend to be similar.

These properties are reminiscent of random-walk graph embeddings, aren't they? But we won't be using any random walks.

How do we take this convolution and integrate it into a neural network? Take a look at Figure 5-21.

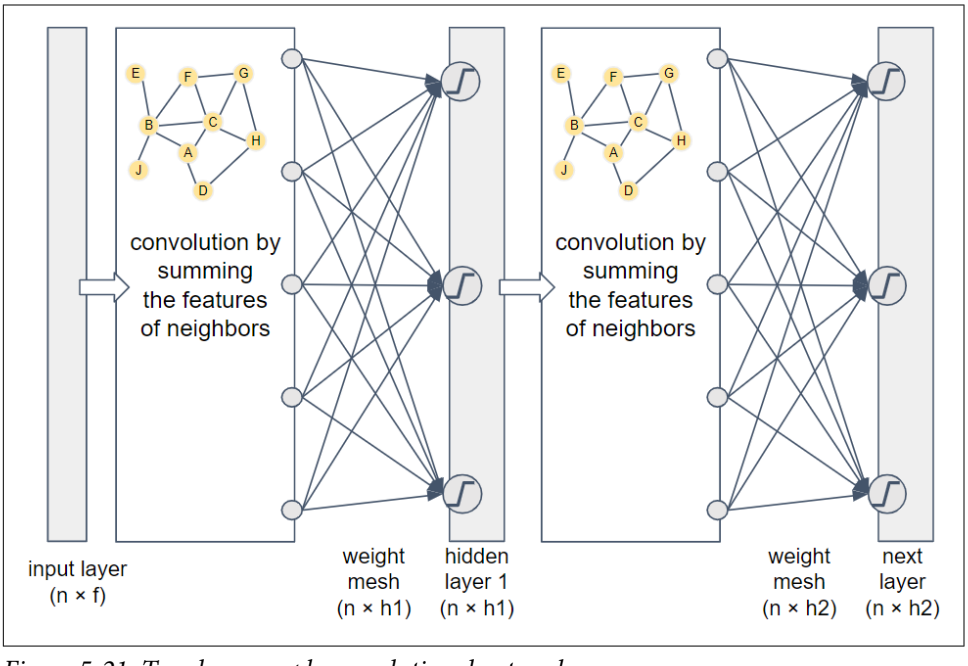


Figure 5-21. Two-layer graph convolutional network

This two-layer network flows from left to right. The input is the feature vectors for all of the graph's vertices. If the feature vectors run horizontally and we stack the vertices vertically, then we get an n × f matrix, where f is the number of features. Next we apply the adjacency-based convolution. We then apply a set of randomized weights (similar to what we did with random-walk graph embedding networks) to merge and reduce the features to an embedding of size h1. Typically h1 < f. Before storing the values in the embedding matrix, we apply an activation function (indicated by the blocky "S" in a circle) which acts as a filter/amplifier. Low values are pushed lower, and high values are pushed higher. Activation functions are used in most neural networks.

Because this is a two-layer network, we repeat the same steps. The only differences are that this embedding may have a different size, where typically h2 ≤ h1, and this weight mesh has a different set of random weights. If this is the final layer, then it's considered the output layer with the output results. By having two layers, the output

embedding for each vertex takes into account the neighbors within two hops. You can add more layers to consider deeper neighbors. Two or three layers often provide the best results. With too many layers, the radius of each vertex's neighborhood becomes so large that it overlaps significantly even with the neighborhoods of unrelated vertices.

Our example here is demonstrating how a GCN can be used in unsupervised learning mode. No training data or target function was provided; we just merged features of vertices with their neighbors. Surprisingly, you can get useful results from an unsupervised, untrained GCN. The authors of GCN experimented with a 3-layer untrained GCN, using the well-known Karate Club dataset.They set the output layer's embedding length of 2, so that they could interpret the two values as coordinate points. When plotted, the output data points showed community clustering which matched the known communities in Zachary's Karate Club.

The GCN architecture is general enough to be used for unsupervised, supervised, semi-supervised, or even reinforcement learning. The only difference between a GCN and a vanilla feed-forward NN is the addition of the step to aggregate the features of a vector with those of its neighbors. Figure 5-22 shows a generic model for how neural networks tune their weights. The graph convolution in GCN affects only the block labeled Forward Propagation Layers. All of the other parts (input values, target values, weight adjustment, etc.) are what determine what type of learning you are doing. That is, the type of learning is decided independently from your use of graph convolution.

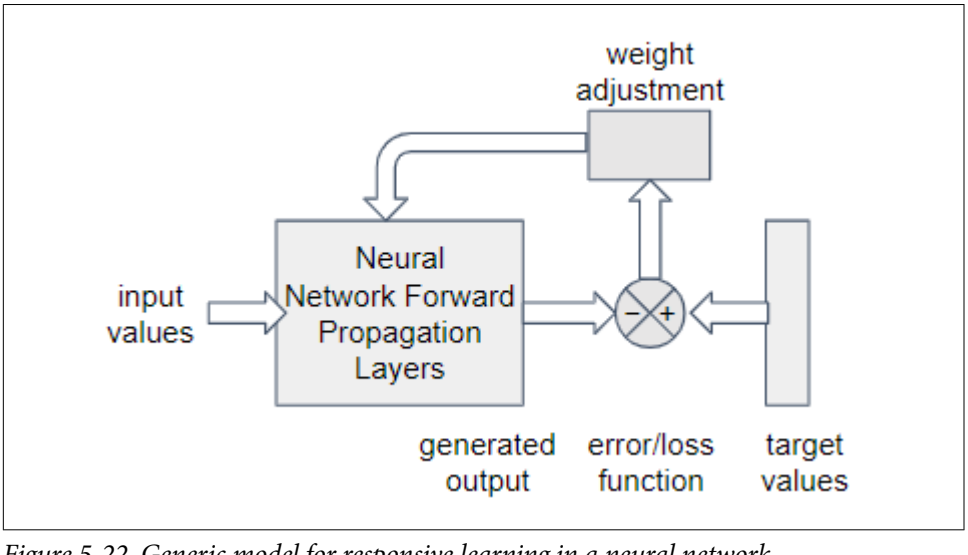


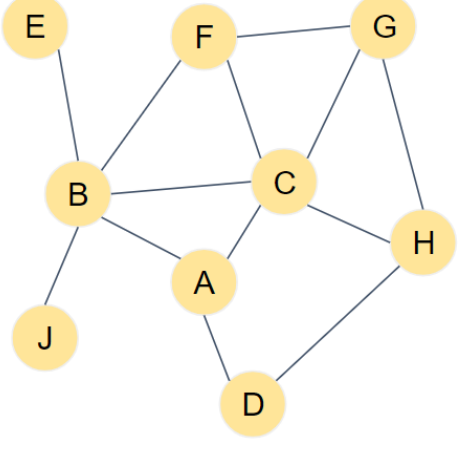
Figure 5-22. Generic model for responsive learning in a neural network

Attention neural networks use a more advanced form of feedback and adjustment. The details are beyond the scope of this book, but graph attention neural networks (GATs) can tune the weight (aka focus the attention) of each neighbor when adding them together for convolution. That is, a GAT performs a weighted sum instead of a simple sum of neighbor's features, and the GAT trains itself to learn the best weights. When applied to the same benchmark tests, GAT outperforms GCN slightly.

Matrix Algebra Formulation

If you like to think in terms of matrices, we can express the convolution process in terms of three simple, standard matrices for any graph: the adjacency matrix A, the identity matrix I, and the degree matrix D. (Feel free to skip this section is this doesn't sound like you.)

Matrix A expresses the connectivity of the graph. Number the vertices 1 to n, and number the rows and columns similarly. The value A(i, j) = 1 if there is an edge from vertex i to vertex j; otherwise A(i,j) is 0. The identity matrix says, "I am myself." I(i, j) = 1 if i = j; otherwise it is 0. It looks like a diagonal line from upper left to lower right. The matrix in Figure 5-23 is (A + I). Note that the 1's in row i tell us the convolution function for vertex i.



A:	**1**	**1**	**1**	**1**	0	0	0	0	0
B:	**1**	**1**	**1**	0	**1**	**1**	0	0	**1**
C:	**1**	**1**	**1**	0	0	**1**	**1**	**1**	0
D:	**1**	0	0	**1**	0	0	0	**1**	0
E:	0	**1**	0	0	**1**	0	0	0	0
F:	0	**1**	**1**	0	0	**1**	**1**	0	0
G:	0	0	**1**	0	0	**1**	**1**	**1**	0
H:	0	0	**1**	**1**	0	0	**1**	**1**	0
J:	0	**1**	0	0	0	0	0	0	**1**

Figure 5-23. A graph and matrix representing the graph's adjacency and identity: (A+I)

What about the regularization? With matrices, instead of performing division, we do multiplication by the inverse matrix. The matrix we want is the degree matrix D, where D(i, j) = degree(i) if i = j; otherwise it is 0. The degree matrix of the graph of Figure 5-16 is shown in the left half of Figure 5-24. At the right is the inverse matrix D-1.

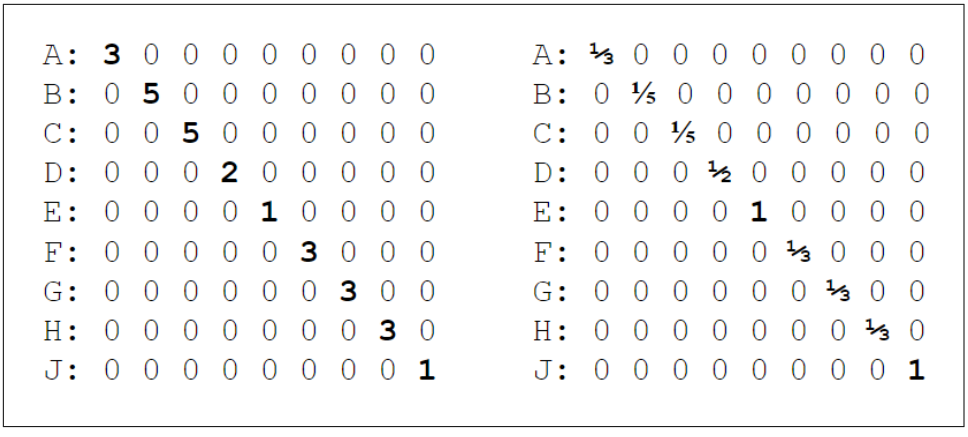


Figure 5-24. The degree matrix D and itself inverse matrix D-1

We can express the computation from layer H0 to the next layer H1 as follows:

$$H_1 = \sigma\left(D^{-1}(A+1)H_0W_1\right)$$

where σ is the activation function, and W is the weight mesh. In Kipf and Welling's paper, they made a tweak to provide a more balanced regularization:

$$H_1 = \sigma\left(D^{-1/2}(A+1)D^{-1/2}H_0W_1\right)$$

GraphSAGE

One limitation of the basic GCN model is that it does a simple averaging of vertex + neighbor features. It seems we would want some more control and tuning of this convolution. Also, large variations in the number of neighbors for different vertices may lead to training difficulties. To address this limitation, Hamilton et al. presented GraphSAGE in 2017 in their paper "Inductive Representation Learning on Large Graphs". Like GCN, this technique also combines information from neighbors, but it does it a little differently. In order to standardize the learning from neighbors, GraphSAGE samples a fixed number of neighbors from each vertex. Figure 5-25 shows a block diagram of GraphSAGE, with sampled neighbors.

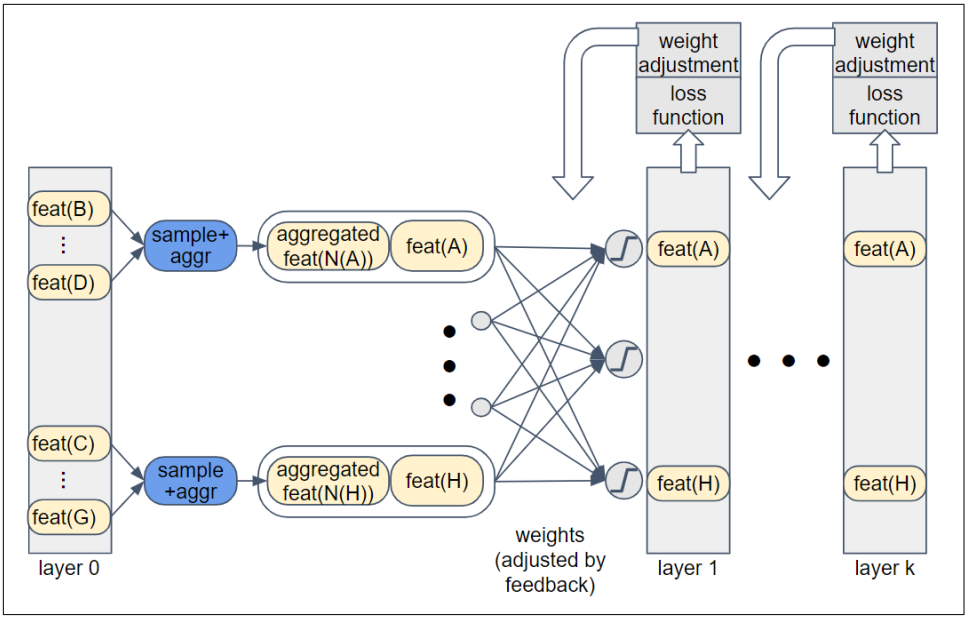


Figure 5-25. Block diagram of GraphSAGE

With GraphSAGE, the features of the neighbors are combined according to a chosen aggregation function. The function could be addition, as in GCNs. Any order-independent aggregation function could be used; LSTM with random ordering and max-pooling work well. The source vertex is not included in the aggregation as it is in GCN; instead, the aggregated feature vectors and the source vertex's feature vector are concatenated to make a double-length vector. We then apply a set of weights to mix the features together, apply an activation function, and store as the next layer's representation of the vertices. This series of sets constitutes one layer in the neural network and the gathering of information within one hop of each vertex. A GraphSAGE network has k layers, each with its own set of weights. GraphSAGE proposes a loss function that rewards nearby vertices if they have similar embeddings, while rewarding distant vertices if they have dissimilar embeddings.

Besides training on the full graph, as you would with GCN, you can train GraphSAGE with only a sample of the vertices. The fact that GraphSAGE's aggregation functions uses equally-sized samples of neighborhoods means that it doesn't matter how you arrange the inputs.That freedom of arrangement is what allows you to train with one sample and then test or deploy with a different sample. Because it can learn from a sample, GraphSAGE performs *inductive learning*. In contrast, GCN directly uses the adjacency matrix, which forces it to use the full graph with the vertices arranged in a particular order. Training with the full data and learning a model for only that data is *transductive learning*.

Whether learning from a sample will work on your particular graph depends on whether your graph's structure and features follow global trends, such that a random subgraph looks similar to another subgraph of similar size. For example, one part of a forest may look a lot like another part of a forest. For a graph example, suppose you have a Customer 360 graph including all of a customer's interactions with your sales team, website, events, their purchases, and all other profile information you have been able to obtain. Last year's customers are rated based on the total amount and frequency of their purchases. It is reasonable to expect that if you used GraphSAGE with last year's graph to predict the customer rating, it should do a decent job of predicting the ratings of this year's customers. Table 5-4 summarizes all the similarities and differences between GCN and GraphSAGE that we have presented.

Table 5-4. Comparison of GCN and GraphSAGE traits

	GCN	GraphSAGE
Neighbors for aggregation	All	sample of n neighbors
Aggregation function	mean	several options
Aggregating a vertex with neighbors?	aggregated with others	concatenated to others
Do weights need to be learned?	not for unsupervised transductive model	yes, for inductive model
Unsupervised?	yes	yes
Supervised?	with modification	with modification
Can be trained on a sample of vertices	no	yes

Summary

Graph-based neural networks put graphs into the mainstream of machine learning. Key takeaways from this section are as follows:

- The graph convolutional neural network (GCN) enhances the vanilla neural network by averaging together the feature vectors of each vertex's neighbors with its own features during the learning process.
- GraphSAGE makes two key improvements to the basic GCN: vertex and neighborhood sampling, and keeping features of vectors separate from those of its neighbors .
- GCN learns transductively (uses the full data to learn only about that data) whereas GraphSAGE learns inductively (uses a data sample to learn a model that can apply to other data samples).
- The modular nature of neural networks and graph enhancements mean that the ideas of GCN and GraphSAGE can be transferred to many other flavors of neural networks.

Comparing Graph Machine Learning Approaches

This chapter has covered many different ways to learn from graph data, but it only scratched the surface. Our goal was not to present an exhaustive survey, but to provide a framework from which to continue to grow. We've outlined the major categories of and techniques for graph-powered machine learning, we've described what characterizes and distinguishes them, and we've provided simple examples to illustrate how they operate. It's worthwhile to briefly review these techniques. Our goal here is not only to summarize but also to provide you with guidance for selecting the right techniques to help you learn from your connected data.

Use Cases for Machine Learning Tasks

Table 5-5 pulls together examples of use cases for each of the major learning tasks. These are the same basic data mining and machine learning tasks you might perform on any data, but the examples are particularly relevant for graph data.

Table 5-5. Use cases for graph data learning tasks

Task	Use case examples
Community detection	Delineating social networks
	Finding a financial crime network
	Detecting an biological ecosystem or chemical reaction network
	Discovering a network of unexpectedly interdependent components or process, such as software procedures or legal regulations
Similarity	Abstraction of physical closeness, inverse of distance
	Prerequisite for clustering, classification, and link prediction
	Entity resolution - finding two online identities that probably refer to the same real-world person
	Product recommendation or suggested action
	Identifying persons who perform the same role in different but analogous networks
Find unknown patterns	Identifying the most common "customer journeys" on your website or in your app
	Discovering that two actions - planning a summer vacation and
	Once current patterns are identified, then noticing changes
Link Prediction	Predicting someone's future purchase or willingness to purchase
	Predicting that business or personal relationship exists, even though it is not recorded in the data
Feature Extraction	Enriching your customer data with graph features, so that your ML training to categorize and model your customers will be more successful
Embedding	Transforming a large set of features to a more compact set, for more efficient computation
Classification (predicting a category)	Given some past examples of fraud, create a model for identifying new cases of fraud
	Predicting categorical outcomes for future vaccine patients, based on test results of past patients
Regression (predicting a numerical value)	Predicting weight loss for diet program participants, based on results of past participants

Once you have identified what type of task you want to perform, consider the available graph-based learning techniques, what they provide, and their key strengths and differences.

Graph-based Learning Methods for Machine Learning Tasks

Table 5-6 lists the graph algorithms and feature extraction methods which we encountered in the chapter. Table 5-7 compares graph neural network methods.

Table 5-6. Features of graph-based learning methods

Task	Graph-based Learning Methods	Comments
Community detection	connected components	one connection to the community is enough
	k-core	at least k connections to other community members
	modularity optimization, e.g. Louvain	relatively higher density of connections inside than between communities
Similarity	Jaccard neighborhood similarity	counts how many relationships in common, for nonnumeric data
	cosine neighborhood similarity	compares numeric or weighted vectors of relationships
	role similarity	defines similarity recursively as having similar neighbors
Find unknown patterns	frequent pattern mining	starts with small patterns and builds to larger patterns
Domain- independent feature extraction	graphlets	systematic list of all possible neighborhood configurations
	PageRank	rank is based on the number and rank of in-neighbors, for directed graphs among vertices of the same type
	closeness centrality	closeness = average distance to any other vertex
	betweenness centrality	how often a vertex lies on the shortest path between any two vertices; slow to compute
Domain- dependent feature extraction	searching for patterns relevant to your domain	custom effort by someone with domain knowledge
Dimensionality reduction and embedding	DeepWalk	embeddings will be similar if the vectors have similar random walks, considering nearness and role; more efficient than SimRank
	node2vec	DeepWalk with directional tuning of the random walks, for greater tuning

Graph Neural Networks: Summary and Uses

The graph neural networks presented in this chapter not only are directly useful in many cases but also templates to show more advanced data scientists how to transform any neural network technique to include graph connectivity in the training. The key is the convolution step, which takes the features of neighboring vertices into account. All of the graph neural network approaches presented can be used for either unsupervised or supervised learning.

Table 5-7. Graph neural networks

Name	Description	Uses
Graph Convolutional Network (GCN)	Convolution: average of neighbor's features	Clustering or classification on a particular graph
GraphSAGE	Convolution: average of a sample of neighbor's features	learning a representative model on a sample of a graph, in addition to clustering or classification
Graph Attention Network (GAT)	Convolution: weighted average of neighbor's features	clustering, classification, and model learning; added tuning and complexity by learning weights for the convolution

Chapter Summary

Graphs and graph-based algorithms contribute to several stages of the machine learning pipeline: data acquisition, data preparation, feature extraction, dimensionality reduction, and model training. As data scientists know, there is no golden ticket, no single technique that solves all their problems. Instead, you work to acquire tools for your toolkit, develop the skills to use your tools well, and gain an understanding about when to use them.

CHAPTER 6
Entity Resolution Revisited

A Note for Early Release Readers

With Early Release ebooks, you get books in their earliest form—the author's raw and unedited content as they write—so you can take advantage of these technologies long before the official release of these titles.

This will be the 11th chapter of the final book.

If you have comments about how we might improve the content and/or examples in this book, or if you notice missing material within this chapter, please reach out to the editor at *gobrien@oreilly.com*.

This chapter uses entity resolution for a streaming video service as an example of unsupervised machine learning with graph algorithms. After completing this chapter, you should be able to:

- Name the categories of graph algorithms which are appropriate for entity resolution as unsupervised learning.
- List three different approaches for assessing the similarity of entities.
- Understand how parameterized weights can adapt entity resolution to be a supervised learning task.
- Interpret a simple GSQL FROM clause and have a general understanding of ACCUM semantics.
- Set up and run a TigerGraph Cloud Starter Kit using GraphStudio.

Goal: Identify Real-World Users and Their Tastes

The streaming video on demand (SVoD) market is big business. Accurate estimates of the global market size are hard to come by, but the most conservative estimate may be $50 billion in 2020[1] with annual growth rates ranging from 11%[2] to 21%[3] for the next five years or so. Movie studios, television networks, communication networks, and tech giants have been merging and reinventing themselves, in hopes of becoming a leader in the new preferred format for entertainment consumption: on-demand digital entertainment, on any video-capable device.

To succeed, SVoD providers need to have the content to attract and retain many millions of subscribers. Traditional video technology (movie theaters and broadcast television) limited the provider to offering only one program at a time per venue or per broadcast region. Viewers had very limited choice, and providers selected content that would appeal to large segments of the public. Home video on VHS tape and DVD introduced personalization; wireless digital video on demand on any personal device has put the power in the hands of the consumer.

Providers no longer need to appeal to the masses. On the contrary, the road to success is microsegmentation: to offer something for everyone. The SVoD giants are assembling sizable catalogs of existing content, as well as spending billions of dollars on new content. The volume of options creates several data management problems. With so many shows available, it is very hard for users to browse. Providers must categorize the content, categorize users, and then recommend shows to users. Good recommendations increase viewership and satisfaction.

While predicting customers' interests is hard enough, the streaming video industry also needs to overcome a multifaceted *entity resolution* problem. Entity resolution, you may recall, is the task of identifying two or more entities in a dataset which refer to the same real-world entity, and then linking or merging them together. In today's market, streaming video providers face at least three entity resolution challenges. First, each user may have multiple different authorization schemes, one for each type of device they use for viewing. Second, corporate mergers are common, which require merging the databases of the constituent companies. For example, Disney+ combines the catalogs of Disney, Pixar, Marvel, and National Geographic Studios. HBO Max brings together HBO, Warner Bros., and DC Comics. Third, SVoD providers may form a promotional, affiliate, or partnership arrangement with another company: a customer may be able to access streaming service A because they are a

1 "Video Streaming Market Report", Grand View Research, February 2021

2 "Video Streaming (SVoD) Report", Statista.com

3 "Video Streaming Market Report", Grand View Research, February 2021

customer of some other service B. For example, customers of AT&T fiber internet service may qualify for free HBO Max service.

Solution Design

Before we can design a solution, let's start with a clear statement of the problem we want to solve.

Title: Problem Statement

Each real-world user may have multiple digital identities. The goal is to discover the hidden connections between these digital identities and then to link or merge them together. By doing so, we will be able to connect all of the information together, forming a more complete picture of the user. In particular, we will know all the videos that a person has watched, in order to get a better understanding of their personal taste and to make better recommendations.

Now that we've crafted a clear problem statement, let's consider a potential solution -- entity resolution. Entity resolution has two parts: deciding which entities are probably the same, and then resolving entities. Let's look at each part in turn.

Learning Which Entities Are The Same

If we are fortunate enough to have training data showing us examples of entities that are in fact the same, we can use supervised learning to train a machine learning model. In this case, we do not have training data. Instead, we will rely on the characteristics of the data itself, looking at similarities and communities to perform unsupervised learning.

To do a good job, we want to build in some domain knowledge. What are the situations for a person to have multiple online identities, and what would be the clues in the data? Here are some reasons why a person may create multiple accounts:

- A user creates a second account because they forgot about or forgot how to access the first one.
- A user has accounts with two different streaming services, and the companies enter a partnership or merge.
- A person may intentionally set up multiple distinct identities, perhaps to take advantage of multiple membership rewards or to separate their behavioral profiles (e.g., to watch different types of videos on different accounts). The personal information may be very different, but the device IDs might be the same.

Whenever the same person creates two different accounts at different moments, there can be variations in some details for trivial or innocuous reasons. The person decides to use a nickname. They choose to abbreviate a city or street name. They mistype. They have multiple phone numbers and email addresses to choose from, and they make a different choice for no particular reason. Over time, more substantial changes may occur, to address, phone number, device IDs, and even to the user's name.

While several situations can result in one person having multiple online identities, it seems we can focus our data analysis on only two patterns. In the first pattern, most of the personal information will be the same or similar, but a few attributes may differ. Even when two attributes differ, they may still be related, such as use of a nickname or a misspelling of an address. In the second pattern, much of the information is different, but one or more key pieces remain the same, such as home phone number or birthdate, and behavioral clues (such as what type of videos they like and what time of day do they watch them) may suggest that two identities belong to the same person.

To build our solution, we will need to use some similarity algorithms and also a community detection or clustering algorithm to group similar entities together.

Resolving Entities

Once we have used the appropriate algorithms to identify a group of entities that we believe to be the same, what will do about it? We want to update the database somehow to reflect this new knowledge. There are two possible ways to accomplish this: merge the group into one entity or link the entities in a special way so that whenever we look at one member of the group, we will readily see the other related identities.

Merging the entities makes sense only when some online identities are considered incorrect, so we want to eliminate them. For example, suppose a customer has two online accounts because they misspelled their name or forgot that they had an account already. Both the business owner and the customer wants to eliminate one account and to merge all of the records (purchase history, game scores, etc.) into one account. Knowing which account to eliminate takes more knowledge of each specific case than we have in our example.

We will take the safer route and simply link entities together. Specifically, we will have two types of entities in the graph: one representing digital identities and the other representing real-world entities. After resolution, the database will show one real-world entity having an edge to each of its digital identities, as illustrated in Figure 6-1.

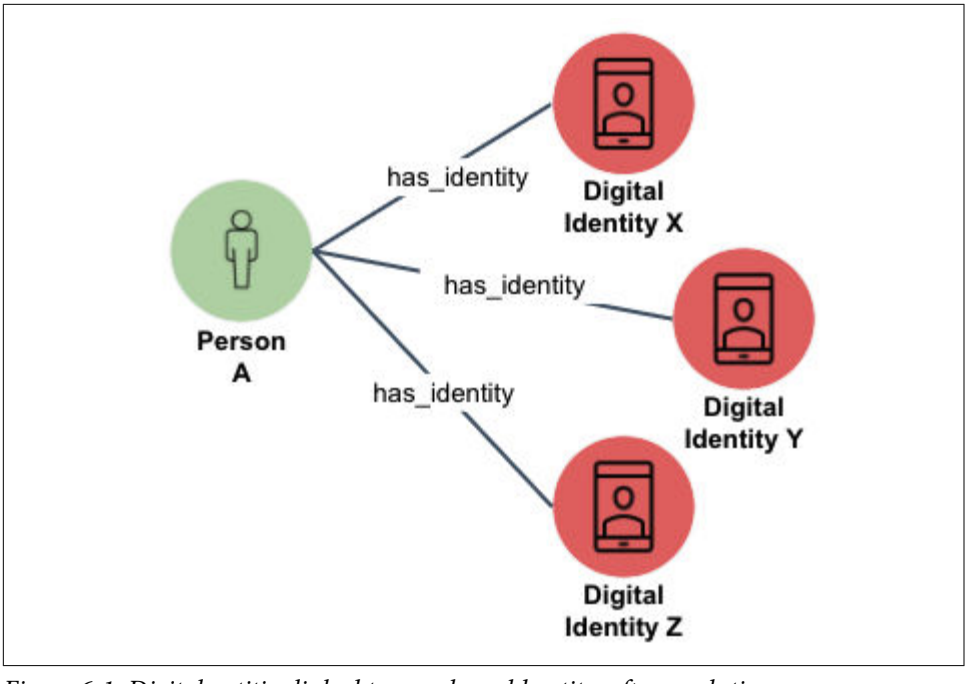


Figure 6-1. Digital entities linked to a real-world entity, after resolution

Implementation

The implementation of graph-based entity resolution we present below is available as a TigerGraph Cloud Starter Kit. As usual, we will focus on using the GraphStudio visual interface. All of the necessary operations could also be performed from a command line interface, but that would lack the visual interface.

Starter Kit

To get and install your Starter Kit, do one of the following::

1. Use a TigerGraph Cloud instance preloaded with this example, go to tgcloud.io and select the Starter Kit called In-Database Machine Learning for Big Data Entity Resolution.
2. Load this Starter Kit onto your own machine which is running TigerGraph, either on-premises or on the cloud, go to www.tigergraph.com/starterkits.
   a. Find In-Database Machine Learning for Big Data Entity Resolution.
   b. Download Data Set and the solution package corresponding to your version of the TigerGraph platform.
   c. Start your TigerGraph instance. Go to the GraphStudio home page.

d. Click Import An Existing Solution, as highlighted in Figure 6-2, and select the solution package which you downloaded

Importing a GraphStudio Solution will delete your existing database. If you wish to save your current design, perform a GraphStudio Export Solution and also gsql backup.

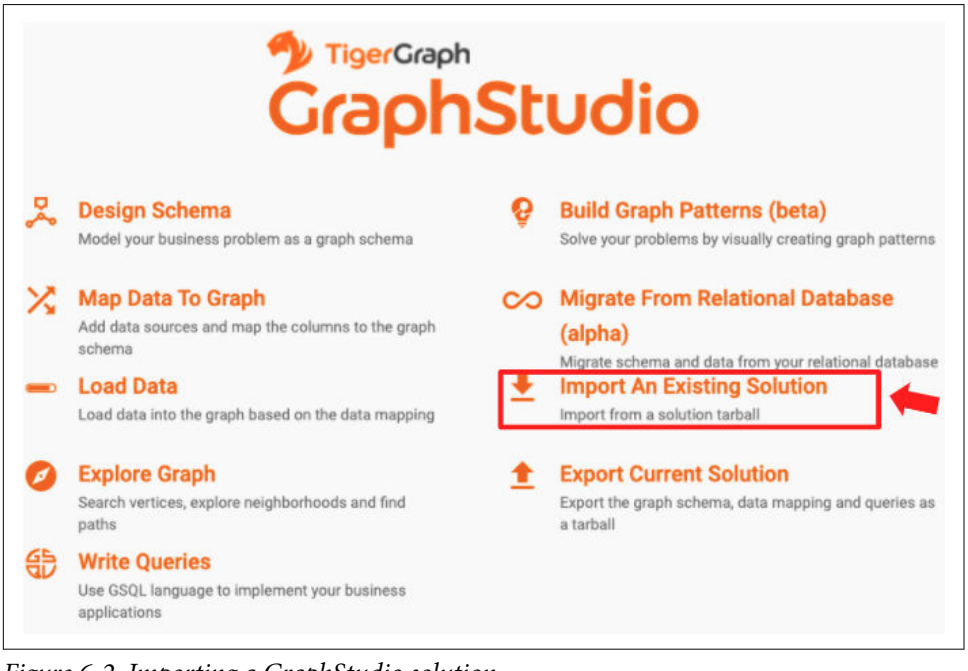


Figure 6-2. Importing a GraphStudio solution

Graph Model

The Starter Kit is preloaded with a graph model based on an actual SVoD business. The name of the graph is *Entity_Resolution*. When you start GraphStudio, you are working at the global graph view. To switch to the Entity_Resolution graph, click on the circular icon in the upper left corner. A dropdown menu will appear, showing you the available graphs and letting you create a new graph. Click on Entity_Resolution.

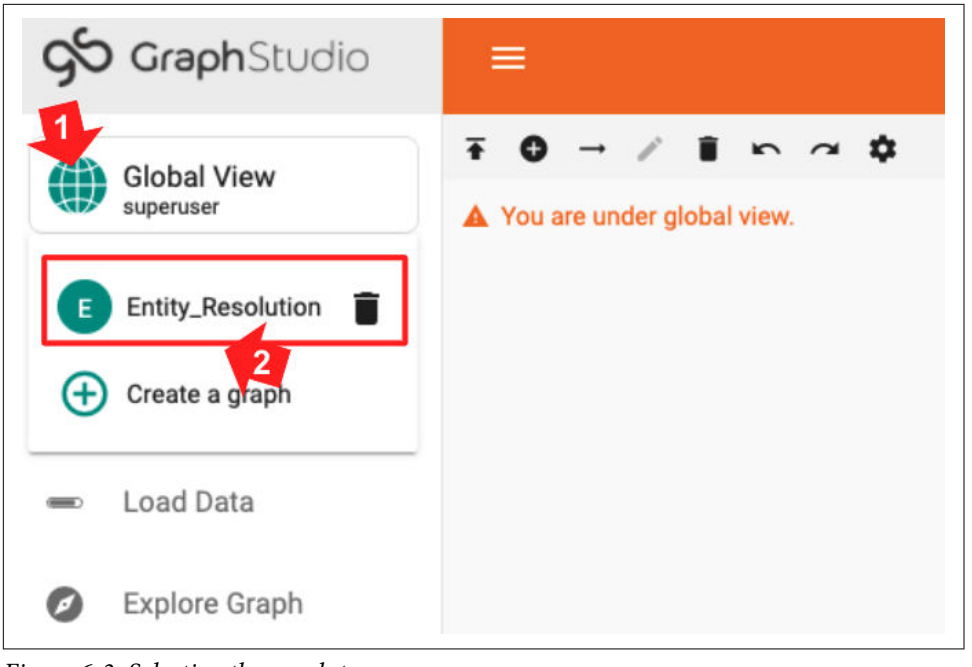


Figure 6-3. Selecting the graph to use

You should now see a graph model or schema like the one in Figure 6-4 in the main display panel.

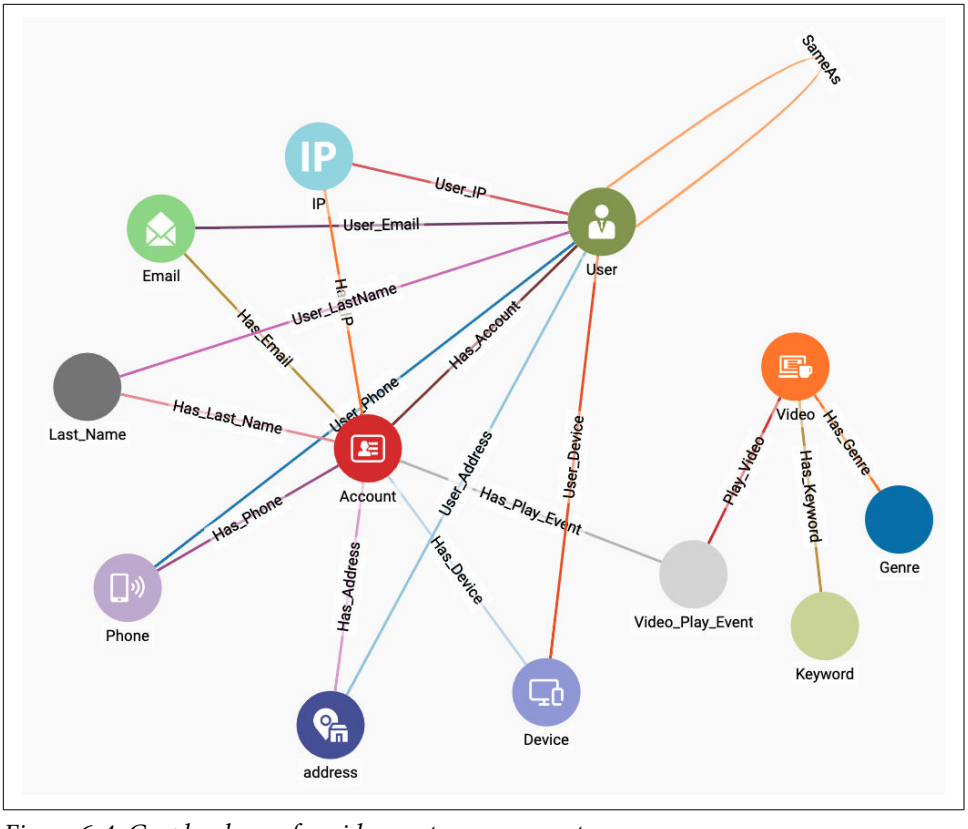


Figure 6-4. Graph schema for video customer accounts

You can see that Account, User, and Video are hub vertices with several edges radiating from them. The other vertices represent the personal information about users and the characteristics of videos. We want to compare the personal information of different users. Following good practice for graph-oriented analytics, if we want to see if two or more entities have something feature in common, e.g., email address, we model that feature as a vertex instead of as a property of a vertex. Table 6-1 gives a brief explanation of each of the vertex types in the graph model. Though the starter kit's data contains much data about videos, we will not focus on the videos themselves in this exercise. We are going to focus on entity resolution of Accounts.

Table 6-1. Vertex types in the graph model

Account	an account for a SVoD user, a digital identity
User	a real-world person. One User can link to multiple Accounts
IP, Email, Last_User, Phone, Address, Device	attributes of an Account. They are represented as vertices instead of internal properties of Account to facilitate linking Accounts/Users that share a common attribute.
Video	a video title offered by an SVoD

Keyword, Genre	attributes of a Video
Video_Play_Event	the time and duration of a particular Account viewing a particular Video

Data Loading

In TigerGraph Starter Kits, the data is included, but it is not yet loaded into the database. To load the data, switch to the Load Data page (step 1 of Figure 6-5), wait a few seconds until the Load button in the upper left of the main panel becomes active, and then click it (step 2). You can watch the progress of the loading in the real-time chart at the right (not shown). Loading the 84K vertices and 270K edges should take only about 40 seconds on the TGCloud free instances; faster on the paid instances.

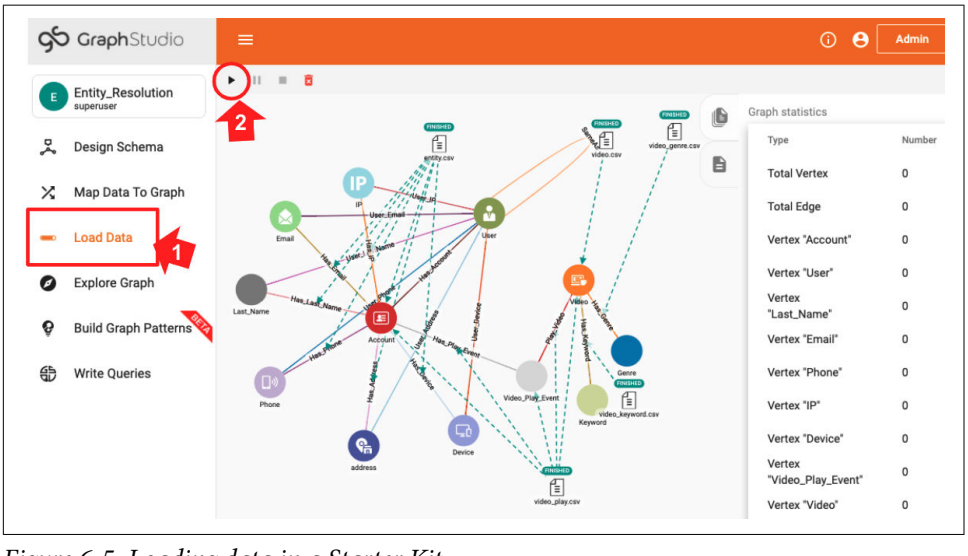


Figure 6-5. Loading data in a Starter Kit

Queries and Analytics

We will analyze the graph and run graph algorithms by composing and executing queries in GSQL, TigerGraph's graph query language. When you first deploy a new Starter Kit, you need to install the queries. Switch to the Write Queries page (step 1 of Figure 6-6). Then at the top right of the list of queries, click the Install All icon (step 2).

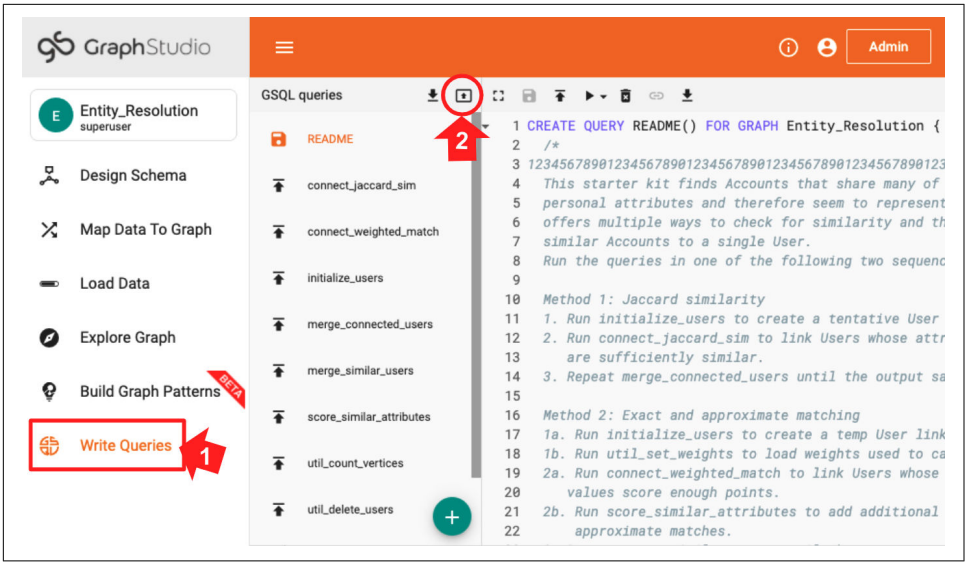


Figure 6-6. Installing queries

For our entity resolution use case, we have a three-stage plan requiring three or more queries.

Initialization
: For each `Account` vertex, create its own `User` vertex and link them. Accounts are online identities, and Users represent real-world persons. We begin with the hypothesis that each `Account` is a real person.

Similarity Detection
: Apply one or more similarity algorithms to measure the similarity between `User` vertices. If we consider a pair to be similar enough, then we create a link between them, using the `SameAs` edge type shown in Figure 6-5.

Merging
: Find the connected components of linked User vertices. Pick one of them to be the main vertex. Transfer all of the edges of the other members of the community to the main vertex. Delete the other community vertices.

For reasons we will explain when we talk about merging, you may need to repeat steps 2 and 3 as a pair, until the Similarity Detection step is no longer creating any new connections.

We will present two different methods for implementing entity resolution in our use case. The first method uses Jaccard similarity (introduced in Chapter 10) to count exact matches of neighboring vertices. Merging will use a Connected Component algorithm. The second method is a little more advanced, suggesting a way to handle

both exact and approximate matches of attribute values. Approximate matches are a good way to handle minor typos or the use of abbreviated names.

Method 1: Jaccard Similarity

For each of the three stages, we'll give a high level explanation, directions for operations to perform in TigerGraph's GraphStudio, what to expect as a result, and a closer look at some of the GSQL code in the queries.

Initialization

Recall in our model that an Account is a digital identity, and a User is a real person. The original database contains only Accounts. The initialization step creates a unique temporary User linked to each Account. And, for every edge from an Account to one of the attribute vertices (Email, Phone, etc.), we create a corresponding edge from the User to the same set of attribute vertices. Figure 6-7 shows an example. The three vertices on the left and the two edges connecting them are part of the original data. The initialization step creates the User vertex and the three dotted-line edges. As a result, each User starts out with the same attribute neighborhood as its Account.

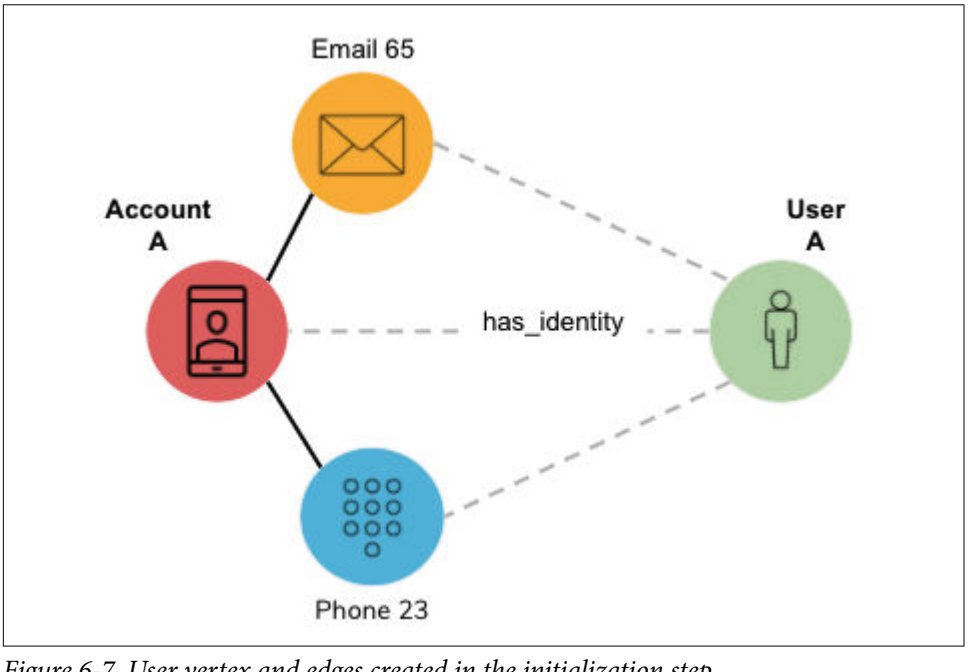


Figure 6-7. User vertex and edges created in the initialization step

Run the GSQL query `initialize_users` by selecting the query name from the list (step 1 in Figure 6-8) and then clicking the Run icon above the code panel (step 2).

This query has no input parameters, so it will run immediately without any additional steps from the user.

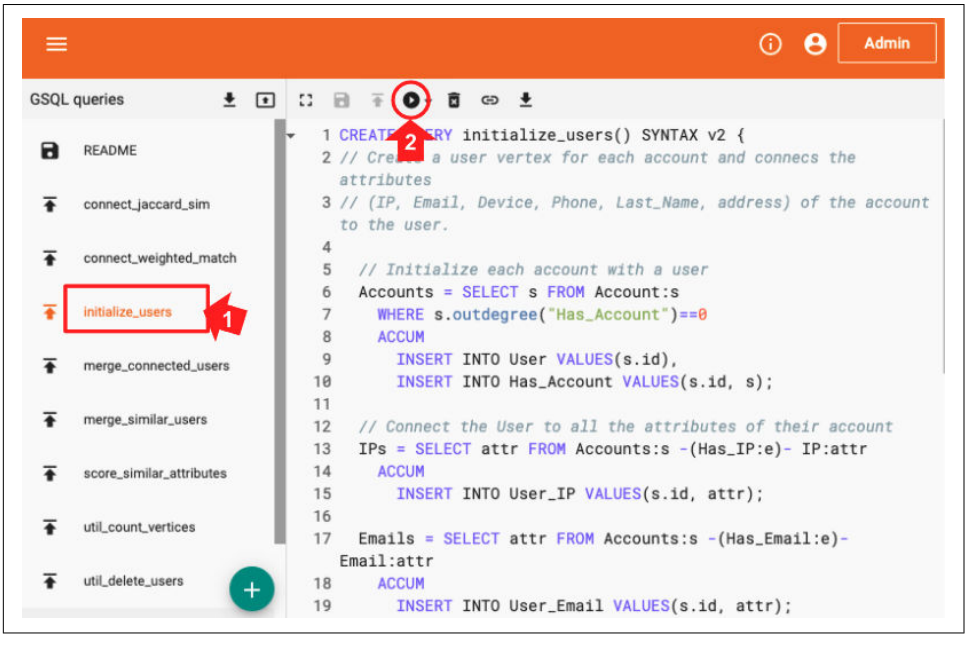


Figure 6-8. Running the `initialize_users` query

Let's take a look at the GSQL code. The block of code below shows the first 20 lines of *initialize_users*. If you are familiar with SQL, then GSQL may look familiar. The comment at the beginning (lines 2 and 3) lists the 6 types of attribute vertices to be included. Lines 6 to 10 create a *User* vertex (line 9) and an edge connecting the *User* to the *Accounts* (line 10) for each *Account* (line 6) which doesn't already have a neighboring *User* (line 7).

```
1   CREATE QUERY initialize_users() SYNTAX v2 {
2   // Create a User vertex for each Account, plus edges to connect attributes
3   // (IP, Email, Device, Phone, Last_Name, Address) of the Account to the User
4
5   // Initialize each account with a user
6   Accounts = SELECT s FROM Account:s
7   WHERE s.outdegree("Has_Account")==0
8   ACCUM
9     INSERT INTO User VALUES(s.id),
10    INSERT INTO Has_Account VALUES(s.id, s);
11
12  // Connect the User to all the attributes of their account
13  IPs = SELECT attr FROM Accounts:s -(Has_IP:e)- IP:attr
14  ACCUM
15    INSERT INTO User_IP VALUES(s.id, attr);
```

```
16
17  Emails = SELECT attr FROM Accounts:s -(Has_Email:e)- Email:attr
18  ACCUM
19    INSERT INTO User_Email VALUES(s.id, attr);
20  // Remaining code omitted for brevity
21  }
```

GSQL's ACCUM clause is an iterator with parallel asynchronous processing. Think of it as "FOR EACH set of vertices and edges which match the pattern in the FROM clause, do the following."

Lines 13 to 15 take care of *IP* attribute vertices: If there is a Has_IP edge from an Account to an IP vertex, then insert an edge from the corresponding User vertex to the same IP vertex. While the alias `s` defined in line 13 refers to an Account, `s.id` in line 15 can refer to a User because the source vertex of a User_IP edge may only be a User, and we recently used `s.id` to create a User (line 9). Lines 17 to 19 take care of *Email* attribute vertices in an analogous way. The code blocks for the remaining four attribute types (*Device, Phone, Last_Name*, and *Address*) have been omitted for brevity.

Similarity Detection

Jaccard similarity counts how many attributes two entities have in common, divided by the total number of attributes between them. Each comparison of attributes results in a yes/no answer; a miss is as good as a mile. Figure 6-9 shows an example where User A and User B each have three attributes; two of those match (Email 65 and Device 87). Therefore, A and B have two attributes in common.

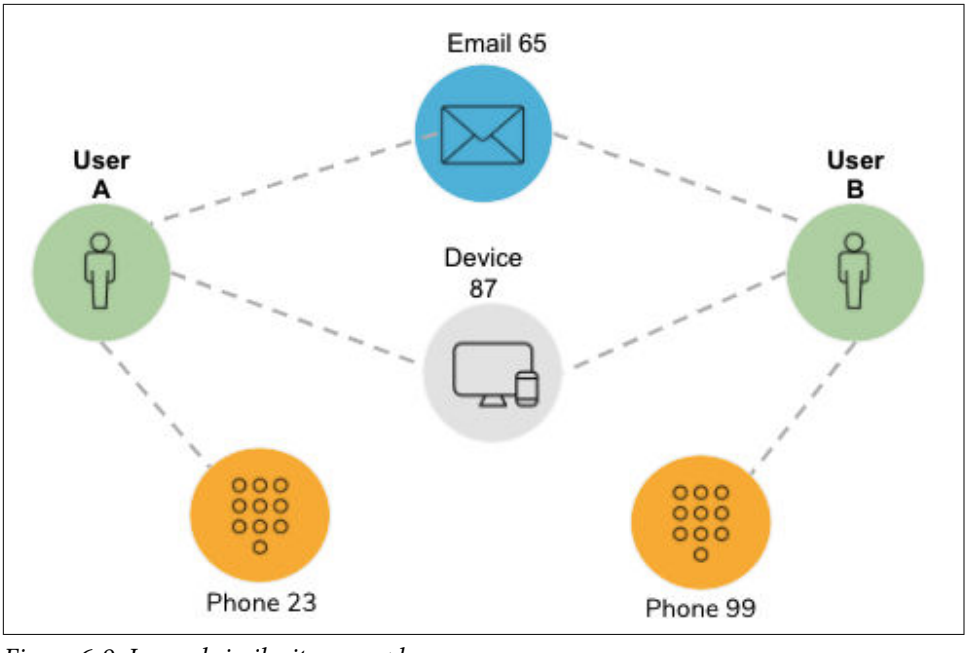


Figure 6-9. Jaccard similarity example

They have a total of four distinct attributes (Email 65, Device 87, Phone 23, and Phone 99); therefore, the Jaccard similarity is 2/4 = 0.5.

Do: Run *connect_jaccard_sim*.

The query *connect_jaccard_sim* computes this similarity score for each pair of vertices. If the score is at or above the given threshold, then create a Same_As edge to connect the two Users. The default threshold is 0.5, but you can make it higher or lower. Jaccard scores range from 0 to 1. Figure 6-10 shows the connections for User vertices 1, 2, 3, 4, and 5, using Jaccard similarity and a threshold of 0.5. For these five vertices, we need communities that range in size from one vertex alone (User 3) to two clusters of three (Users 1 and 2).

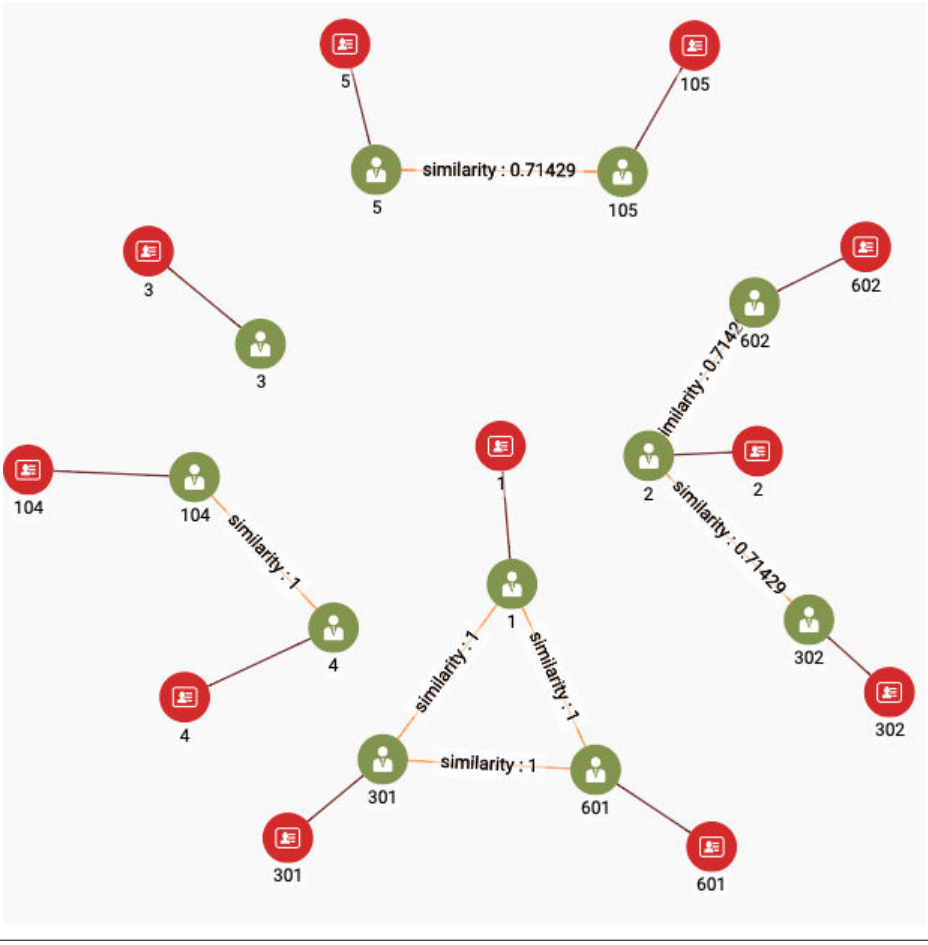


Figure 6-10. Connections for User vertices 1, 2, 3, 4, and 5, using Jaccard similarity and a threshold of 0.5

How to Visualize User Communities

You can use the Explore Graph page in TigerGraph GraphStudio to create a display like the one in Figure 6-10.

1. Click the Select Vertices icon at the top of the left-side menu (step 1 in Figure 6-11).
2. Choose the User type vertex. Enter the vertex ID 1 and click the Select icon. (step 2 in Figure 6-11). User 1 will appear in the display pane. Repeat for vertex IDs 2, 3, 4, and 5.

3. Shift-click the vertices so that all of them are selected (step 3, Figure 6-12).
4. Click the Expand From Selected Vertices icon, the next item in the left-side menu (step 4).
5. You are now presented with a checklist of all the edge types you wish to traverse, followed by a checklist of all the target vertex types. We want to include only the User and Account vertex types (step 5). This specifies a one-hop exploration.
6. We need to explore multiple hops, including the full communities. We don't know the diameter of the communities, but let's just guess that three hops is enough. Click the Add Expansion Step button at the bottom (step 6). Another set of checklists appear. Again, select only the User and Account vertex types. This is the second hop. Repeat these steps to set your third hop.
7. Click the Expand button above the checklists (step 7).

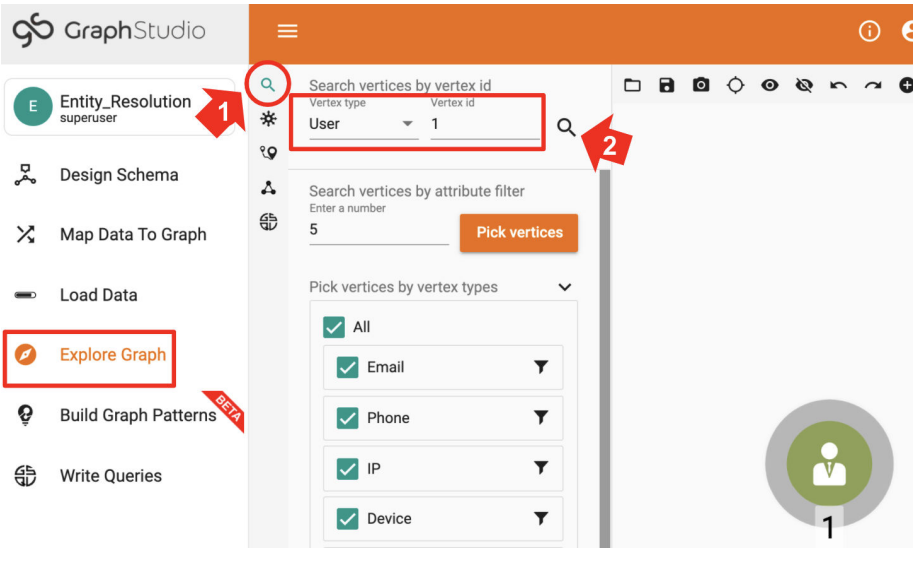


Figure 6-11. Selecting vertices on the Explore Graph page.

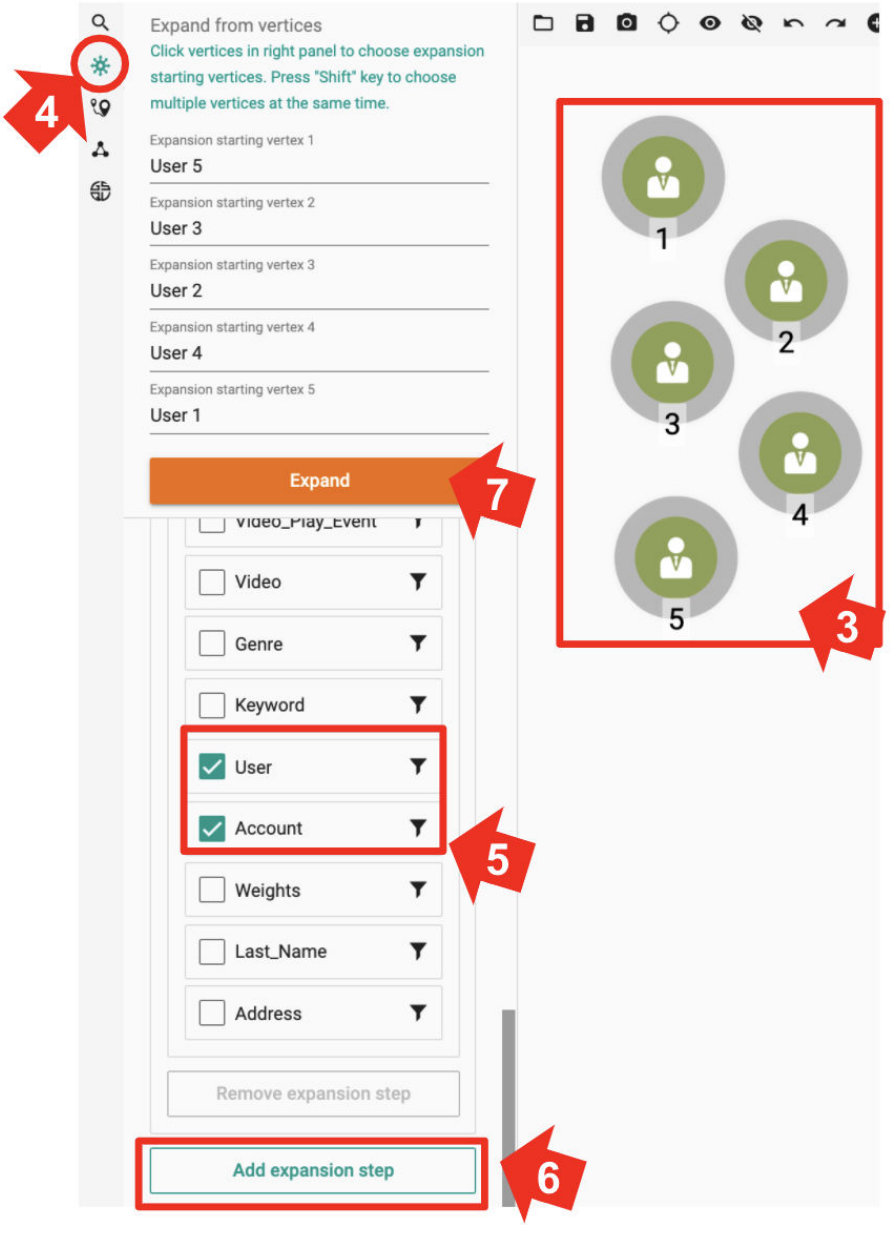


Figure 6-12. Expanding from vertices

Rather than showing the full code, we'll just focus on a few excerpts. In the first code snippet, we'll show how to count the attributes in common between every pair of Users, using a single SELECT statement. The statement uses pattern-matching to describe how two such Users would be connected, and then it uses an accumulator to count the occurrences.

```
1   Others = SELECT B FROM
2   Start:A -()- (IP|Email|Phone|Last_Name|Address|Device):n -()- User:B
3   WHERE B != A
4   ACCUM
5     A.@intersection += (B -> 1); // tally each path A->B,
6     @@path_count += 1            // count the total number of paths
```

GSQL's FROM clause describes a left-to-right path moving from vertex to vertex via edges. Each sequence of vertices and edges which fit the requirements form one "row" in the temporary "table" of results, which is passed on to the ACCUM and POST-ACCUM clauses for further processing.

The FROM clause in lines 1 to 2 presents a graph path pattern to search for. `FROM User:A -()- (IP|Email|Phone|Last_Name|Address|Device):n -()- User:B` defines a two-hop path pattern:

- `User:A` means start from a User vertex, aliased to `A`,
- `-()-` means pass through any edge type
- `(IP|Email|Phone|Last_Name|Address|Device):n` means arrive at one of these six vertex types, aliased to `n`
- `-()-` means pass through another edge of any type, and finally
- `User:B` means arrive at a User vertex, aliased to `B`

In line 3, (`WHERE B != A`) ensures that we skip the situation of a loop where A = B. Line 4 announces the start of an ACCUM clause. Line 5 (`A.@intersection += (B -> 1); // tally each path A->B`) is a good example of GSQL's support for parallel processing and aggregation: for each path from A to B, append a key → value record attached to A. The record is (B, +=1). That is, if this is the first record associating B with A, then set the value to 1. For each additional record where B is A's target, then increment the value by 1. Hence, we're counting how many times there is a connection from A to B, via one of the six specified edge types. Line 6 (`ACCUM @@path_count += 1`) is just for bookkeeping purposes, to count now many of these paths we find.

Let's look at one more code block, the final computation of Jaccard similarity and creation of connections between Users.

```
1   Result = SELECT A FROM User:A
2   ACCUM FOREACH (B, overlap) IN A.@intersection DO
3   FLOAT score = overlap*1.0/(@@deg.get(A) + @@deg.get(B) - overlap),
4   IF score > threshold THEN
5     INSERT INTO EDGE SameAs VALUES (A, B, score), // FOR Entity Res
6     @@insert_count += 1,
7     IF score != 1 THEN
8       @@jaccard_heap += SimilarityTuple(A,B,score)
9     END
10  END
11  END;
```

This SELECT block does the following:

1. For each User A, iterate over its record of similar Users B and the number of common neighbors, aliased to `overlap`.
2. For each such pair (A, B), compute the Jaccard score, using `overlap` as well as the number of qualified neighbors of A and B (`@@deg.get(A)` and `@@deg.get(B)`), computed earlier.
3. If the score is greater than the threshold, insert a SameAs edge between A and B.
4. `@@insert_count` and `@@jaccard_heap` are for reporting statistics and are not essential.

Merging

In our third and last stage, we merge together the connected communities of User vertices which we created in the previous step. For each community, we will select one vertex to be the survivor or lead. The remaining members will be deleted; all of the edges from an Account to a non-lead will be redirected to point to the lead User.

Do: run query *merge_connected_users*. Look at the JSON output. Note whether t says `converged = TRUE` or `FALSE`.

Figure 6-13 displays the User communities for Accounts 1, 2, 3, 4, and 5. The user communities have been reduced to a single User (real person). Each of those Users links to one or more Accounts (digital identities). We've achieved our entity resolution.

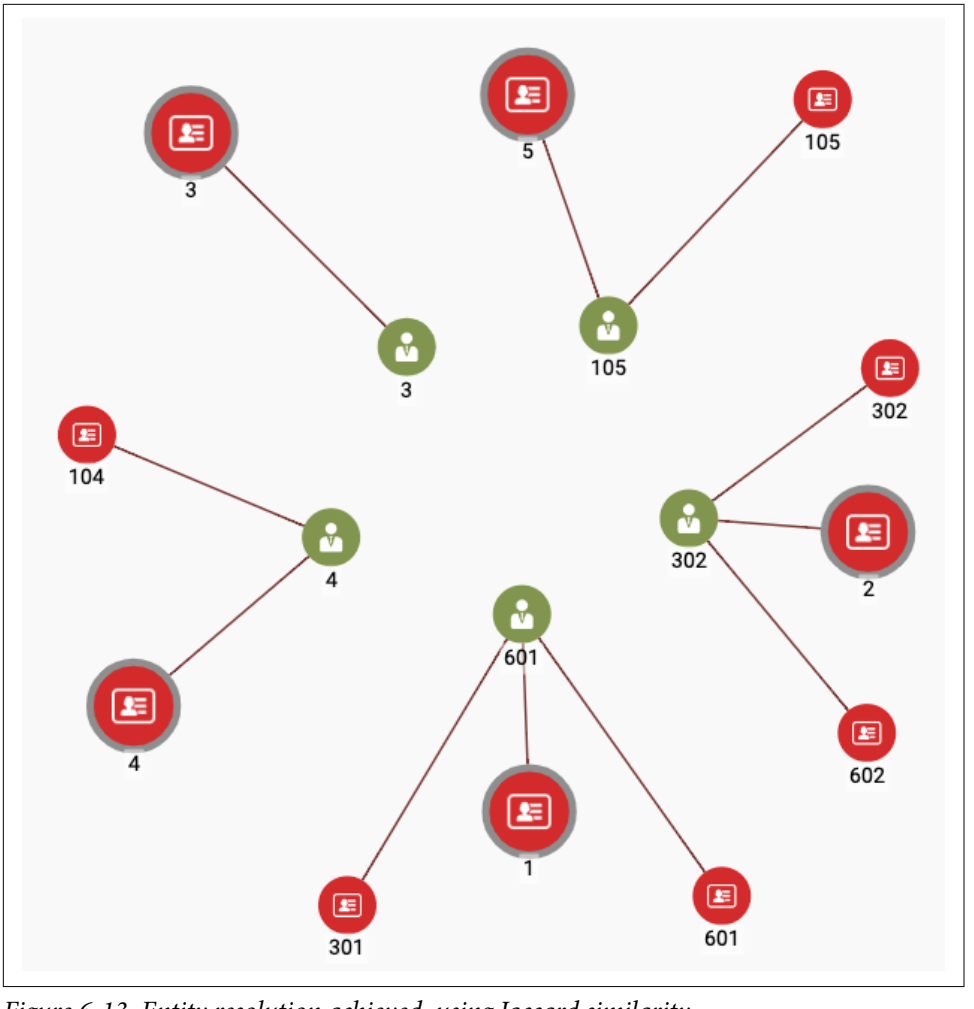


Figure 6-13. Entity resolution achieved, using Jaccard similarity

The *merge_connected_users* algorithm has four stages:

1. Find connected users using the connected component algorithm.
2. In each component, select a lead user.
3. In each component, redirect the attribute connections from other users to the lead user.
4. Delete the users that are not the lead user and all of the Same_As edges.

We'll take a closer look at the GSQL code for the Connected Components algorithm.

```
1   Users = {User.*};
2   Updated_users = SELECT s FROM Users:s
3     POST-ACCUM
4       s.@min_user_id = s;
5
6   WHILE (Updated_users.size() > 0) DO
7     IF verbose THEN PRINT iteration, Updated_users.size(); END;
8     Updated_users = SELECT t
9      FROM Updated_users:s -(SameAs:e)- User:t
10      // Propagate the internal IDs from source to target vertex
11     ACCUM t.@min_user_id += s.@min_user_id   // t gets the lesser of t & s ids
12     HAVING t.@min_user_id != t.@min_user_id' // accum is accum's previous val
13      ;
14    iteration = iteration + 1;
15  END;
```

Line 1 initializes the vertex set called `Users` to be all User vertices. Lines 2 to 4 initializes an accumulator variable `@min_user_id` for each User. The initial value is the vertex's internal ID (different from the externally visible ID). This variable will store the algorithm's current best guess at the community ID for this vertex: all vertices having the same value of @min_user_id belong to the same community. It's important to note that this accumulator is a MinAccum. Whenever you input a new value to a MinAccum, it retains the lesser of its current value and the new input value.

Lines 8 to 11 says that for each connected User pair from `s` to `t`, set `t.@min_user_id` to the lesser of `s` and `t`'s community IDs. Line 12 says that the output set (`Updated_users` in line 8) should contain only those User vertices who updated their community ID in this round. Note the tick mark ' at the end of the line; this is actually a modifier for the accumulator t.@min_user_id. It means "the previous value of the accumulator"; this lets us easily compare the previous to current values. When no vertices have changed their ID, then we can exit the WHILE loop (line 6).

Are We There Yet?

It might seem that one pass through the three steps – initialize, connect similar entities, and merge connected entities – should be enough. The merging, however, can create a situation in which new similarities arise. Take a look at Figure 6-14, which depicts the attribute connections of User 302 after Users 2 and 602 have been merged into it. Accounts 2, 302, and 602 remain separate, so you can see how each of them contributed some attributes. Because User 302 has more attributes than before, it is now possible that it is more similar than before to some other (possibly newly merged) User. Therefore, we should run another round of similarity connection and merge. Repeat these steps until no new similarities arise.

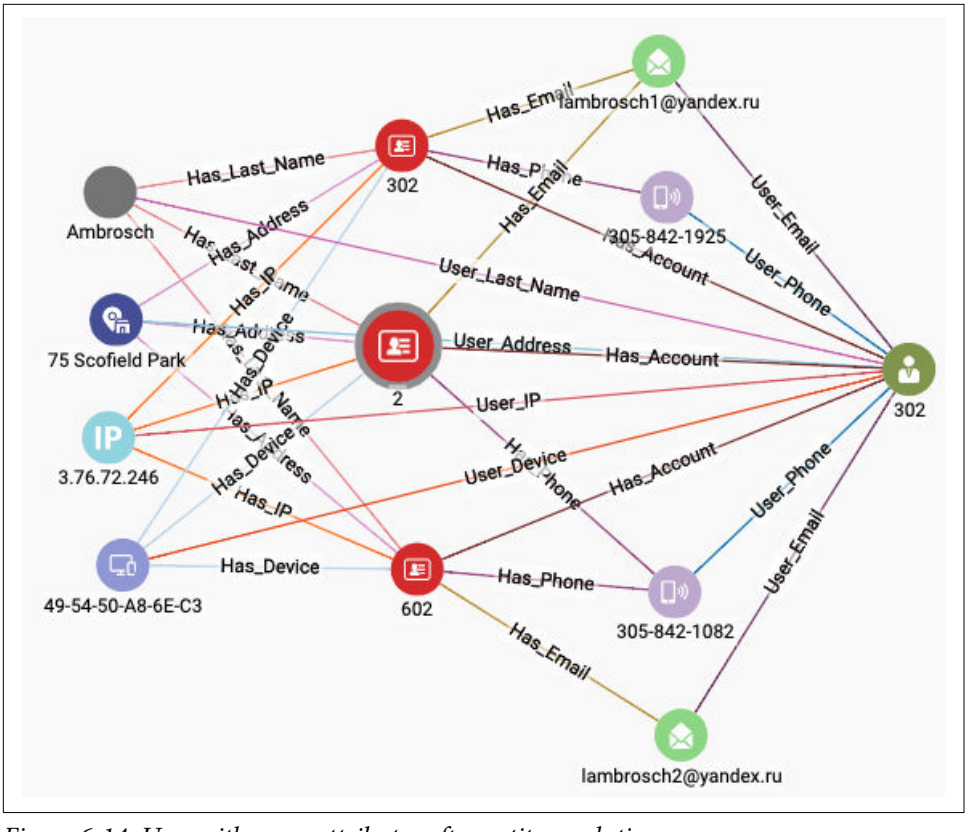


Figure 6-14. User with more attributes after entity resolution

For convenient reference, here is the sequence of queries we ran for simple entity resolution using Jaccard similarity:

Run *initialize_users*

Run *connect_jaccard_sim*

Run *merge_connected_users*

Repeat steps 2 & 3 until the output of *merge_connected_users* says conversed = TRUE

Reset

After you've finished, or at any time, you might want to restore the database to its original state. You need to do this if you want to run the entity resolution process from the start again. The query *util_delete_users* will delete all User vertices and all edges connecting to them. Note that you need to change the input parameter are_you_sure from FALSE to TRUE. This manual effort is put in as a safety precaution.

Deleting bulk vertices (*util_delete_users*) or creating bulk vertices (*initialize_users*) can take tens of seconds in the modestly sized free TigerGraph Cloud instances. Go to the Load Data page to check the live statistics for User vertices and User-related edges to see if the creation or deletion has finished.

Method 2: Scoring Exact and Approximate Matches

The previous section demonstrated a nice and easy graph-based entity resolution technique, but it is too basic for real-world use. It relies on exact matching of attribute values, whereas we need to allow for almost-the-same values, which arise from unintentional and intentional spelling variations. We also would like to make some attributes more important than others. For example, if you happen to have date-of-birth information, you might be strict about this attribute matching exactly. While persons can move, have multiple phone numbers and email addresses, they can only have one birthdate. In this section, we will introduce weights to adjust the relative importance of different attributes. We will also provide a technique for approximate matches of string values.

If you already used your starter kit to run Method 1, be sure to reset it. (See the Reset section at the end of Method 1.)

Initialization

We are still using the same graph model with User vertices representing real persons and Account vertices representing digital accounts. So, we are still using the initialize_users query to set up an initial set of User vertices.

We are adding another initialization step. We are going to assign weights to each of the six attributes: IP, Email, Phone, Address, Last_Name, and Device. If this were a relational database, we would store those weights in a table. Since this is a graph, we are going to store them in a vertex. We only need one vertex, because one vertex can have multiple attributes. However, we are going to be even more fancy. We are going to use a map type attribute, which will have six key → value entries. This allows us to use the map like a lookup table: tell me the name of the key (attribute name), and I'll tell you the value (weight).

Do:

1. Run *initialize_users*. Check the graph statistics on the Load Data page to make sure that all 900 Users and related edges have been created.

2. Run *util_set_weights*. The weights for the six attributes are input parameters for this query. Default weights are included, but you may change them if you wish. If you want to see the results, run *util_print_vertices*.

Scoring Weighted Exact Matches

We are going to do our similarity comparison and linking in two phases. In phase one, we are still checking for exact matches because exact matches are more valuable than approximate matches; however, those connections will be weighted. In phase two, we will then check for approximate matches for our two attributes which have alphabetic values: Last_Name and Address.

In weighted exact matching, we create weighted connections between Users, where higher weights indicate stronger similarity. The net weight of a connection is the sum of the contributions from each attribute that is shared by the two Users. Figure 6-15 illustrates the weighted match computation. Earlier, during the initialization phase, you established weights for each of the attributes of interest. In the figure, we use the names `wt_email` and `wt_phone` for the weights associated with matching Email and Phone attributes, respectively.

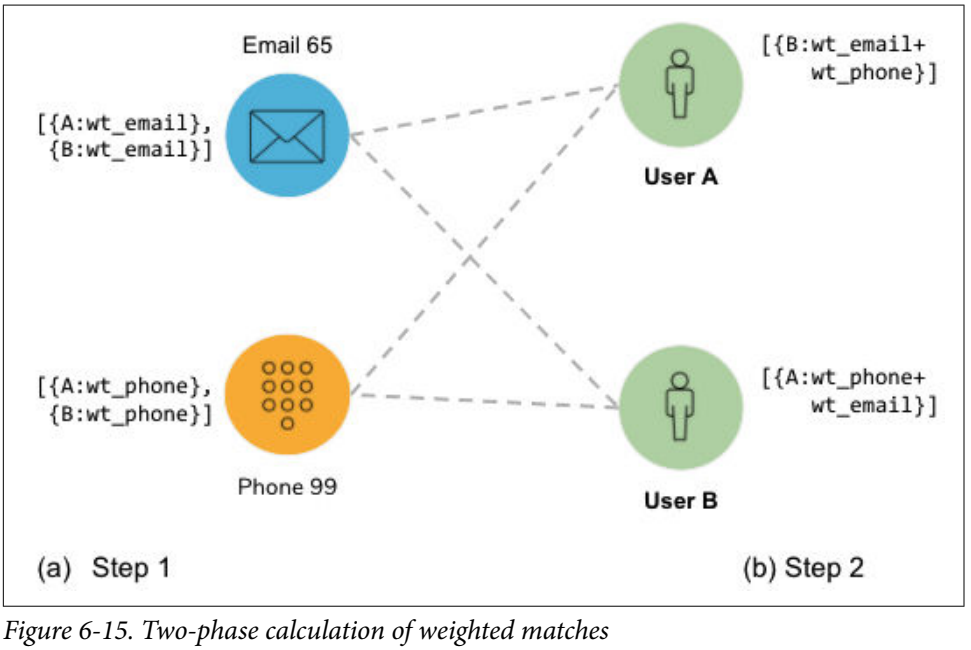


Figure 6-15. Two-phase calculation of weighted matches

The weighted match computation has two steps. In step 1, we look for connections from Users to Attributes and record a weight on each attribute for a connection to each User. Both User A and User B connect to Email 65, so Email 65 records

`A:wt_email` and `B:wt_email`. Each User's weight needs to be recorded separately. Phone 99 also connects to Users A and B so it records analogous information.

In step 2, we look for the same connections but in the other direction, with Users as the destinations. Both Email 65 and Phone 99 have connections to User A. User A aggregates their records from step 1. Note that some of those records refer to User A. User A ignores those, because it is not interested in connections to itself! In this example, it ends up recording `B:(wt_email + wt_phone)`. We use this value to create a weighted Same_As edge between Users A and B. You can see that User B has equivalent information about User A.

Do: run *connect_weighted_match*.

Figure 6-16 shows one of the communities generated by *connect_weighted_match*. This particular community is the one containing User/Account 5. The figure also shows connections to two attributes, Address and Last_Name. The other attributes such as Email were used in the scoring but are not shown, to avoid clutter.

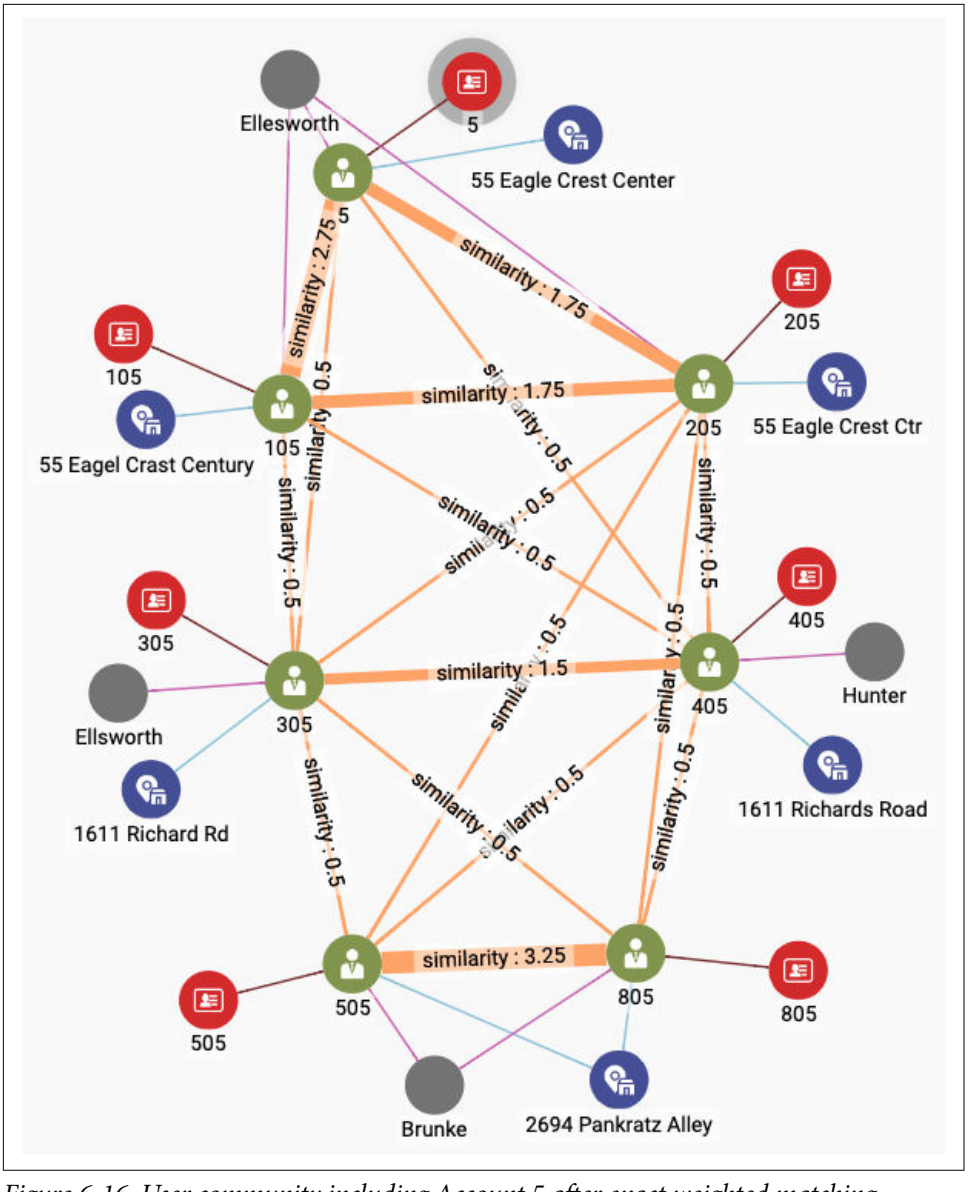


Figure 6-16. User community including Account 5 after exact weighted matching

The thickness of the SameAs edges indicates the strength of the connection. The strongest connection is between Users 505 and 805 at the bottom of the screen. In fact, we can see three subcommunities of Users among the largest community of seven members:

- Users 5, 105, and 205 at the top. The bond between Users 5 and 105 is a little stronger, for reasons not shown. All three share the same last name. They have similar addresses.
- Users 305 and 405 in the middle. Their last names and address are different, so some of the attributes not shown must be the cause of their similarity.
- Users 505 and 805 at the bottom. They share the same last name and address, as well as other attributes.

Scoring Approximate Matches

We can see (in Figure 6-16) that some Users have similar names (Ellsworth vs. Ellesworth) and similar addresses (Eagle Creek Center vs. Eagle Crest Ctr). A scoring system that looks only for exact matchings gives us no credit for these near misses. An entity resolution system would like to be able to assess the similarity of two text strings and to assign a score to the situation. Do they differ by a single letter, like Ellsworth and Ellsworth? Are letters transposed, like Center and Cneter? Computer scientists like to think of the *edit distance* between two text strings: how many single-letter changes of value or position are needed to transform string X into string Y?

We are going to use the Jaro-Winkler similarity to measure the similarity between two strings, an enhancement of Jaro similarity. Given two strings s1 and s2, who have m matching characters and t transformation steps between them, then their Jaro similarity is defined as shown in Equation 6-1.

Equation 6-1. Definition of Jaro similarity

$$Jaro(s1, s2) = \frac{1}{3}\left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m}\right)$$

If the strings are identical, then m = |s1| = |s2|, and t =0, so the equation simplifies to (1 + 1 + 1)/3 = 1. On the other hand, if there are no letters in common, then the score = 0. We then multiply this score by the weight for the attribute type. For example, if the attribute's weight is 0.5, and if the JaroWinkler similarity score is 0.9, then the net score is 0.5 * 0.9 = 0.45.

Jaro-Winkler similarity takes Jaro as a starting point and adds an additional reward if the beginnings of each string, reading from the left end, match exactly.

Do: Run *score_similar_attributes*.

The *score_similar_attributes* query considers the User pairs which already are linked by a Same_As edge. It computes the weighted Jaro-Winkler similarity for the Last_Name and the Address attributes, and adds those scores to the existing similarity score. We chose Last_Name and Address because they are alphabetic instead of

numeric. This is an application decision rather than a technical one. Figure 6-17 shows the results after adding in the scores for the approximate matches.

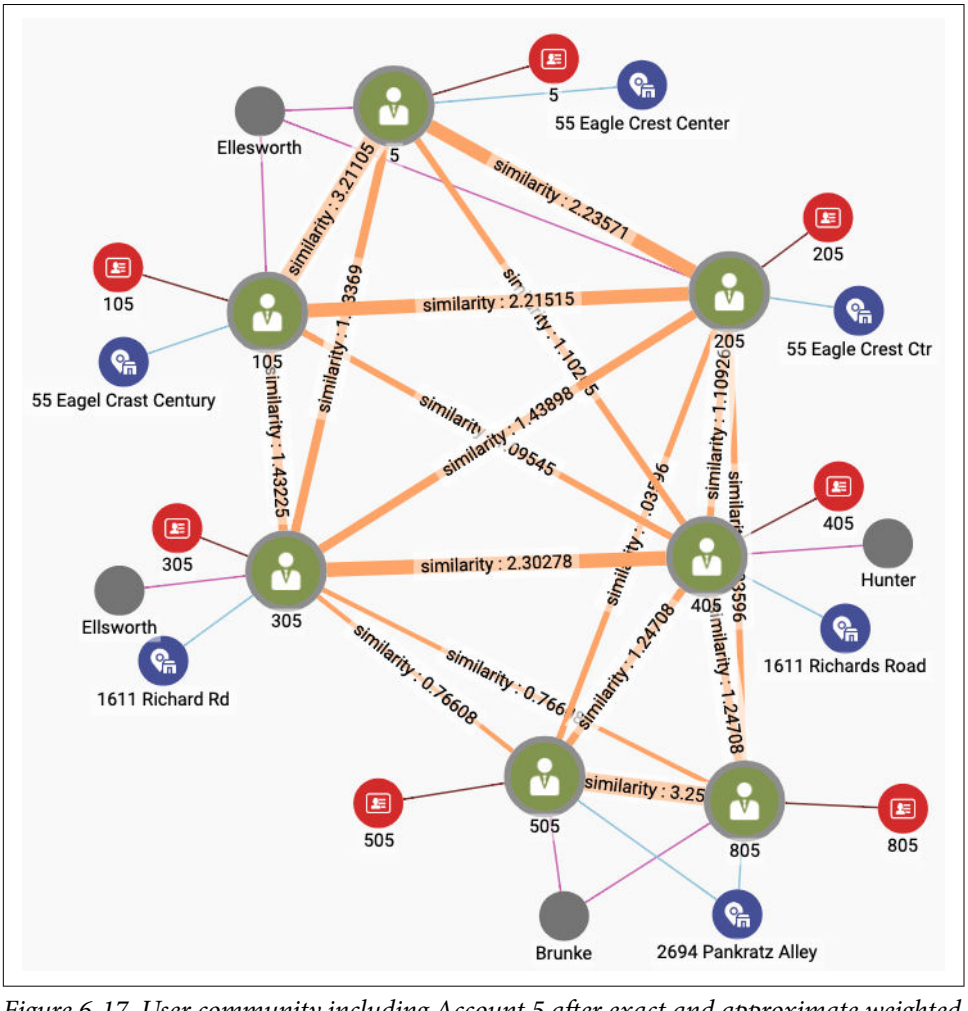


Figure 6-17. User community including Account 5 after exact and approximate weighted matching

Comparing Figures 11-16 and 11-17, we notice the following changes:

- The connections among Users 1, 105, and 205 have strengthened due to their having similar addresses.
- User 305 is more strongly connected to the trio above due to a similar last name.
- The connection between 305 and 405 has strengthened due to their having similar addresses.

- User 405 is more strongly connected to Users 505 and 805 due to the name Hunter having some letters in common with Brunke. This last effect might be considered an unintended consequence of the Jaro-Winkler similarity measure not being as judicious as a human evaluator would be.

Comparing two strings is a general purpose function which does not require graph traversal, so we have implemented it as a simple string function in GSQL. Because it is not yet a built-in feature of the GSQL language, we took advantage of GSQL's ability to accept a user-supplied C++ function as a user-defined function (UDF). The UDFs for `jaroDistance(s1, s2)` and `jaroWinklerDistance(s1, s2)` are included in this starter kit. You can invoke them from within a GSQL query anywhere that you would be able to call a built-in string function.

The code snippet below shows how we performed the approximate matching and scoring for the Address feature:

```
1   Connected_users = SELECT A
2     // Find all linked users, plus each user's address
3     FROM Connected_users:A -(SameAs:e)- User:B,
4        User:A -()- Address:A_addr,
5        User:B -()- Address:B_addr
6     WHERE A.id < B.id    // filter so we don't count (A,B) & (B,A)
7     ACCUM @@addr_match += 1,
8     // If addresses aren't identical compute JaroWinkler * weight
9     IF do_address AND A_addr.val != B_addr.val THEN
10      FLOAT sim = jaroWinklerDistance(A_addr.id,B_addr.id) * addr_wt,
11      @@sim_score += (A -> (B -> sim)),
12      @@string_pairs += String_pair(A_addr.id, B_addr.id),
13      IF sim != 0 THEN @@addr_update += 1 END
14    END
15  ;
```

Lines 3 to 5 are an example of a *conjunctive path pattern*, that is, a compound pattern comprising several individual patterns, separated by commas. The commas act like boolean AND. This conjunctive pattern means "find a User A linked to a User B, and find the Address connected to A, and find the Address connected to B." Line 6 filters out the case where A = B and prevents a pair (A, B) from being processed twice.

Line 9 filters out the case where A and B are different but have identical addresses. If their addresses are the same, then we already gave them full credit when we ran *connect_weighted_match*. Line 10 computes the weighted scoring, using the jaroWinklerDistance function and the weight for Address. Line 11 stores this value in a lookup table temporarily. Lines 12 and 13 are just to record our activity, for informative output at the end.

Merging Similar Entities

In Method 1, we had a simpler scheme for deciding whether to merge two entities: if their Jaccard score was greater than some threshold, then we created a SameAs edge. The decision was made. Merge everything that has a SameAs edge. We want a more nuanced approach now. Our scoring has adjustable weights, and the SameAs edges record our scores. We can use another threshold score to decide which Users to merge.

Take another look at Figure 6-17. We can see the effect of threshold score for merging. If we set the threshold at 3.0, there would be only two merges in this community: (5, 105) and (505, 805). If we set it at 2.0, we will get the three communities that we spoke about earlier: (5, 105, 205), (305, 405), and (505, 805). If we set the threshold at 1.0, all seven Users will be merged into 1.

We only need to make two small changes to merge_connected_users to let the user set a threshold.

- Add a threshold parameter to the query header:

```
CREATE QUERY merge_connected_users(FLOAT threshold=1.0, BOOL ver
bose=FALSE)
```

- In the connected component algorithm, add a WHERE clause to check the SameAs edge's similarity value:

```
1   WHILE (Updated_users.size() > 0) DO
2     IF verbose THEN PRINT iteration, Updated_users.size(); END;
3     Updated_users = SELECT t
4       FROM Updated_users:s -(SameAs:e)- User:t
5       WHERE e.1  similarity > threshold
6         // Propagate the internal IDs from source to target vertex
7           // t gets the lesser of t & s ids
8       ACCUM t.@min_user_id += s.@min_user_id
9           // accum' is accum's previous val
10      HAVING t.@min_user_id != t.@min_user_id'
11    ;
12    iteration = iteration + 1;
13  END;
```

Run *merge_connected_users*. Pick a threshold value and see if you get the result that you expect.

Figure 6-18 shows the three different merging results for threshold values of 1.0, 2.0, and 3.0, to the community we have been following.

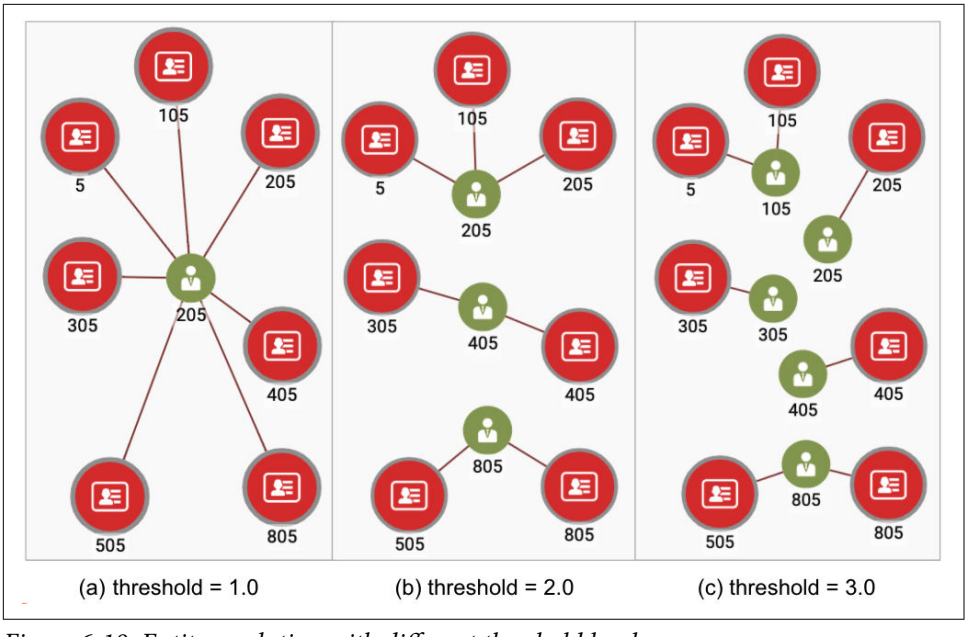


Figure 6-18. Entity resolution with different threshold levels

That concludes our second and more nuanced method of entity resolution.

For convenient reference, here is the sequence of queries we ran for entity resolution using weighted exact and approximate matching:

1. Run initialize_users
2. Run util_set_weights
3. Run connect_weighed_match
4. Run score_similar_attributes
5. Run merge_connected_users
6. Repeat steps 3, 4 and 5 until the output of merge_connected_users says conversed = TRUE

Chapter Summary

In this chapter we saw how graph algorithms and other graph techniques can be used for more sophisticated entity resolution than the simple entity resolution presented earlier in this book. Similarity algorithms and the connected component algorithm play key roles. We considered several schemes for assessing the similarity of two entities: Jaccard similarity, weighted sum of exact matches, and Jaro-Winkler similarity for comparing text strings.

These approaches can readily be extended to supervised learning if training data becomes available. There are a number of model parameters that can be learned, to improve the accuracy of the entity resolution: the scoring weights of each Attribute for exact matching, tuning the scoring of approximate matches, and thresholds for merging similar Users.

We saw how we use the FROM clause in GSQL queries to select data in a graph, by expressing a path or pattern. We also saw examples of the ACCUM clause and accumulators being used to compute and store information such as the common neighbors between vertices, a tally, an accumulating score, or even an evolving ID value, marking a vertex as a member of a particular community.

This chapter showed us how graph-based machine learning can improve the ability of enterprises to see the truth behind the data. In the next chapter, we'll apply graph machine learning to one of the most popular and important use cases: product recommendation.

About the Authors

Victor Lee is Head of Product Strategy and Developer Relations at TigerGraph. His Ph.D. dissertation was on graph-based similarity and ranking. Dr. Lee has co-authored book chapters on decision trees and dense subgraph discovery. Teaching and training have also been central to his career journey, with activities ranging from developing training materials for chip design to writing the first version of TigerGraph's technical documentation, from teaching 12 years as a full-time or part-time classroom instructor to presenting numerous webinars and in-person workshops.

Phuc Kien Nguyen is a data scientist at ABN Amro Bank in Amsterdam. For the past five years, he has helped develop solutions and machine learning models to combat financial crime. He holds an MSc degree in Information Architecture from Delft University of Technology. Next to his day-to-day job, he writes articles at Medium about data science and network analytics. He has a great passion for storytelling, especially through video games. In his spare time, he loves to play football and catch up with the latest development in technology.

Xinyu Chang is responsible for designing and supporting sophisticated graph database and analytics deployments around the globe. He excels at meeting customer needs with solutions that enable them to achieve better outcomes. Xinyu co-authored the GSQL query language with Dr. Alin Deutsch and others at TigerGraph. He has a Master's degree in Computer Science from Kent State University and a Bachelore's degree in Computer Science and Japanese from Dalian University of Foreign Languages.