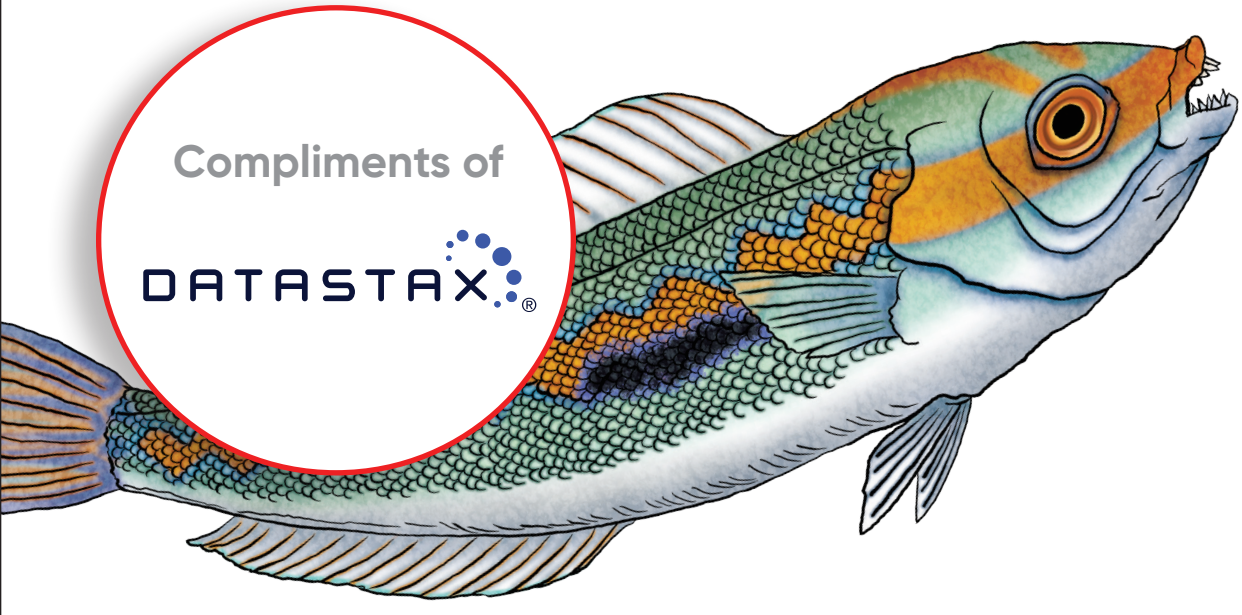
O'REILLY®

The Practitioner's Guide to Graph Data

Applying Graph Thinking and Graph Technologies to Solve Complex Problems



Denise Koessler Gosnell &
Matthias Broecheler

O'REILLY®

The Practitioner's Guide to Graph Data

Graph data closes the gap between the way humans and computers view the world. While computers rely on static rows and columns of data, people navigate and reason about life through relationships. This practical guide demonstrates how graph data brings these two approaches together. By working with concepts from graph theory, database schema, distributed systems, and data analysis, you'll arrive at a unique intersection known as graph thinking.

Authors Denise Koessler Gosnell and Matthias Broecheler show data engineers, data scientists, and data analysts how to solve complex problems with graph databases. You'll explore templates for building with graph technology, along with examples that demonstrate how teams think about graph data within an application.

- Build an example application architecture with relational and graph technologies
- Use graph technology to build a Customer 360 application, the most popular graph data pattern today
- Dive into hierarchical data and troubleshoot a new paradigm that comes from working with graph data
- Find paths in graph data and learn why your trust in different paths motivates and informs your preferences
- Use collaborative filtering to design a Netflix-inspired recommendation system

Denise Koessler Gosnell, PhD, is chief data officer of DataStax. She's built and patented dozens of processes related to graph theory, graph algorithms, and graph data applications.

Matthias Broecheler, PhD, is chief technologist at DataStax. A graph database expert, he invented the Titan graph database.

"A must-have addition to any coder's arsenal of references. Both authors are exemplars in the field of graph theory, architecture, and principles."

—**Theodore C. Tanner Jr.**
Global Chief Technology Officer and Chief Architect, Watson Health

"This book does a masterful job of leveling up your graph database knowledge. It's the perfect starting point for newcomers, and even the most seasoned practitioners will learn new things."

—**Matthew Russell**
Chief Executive Officer, Strongest AI, and author of *Mining the Social Web*

AI / SEMANTICS

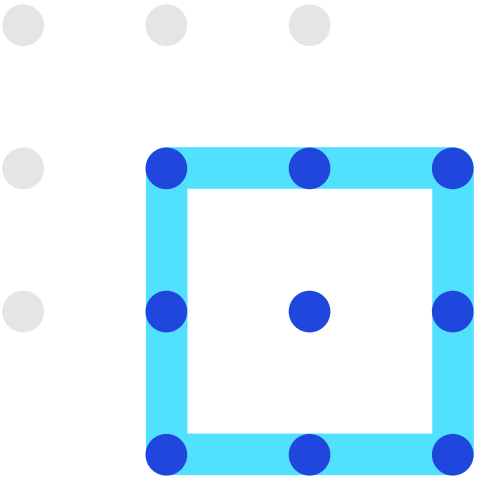
US $69.99 CAN $92.99

ISBN: 978-1-492-04407-9

9 781492 044079 56999

Twitter: @oreillymedia
facebook.com/oreilly

Your relational database isn't cutting it.



Discover the most powerful way to scale with NoSQL and Apache Cassandra™

Read More

"We (ICE Data Services) compile quotes from almost every market in the world in near real-time and create synthetic products leveraging DataStax Enterprise (DSE). Our ability to deliver key, reliable, real-time data products enables our customers to continuously calculate risk, accurately price assets, and power their mission critical financial platforms."

Steve Hirsch, Chief Data Officer, ICE and NYSE

ACI UNIVERSAL PAYMENTS · DELTA · ebay · INTUIT · macy's · NYSE

DATASTAX

#DataStax

The Practitioner's Guide to Graph Data

Denise Koessler Gosnell and Matthias Broecheler

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

The Practitioner's Guide to Graph Data
by Denise Koessler Gosnell and Matthias Broecheler

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: Jonathan Hassell
Developmental Editor: Jeff Bleiel
Production Editor: Nan Barber
Copyeditor: Arthur Johnson
Proofreader: Josh Olejarz

Indexer: Ellen Troutman-Zaig
Interior Designer: David Futato
Cover Designer: Karen Montgomery
Illustrator: Rebecca Demarest

April 2020: First Edition

Revision History for the First Edition
2020-03-26: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9781492044079* for release details.

978-1-492-04407-9

[LSI]

CHAPTER 3

Getting Started: A Simple Customer 360

We have often seen technical teams understand the benefits of graph thinking in the context of discussing a data problem that most large businesses face: trying to extract value across disparate data sources. Standing at a whiteboard sketching out the problem inevitably produces one hairy graph.

You can imagine this same scenario. You are drawing at a whiteboard and actively discussing how your system's data is spread in different silos across the company's systems. Your team agrees that what it really needs is direct access to your customers and their data. To illustrate this, almost every time, your coworker draws the customer at the center of the whiteboard and connects the relevant data to the customer. After stepping back, you all realize your colleague just drew a graph.

In our experience, these whiteboarding exercises illustrate the power of using graph thinking to build a data management solution. Graph applications start with data management because, either conceptually or physically, previous technology choices forced us to shape graph data into tabular solutions. The problem is that tabular-shaped data is no longer a one-size-fits-all design for today's applications.

This is especially true for those applications that have to cater to the user's demand for personalized context. The rising demand for personalization has put top-down pressure on data availability and relevancy. This pressure has forced organizations to integrate disparate data and ensure that data ties users to their digital experience.

When teams huddle around the drawing board to re-architect their systems to deliver personalization, they encounter a new problem. How does a single system unify data, function in real time, and relate the data back to the end user? Existing relational tools are great for those parts of the process that require the data to fit well in a row-and-column format.

However, relational tools are not well suited for delivering certain shapes of data—specifically, deeply connected data.

At this point in the whiteboarding session, we have reached a significant discussion topic: identifying and comparing solutions. The solution design process often introduces multiple technologies. The subsequent debate around which technology to choose can be divisive and never-ending.

Chapter Preview: Relational Versus Graph

To address this common pressure point, the main goals of this chapter are as follows:

1. Define and formalize a common starting application for graph data
2. Build out an example application architecture with relational and graph technologies
3. Provide a guide for making the right choice for your system's needs

Throughout the rest of this chapter, we are going to introduce and motivate the use case that we just described in the whiteboard story. Afterwards, we are going to step through the implementation details of this type of application, starting with a relational system. Then we will follow the same process with a graph system. We will close this chapter with a discussion of how to select which technology is best for your application. Getting this right will help you find roots in the seemingly circular and never-ending debate over when, where, and how to apply graph technology to resolve your data management needs.

The Foundational Use Case for Graph Data: C360

As we illustrated with the whiteboard story, tech teams all over the world are realizing the utility of graph data to solve their data management problems. For this type of problem, the difference between old and new solutions lies in the usefulness of modeling, storing, and retrieving relationships within your data.

Applications that aim to focus on the relationships in their data have the initial challenge of transforming and unifying data across relational systems. This transformation requires us to reorganize our thinking and processes from organizing entities to organizing relationships. Similar to the whiteboard drawing from earlier, the new approach to organizing your data according to its relationships is usually very close to what we have in Figure 3-1.

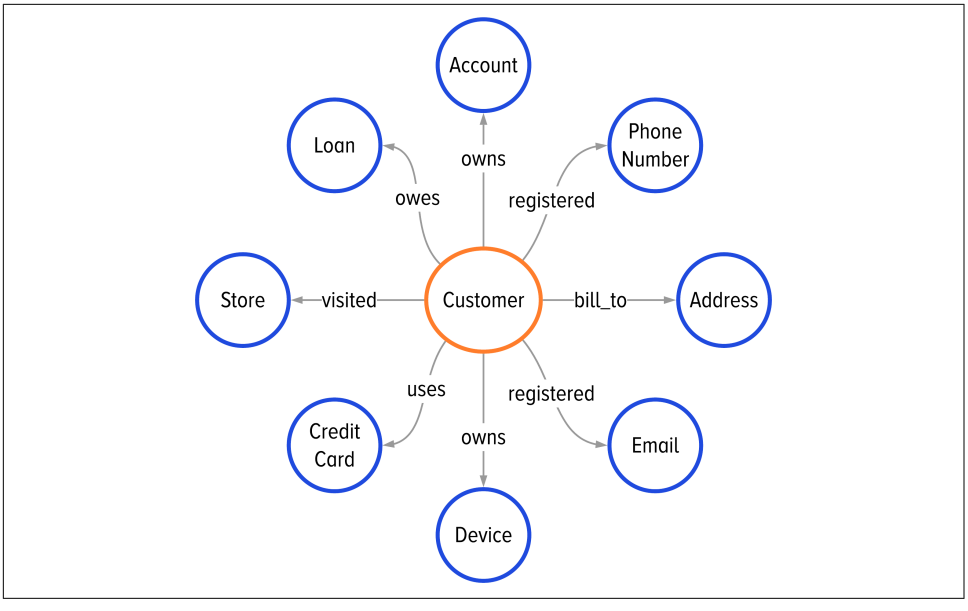


Figure 3-1. An exercise in graph thinking to yield a conceptual graph model

Adopters of graph technologies independently converged on naming this type of solution a *Customer 360* application, commonly abbreviated C360. The vision of a C360 project, like what is illustrated in Figure 3-1, is to engineer an application around the relationships between the important entities in your business.

You can envision the goal of a C360 application; there is a central object, your customer, and the customer's relationships to other integral pieces of data. These pieces of data are likely those that are most relevant to your business's domain. Commonly, we see teams start with the customer's family, methods of payment, or important identification details. This particular application within financial services is designed to answer the following types of questions about your customer:

1. Which credit cards does this customer use?
2. Which accounts does this customer own?
3. Which loans does this customer owe?
4. What do we know about this customer?

The idea to unify consumer data into a single application is not new. Existing solutions such as data warehouses or data lakes provide single systems in which consumer data is stored. The problem here isn't in the integration of a business's data but in its accessibility. The era of graph thinking made us revisit these solutions in search

of a way to make this data more available and representative of the individual's experience.

Think of it this way: would you rather spend the day fishing, or would you like to get quick access to your dinner?

The difference between fishing for or ordering your dinner is similar to putting your data in a data lake or organizing your data for quick retrieval. The demands of today's digital applications require architects to focus on the quick delivery of data. Graph technologies allow architects to build deeply connected retrieval systems to complement longer search expeditions across data lakes.

Why Do Businesses Care About C360?

Consumers interact with your company in an omnichannel fashion; they seamlessly transition from your mobile or web applications to your social media feeds and physical storefronts. Across all of these channels, they are experiencing an integrated perception of your brand. Companies that match this expectation by creating a unified digital experience are seeing revenue increases of up to 10%. These revenue increases are measured to be two to three times faster than those of companies that have not unified their customers' digital experiences.[1]

The secret ingredient behind this observed revenue growth is an application that unifies all customer data. Bringing together all of your customers' data into an application mirrors each customer's experience with your brand. In other words, it is a C360 application.

There are myriad creative examples of early innovators who have deployed interesting C360 applications. One of the more unique examples comes from Baidu (the Google of China) and Kentucky Fried Chicken. Through a unified data platform, Baidu teamed up with KFC to deliver order recommendations. Their collaborative solution identifies customers, accesses their order history, and returns order recommendations. This integration of data across these two industries has proven to be a unique and profitable example of C360 technologies.

A C360 application is the starting place for implementing graph thinking in your business. Getting this right provides a solid foundation for introducing graph data into your system's architecture. We have found that one of the most common mistakes made by architects and system designers is to move too quickly from the conceptual model to implementation details with graph technologies. There is more to consider here, and we want to use our experience to guide you through your own evaluation throughout the rest of this chapter.

1 Mark Abraham, et al. "Profiting from Personalization." Boston Consulting Group, May 8, 2017. *https://www.bcg.com/publications/2017/retail-marketing-sales-profiting-personalization.aspx*.

Implementing a C360 Application in a Relational System

The goal of this section is to briefly introduce how to build out a relational system to store the C360 data. This section does not serve as a complete introduction to this class of system architecture. Rather, our goal is to introduce the minimum needed to understand the complexities of using a relational system for a C360 application.

To illustrate the process from data modeling through queries, we will be using the same data introduced in Table 2-1. For convenience, we are sharing the table of data again in this chapter—see Table 3-1. For a complete refresher on the generation, meaning, and details of this data, refer to the discussion in "Data for Our Running Example" on page 23.

Table 3-1. A sample of the data created to illustrate the technology choices in this chapter

customer_id	name	acct_id	loan_id	cc_num
customer_0	Michael	acct_14	loan_32	cc_17
customer_1	Maria	acct_14	none	none
customer_2	Rashika	acct_5	none	cc_32
customer_3	Jamie	acct_0	loan_18	none
customer_4	Aaliyah	acct_0	[loan_18, loan_80]	none

The two technologies we will be using to illustrate a relational implementation are SQL and Postgres. SQL, which stands for Structured Query Language, is the programming language used to communicate with a relational database. We have chosen to use the Postgres RDBMS because of its wide applicability and origins within the open source community.

Data Models

After agreeing on a conceptual model, like that shown in Figure 2-1, you can move on to the design of your relational database. Traditionally, you would create an *entity-relationship diagram*, or ERD. An ERD is a logical representation of your data model and is a typical starting point for a relational database design.

In Figure 3-2, each square represents an entity that will become a table in the relational database. The *attributes*, or descriptive properties about each entity, are listed within each square. As already seen in the data, each entity will have a unique identifier. A customer will be uniquely identified by its `customer_id`, an account by its `acct_id`, and so on. Customers also have names and, in larger applications, other attributes.

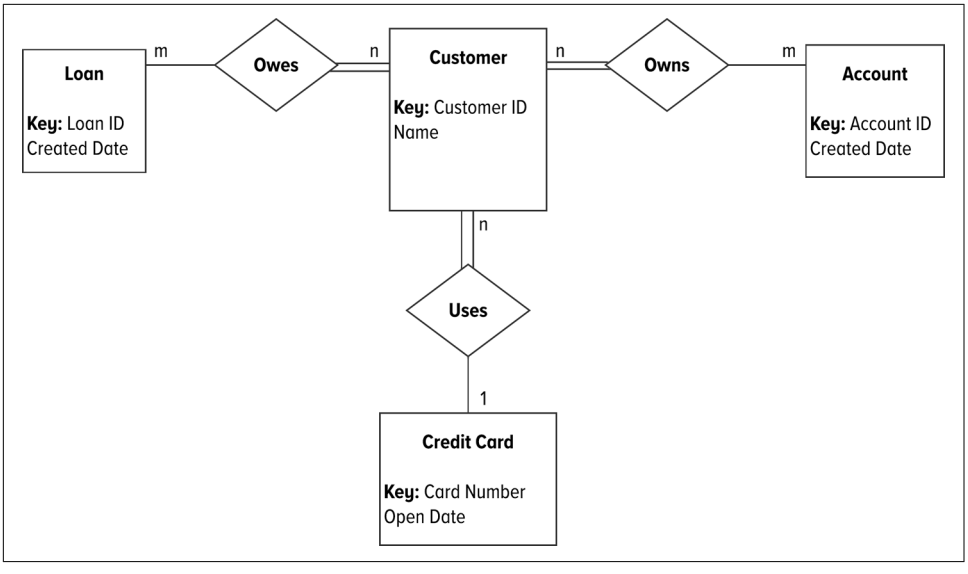


Figure 3-2. An entity-relationship diagram for a relational implementation of this C360 application

The diamond shapes between entities in Figure 3-2 represent the connection from one entity to another. The *cardinality* of the connection is indicated above and below or to the left and right of the diamond shape. In this data, we have two types of connections: one-to-many and many-to-many.

Let's start with the *one-to-many* connection that is shown between customers and credit cards. In this example, a customer can have many credit cards, but a credit card can only have one customer. This one-to-many connection describes the *cardinality* between customers and credit cards and is illustrated with the `n to 1` connection between customers and credit cards in Figure 3-2.

The other type of connection we see in our data is a *many-to-many* connection. There are two many-to-many connections in our data: customers to accounts and customers to loans. We know from our data that a customer can have many accounts, and one account can have many customers. The same is true for loans. We say that customers to loans is a many-to-many connection and illustrate this in Figure 3-2 with the `n to m` notation on the connection.

Before creating tables and inserting data, we need to translate our logical data model into a physical data model. Specifically, we need to translate the entities and connections from the ERD illustrated in Figure 3-2 into tables with primary and foreign keys.

We need two types of keys for this implementation: primary and foreign keys. A *primary key* is a uniquely identifying piece of data, such as a customer's ID or credit card number, that we will use to access the information in its table. A *foreign key* is a uniquely identifying piece of data that we will use to access the information in a different table, such as storing a customer's ID alongside their credit card information. We store a customer's ID with the customer's credit card information so that we can use it to look up all of their information in a different table, namely in the customer table.

Let's take a look at how the keys and data from Figure 3-2 map into the physical data model shown in Figure 3-3.

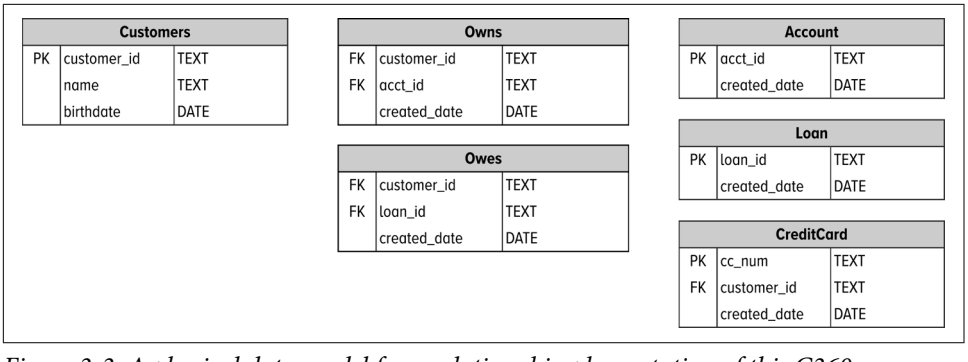


Figure 3-3. A physical data model for a relational implementation of this C360 application

We expected to see at least four tables in Figure 3-3—one table for each entity. Specifically, Figure 3-3 has one table per entity type: customer, account, loan, and credit card. For each of those tables, we see additional attributes that describe the entity. The most important attribute for each of these entities is its primary key. Each primary key is indicated with a `PK` next to the row. The primary keys we have for each table are the `customer_id`, `acct_id`, `loan_id`, and `cc_num`, respectively. These are the unique identifiers that we will use to look up a specific row of information in the table.

Before we talk about the other two tables in Figure 3-3, let's examine the `CreditCard` table. This table has both a primary key and a foreign key. We are using a foreign key in this table to track the one-to-many relationship we created in our ERD. The `customer_id` is the foreign key (indicated with an `FK`) that will give us the ability to relate the credit card information back to a unique customer. Building a one-to-many relationship into your physical data model can be as easy as adding a foreign key to join you back to another entity table.

The last two tables to understand in our physical data model are the `Owns` and `Owes` tables. These tables are *join tables* that allow us to physically store the many-to-many connections in our data. The `Owns` table stores the many connections observed

between customers and the accounts that they own. The `Owes` table stores the many connections observed between customers and the loans that they owe. Since each customer can own an account only once and can owe a loan only once, the primary key of these join tables is a compound of the two foreign keys.

For example, the `Owns` table stores at least two pieces of information about each row in the table: the customer's unique identifier and the account's unique identifier. Given one row from this table, we can access both the customer's unique identifier to join back to the customer table and the account's unique identifier to join back to the account table. This join table is a common way to represent a many-to-many connection in a relational system.

Relational Implementation

Given our physical data model, let's walk through creating the tables and inserting our sample data from Table 3-1 into the tables.

First, we want to create the customer table. The final data model for this is shown in Figure 3-4.

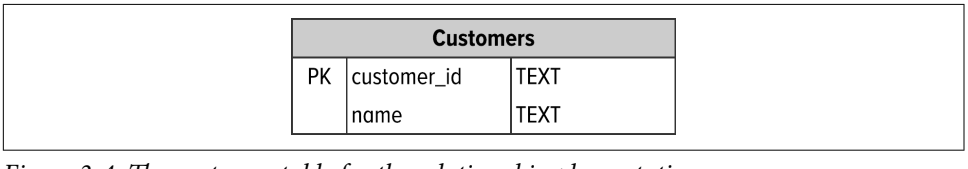


Figure 3-4. The customer table for the relational implementation

The SQL statement for creating the customer table is:

```
CREATE TABLE Customers ( customer_id TEXT,
                         name TEXT,
                         PRIMARY KEY (customer_id));
```

Our data has five customers. Let's insert those five customers into this customer table:

```
INSERT INTO Customers (customer_id, name) VALUES
  ('customer_0', 'Michael'),
  ('customer_1', 'Maria'),
  ('customer_2', 'Rashika'),
  ('customer_3', 'Jamie'),
  ('customer_4', 'Aaliyah');
```

Our data in our relational database now has one table with five entries, as shown in Figure 3-5.

Customers	
customer_0	Michael
customer_1	Maria
customer_2	Rashika
customer_3	Jamie
customer_4	Aaliyah

Figure 3-5. The customer data within our relational database

Next, let's add the other three entity tables for `Accounts`, `Loans`, and `CreditCards`. Their final data models are shown in Figure 3-6.

Account		
PK	acct_id	TEXT
	created_date	DATE

Loan		
PK	loan_id	TEXT
	created_date	DATE

CreditCard		
PK	cc_num	TEXT
FK	customer_id	TEXT
	created_date	DATE

Figure 3-6. The account, loan, and credit card tables for the relational implementation

We will start by creating the two tables for `Accounts` and `Loans`:

```
CREATE TABLE Accounts ( acct_id TEXT,
                        created_date DATE DEFAULT CURRENT_DATE,
                        PRIMARY KEY (acct_id));

CREATE TABLE Loans ( loan_id TEXT,
                     created_date DATE DEFAULT CURRENT_DATE,
                     PRIMARY KEY (loan_id));
```

Next, let's insert the data for `Accounts` and `Loans`:

```
INSERT INTO Accounts (acct_id) VALUES
  ('acct_0'),
  ('acct_5'),
  ('acct_14');

INSERT INTO Loans (loan_id) VALUES
  ('loan_18'),
  ('loan_32'),
  ('loan_80');
```

At this point, we have one last entity table to create in our relational database: the table for `CreditCards`. Because credit cards have a one-to-many relationship with customers, we also need to insert the customer's ID as a foreign key. We create this table with:

```
CREATE TABLE CreditCards
    ( cc_num TEXT,
      customer_id TEXT NOT NULL,
      created_date DATE DEFAULT CURRENT_DATE,
      PRIMARY KEY (cc_num),
      FOREIGN KEY (customer_id) REFERENCES Customers(customer_id));
```

Looking back at the data from Table 3-1, we find each unique credit card and the customer who owns that card. From this information, we can create the following statements to insert this data into our relational database:

```
INSERT INTO CreditCards (cc_num, customer_id) VALUES
  ('cc_17', 'customer_0'),
  ('cc_32', 'customer_2');
```

Our relational database now has a total of four tables with data; see Figure 3-7.

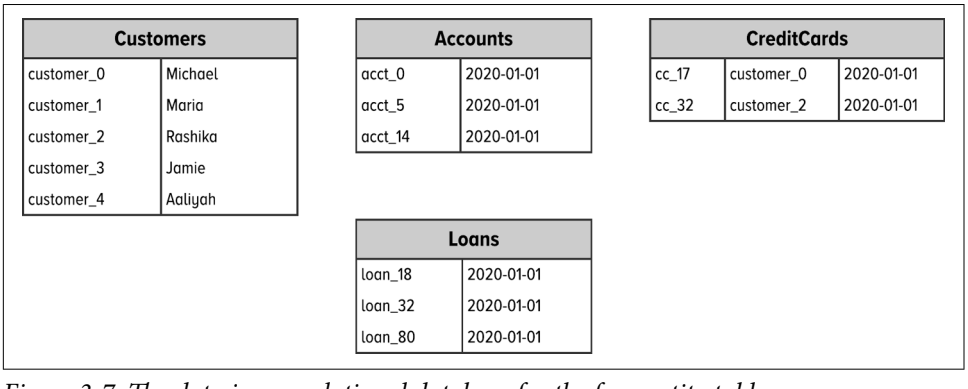


Figure 3-7. The data in our relational database for the four entity tables

The last two tables to create in our relational implementation are those for the many-to-many connections from `Customers` to `Accounts` and `Loans`. First, let's create the table that will join Customers to Accounts, as illustrated in Figure 3-8.

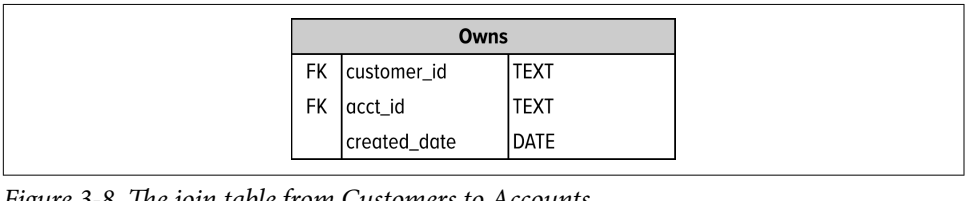


Figure 3-8. The join table from Customers to Accounts

In SQL, we create this table with:

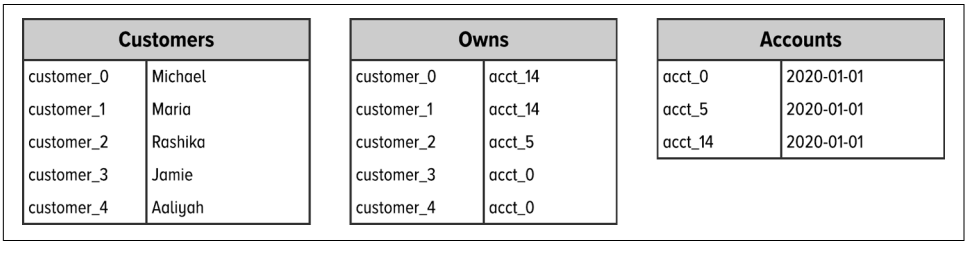
```
CREATE TABLE Owns ( customer_id TEXT NOT NULL,
                    acct_id TEXT NOT NULL,
                    created_date DATE DEFAULT CURRENT_DATE,
                    PRIMARY KEY (customer_id, acct_id),
```

```
                    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),
                    FOREIGN KEY (acct_id) REFERENCES Accounts(acct_id));
```

Remembering the data from Table 3-1, we find the following data to insert into the `Owns` table:

```
INSERT INTO Owns (customer_id, acct_id) VALUES
  ('customer_0', 'acct_14'),
  ('customer_1', 'acct_14'),
  ('customer_2', 'acct_5'),
  ('customer_3', 'acct_0'),
  ('customer_4', 'acct_0');
```

Now that we have some data in our `Owns` table (see Figure 3-9), we can see how to associate the data from a customer and an account in our data.



Customers

customer_0	Michael
customer_1	Maria
customer_2	Rashika
customer_3	Jamie
customer_4	Aaliyah

Owns

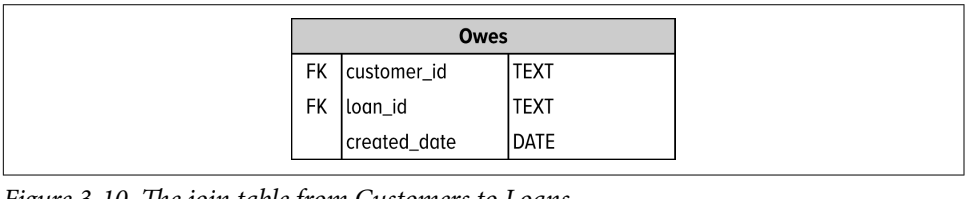
customer_0	acct_14
customer_1	acct_14
customer_2	acct_5
customer_3	acct_0
customer_4	acct_0

Accounts

acct_0	2020-01-01
acct_5	2020-01-01
acct_14	2020-01-01

Figure 3-9. The customer, account, and join table data

The final step in creating our relational database is to create the `Owes` table to associate a customer to their loan, and vice versa. The final data model for this join table is shown in Figure 3-10.



Owes

FK	customer_id	TEXT
FK	loan_id	TEXT
	created_date	DATE

Figure 3-10. The join table from Customers to Loans

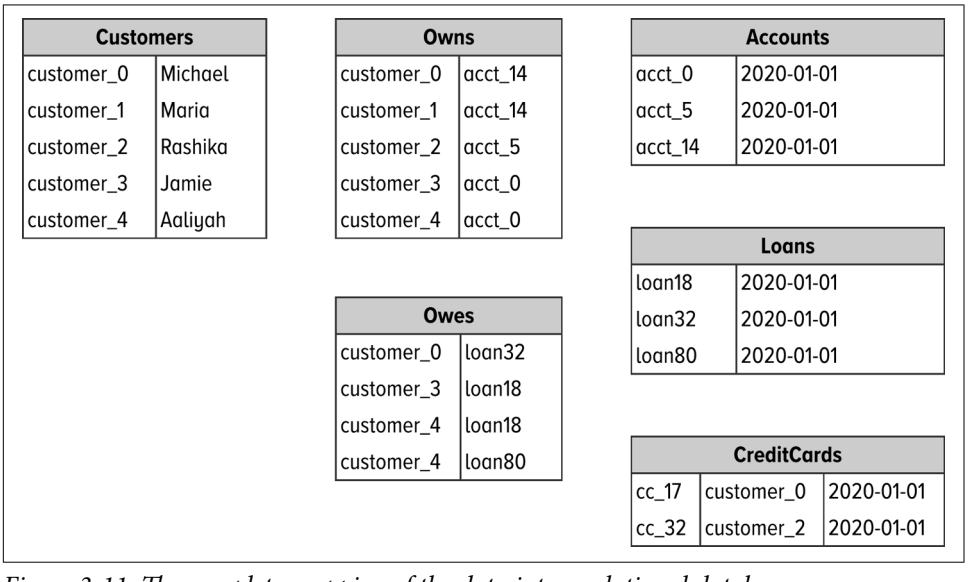
In SQL, we create this table in our relational database via:

```
CREATE TABLE Owes ( customer_id TEXT NOT NULL,
                    loan_id TEXT NOT NULL,
                    created_date DATE DEFAULT CURRENT_DATE,
                    PRIMARY KEY (customer_id, loan_id),
                    FOREIGN KEY (customer_id) REFERENCES Customers(customer_id),
                    FOREIGN KEY (loan_id) REFERENCES Loans(loan_id));
```

Finally, we can extract one last connection from the data in Table 3-1 and insert all observations of customers who owe loans into the `Owes` table:

```
INSERT INTO Owes (customer_id, loan_id) VALUES
  ('customer_0', 'loan_32'),
  ('customer_3', 'loan_18'),
  ('customer_4', 'loan_18'),
  ('customer_4', 'loan_80');
```

The full picture of our data in our relational database is shown in Figure 3-11.



Customers

customer_0	Michael
customer_1	Maria
customer_2	Rashika
customer_3	Jamie
customer_4	Aaliyah

Owns

customer_0	acct_14
customer_1	acct_14
customer_2	acct_5
customer_3	acct_0
customer_4	acct_0

Owes

customer_0	loan32
customer_3	loan18
customer_4	loan18
customer_4	loan80

Accounts

acct_0	2020-01-01
acct_5	2020-01-01
acct_14	2020-01-01

Loans

loan18	2020-01-01
loan32	2020-01-01
loan80	2020-01-01

CreditCards

cc_17	customer_0	2020-01-01
cc_32	customer_2	2020-01-01

Figure 3-11. The complete mapping of the data into a relational database

Example C360 Queries

Now that the data is in our relational database, we need to ask the four fundamental queries for our C360 application:

1. Which credit cards does this customer use?
2. Which accounts does this customer own?
3. Which loans does this customer owe?
4. What do we know about this customer?

For our relational system, we are asking these four questions in a specific order for two reasons. First, we want to start slowly with a natural progression toward asking more detailed questions about a person from a database. Second, we structured these questions so that the technical implementation builds upon each statement to conclude with the final SQL statement.

Query: Which credit cards does this customer use?

First, let's use our relational database to query for the credit cards owned by `customer_0`. The data for this query is directly available from the `CreditCards` table. If we just want the credit card information, we can query the table with the following SQL query:

```
SELECT * from CreditCards WHERE customer_id = 'customer_0';
```

This query will return the following data:

cc_num	customer_id	created_date
cc_17	customer_0	2020-01-01

It is likely that you really wanted to view the customer's data alongside their credit card information. This requires us to join the `Customers` table with the `CreditCards` table. In SQL, this would be done via:

```
SELECT Customers.customer_id,
       Customers.name,
       CreditCards.cc_num,
       CreditCards.created_date
FROM Customers
LEFT JOIN CreditCards ON (Customers.customer_id = CreditCards.customer_id)
WHERE Customers.customer_id = 'customer_0';
```

This query will return the following data:

Customers.customer_id	Customers.name	CreditCards.cc_num	CreditCards.created_date
customer_0	Michael	cc_17	2020-01-01

Getting access to a customer alongside their credit card information requires only one join statement because of the one-to-many relationship between customers and credit cards. When we need to look at the data about customers and their accounts, things get a little tricky.

Query: Which accounts does this customer own?

Next, let's query the relational database to answer the question: which accounts does `customer_0` own? For this query, we will need to use the join table `Owns` to join together the customer table with the account table. The SQL query for this is:

```
SELECT Customers.customer_id,
       Customers.name,
       Accounts.acct_id,
       Accounts.created_date
FROM Customers
LEFT JOIN Owns ON (Customers.customer_id = Owns.customer_id)
```

```
LEFT JOIN Accounts ON (Accounts.acct_id = Owns.acct_id)
WHERE Customers.customer_id = 'customer_0';
```

This query starts by accessing the data about `customer_0` from the customer table. Next, we find all foreign key pairs from the `Owns` table that have a matching `customer_id`. There is only one entry in the `Owns` table for this customer because `customer_0` owns only one account. From here, we follow the foreign keys of the accounts over to the `Accounts` table to extract the account information. The resulting data looks like:

Customers.customer_id	Customers.name	Accounts.acct_id	Accounts.created_date
customer_0	Michael	acct_14	2020-01-01

Query: Which loans does this customer owe?

The next question uses the same structure but traces from the customer table to the `Loans` table by using the `Owes` join table. This question is asking for the customer's information alongside their loan details. For this query, let's use the data about `customer_4`. The SQL statement for this query is:

```
SELECT Customers.customer_id,
       Customers.name,
       Loans.loan_id,
       Loans.created_date
FROM Customers
LEFT JOIN Owes ON (Customers.customer_id = Owes.customer_id)
LEFT JOIN Loans ON (Loans.loan_id = Owes.loan_id)
WHERE Customers.customer_id = 'customer_4';
```

The resulting data is:

Customers.customer_id	Customers.name	Loans.loan_id	Loans.loan_id
customer_4	Aaliyah	loan_18	2020-01-01
customer_4	Aaliyah	loan_80	2020-01-01

Query: What do we know about this customer?

Each of these queries is building up the individual pieces required to ask the main query for a C360 application: for a specific customer, tell me everything we know about them. This query brings together each of the three previous queries into one statement. The following SQL statement uses all six tables across our relational database to find all information about one customer. Let's use `customer_0` again in this final example:

```
SELECT Customers.customer_id,
       Customers.name,
       Accounts.acct_id,
```

```
        Accounts.created_date,
        Loans.loan_id,
        Loans.created_date,
        CreditCards.cc_num,
        CreditCards.created_date
FROM Customers
LEFT JOIN Owns ON (Customers.customer_id = Owns.customer_id)
LEFT JOIN Accounts ON (Accounts.acct_id = Owns.acct_id)
LEFT JOIN Owes ON (Customers.customer_id = Owes.customer_id)
LEFT JOIN Loans ON (Loans.loan_id = Owes.loan_id)
LEFT JOIN CreditCards ON (Customers.customer_id = CreditCards.customer_id)
WHERE Customers.customer_id = 'customer_0';
```

This will transform the data about `customer_0` from across the database into the following:

customer_id	name	acct_id	created_date	loan_id	created_date	cc_num	created_date
customer_0	Michael	acct_14	2020-01-01	loan_32	2020-01-01	cc_17	2020-01-01

The four questions we demonstrated in this section are just scratching the surface of the SQL query language. And we addressed only the fundamentals of SQL: `SELECT-FROM-WHERE` with basic joins. Even though our questions can be stated very simply, the required queries become increasingly complex. It is even harder to follow the data throughout this system to understand which data is related to which customer.

Implementing a C360 Application in a Graph System

Now that we have walked through a relational implementation, let's dig into transforming our sample data into a graph database implementation. Let's revisit the conceptual model, shown in Figure 3-12, before we dig into the implementation details in this section.

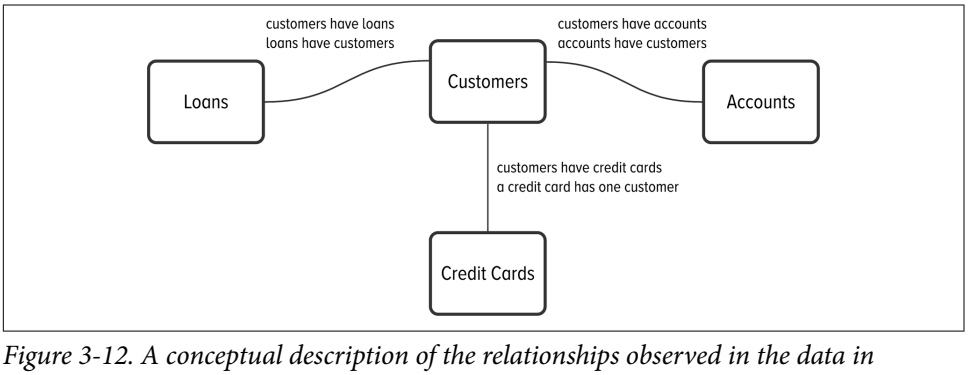


Figure 3-12. A conceptual description of the relationships observed in the data in Table 3-1

For this example, we are going to use the Gremlin query language—the most widely implemented graph query language— and DataStax Graph schema APIs. We are choosing to use Gremlin due to its wide adoption across the graph database community and its roots in open source. Our overarching objective in this book is to build up to implementing graphs within a distributed, partitioned environment. Given this goal, we will be using the DataStax Graph schema APIs to build up to working with distributed graphs.

Data Models

Compared to relational models, there is a smaller transition from a conceptual model to a graph data model. This lower bar illustrates the power of a database implementation that more closely represents your natural way of expressing data.

Using the GSL from "The Graph Schema Language" on page 33, Figure 3-13 contains a property graph model for our example data. The first benefit to notice is the shorter transition from conceptual (Figure 3-12) to logical data modeling for graph implementations.

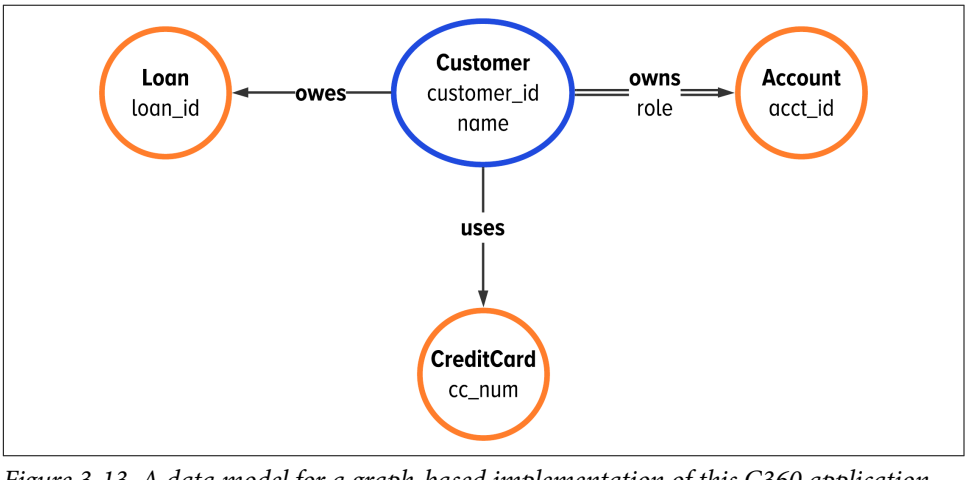


Figure 3-13. A data model for a graph-based implementation of this C360 application

There are four vertex labels in Figure 3-13: `Customer`, `Account`, `CreditCard`, and `Loan`. These vertex labels are shown in bold on the entities in the data model. There are three edge labels in Figure 3-13: `owns`, `uses`, and `owes`. These edge labels are shown in bold on the relationships in the data model.

Last, we find a few places in Figure 3-13 where we are using properties. A `Customer` vertex will have two properties: `customer_id` and `name`. You can see the properties for each vertex listed underneath the vertex labels. And we have also included a `role` on the `owns` edge label.

Graph Implementation

With graph databases, our first implementation step will be to create the graph so that we can add the graph's schema. Once we have set up the schema, we will be ready to insert data into the database.

The code for creating a graph is as follows:

```
system.graph("simple_c360").create()
```

We took care of installing and setting up the graphs for you in the provided technical assets. If you want to dig into those steps on your own, you can find the step-by-step instructions in the DataStax Docs. We will not be covering those topics in this book.

Let's dive straight into creating graph schema. If you would like, you can follow along in the DataStax Studio Notebook we created for this chapter, `Ch3_SimpleC360`. DataStax Studio gives you a notebook environment for developing with DataStax products and is the best way to implement this book's examples. The notebooks are available in our book's GitHub repository.

Creating your graph's schema

First, let's create the `customer` vertex label. Our customer data has a unique ID and a name:

```
schema.vertexLabel("Customer").
       ifNotExists().
       partitionBy("customer_id", Text).
       property("name", Text).
       create();
```

Let's finish the vertex label creation by adding the vertex labels for accounts, loans, and credit cards:

```
schema.vertexLabel("Account").
       ifNotExists().
       partitionBy("acct_id", Text).
       create();

schema.vertexLabel("Loan").
       ifNotExists().
       partitionBy("loan_id", Text).
       create();

schema.vertexLabel("CreditCard").
       ifNotExists().
       partitionBy("cc_num", Text).
       create();
```

At this point, we have four tables in the database—one table per vertex label. The last step is to add the relationships from the customer to each of the other entities in the data model.

For this example, we selected to model the edges coming out of the customer vertex and into the other vertex types. These edges have direction; it comes from the customer and goes to `Accounts`, `Loans`, and `CreditCards`. When we create an edge label, this direction matters. Let's look at an example for creating the `owes` relationship between a customer and their account:

```
schema.edgeLabel("owes").
       ifNotExists().
       from("Customer").
       to("Loan").
       create();
```

The direction of this edge label is set with the `from` and `to` steps. The edge comes from the vertex label `Customer` and goes to the `Loan` vertex label.

There are two more edge labels to create: one from the customer to their credit card and another from the customer to their account. The `owns` edges will also have the `role` property stored on the edge:

```
schema.edgeLabel("uses").
       ifNotExists().
       from("Customer").
       to("CreditCard").
       create();

schema.edgeLabel("owns").
       ifNotExists().
       from("Customer").
       to("Account").
       property("role", Text).
       create();
```

We read the label to say the edge `owns` is coming *from* the customer and going *to* the account and has a property called `role`.

Inserting your graph data

With our graph schema in place, we can add our sample data into this graph database. We will start by adding one piece of data—the vertex for `Michael`:

```
michael = g.addV("Customer").
            property("customer_id", "customer_0").
            property("name", "Michael").
            next();
```

When adding vertices into your graph, the `addV` step requires you to provide the full primary key. Otherwise, you will see an error like the one shown in Figure 3-14.

Figure 3-14. An example of an error you may experience if you forget to include the full primary key when inserting a new vertex

Next, let's add the vertices for Michael's account, loan, and credit card:

```
acct_14 = g.addV("Account").
          property("acct_id", "acct_14").
          next();

loan_32 = g.addV("Loan").
          property("loan_id", "loan_32").
          next();

cc_17 = g.addV("CreditCard").
        property("cc_num", "cc_17").
        next();
```

The `next()` step is a terminal step in Gremlin. It returns the first result from the end of a traversal. In the preceding example, we are returning the vertex object that we just added into the graph and storing it into an in-memory variable.

Now, we have four disconnected pieces of data in our graph database. As before, we stored each vertex object in variables called `acct_14`, `loan_32`, and `cc_17` to be used later. The database essentially has four vertices with no edges, as seen in Figure 3-15.

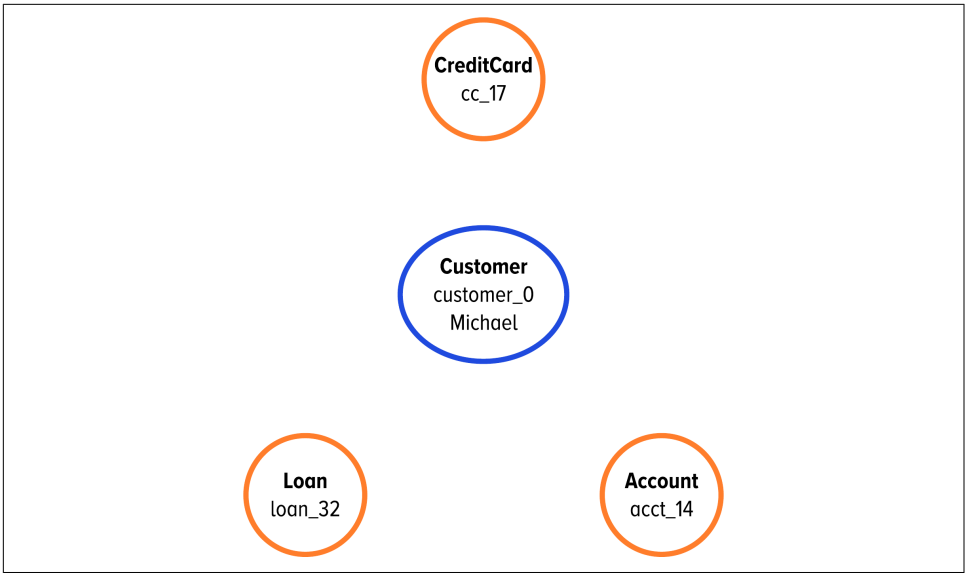


Figure 3-15. The data currently in our graph database

Let's introduce some connectivity between the data. To do so, we need to add three edges from `customer_0` to the other vertices. Using the variables we just created, we can add an edge from the vertex `Michael` to the vertices `account`, `loan`, and `credit Card`, respectively:

```
g.addE("owns").
  from(michael).
  to(acct_14).
  property("role", "primary").
  next();

g.addE("owes").
  from(michael).
  to(loan_32).
  next();

g.addE("uses").
  from(michael).
  to(cc_17).
  next();
```

When adding edges into the database, we start by identifying the vertex from which the edge will be coming. In the preceding example, this is `Michael` because all edges will be starting *from* `Michael` and going *to* other pieces of data. These three edges create the first connected view of our data in our graph database, as shown in Figure 3-16.

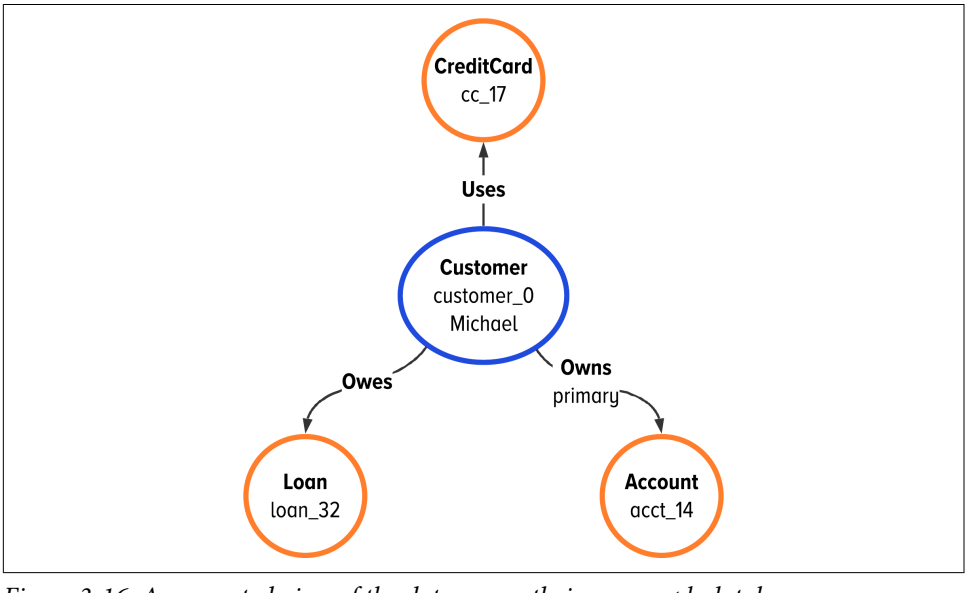


Figure 3-16. A connected view of the data currently in our graph database

From the example we already walked through, we know that `Maria` shares an account with `Michael`. Let's add the vertex for `Maria` and connect it to the account vertex we already created (see Figure 3-17):

```
maria = g.addV("Customer").
          property("customer_id", "customer_1").
          property("name", "Maria").
          next();

g.addE("owns").
  from(maria).
  to(acct_14).
  property("role", "limited").
  next();
```

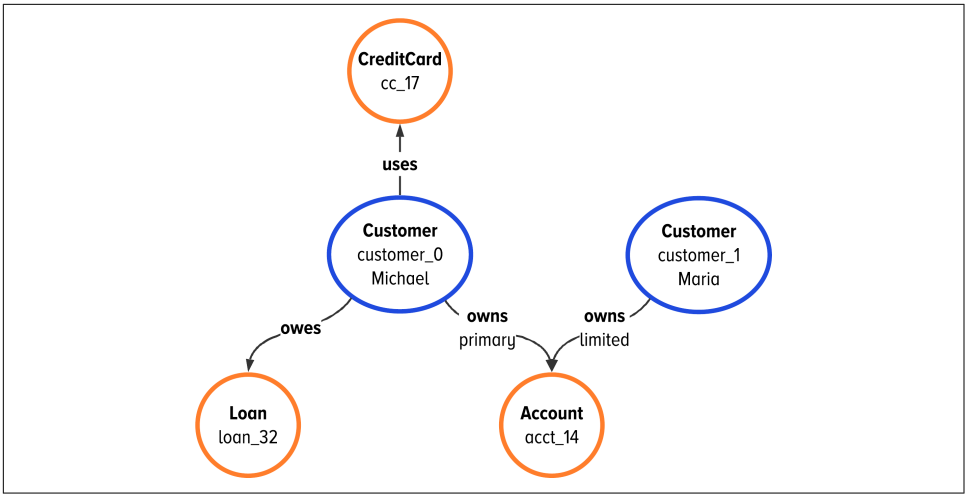


Figure 3-17. The connected view of Michael's and Maria's data in our graph database

Let's finish up this example by adding the vertices and edges about the remaining three customers:

```
// Data Insertion for Rashika
rashika = g.addV("Customer").
          property("customer_id", "customer_2").
          property("name", "Rashika").
          next();
acct_5 = g.addV("Account").
           property("acct_id", "acct_5").
           next();
cc_32 = g.addV("CreditCard").
          property("cc_num", "cc_32").
          next();
g.addE("owns").
  from(rashika).
  to(acct_5).
  property("role", "primary").
  next();
g.addE("uses").
  from(rashika).
  to(cc_32).
  next();

// Data Insertion for Jamie
jamie = g.addV("Customer").
          property("customer_id", "customer_3").
          property("name", "Jamie").
          next();
acct_0 = g.addV("Account").
           property("acct_id", "acct_0").
           next();
```

```
loan_18 = g.addV("Loan").
          property("loan_id", "loan_18").
          next();
g.addE("owns").
  from(jamie).
  to(acct_0).
  property("role", "primary").
  next();
g.addE("owes").
  from(jamie).
  to(loan_18).
  next();

// Data Insertion for Aaliyah
aaliyah = g.addV("Customer").
         property("customer_id", "customer_4").
         property("name", "Aaliyah").
         next();
loan_80 = g.addV("Loan").
          property("loan_id", "loan_80").
          next();
g.addE("owns").
  from(aaliyah).
  to(acct_0).
  property("role", "primary").
  next();
g.addE("owes").
  from(aaliyah).
  to(loan_80).
  next();
g.addE("owes").
  from(aaliyah).
  to(loan_18).
  next();
```

These final statements complete the insertion of the sample data into our graph database. Figure 3-18 shows the final view of the data in the database.

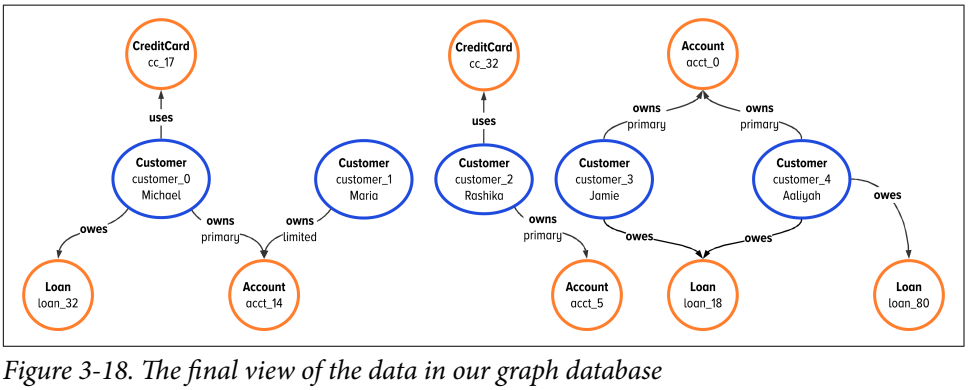


Figure 3-18. The final view of the data in our graph database

Graph traversals

The Gremlin statements in this section are our first graph database queries. A graph database query is also called a *graph traversal*.

Graph traversal
: A graph traversal is an iterative process of visiting the vertices and edges of a graph in a well-defined order.

When using Gremlin, you start your traversals with a *traversal source*.

Traversal source
: A traversal source wraps two concepts together: the graph data you are traversing and traversal strategies, such as exploring data without indexes. The traversal sources you will use for examples in this book are `dev` (for development) and `g` (for production).

The queries in this section used the `g` traversal source. We will come back to using the `g` traversal source in Chapter 5 and in the production chapters.

For the rest of this chapter, we will be using the `dev` traversal source. We will always use the `dev` traversal source in this book when we are developing our graph traversals, like in this chapter, in Chapter 4, and in the development chapters. We use the `dev` traversal source because it allows us to explore our graph data without indexes on the data.

From here, let's move on to implementing the same queries as before, but with our graph data.

Example C360 Queries

We like to think of querying a graph database, loosely, as the reverse of an SQL query. The common relational querying mindset is `SELECT-FROM-WHERE`. In graph, we are essentially asking the traversal to follow a similar pattern in reverse: `WHERE-JOIN-SELECT`.

You can think of a Gremlin query as beginning with `WHERE` you need to start from in your graph data. Then you tell the database to use relationships from your starting location to `JOIN` different pieces of data together. Last, you tell the database which data to `SELECT` and return. For a C360 application, our query loosely follows this `WHERE-JOIN-SELECT` pattern and is a great starting point for learning how to query a graph database.

With that in mind, let's revisit our C360 application queries, and then we will answer each question using the Gremlin query language and our graph database:

1. Which credit cards does this customer use?

2. Which accounts does this customer own?
3. Which loans does this customer owe?
4. What do we know about this customer?

Query: Which credit cards does this customer use?

First, let's use our graph database to query for the credit cards owned by `customer_0`. We can't just query for any credit card; we need to first access the vertex for `customer_0` and then walk to the credit card that is adjacent (connected) to `customer_0`. In Gremlin:

```
dev.V().has("Customer", "customer_id", "customer_0"). // WHERE
        out("uses").                                  // JOIN
        values("cc_num")                              // SELECT
```

The language to the right of `//` in each line of code is an in-line comment to describe the logic happening in the code at left.

This query will return the following data:

```
"cc_17"
```

Let's break down this Gremlin query into the `WHERE-JOIN-SELECT` pattern. The first part of the query is `dev.V().has("Customer", "customer_id", "customer_0")`. We say that this step is finding *where* you are starting your graph traversal; we are starting by finding a vertex with label `Customer` that has `customer_id` equal to `customer_0`. The second step in this traversal is `out("uses")`. This step is *joining* the customer to their credit card data. The last step is to pick the data you want to return. This is the `values("cc_num")` step. This part of the Gremlin traversal is specifying which data to *select* and return to the end user.

Whenever you see the word *traversal*, you can associate it with the idea of walking. To us, a graph traversal is a walk through your graph data. When we write graph traversals, we picture walking to and from pieces of graph data in our minds.

Let's go back to the graph query we just wrote to show you how we think of traversals as walking around graph data. In the first part of the graph query, we found a single vertex as our starting place: `customer_0`. From this customer, we needed to walk through the outgoing edge labeled `uses`. We walked through this edge using the `out()` step in Gremlin so that we could arrive at the credit card vertex. Once we were at the credit card vertex, we could look at the properties on the vertex. Specifically, we wanted access to Michael's credit card number: `cc_17`.

For the best performance, we advise that you always start your traversal from a specific vertex via the full primary key. For Apache Cassandra users, this is the same as providing the full primary key for a CQL query.

It helps to have a copy of Figure 3-13 to look at as you start practicing your first graph traversals. From an image on paper, you can see where you need to start and end. This is just like using a map for navigation, but in this case, you are walking around your data. With graph data, you can use your graph model to find your starting place, find your ending place, and translate the walk between them into your Gremlin statements. With enough practice, you will eventually be able to do all of this in your head.

Query: Which accounts does this customer own?

The next C360 query for our application wants to know which accounts a specific customer owns. Following the same pattern as before, we are going to access the vertex for `customer_0` and walk to the account vertex. From the account vertex, we can access the unique ID for the account:

```
dev.V().has("Customer", "customer_id", "customer_0").// WHERE
        out("owns").                                 // JOIN
        values("acct_id")                            // SELECT
```

As before, this query follows the `WHERE-JOIN-SELECT` pattern. The first part of this Gremlin query is similar to a *where* statement: `dev.V().has("Customer", "customer_id", "customer_0")`. We say that this step is finding *where* you are starting your graph traversal.

The second step in this traversal is like a *join* statement: `out("owns")`. This step is walking through the `owns` relationship coming out of the customer to *join* the customer to their data. The last step *selects* the data to return to the end user, specifically the account id: `values("acct_id")`. This query will return the following data:

```
"acct_14"
```

Let's try the same query again, but this time we would like to display the customer's name alongside their account ID. To do this, we need to remember the data we have visited as we walked through the graph. This introduces two new Gremlin steps: `as()` and `select()`. The `as()` step is similar to labeling the data as you walk through your graph, like leaving breadcrumbs behind as you walk around a maze.

Once we are done, we can recall the visited data with the other new step: `select()`. We use the `select()` step to return the data from the query:

```
dev.V().has("Customer", "customer_id", "customer_0"). // WHERE
          as("customer").                             // LABEL
        out("owns").                                  // JOIN
          as("account").                              // LABEL
        select("customer", "account").                // SELECT
          by(values("name")).                  // SELECT BY (for the customer)
          by(values("acct_id"))                // SELECT BY (for the account)
```

As before, this query follows the same `WHERE-JOIN-SELECT` pattern, with two additions. This query adds in the need to `SAVE` and `SELECT` specific data points from the query.

Let's walk through the steps in this query.

Once again, we start with *where* we need to go in our graph data, `dev.V().has("Customer", "customer_id", "customer_0")`. We want to remember this data for later, so we save the data with the step `as("customer")`. We continue to follow the pattern as before, *joining* the customer to their account data by walking through the `owns` edge. Now we have arrived at the account vertex. We want to save this vertex by using `as()`, like before. Last, we need to *select* multiple pieces of data. We do this with `select("customer", "account")`.

The remaining two steps that use `by` are important to call out. This step helps us shape the results of our query. After the `select("customer", "account")` step, we have two vertex objects: the customer and account vertices, respectively. Our original query wanted to access the customer's name and account ID. That is where the `by` step comes in. We want to view the customer according to their name and the account according to its ID. The `by` steps are applied in order to the vertex objects.

This query returns the following JSON:

```
{
  "customer": "Michael",
  "account": "acct_14"
}
```

Query: Which loans does this customer owe?

So far, we have seen three graph traversals and two different ways to select data from your graph. Next, let's explore the third query for our C360 application. This query wants to access the loans associated to a customer. Let's use `customer_4` for this example since she has multiple loans in our dataset. In this query we just want to look at the loan IDs:

```
dev.V().has("Customer", "customer_id", "customer_4"). // WHERE
        out("owes").                                  // JOIN
        values("loan_id")                             // SELECT
```

This query follows the same `WHERE-JOIN-SELECT` pattern that we saw in the previous section. This query will return the following data:

```
"loan_18",
"loan_80"
```

Query: What do we know about this customer?

The final query of a C360 application is to access an individual customer and all of their relevant data. The query for this will start at `customer_0` and walk through *all* outgoing edges that are connected to `customer_0`. Then we return the data from all vertices that are in this first neighborhood of `customer_0`. This query gives us all of the data about `customer_0`:

```
dev.V().has("Customer", "customer_id", "customer_0"). // WHERE
        out().                                        // JOIN
        elementMap()                                  // SELECT *
```

This query will return the data shown in Example 3-1.

Example 3-1.

```
{
  "id": "dseg:/CreditCard/cc_17",
  "label": "CreditCard",
  "cc_num": "cc_17"
},
{
  "id": "dseg:/Loan/loan_32",
  "label": "Loan",
  "loan_id": "loan_32"
},
{
  "id": "dseg:/Account/acct_14",
  "label": "Account",
  "acct_id": "acct_14"
}
```

Example 3-1 shows everything stored in DataStax Graph about each vertex: an internal `id`, the vertex's `label`, and then all properties. Let's inspect the JSON that describes Michael's credit card. First, there is an `"id": "dseg:/CreditCard/cc_17"`. This is the internal identifier used in DataStax Graph to describe that piece of data. The internal `id` in DataStax Graph is a URI, or Uniform Resource Identifier. Next, we see the vertex's label, `"label": "CreditCard"`. Last, we see the only property we

stored in the graph about credit cards: `"cc_num": "cc_17"`. We interpret the JSON about the loan and account vertices similarly.

These traversals are the base of what is required to extract the data in your C360 application. We recommend keeping a copy of your graph data model nearby when you are first starting to write graph traversals. Once you understand the basic steps, you can use an image of your data model to walk from your starting point to your destination. After some practice, this is an art that you may be able to visualize in your head as if you were the one walking around the data.

We constructed this example to show that graph applications can make data retrieval easier. As seen in this section, there were significantly fewer steps to the query, and they were easier to follow. The adjustment from relational to graph query languages requires an adjustment in your mentality to traversing or walking through your data. The learning curve is steep; we don't want to hide that. However, once you can picture yourself walking through your graph data, writing graph queries can be as simple as learning a new set of tools.

Relational Versus Graph: How to Choose?

Through the lens of a C360 application, let's consider the benefits and drawbacks of a relational database implementation versus those of a graph database implementation. In considering these two technologies, we are going to compare them in four areas. We are going to examine each technique's approach to data modeling, representing relationships, and query languages.

Relational Versus Graph: Data Modeling

There are quantitative and subjective items to consider when comparing the differences in data modeling for relational or graph databases. The quantitative arguments around data model design will point toward a relational system as the clear winner due to the higher volume of resources and production usage of relational systems. The techniques, tricks, and optimizations for a relational system are very well documented and accessible for all members and abilities within a development team.

On a more subjective note, data modeling techniques with graph technologies are significantly more intuitive. Specifically, when using graph technology, the human-to-computer translation of data is preserved; the way you think about your data is nearly the same as the way you would represent it digitally in a computer. This shorter translation from human intuition to machine representation allows you to extract deeper insights about the relationships in your data. This arguably makes graph technology easier to use over the system design required for the same implementation in a relational database.

Relational Versus Graph: Representing Relationships

There has been and continues to be a growing demand for modeling and storing relationships within a database. This has created both good and bad news for relational systems. The good news, as stated before, is that the tips, tricks, and techniques for modeling relationships with relational technology are well documented. Adding relationships to an existing relational database can be as simple as adding a join table or foreign key constraint. With the new join table or foreign key, relationships are queryable and accessible. Essentially, getting the data into the system is well documented and relatively easy for the developer.

On the downside, getting the relationships back out of a relational system in a meaningful way has a much steeper curve; it is very difficult to reason about the relationships stored in a relational system because of the large gap from idea to implementation to machine. The process from conversation to modeling to reasoning is much more disconnected with relational technology than it is with graph technology. The disconnect lies in the mental transformation required to map your human understanding of data into relational models and down to tables. This translation requires significant mental interpretation to follow and reason about relationships within the data stored in a relational database.

Graph technologies were created from this gap. If there is a need to model and reason about relationships in your data, graph technologies provide a more seamless transition from human understanding to machine representation of your data and back. The crux of this stance is whether or not relationships exist within your data and are useful for deeper analytics and reasoning. If you need to model and reason about relationships in your data, then graph technologies are the way to go.

Relational Versus Graph: Query Languages

There are three aspects we would like to examine when comparing query languages for the two systems: language complexity, query performance, and expressiveness.

First, let's talk about what we mean by language complexity. After designing relationships into your system, the query language introduces additional complexity to your evaluation of the database within your architecture. At this level, it is the query language that will highlight all of the complexities or simplifications that were made during the implementation process. The additional complexity is experienced as queries are developed and lengthened to pull together the required data.

Teams often measure query language complexity by query development time, maintainability, and ease of transferring knowledge. When you are considering SQL and Gremlin, these comparisons come down to adoption maturity and personal preference. SQL is the clear winner in language maturity. However, we see the scales tip toward Gremlin for deeply nested queries or those requiring a large number of joins.

The next evaluation of query languages measures query performance. Query performance measures a multifaceted and complex dependency of database-tuning exercises from indexing, partitioning, load balancing, and more optimizations than will fit in this book.

When we are considering the scope of a C360 application in a small deployment, it is likely that the queries against a properly indexed relational system will consistently outperform the same queries in a graph database. This is because the queries for the simplistic C360 application are very shallow graph queries; the queries stay within the first neighborhood of the customer. As graph queries get deeper, like what we will see in the next chapter, the performance debate between graph technology and relational technology heavily favors graph solutions.

The last comparison to make considers query language expressiveness. In our experience, the expressiveness of graph query languages solidifies the power of using graph data in an application. The difference in query complexity between the two systems illustrates that a more expressive language like Gremlin is a significant improvement for querying relationships in your system. Graph query languages on top of graph databases allow for a significant reduction in the code required to access and extract relationships from data. Only time will allow graph technologies to mature to the same levels as relational standards.

Relational Versus Graph: Main Points

For a loose summation, the points that we can make for each option, see Table 3-2.

Table 3-2. A summary of considerations when choosing a relational or graph database for a C360 application

	Relational	Graph Databases
Data modeling	Well documented	Digital representation matches human interpretation
Representing relationships in data	Known limitations and complexities	More intuitive representation
Queries	Well documented	Steep learning curve
	Difficulty when querying many relationships together	More expressive query language

For any area in which you can compare these two technologies, the advantages and disadvantages for either choice come down to maturity. The adoption, documentation, and community are much more evolved for relational technologies than for graph technologies. This maturity likely translates to lower risk and faster execution for traditional applications. Today, graph technologies cannot compete with relational in the categories of maturity and time to delivery for a new application.

On the other hand, relational technology is reaching its limits for delivering valuable insights into relationships within data. This is a significant problem because relationships naturally occur within data and are instrumental in delivering improved insights into your business. In this regard, graph technology is the better option for applications that require relationships to make business decisions. It is the best choice for delivering and reasoning on relationships within your data, which is not achievable at depth and scale with relational databases.

Summary

The power and vision of implementing a C360 application with graph technology is directly correlated to your business's need for accessing related data across your organization.

Let's unpack what we mean by that.

We have consulted with many enterprises that made specific technology choices over the past decade that in turn led to the construction of data silos. These data silos separated the data relevant to the core entities of their business, such as the customer. From there, recent approaches led to the integration of important data into large monolith systems, such as data lakes. The pain points here were not in the integration of the company's data but in its accessibility.

Who wants to spend time and resources fishing for valuable data in a data lake instead of using a system designed to retrieve valuable data?

For these enterprises, the advent of graph thinking has guided the next iteration of their data architecture. Their goal is to build with technologies that make their data available and representative of their customers' experience. This combination of availability and representation has been and continues to be the driving momentum behind graph technology.

Graph technologies are enabling the next iteration of enterprise data architectures in a way that was previously unachievable. We delved into one version of graph data management in this chapter. Namely, we explored the application and implementation details of a Customer 360 application, a customer-specific use case for graph technology. However, this same template for building data-centric applications with graph technology applies to non-customer-facing applications.

We have seen companies build similar systems around the businesses they interact with—kind of like a Business 360. The applications that organize and deliver all information about important interactions within their business are saving significant overhead in cross-departmental communication. For example, imagine all of the different departments you have to collaborate with in your company to find out the most recent interaction between your company and another vendor. The information for

this request is spread across finance, marketing, sales, customer relations, and likely other departments. The solution to this B2B problem requires the same template as we have described throughout this chapter.

Given the vision for this style of application, the next criteria to evaluate are the time and cost of implementation. These choices likely involve comparing vendors and existing tools, such as using relational or graph technology for your C360 application.

Why Not Relational?

We get this question all the time: "I can build a C360 application with an RDBMS, so why not use what I already know?"

The short version of our response to this question: relational is great for tabular data, graph is better for complex data. Otherwise, the two are remarkably similar. At its root, your choice comes down to the complexity of your data and what value you want to get out of it.

In the longer version, the key is how your business values time: time spent on engineering custom solutions and time spent on waiting for queries. The differences are quite clear when your business needs to answer deeper or unplanned queries. Relational systems require architectural changes, adding tables, and building your own query languages. Graph systems require augmenting your schema and inserting more data.

Essentially, graph technology makes it easier to work with complex data, whereas relational technology is easier for simple (i.e., tabular) data. The depth and complexity your project needs to expand into will help make this choice clearer for you.

Making a Technology Choice for Your C360 Application

The decision between relational and graph technology ultimately comes down to your C360 application's full scope. Generally speaking, our experiences have shown us that if your application aims only to unify disparate data sources, then you will achieve the best results from properly tuning a relational system. This realization and commitment to the sole function of your application will save development resources and more quickly get to final delivery for the production system.

On the other hand, if a data management solution or C360 application is a starting point for your data architecture, then the steep learning curve to graph databases will deliver more value in the long run. Graph technologies enable more intuitive reasoning about the relationships that exist across your data. The business objectives that require insight into relationships also require graph technology behind them.

Let us be clear on our points here. The example in this chapter is incredibly primitive. Anything more realistic and more elaborate starts to become a stretch for RDBMS.

And realistic data contains elaborate relationships within it. If your business needs access to these relationships, then you need graph technology.

If you are going to just build a simple C360 system and nothing more, use relational technology. If you want to understand and explore the connectedness within your data, use graph technology. There are pluses and minuses for each choice, but for the scenario we have set up in this chapter, graph technology is the winner.

Whichever data problem your business faces, be aware that those teams with the need to build *and extend* a foundation are turning to graph technology. A successful integration of graph technology into your architecture needs to start with a C360 application as its foundation and build from there. With a C360 application as a foundation, your business is set up to go after deeper graph traversals for more valuable insights from your data. In the next chapter, we are going to extend our simplistic C360 application to a more complete scenario that will highlight how graph technology and RDBMS diverge in terms of ease of use and time to market.

CHAPTER 4

Exploring Neighborhoods in Development

To get to the next phase in graph application development, we are going to build upon the simple Customer 360 (C360) application from Chapter 3. We'll add a few more layers, or neighborhoods, onto that example to illustrate the next wave of concepts in graph thinking.

Adding data to our example provides a more realistic picture of the complexity of data modeling, querying, and applying graph thinking to our customer-centric financial data.

We consider the transition from the basic example in Chapter 3 to the complexity in this chapter to be analogous to steps in the process of learning how to scuba dive. What we did in Chapter 3 was like starting to learn how to scuba dive in a wading pool; it is not really clear what the point is when you are in water that shallow. But we needed to start from a familiar place. The examples in this chapter are like scuba diving in a deep pool. Afterwards, we will be ready to head into more interesting depths in Chapter 5.

Chapter Preview: Building a More Realistic Customer 360

There are three main sections within this chapter.

In the first section, we will explore and explain graph thinking to present best practices in graph data modeling. We will do this by adding more neighborhoods of data to our C360 example so that we can answer the following questions:

1. What are the most recent 20 transactions involving Michael's account?
2. In December, at which vendors did Michael shop, and with what frequency?

3. Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan. (Query 3 is an example of personalization.)

Throughout this initial section, we will follow query-driven design to illustrate common best practices for creating a property graph data model. Topics include mapping your data to vertices or edges, modeling time, and common mistakes.

In the next section, we will build up deeper Gremlin queries. These queries walk through three, four, and five neighborhoods of data. We will also introduce how to use properties to slice, order, and range over graph data, and we will discuss querying in time windows. By the end of this section, we will have illustrated all of the data, technical concepts, and data modeling that we planned for our example.

We will end the chapter by revisiting the basic queries to introduce some more advanced querying techniques. These techniques are most commonly a part of trying to format your query results into a more user-friendly structure.

This content sets us up to present the final, production-quality schema for this example, which we will do in Chapter 5.

Graph Data Modeling 101

During the early days of working with graph databases backed by Apache Cassandra, my team was sitting around the couches in the living room of our venture-backed startup. We were whiteboarding a graph data model for storing healthcare data in a graph database.

We quickly agreed that doctors, patients, and hospitals were our primary entities of importance, and therefore they would be vertices. Everything else after that was a debate. Vertices, edges, properties, and names: everyone had a defensible opinion about everything. Our most memorable disagreements were polarizing. What should we name the edges between doctors and patients? All of these entities live or work somewhere; how do we model addresses? Is country a vertex or a property, or should it be left out of our model?

It was a difficult conversation. It took much longer than we had expected to arrive at a design consensus, and none of us really felt comfortable with it.

Since that design session, each time I advise a graph team around the world, I can feel similar tensions and see similar design consensus. The tensions are always real, always there, and always observable.

This section is all about helping your team have a more constructive discussion about your graph data model. To accomplish this, we want to walk through three sections of advice for creating a good graph data model. Those sections of advice will be:

1. Should This Be a Vertex or an Edge?
2. Lost yet? Walk Me Through Direction.
3. A Graph Has No Name: Common Mistakes in Naming

We selected these topics for two reasons. First, these topics cover most of the points of contention you will encounter during the modeling process. Second, these topics support where we are in the development of the running example for these chapters. Details for deeper and more advanced modeling advice will be introduced when we get there.

Should This Be a Vertex or an Edge?

This is the most debatable topic about property graph modeling. From the middle of the most heated debates, we've grabbed a number of tips for creating graph data models.

Let's start our tips at the beginning. In our world, the beginning is where you want to start your graph traversals.

Rule of Thumb #1

If you want to start your traversal on some piece of data, make that data a vertex.

To unpack our first tip, let's revisit one of the queries we constructed in Chapter 3:

```
Which accounts does this customer own?
```

There are three pieces of data required to answer that question: customers, accounts, and a connection from which customer owns an account. Think about how you could use that data to "find all accounts owned by Michael." There are two ways to translate this statement into a database query: "Michael owns accounts" or "accounts owned by Michael."

Let's talk about the first option: starting with Michael to find his accounts. This means that you are starting with data about people—specifically, the piece of data about Michael. In your head, when you find a starting place for a query, you would want to translate that data to being a vertex label in your graph model. With this, we have our first vertex label for our graph model: customers.

Consider the second way to find this information: you could first find all accounts and then keep only those that are owned by Michael. In this case, you are starting with the data about accounts. Now we have a second vertex label for our graph model: accounts.

This sets us up for the next tip on how to find the edges in your data.

Rule of Thumb #2

If you need the data to connect concepts, make that data an edge.

For the query we are working with, we know that Michael will be a vertex label and that his account is another vertex label. That leaves the concept of ownership, and yes, you guessed it—it will be the edge. The concept of ownership links a customer to an account for our example data.

To find the edges in your model, examine your data. You find your edges from within the information that links concepts together *and* to which you have access.

When working with graph data, these edges are the most important piece of your graph model. Edges are why you need graph technology in the first place.

Putting these two together, you can derive the following rule for labeled property graph models.

Rule of Thumb #3

Vertex-Edge-Vertex should read like a sentence or phrase from your queries.

Our advice here is to write out how you want to query with your data into short phrases like "customer owns account." Identifying these queries and phrases remains a simple way to identify how you want to map your data into graph objects in a property graph database, as shown in Figure 4-1.

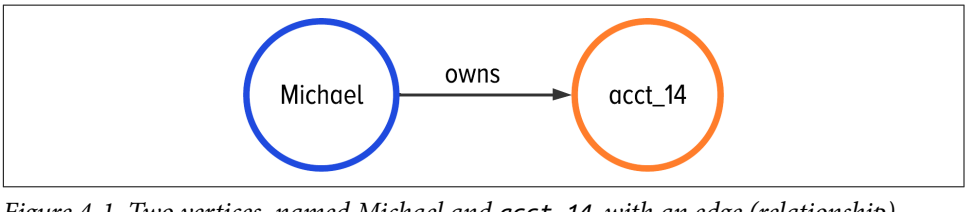


Figure 4-1. Two vertices, named Michael and `acct_14`, with an edge (relationship) titled, `owns`; this illustrates an example of translating short phrases of noun-verb-noun to a property graph model: Michael owns account 14

Generally speaking, written forms of your graph queries will translate verbs to edges and nouns to vertices.

This isn't the first time the graph community has worked with semantic phrases and graph data. Those of you from the semantic community are likely shouting, "We've seen this before!" And you are right; we have.[1]

Putting recommendations #2 and #3 together yields a specific way to translate how you think into graph objects.

Rule of Thumb #4

Nouns and concepts should be vertex labels. Verbs should be edge labels.

Depending on how you think, there are times at which tip #3 and tip #4 can create ambiguous scenarios. We want to delve into some semantics here to help you navigate different ways that people see and think about data.

Specifically, if you think "Michael owns an account," then "owns" should be an edge label. This is a case in which you are thinking actively about the relationship between Michael and his account. And this active line of thought translates `owns` to a verb that connects two pieces of data together. This is how we arrive at "owns" as an edge label.

However, there are cases in which you may see this same scenario differently. Namely, if you are thinking "We need to represent the concept of ownership between Michael and his account," then ownership should be a vertex label. In that case, you are thinking of ownership as a noun—that is, an entity. The difference is that in this case, it is likely that the ownership needs to be identifiable. You probably are trying to relate ownership in other ways. In these cases, other questions you may plan on asking are, "Who established that ownership?" or "Who does the ownership transfer to if the primary agent dies?"

We acknowledge that we are getting into the weeds here. But we know that you will eventually find yourself in the weeds as well. We hope that the guidance we are providing will help you find your way back up and out.

Our first four tips introduced the fundamentals for identifying vertices and edges in your graph data. Let's walk through how to reason about the direction of your edge labels.

1 Ora Lassila and Ralph R. Swick, "Resource Description Framework (RDF) Model and Syntax Specification," 1999. *https://oreil.ly/zWcnO*

Lost Yet? Let Us Walk You Through Direction

The questions and queries for this chapter integrate more data into our model. Specifically, we want to add transactions into our data so that we can answer questions like:

```
What are the most recent 20 transactions involving Michael's account?
```

To answer this query, we need to add transactions into our data model. And these transactions need to give us a way to model and reason about how transactions withdraw and deposit money between the accounts, loans, and credit cards.

When you first start writing graph queries and iterating on data models, it is very easy to get turned around in your data model. Direction of an edge label is a difficult thing to reason around, which is why we make the following recommendation.

Rule of Thumb #5

When in development, let the direction of your edges reflect how you would think about the data in your domain.

Tip #5 infers the direction of an edge label as you combine and apply the advice from the previous four tips. At this point, the pattern of Vertex-Edge-Vertex should be easily read as subject-verb-object sentences.

Therefore, the edge label's direction comes from subject and goes to object.

Coming up with edge labels between transactions is a discussion we have seen play out many times. Let's follow through our thought process to detail how we reasoned about modeling something like a transaction in a graph.

An evolution of modeling transactions in a graph

Think about how you would first add transactions into your graph model. You likely are thinking about how an account transacts with other accounts, or something like we are showing in Figure 4-2.

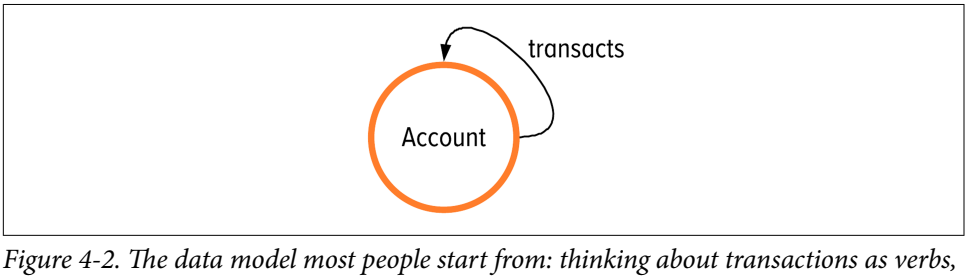


Figure 4-2. The data model most people start from: thinking about transactions as verbs, with phrases like "this account transacts with that account"

The model for Figure 4-2 doesn't work for our example because it uses the idea of a transaction as a verb, whereas our questions use transactions as nouns. We want to know things like an account's most recent transactions and which transactions are loan payments. In this light, we are really thinking about transactions as nouns.

Therefore, transactions need to be vertex labels in our example.

Now we need to reason about the direction of the edges. Most people start with modeling edge direction to follow the flow of money, as shown in Figure 4-3.

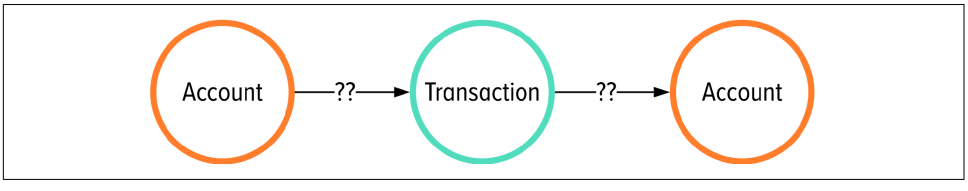


Figure 4-3. Modeling edge direction according to the flow of money

The challenge with a model like Figure 4-3 is to come up with intuitive names for the edges that make it easy to answer our chapter's questions. The edge direction in Figure 4-3 models the flow of money and is awkward for how we are using transactions in our questions. Would we say, "This account had money withdrawn from it via this transaction"? Let's hope not.

So Figure 4-3 isn't going to work for our example, either.

Let's recall our chapter's questions and reason about how we use transactions in the queries. We came up with the following subject-verb-object sentences for the context in which we are using transactions in our example:

1. Transactions withdraw from accounts.
2. Transactions deposit to accounts.

These two phrases might work; let's see how this would work with data. In our data, we could model a transaction and how it interacted with accounts as shown in Figure 4-4.

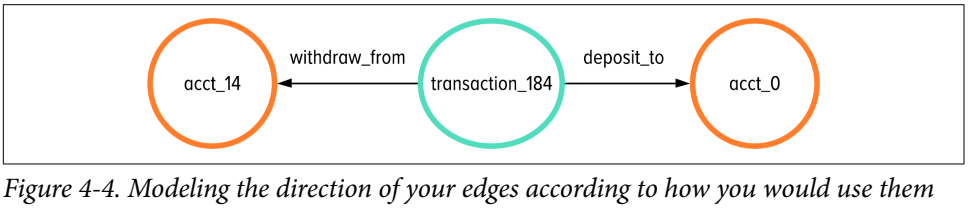


Figure 4-4. Modeling the direction of your edges according to how you would use them in your queries

For the example in this chapter, we think that Figure 4-4 makes it reasonably easy to use our model to answer our questions. This gives us direction for both of our labels: the edge labels will flow from a `Transaction` and go to an `Account`. The schema is shown in Figure 4-5.

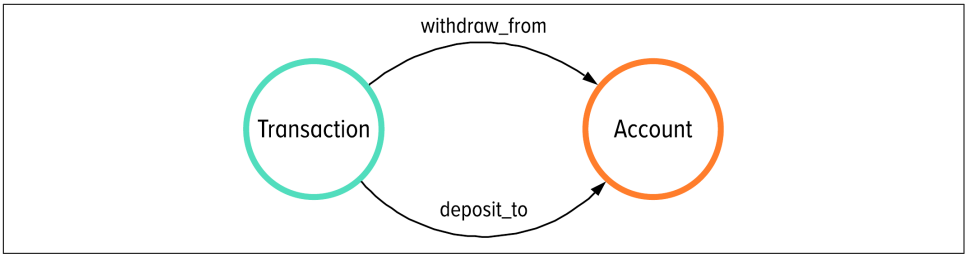


Figure 4-5. Modeling the direction of your edges according to how you would think about the data in your domain

By breaking down your queries into short, active phrases of the structure subject-verb-object, you will be able to naturally find what needs to be a vertex or edge label in your graph model. Then the edge label's direction will come from the subject and go to the object.

Let's zoom out from the nuances of modeling direction for transactions and get back to the final main element of a graph's schema: properties.

When do we use properties?

Let's repeat the first query that will use the transaction vertices:

```
What are the most recent 20 transactions involving Michael's account?
```

The short version of our query from above translates to the following short phrases:

1. Michael owns account
2. Transactions withdraw from his account
3. Select the most recent 20 transactions

So far, we can walk through customers, accounts, and transactions within our graph. Now our question asks for the 20 most recent transactions from an account. This means that we need to subselect our transactions to include only the most recent ones.

Therefore, we will want the ability to filter transactions by time. This brings us to our last tip related to data modeling decisions.

Rule of Thumb #6

If you need to use data to subselect a group, make it a property.

Ordering transactions by time requires us to have that value stored in our graph model: enter properties. This is a great use of a property on the transaction vertex so that we can subselect those vertices in our model. Figure 4-6 shows how we would add time into our ongoing example.

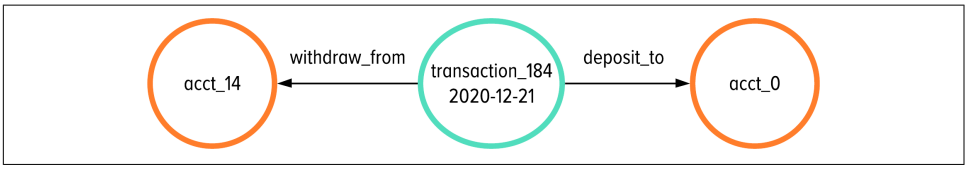


Figure 4-6. Modeling time as a property on the transaction vertex so that we can subselect to query for only the most recent transactions

Together, tips #1–6 give you a great starting point for identifying what will be a vertex, an edge, or a property in your graph data model. We have one last section of data modeling best practices to consider before we start the implementation details for this chapter.

A Graph Has No Name: Common Mistakes in Naming

The callouts in the upcoming section are common mistakes. Each mistake is followed by our bad-better-best recommendations.

Arriving at a consensus on what something should be named and maintained with your codebase is surprisingly difficult. We have three topics on which teams commonly waste their valuable time in bikeshedding how to address naming conventions in their graph data model.

Pitfalls in Naming Conventions #1

Using the word `has` as an edge label.

One of the most common mistakes we see comes from naming all of your edges with the label `has`, as shown on the left side of Figure 4-7. This is a mistake in naming because the word `has` does not provide meaningful context regarding the edge's purpose or direction.

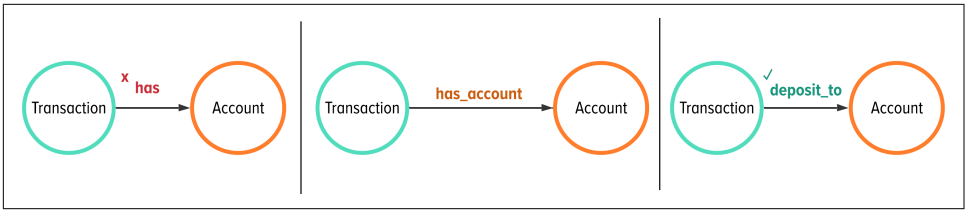


Figure 4-7. From left to right: the bad, better, and recommended ways to name your edges

If your graph model uses `has` for its edge labels, we have two recommendations for you. A better edge label would have the form `has_{vertex_label}`, as shown in the center in orange in Figure 4-7. This type of name allows you to have more specificity in your graph queries while also providing a more meaningful name to maintain in your codebase.

The preferred solution to this problem is shown in green at far right in Figure 4-7. This recommendation advises you to use an active verb that communicates meaning, direction, and specificity to your data. We are going to use the edge labels `deposit_to` and `withdraw_from` to connect transactions to the accounts in our examples.

After meaningful edge labels have been selected, it is also a common mistake to create property names that do not help uniquely identify your data. This brings us to our next pitfall in property graph modeling.

Pitfalls in Naming Conventions #2

Using the word `id` as a property.

The concept of which pieces of data uniquely identify an entity is a deep topic. Using a property key called `id` is a bad decision because it is not descriptive of what it is referring to. Additionally, `id` is a naming clash with the internal naming conventions within Apache Cassandra and is not supported in DataStax Graph.

A slightly better convention would be to name the property that uniquely identifies your data with `{vertex_label}_id`, as shown at center in Figure 4-8. We use this a few times throughout the book because we are working with synthetic examples, and this type of identifier is perfectly fine if you use randomly generated identifiers, like UUIDs (universally unique identifiers). However, you will see us move to using more descriptive identifiers when we work with open source data. These identifiers represent concepts that uniquely identify entities within their domain, such as social security numbers, public keys, and domain-specific universally unique identifiers.

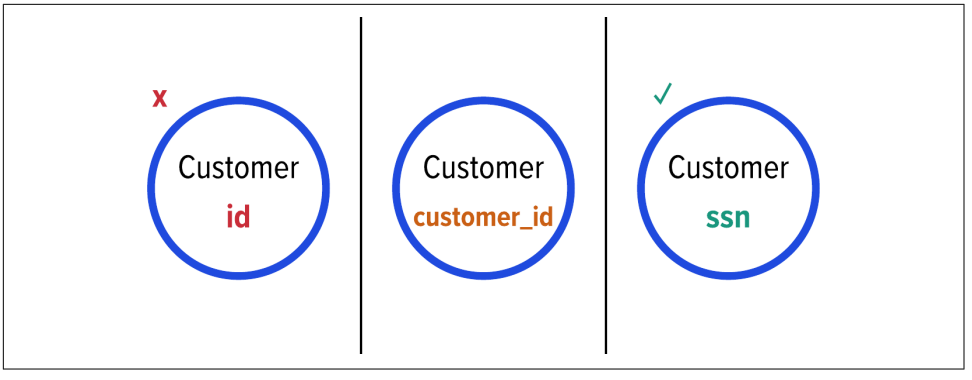


Figure 4-8. From left to right, the bad, better, and recommended ways to name a property to uniquely identify your data.

This brings us to the last and debatably most important mistake that we see throughout application codebases.

Pitfalls in Naming Conventions #3

Inconsistent use of casing.

When it comes to casing, the best approach follows the language conventions that you are writing in. Some languages have style guides that promote `CamelCase`, whereas others prefer `snake_case`. For the examples in this book, we plan to follow the following casing and styles:

1. Capital `CamelCase` for vertex labels
2. Lowercase `snake_case` for edge labels, property keys, and example data

This last tip feels a bit pedantic to even bring up in a graph book. We are mentioning it because consistency in naming conventions tends to be forgotten, creating expensive roadblocks for teams during the last stretch of getting their graph technology into production. The more trivial these tips seem to your team, the better off you probably already are in making sure to remember them.

Our Full Development Graph Model

The previous discussion of graph data modeling illustrated how we broke down our first query to evolve the example from Chapter 3. In this section, we want to build up the remaining elements in our data model to answer all the questions for this chapter's example.

The example in this chapter adds schema and data that enable our application to answer the following three questions:

1. What are the most recent 20 transactions involving Michael's account?
2. In December, at which vendors did Michael shop, and with what frequency?
3. Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

We have already stepped through how to model the first question. Let's take a closer look at it.

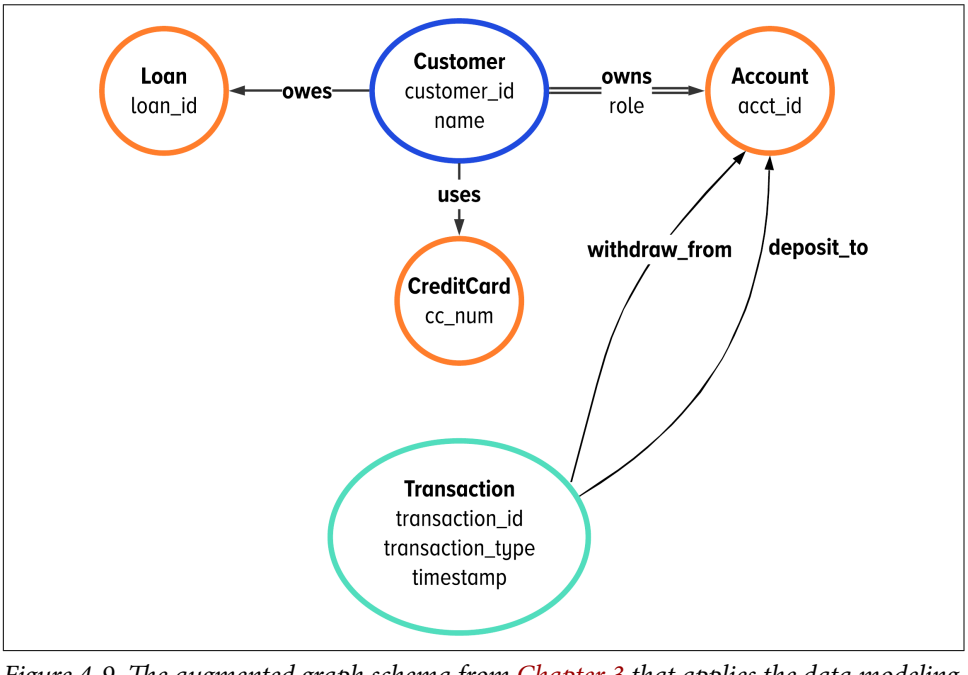


Figure 4-9. The augmented graph schema from Chapter 3 that applies the data modeling principles to answer the first query of our expanded example

The graph schema in Figure 4-9 applies the principles we built up to answer the first question into a graph data model. The new vertex label is `Transaction`, with two new edge labels to the `Account` vertex: `withdraw_from` and `deposit_to`, respectively. We discussed how and where to model time in our graph, which you see in Figure 4-9 with `timestamp` on the `Transaction` vertex.

Next, let's consider this chapter's remaining questions for our example in this chapter by modeling the queries:

```
1. In December, at which vendors did Michael shop, and with what frequency?

2. Find and update the transactions that Jamie and Aaliyah most value:
     their payments from their account to their mortgage loan.
```

To arrive at a data model for these questions, let's apply the thought processes we introduced in "Graph Data Modeling 101" on page 82. Following the advice there, we came up with three statements about transactions:

1. Transactions charge credit cards.
2. Transactions pay vendors.
3. Transactions pay loans.

From these statements, we can find the rest of our required schema elements. First, we need a new vertex label to represent where our customers shop: `Vendor`. Next, we need an edge label, `pay`, for a transaction to the `Loan` or `Vendor` vertex labels. Last, we need another edge label, `charge`, to indicate that a transaction charges a credit card.

Bringing all of this together, we have the schema shown in Figure 4-10.

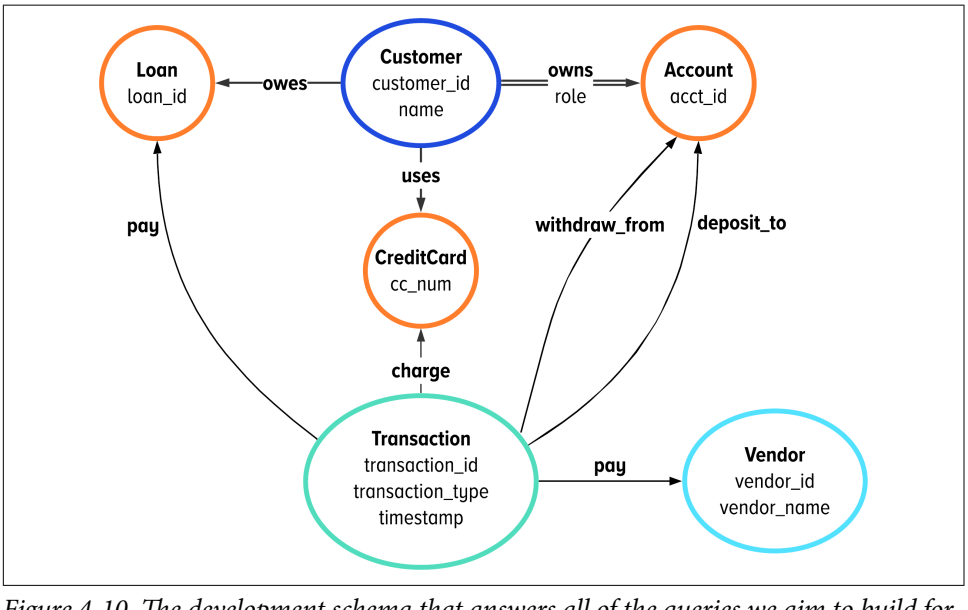


Figure 4-10. The development schema that answers all of the queries we aim to build for this example

Before We Start Building

We reduced the full perspective on graph data modeling to include only the practices that we need for our current example. Beyond these core principles, you will find

edge cases about your data that are not covered here. That is expected. We are teaching a thought process and selected the principles here as a starting guide for modeling your data like a graph.

If we could ensure you understood one concept about graph data modeling, it would be the following: modeling your data as a graph is just as much of an art as it is engineering. The art of the data modeling process involves creating and evolving your perspective on your data. This evolution translates your mindset into the paradigm of relationship-first data modeling.

When you find new modeling cases in this book or in your own work, ask the following questions about what you are modeling to help develop your own reasoning:

1. What does this concept mean to the end user of the application?
2. How are you going to read this data in your application?

Defining your data model is the first step in applying graph thinking to your application. Focus on the data you can integrate, the queries you want to ask, and what this will mean to your end user. When combined, those three concepts articulate how we see, model, and use graph data within an application.

Our Thoughts on the Importance of Data, Queries, and the End User

To help you learn and apply our perspective to building your own graph model, let's walk through the importance of data, queries, and the end user.

Our first piece of advice is to focus on the data you have. It is easy to boil the ocean by modeling your industry's entire graph problem; avoid this rabbit hole! Your graph model will evolve if you keep centered on getting to production with the data with which your application will be working.

Second, apply the practice of query-driven design. Build your data model to accommodate only a predefined set of graph queries. A common red herring we run into on this topic is those applications that aim to create open traversals across any discoverable data in a graph. For developmental purposes, the ability to explore and discover makes sense. However, for production use, an application with open traversal access can introduce a myriad of concerns.

For security, performance, and maintenance implications, we strongly advise teams not to create production platforms with unbounded and unlimited traversals. The warning sign we see is a lack of specificity for your graph application. We know this perspective is very hard to apply when you are first exploring graph data. We see the

line here as setting expectations between what you want to do during development versus what you want to push to production in a distributed production application.

Last and most importantly, you have to consider what the data means to your end user. Everything from selecting naming conventions to the objects in your graph will be interpreted by someone else: your team members or your application users. Naming conventions and graph objects are interpreted and maintained by your engineering team members; choose them wisely.

Ultimately, your graph data will be presented to an end user through your application. Spend time designing your data architecture, models, and queries to present information that is most meaningful to them.

When combined, these three concepts articulate how we see, model, and use graph data within an application. Again, the three concepts are to build with the data you have, follow query-driven design, and design for your end user. Following these design principles will help get you unstuck during those difficult data modeling discussions and prepare your application to be the best use of graph data the industry has ever seen.

Implementation Details for Exploring Neighborhoods in Development

Our schema from Figure 4-10 requires only two new vertex labels: `Transaction` and `Vendor`. What you have practiced a few times prior to now is how to take a schema drawing and translate it into code. We showed the schema in Figure 4-10, and in Example 4-1 we show you the code.

Example 4-1.

```
schema.vertexLabel("Transaction").
       ifNotExists().
       partitionBy("transaction_id", Int).
       property("transaction_type", Text).
       property("timestamp", Text).
       create();

schema.vertexLabel("Vendor").
       ifNotExists().
       partitionBy("vendor_id", Int).
       property("vendor_name", Text).
       create();
```

In case you are wondering, we are using `Text` as the data type for timestamp to make it easier to teach concepts in our upcoming examples. We will be using the ISO 8601 standard format stored as text.

In addition to these vertex labels, we added relationships between the `Transaction` vertex and the other vertex labels in this graph. Let's start with the new edge labels between the `Transaction` and `Account` vertex labels. The schema code for the new edge labels is shown in Example 4-2.

Example 4-2.

```
schema.edgeLabel("withdraw_from").
       ifNotExists().
       from("Transaction").
       to("Account").
       create();

schema.edgeLabel("deposit_to").
       ifNotExists().
       from("Transaction").
       to("Account").
       create();
```

These two edges model how money moves to and from an account within your bank. In Example 4-3, we add in the rest of the edge labels in our example:

Example 4-3.

```
schema.edgeLabel("pay").
       ifNotExists().
       from("Transaction").
       to("Loan").
       create();

schema.edgeLabel("charge").
       ifNotExists().
       from("Transaction").
       to("CreditCard").
       create();

schema.edgeLabel("pay").
       ifNotExists().
       from("Transaction").
       to("Vendor").
       create();
```

These last three edge labels complete the edges we will need to describe transactions between the assets in our example.

Generating More Data for Our Expanded Example

As examples grow, so too does the data. We wrote a small data generator to expand the data from Chapter 3 to include our data model from Figure 4-10. If you are interested in the data generation process for this chapter, you have two options.

Your first option is to use the bash scripts to reload the exact same data you will see in the upcoming examples. We will teach you about this tool and process in Chapter 5, but you are welcome to preview the loading script in the GitHub repository. We recommend using the scripts throughout this book if you would like the examples you are running locally to match the results we show in the text.

Your second option is to dive into and execute our data generation code. We provided our code in a separate Studio Notebook called `Ch4_DataGeneration`. We recommend this option if you want to dig into creating fake data with Gremlin and the methods we used.

An Important Warning About the Data Generation Process

If you rerun the data insertion process in your Studio Notebook, the results in your local graph will not precisely match the results printed in this text. If you want the data to match precisely, we recommend importing the exact same graph structure via DataStax Bulk Loader. You will find all of this in the accompanying technical materials.

Up to this point, we have accomplished many tasks. We explored our first set of data modeling tips, created a development model, looked at the schema code, and inserted data.

The last main task is to use the Gremlin query language to walk around our model and answer questions about our data.

Basic Gremlin Navigation

The main objective of this chapter is to illustrate a real-world graph schema that walks through multiple neighborhoods of graph data.

For your reference, we will use the words *walk*, *navigate*, and *traverse* interchangeably throughout this book to mean that we are writing graph queries.

Everything in this chapter up until now was required to set up answering the following three questions in this section:

1. What are the most recent 20 transactions involving Michael's account?
2. In December, at which vendors did Michael shop, and with what frequency?
3. Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

Let's walk through the queries and their results. Then, in the chapter's final section on Advanced Gremlin, we will delve a bit deeper into how to shape the result payload.

Our recommendation is that you find a way to reference Figure 4-10 as you practice the queries in the upcoming sections. We recommend doing this because your schema functions as your map; you need to know where you are so that you can walk in the right direction to your destination.

Query 1: What are the most recent 20 transactions involving Michael's account?

Let's start with some pseudocode in Example 4-4 to think about how we are going to walk through our data to answer this first question.

Example 4-4.

```
Question: What are the most recent 20 transactions involving Michael's account?
Process:
    Start at Michael's customer vertex
    Walk to his account
    Walk to all transactions
    Sort them by time, descending
    Return the top 20 transaction ids
```

We used the process outlined in Example 4-4 to create the Gremlin query in Example 4-5.

Example 4-5.

```
1 dev.V().has("Customer", "customer_id", "customer_0"). // the customer
2         out("owns").                      // walk to his account
3         in("withdraw_from", "deposit_to"). // walk to all transactions
4         order().                          // sort the vertices
5           by("timestamp", desc).   // by their timestamp, descending
6         limit(20).                        // filter to only the 20 most recent
7         values("transaction_id")          // return the transaction_ids
```

A sample of the results:

```
    "184", "244", "268", ...
```

Let's dig into this query one step at a time.

On line 1, `dev.V().has("Customer", "customer_id", "customer_0")` looks up a vertex according to its unique identifier. Then on line 2, the step `out("owns")` walks through the outgoing `owns` edge to the `Account` vertices for this customer. In this case, Michael has only one account.

At this point, we want to access all transactions. On line 3, the `in("withdraw_from", "deposit_to")` step does just that: we walk through the incoming edge labels to access transactions. At line 4, we are on the transaction vertices.

We left a detail out of "An evolution of modeling transactions in a graph" on page 86 that we want to bring up now. The simplicity of line 3 in Example 4-5 was also part of the motivation that led to how we designed the edges in our data model. This first query was much harder to write and reason about when the edges were going in different directions.

The `order()` step on line 4 indicates that we need to provide some sort of order to the vertices, which are transactions. We specify the sort order on line 5 with the `by("time stamp", desc)` step. This means that we are going to access, merge, and sort all `Transaction` vertices according to their timestamp. Then we want to select only the 20 most recent vertices with `limit(20)`. Last, on line 7, we want to get access to the `transaction_ids`, so we select them via the `values("transaction_id")` step.

This query will return a list of values that contains the `transaction_id` for each of the 20 most recent transactions across all of the customer's accounts.

Imagine how much more powerful this would be to display for the end user. They would be able to see the details that are most relevant to them instead of navigating multiple screens to join this data together in their head. This type of query is vital in understanding how to personalize your application to what a customer most cares about.

Query 2: In December 2020, at which vendors did Michael shop, and with what frequency?

For this second question, let's start with an outline of the query in Example 4-6 to think about how we are going to walk through our data to answer the question.

Example 4-6.

```
Question: In December 2020, at which vendors did Michael shop, and with what frequency?
Process:
    Start at Michael's customer vertex
    Walk to his credit card
    Walk to all transactions
    Only consider transactions in December 2020
    Walk to the vendors for those transactions
    Group and count them by their name
```

We start the process outlined in Example 4-6 in Example 4-7 and complete it in Example 4-8. In preparation for this query, we used the ISO 8601 timestamp standardization in our data to make it easier to range on dates. In the ISO 8601 standard, timestamps are commonly formatted as `YYYY-MM-DD'T'hh:mm:ss'Z'`, where `2020-12-01T00:00:00Z` represents the very beginning of December in 2020.

Example 4-7.

```
1 dev.V().has("Customer", "customer_id", "customer_0"). // the customer
2         out("uses").                          // Walk to his credit card
3         in("charge").                         // Walk to all transactions
4         has("timestamp",                      // Only consider transactions
5             between("2020-12-01T00:00:00Z",   // in December 2020
6                     "2021-01-01T00:00:00Z")).
7         out("pay").                           // Walk to the vendors
8         groupCount().                         // group and count them
9           by("vendor_name")                   // by their name
```

The results are:

```
{
  "Nike": "2",
  "Amazon": "1",
  "Target": "3"
}
```

Randomization affects the results of query 2. If you use the data generation process instead of loading the data, your graph may have a slightly different structure and therefore different counts for query 2.

The setup for Example 4-7 follows a similar access pattern as before, where we start at a customer and then traverse to a neighboring vertex. We start at `customer_0` and walk to their credit cards and then to transactions. On lines 4 through 6, we are using a way to filter your data during a traversal. Here, we are filtering all vertices according

to their timestamps in a specific range. Specifically, `has("timestamp", between("2020-12-01T00:00:00Z", "2021-01-01T00:00:00Z"))` sorts and returns all transactions that have a timestamp during the month of December in the year 2020.

At line 7, following our schema, we walk to the vendors with the `out("pay")` step. Finally, we want to return the vendor's name along with how many times a transaction was observed with that vendor. We do this on lines 8 and 9 with `group Count().by("vendor_name")`.

In addition to `between`, Table 4-1 lists the most popular predicates you can use to range on values. Please refer to the book by Kelvin Lawrence for the full table of predicates.[2]

Table 4-1. Some of the most popular predicates that you can use to range on values

Predicate	Usage
eq	Equal to
neq	Not equal to
gt	Greater than
gte	Greater than or equal to
lt	Less than
lte	Less than or equal to
between	Between two values excluding the upper bound

You may be wondering: what if we wanted to order the output of Example 4-7?

If you wanted to return the results in a decreasing order, you would do that by adding in the `order().by()` pattern, shown on lines 10 and 11 in Example 4-8.

Example 4-8.

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         out("uses").
3         in("charge").
4         has("timestamp",
5             between("2020-12-01T00:00:00Z",
6                     "2021-01-01T00:00:00Z")).
7         out("pay").
8         groupCount().
9           by("vendor_name").
```

2 Kelvin Lawrence, *Practical Gremlin: An Apache TinkerPop Tutorial*, January 6, 2020, *https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html*.

```
10        order(local).         // Order the map object
11          by(values, desc)    // according to the groupCount map's values
```

The results are now:

```
{
  "Target": "3",
  "Nike": "2",
  "Amazon": "1"
}
```

We threw in the use of scope in a traversal at line 10 with the step `order(local)`.

Scope
: Scope determines whether the particular operation is to be performed to the current object (local) at that step or to the entire stream of objects up to that step (global).

For a visual explanation of scope in a traversal, consider Figure 4-11.

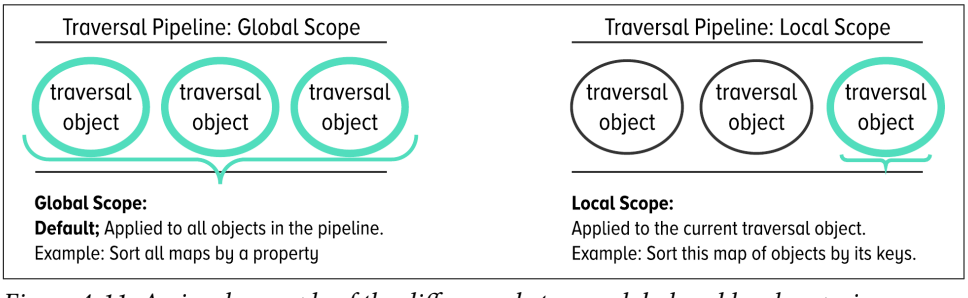


Figure 4-11. A visual example of the difference between global and local scope in a Gremlin traversal

To explain it simply, at the end of line 9, we needed to order the object in the pipeline, which is a map. The use of `local` on line 10 tells the traversal to sort and order the items within the map object. Another way to think about this is that we want to order the entries *within* the map. We do that by indicating that the scope is local to the object itself.

The best way to understand traversal scope is to play with different queries in your Studio Notebook and see how the scope affects the shape of your results. More great visual diagrams on understanding the flow of data and object types are available on the DataStax Graph documentation pages.

If you ever question what object type you have in the middle of developing a Gremlin traversal, add `.next().getClass()` to where you are in your traversal development. This will inspect the objects at this point in your traversal and give you their class.

Query 3: Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

The advantage of using a graph database really starts to show as we walk through multiple neighborhoods of data, as we will be doing with this third and last query. Here, we are accessing and mutating data across five neighborhoods of data in our graph. We are going to break this query down into three steps: access, mutation, and then validation.

The first simplification we are going to make to our account is to reduce the scope of the query. We know that Jamie and Aaliyah share only one account: `acct_0`. Therefore, to further simplify our query, we can focus on walking from only one person; we choose Aaliyah.

This brings us to the first shorter query we want to build:

Query 3a: Find Aaliyah's transactions that are loan payments. Before we can update important transactions, we need to find the important ones. The transactions we are looking for are those that indicate a loan payment from Aaliyah's joint account to Jamie and Aaliyah's mortgage. Let's outline our approach in pseudocode in Example 4-9 to think about how we are going to walk through our data to answer the question.

Example 4-9.

```
Question: Find Aaliyah's transactions that are loan payments
Process:
    Start at Aaliyah's customer vertex
    Walk to her account
    Walk to transactions that are withdrawals from the account
    Go to the loan vertices
    Group and count the loan vertices
```

We used the process outlined in Example 4-9 to create the Gremlin query in Example 4-10.

Example 4-10.

```
dev.V().has("Customer", "customer_id", "customer_4"). // accessing Aaliyah's vertex
        out("owns").                                  // walking to the account
        in("withdraw_from").                          // only consider withdraws
        out("pay").                                   // walking out to loans or vendors
        hasLabel("Loan").                             // limiting to only loan vertices
        groupCount().                                 // groupCount the loan vertices
          by("loan_id")                               // by their loan_id
```

The results for the sample data will look like:

```
{
  "loan80": "24",
  "loan18": "24"
}
```

Let's step through Example 4-10. On line 1, we start by accessing the customer and walking to their account. On line 2, we traverse to Aaliyah's account. Recalling the schema, we walk through the incoming edge `withdraw_from` to access account withdrawals on line 3.

On line 4, we walk through the `pay` edge label to arrive at either `Loan` or `Vendor` vertices. The `hasLabel("Loan")` step on line 5 is a filter that eliminates all vertices at this point that are not loans. This means we are now considering only the assets into which a payment has been made from the account *and* that are loans. On line 6, we group and count those loan vertices according to their unique identifier, as indicated on line 7.

The result payload indicates that this account has made 24 payments into each loan within the system.

Next, we want to go a step further and update the data in this traversal to indicate which transactions are mortgage payments.

Query 3b: Find and update the transactions that Jamie and Aaliyah most value: their payments from their checking account to their mortgage, loan_18.

The traversal required to accomplish this query is a mutating traversal. All we mean by *mutating traversal* is that it updates data in the graph as a part of the traversal. Example 4-11 shows how we can use the traversal above to write properties on the transactions that go from the account and into `loan_18`, because `loan_18` is Jamie and Aaliyah's mortgage loan.

Example 4-11.

```
1 dev.V().has("Customer", "customer_id", "customer_4"). // accessing Aaliyah's vertex
2         out("owns").                                  // walking to the account
3         in("withdraw_from").                          // only consider withdraws
4         filter(
5                out("pay").                            // walking to loans or vendors
6                has("Loan", "loan_id", "loan_18")).    // only keep loan_18
7         property("transaction_type",  // mutating step: set the "transaction_type"
8                  "mortgage_payment"). // to "mortgage_payment"
9         values("transaction_id", "transaction_type")  // return transaction & type
```

The results are:

```
"144", "mortgage_payment",
"153", "mortgage_payment",
"132", "mortgage_payment",
...
```

Example 4-11 starts the same as the first part of our query. The new portion of this traversal spans lines 4 through 6 with the `filter(out("pay").has("Loan", "loan_id", "loan_18"))` steps. Here, we allow only the transactions that are connected to the `loan_18` vertex to continue down the pipeline. This is because `loan_18` is Jamie and Aaliyah's mortgage loan. On line 7, we mutate the transaction vertices by changing "transaction_type" to "mortgage_payment." At the end of this traversal on line 9, we want to return the `transaction_id` along with its new property, its `transaction_type`.

Query 3c: Verify that we didn't update every transaction. At this point, it is very helpful to make sure that we did not update all of Aaliyah's transactions with `mortgage_payment`. We can do that with a quick check, shown in Example 4-12.

Example 4-12.

```
// check that we didn't update every transaction
1 dev.V().has("Customer", "customer_id", "customer_4"). // at the customer vertex
2         out("owns").                 // at the account vertex
3         in("withdraw_from").         // at all withdrawals
4         groupCount().                // group and count the vertices
5           by("transaction_type")     // according to their transaction_type
```

The results from the Studio Notebook are shown below. We set `unknown` as the default value during the data loading process also shown in the Studio Notebook:

```
{
  "mortgage_payment": "24",
  "unknown": "47"
}
```

This query does a quick check to validate that we properly mutated our data. Combining lines 1 through 3, we process all of the transactions from Aaliyah's bank account. At line 4, we do a `groupCount()` for all of those vertices according to the value stored in the `transaction_type` property. Here, we see that we correctly updated only the 24 transactions that are mortgage payments to `loan_18`. This validates that our mutation query properly updated our graph structure.

This section started out with three questions, and the last three examples answered them using the Gremlin query language.

We stepped through the basic queries to show you where to start. Get your basic graph walks ironed out before you start exploring the full flexibility and expressivity of the Gremlin query language. We always recommend iterating through Gremlin steps in development mode to find the basic walks that accomplish your queries. This means we are asking you to execute line 1 of a Gremlin query and look at the results. Then execute lines 1 and 2 and look at the results, and so on.

After you have mapped out your basic walks, you can try out more advanced Gremlin. At this point in development, it is very common to find ways to create specific payload structures to pass back to your endpoint.

We will cover the most popular strategies for building JSON with Gremlin in the next section.

Advanced Gremlin: Shaping Your Query Results

The goal of this section is to build up a more advanced version of our Gremlin query that answers a new question:

```
Is there anyone else who shares accounts, loans, or credit cards with Michael?
```

We would like to introduce a new question to demonstrate advanced Gremlin concepts within a small neighborhood of data. Once you understand how these concepts apply to this question, we invite you to use the accompanying notebook for this chapter to implement the concepts for the other queries introduced in "Basic Gremlin Navigation" on page 97.

We will work through shaping the results of our new query in a few stages. They are:

1. Shaping query results with the `project()`, `fold()`, and `unfold()` steps
2. Removing data from the results with the `where(neq())` pattern
3. Planning for robust result payloads with the `coalesce()` step

For anyone diving deeper into the world of Gremlin queries, we highly recommend the detail and explanations in the book *Practical Gremlin: An Apache TinkerPop Tutorial* by Kelvin Lawrence.[3]

3 Kelvin Lawrence, *Practical Gremlin: An Apache TinkerPop Tutorial*, January 6, 2020, *https://kelvinlawrence.net/book/Gremlin-Graph-Guide.html*.

Shaping Query Results with the project(), fold(), and unfold() Steps

When we start writing a new query, we like to slowly build up its required pieces. One of the most useful Gremlin steps is the `project()` step, because it helps us build up a specific map of data from our query. Let's start building our query out by defining the three keys we want to have in our map: `CreditCardUsers`, `AccountOwners`, and `LoanOwners`.

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3           by(constant("name or no owner for credit cards")).
4           by(constant("name or no owner for accounts")).
5           by(constant("name or no owner for loans"))
```

This query structure is the base of what we are building toward. We want to start with a specific person in this example—namely Michael. Then we want to create a data structure that will have three keys: `CreditCardUsers`, `AccountOwners`, and `LoanOwners`. We create this map with the `project()` step on line 2. The arguments to the `project()` step are the three keys. For each key in the `project()` step, we want to have a `by()` step. Each `by()` modulator creates the values associated to the keys:

1. The `by()` modulator on line 3 will create a value for the `CreditCardUsers` key.
2. The `by()` modulator on line 4 will create a value for the `AccountOwners` key.
3. The `by()` modulator on line 5 will create a value for the `LoanOwners` key.

Let's take a look at the results at this point:

```
{
  "CreditCardUsers": "name or no owner for credit cards",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

This is a good baseline to work from. Next, let's walk through our graph structure to start to populate the values in our map. We will start with the data for the first key: finding people who share a credit card with Michael.

Thinking back to our schema, we will need to walk through the `uses` edge to get to the credit cards. Then we will walk back through the `uses` edge to get back to people. After that, we want to access their names. In Gremlin, we would add this walk on lines 3, 4, and 5:

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5            values("name")).
```

```
6         by(constant("name or no owner for accounts")).
7         by(constant("name or no owner for loans"))
```

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5            values("name")).
6         by(constant("name or no owner for accounts")).
7         by(constant("name or no owner for loans"))
```

The only steps we added were to walk from Michael out to his credit card via the `uses` edge on line 3. Then, on line 4, we walk back to all people who use that credit card. The resulting payload is:

```
{
  "CreditCardUsers": "Michael",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

This confirms what we know: Michael didn't share any credit cards with other people. We expected to see his name in the result set.

Now let's do the same thing for the next key in our map: `AccountOwners`. Here, we want to walk out the `owns` edge to the account vertex and back to the person vertex:

```
1 dev.V().has("Customer", "customer_id", "customer_0").
2           project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5            values("name")).
6         by(out("owns").
7            in("owns").
8            values("name")).
9         by(constant("name or no owner for loans"))
```

Let's look at the resulting payload:

```
{
  "CreditCardUsers": "Michael",
  "AccountOwners": "Michael",
  "LoanOwners": "name or no owner for loans"
}
```

Looking at this data, we do not see what we would expect. We expected to see Maria as a resulting value for `AccountOwners`. Maria does not show up because Gremlin is lazy; it returns the first result, not all results. We need to add a barrier to force all results to finish and return.

The barrier that we like to use here is `fold()`. The `fold()` step will wait for all of the data to be found and then roll up the results into a list. This is a bonus, because now we can build up specific data type rules for our application. The adjusted query reads:

```
dev.V().has("Customer", "customer_id", "customer_0").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
           values("name").
           fold()).
        by(out("owns").
           in("owns").
           values("name").
           fold()).
        by(constant("name or no owner for loans"))
```

The shape of the data in the resulting payload is what we were expecting to see:

```
{
  "CreditCardUsers": [
    "Michael"
  ],
  "AccountOwners": [
    "Michael",
    "Maria"
  ],
  "LoanOwners": "name or no owner for loans"
}
```

Let's complete the construction of our map by adding in the statements in the last `by()` step. These statements need to walk from Michael out to his loan and then back. The query and result set are:

```
dev.V().has("Customer", "customer_id", "customer_0").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
           values("name").
           fold()).
        by(out("owns").
           in("owns").
           values("name").
           fold()).
        by(out("owes").
           in("owes").
           values("name").
           fold())
```

```
{
  "CreditCardUsers": [
    "Michael"
  ],
  "AccountOwners": [
```

```
    "Michael",
    "Maria"
  ],
  "LoanOwners": [
    "Michael"
  ]
}
```

```
dev.V().has("Customer", "customer_id", "customer_0").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
           values("name").
           fold()).
        by(out("owns").
           in("owns").
           values("name").
           fold()).
        by(out("owes").
           in("owes").
           values("name").
           fold())
```

```
{
  "CreditCardUsers": [
    "Michael"
  ],
  "AccountOwners": [
    "Michael",
    "Maria"
  ],
  "LoanOwners": [
    "Michael"
  ]
}
```

At this point, we have the expected results. We see that Michael shares an account with Maria. And we see that Michael doesn't share credit cards or loans with anyone else.

For some applications, it isn't helpful to return that Michael shares a credit card with himself. Let's dive into how we would remove Michael from this resulting payload.

Removing Data from the Results with the where(neq()) Pattern

It might be useful for you to eliminate Michael from the result set. We can do that by using the `as()` step to store Michael's vertex, and then eliminate it from the result set. You can remove a vertex from your pipeline with the step `where(neq("some_stored_value"))`.

The next version of our query, in which we have directly applied this step to each section, is shown in Example 4-13.

Example 4-13.

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold()).
8         by(out("owns").
9            in("owns").
10             where(neq("michael")).
11           values("name").
12           fold()).
13        by(out("owes").
14           in("owes").
15             where(neq("michael")).
16           values("name").
17           fold())
```

The full results of Example 4-13 are shown below:

```
{
  "CreditCardUsers": [],
  "AccountOwners": [
    "Maria"
  ],
  "LoanOwners": []
}
```

The main additions to our query occur on lines 1, 5, 10, and 15 in the above query. On line 1, we store the vertex for Michael with the `as("michael")` step. Let's take a look at what is happening with `where(neq("michael"))` on line 5, which is the same thing that is happening on lines 10 and 15.

To understand what is happening on line 5, you need to remember where you are in your graph. At the end of line 4, we are on `Customer` vertices. Specifically, we are processing customers that share an account with `Michael`. This is where the `where(neq("michael"))` step comes in. We want to apply a true/false filter to every vertex in the pipeline. The true/false filter test is whether or not that vertex is equal to Michael: `where(neq("michael"))`. If the vertex is Michael, line 5 eliminates it from the traversal. If the vertex is not Michael, the vertex passes through the filter and remains in the pipeline.

Planning for Robust Result Payloads with the coalesce() Step

Depending on your team's data structure rules, checking whether or not a value in your data payload is an empty list may not be preferred. We can help design around that.

We can implement try/catch logic so that your query doesn't return an empty list. We will step through this for the first key in the map: `CreditCardUsers`. After we step through that, we will add in the full query details for the two remaining `by()` steps.

Let's rewind and go back to just building up the JSON payload for the value associated to `CreditCardUsers`. We are starting from here:

```
dev.V().has("Customer", "customer_id", "customer_0").as("michael").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
             where(neq("michael")).
           values("name").
           fold()).
        by(constant("name or no owner for accounts")).
        by(constant("name or no owner for loans"))
```

```
{
  "CreditCardUsers": [],
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

You can implement try/catch logic in Gremlin with the `coalesce()` step. We want to shape the results so that there is always a value in the lists for each key, like `"CreditCardUsers": ["NoOtherUsers"]`. Let's start by seeing how to integrate the `coalesce` step into our query:

```
dev.V().has("Customer", "customer_id", "customer_0").as("michael").
        project("CreditCardUsers", "AccountOwners", "LoanOwners").
        by(out("uses").
           in("uses").
             where(neq("michael")).
           values("name").
           fold().
           coalesce(constant("tryBlockLogic"),    // try block
                    constant("catchBlockLogic"))).// catch block
        by(constant("name or no owner for accounts")).
        by(constant("name or no owner for loans"))
```

The resulting payload is:

```
{
  "CreditCardUsers": "tryBlockLogic",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

When you use the `coalesce()` step in line 8, it takes two arguments. The first argument is on line 8 and can be thought of as the try block logic. The second argument is on line 9 and can be thought of as the catch block logic.

If the try block logic succeeds, then the resulting data is passed down the pipeline. In this case, for illustrative purposes, we used something that would definitely succeed: the `constant()` step. This step returned the string `"tryBlockLogic"` that we see in the resulting payload. The `constant()` step is useful for many reasons, one of which is that it can serve as a placeholder while you build up more complicated queries. This is how we are using it here.

Should the first argument of the `coalesce()` step fail on line 8, the second argument will execute on line 9. Let's look at how we can use this to populate what we want in our data payload:

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold().
8            coalesce(unfold(),                    // try block
9                     constant("NoOtherUsers"))). // catch block
10        by(constant("name or no owner for accounts")).
11        by(constant("name or no owner for loans"))
```

```
{
  "CreditCardUsers": "NoOtherUsers",
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

On line 8, the logic that we added to the try block is the `unfold()`. This is trying to take the results from the previous step and successfully unfold them. The results at this point in the pipeline are an empty list `[]`. In Gremlin, you cannot unfold an empty object. This throws an exception that is caught by the try block. Therefore, we execute line 9, the second argument of the `coalesce()` step: `constant("NoOtherUsers")`. This is why we see the entry `"CreditCardUsers": "NoOtherUsers"` in our result payload.

Regrettably, we lost our guaranteed list structure. We can add that back in with a `fold()` after the `coalesce()` step:

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold().
8            coalesce(unfold(),
9                     constant("NoOtherUsers")).fold()).
10        by(constant("name or no owner for accounts")).
11        by(constant("name or no owner for loans"))
```

```
{
  "CreditCardUsers": [
    "NoOtherUsers"
  ],
  "AccountOwners": "name or no owner for accounts",
  "LoanOwners": "name or no owner for loans"
}
```

The steps we added from line 5 to line 9 create a predictable data structure to exchange throughout your application. It will be well-formatted JSON about which other applications can reason.

Next, we need to add this try/catch logic to each `by()` step. The full logic pattern to add at the end of each `by()` step in our full query is:

```
coalesce(unfold(),                  // try to unfold the names
         constant("NoOtherUsers")). // inject this string if there are no names
fold()                              // structure the results into a list
```

This Gremlin pattern ensures we have a nonempty list in the resulting payload. The full query and its results are:

```
1 dev.V().has("Customer", "customer_id", "customer_0").as("michael").
2         project("CreditCardUsers", "AccountOwners", "LoanOwners").
3         by(out("uses").
4            in("uses").
5              where(neq("michael")).
6            values("name").
7            fold().
8            coalesce(unfold(),
9                     constant("NoOtherUsers")).fold()).
10         by(out("owns").
11           in("owns").
12             where(neq("michael")).
13           values("name").
14           fold().
15           coalesce(unfold(),
16                    constant("NoOtherUsers")).fold()).
```

```
17          by(out("owes").
18             in("owes").
19               where(neq("michael")).
20             values("name").
21             fold().
22             coalesce(unfold(),
23                      constant("NoOtherUsers")).fold())
```

```
{
  "CreditCardUsers": [
    "NoOtherUsers"
  ],
  "AccountOwners": [
    "Maria"
  ],
  "LoanOwners": [
    "NoOtherUsers"
  ]
}
```

We find that iterative building and stepping through Gremlin steps is the best way to wrap your head around the query language. This book is about teaching you our thought processes, and this is how we think through using Gremlin. There is more than one way to write a graph query; we hope you are curious about using other steps to process the same data. Figuring this out can be as easy as opening up a Studio Notebook and exploring new steps on your own.

Moving from Development into Production

Bringing back our scuba analogy from the beginning of this chapter, our time training in the pool has come to a close. As we see it, the progression through the technical examples in this chapter is just like learning buoyancy control or deepwater troubleshooting within a pool. At some point, you have learned everything you can from practicing in a controlled environment.

With the foundation we have built over the past few chapters, it is time to take the leap out of your development environment and build a production-ready graph database.

Before you get too concerned, this doesn't mean you are supposed to know everything there is to know about graph data. There are still myriad topics we are continuing to explore ourselves.

What it does mean, however, is that we think you are ready to move into a deeper understanding of using graph data in distributed systems. We set up this example to get you ready for one last step down into the physical data layer of understanding graph data structures in Apache Cassandra. Specifically, the upcoming chapter will show you how to optimize your graph structures for distributed applications.

While illustrating how we think through graph data, we purposefully set up some traps in the example in this chapter. In the next chapter, we will show these traps to you and walk you through their resolution. This upcoming chapter will be the last chapter that uses our C360 example, as it will describe the final iteration in creating a production-quality graph schema for this example.

CHAPTER 5

Exploring Neighborhoods in Production

When you use DataStax Graph, you are working with graph data in Cassandra. And if you have been following along and executing the implementation details from the last two chapters, you have already been using it.

The paradigm shift from working with a traditional database to working with Apache Cassandra is that we write our data according to how we are going to read it.

To illustrate how we apply this, the examples in Chapters 3 and 4 used but skipped over fundamental topics of working with graph data in Apache Cassandra. Concepts like edge direction and partition key design are fundamental to building a production-quality, scalable, and distributed graph data model.

We are going to dig deeply into the topics of distributed data to set you up for a successful use of distributed graph technology within your production stack.

Recall that we mentioned at the end of Chapter 4 that we purposely set up some traps. Our example built up the schema shown in Figure 5-1 and aimed to use queries like we have in Example 5-1.

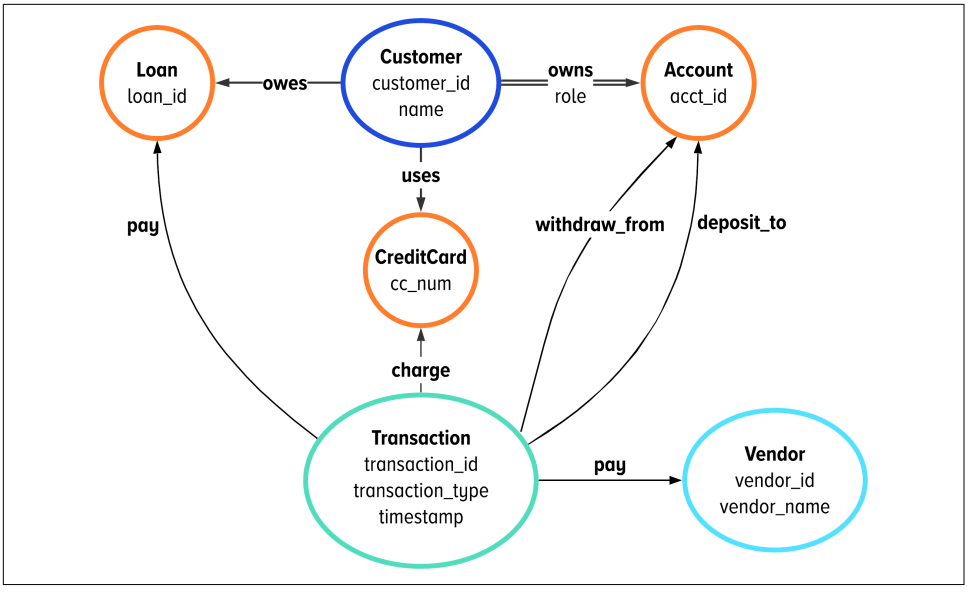


Figure 5-1. The developmental data model for a graph-based implementation of a C360 application from the previous chapter

We need to connect two concepts together so you can see the whole picture. First, all of our queries have used the development traversal source `dev.V()`. The development traversal source in DataStax Graph enables you to walk around your data without worrying about indexing strategies. Second, our queries walk from an account vertex to transactions. The query in Example 5-1 uses the production traversal source `g.V()`. If you try to run the query in Example 5-1 in DataStax Studio, you will see something like the execution error in Example 5-1.

Example 5-1.

```
g.V().has("Customer", "customer_id", "customer_0"). // the customer
      out("owns").                                  // walk to their account(s)
      in("withdraw_from", "deposit_to")             // access all transactions
```

Table 5-1. An example of an execution error due to trying to walk an edge in the reverse direction without an index

Execution error
com.datastax.bdp.graphv2.engine.UnsupportedTraversalException:
One or more indexes are required to execute the traversal

This error is tied to the representation of graph data structures on disk. In the rest of this chapter, we take a peek under the hood to explain the *why* and then apply the *how*.

Chapter Preview: Understanding Distributed Graph Data in Apache Cassandra

The primary intent of this chapter is to introduce design and operational recommendations for modeling data efficiently prior to entering production. For that, this chapter builds on the example from Chapter 4 by detailing how graph data structures operate in Apache Cassandra.

At the end of this chapter, you will have a list of 10 data modeling recommendations to apply to any new problem. We will use these same tips throughout the remaining examples in this book, too.

We selected the next set of technical topics to illustrate the minimum required set of concepts for building production-quality distributed graph applications. This chapter has three main sections that align with the accompanying notebook and technical materials.

The first section of this chapter revisits the topics we used but did not explain in Chapter 4. Here, we introduce the fundamentals of distributed graph structures to model our queries from that chapter. Namely, you will learn about partition keys, clustering columns, and materialized views.

The second section applies the concepts of distributed graph structures to our second set of data modeling recommendations. We will introduce Cassandra topics such as denormalization, revisit edge direction, and talk about loading strategies. These tips represent data modeling decisions that we recommend for production-quality, distributed graph schema.

The last section walks through the final iteration of our C360 example. We will explain the schema code that applies the concepts of materialized views and indexing strategies. And we will go through one last iteration of our Gremlin queries to use the new optimizations.

Altogether, the thought process and development in Chapters 3, 4, and 5 represent the development life cycle of designing, exploring, and finalizing the models and queries for your first application with distributed graph data.

Let's get started by taking a final step down into the physical data layer of working with graph data in Apache Cassandra.

Working with Graph Data in Apache Cassandra

This section looks at the fundamental concepts of working with graph data structures in Apache Cassandra: primary keys, partition keys, clustering columns, and materialized views.

We are going to discuss these Cassandra data modeling topics from a graph user's perspective.

First, we will talk about what you need to know about vertices, and then we will go over what you need to know about edges. For vertices, you need to know about primary keys and partition keys. For edges, you need to know about clustering columns and materialized views.

Let's get started with the concept that connects everything: the primary key.

The Most Important Topic to Understand About Data Modeling: Primary Keys

A major challenge of building a good data model within a distributed system is determining how to uniquely identify your data with primary keys.

You have already worked with one of the simplest forms of a primary key: the partition key.

Partition key
: The partition key is the first element of a primary key in Apache Cassandra. The partition key is the part of the primary key that identifies the location of the data in a distributed environment.

From a user's perspective, the entire primary key is required for you to access your data from the system. The partition key is just the first piece of the primary key.

Primary key
: The primary key describes a unique piece of data in the system. In DataStax Graph, a primary key can be made up of one or more properties.

You have already been using and working with primary and partition keys. In DataStax Graph, you specify the desired primary key in the schema API. We saw the simplest version of a primary key—just one partition key—in the previous chapter with:

```
schema.vertexLabel("Customer").
       ifNotExists().
       partitionBy("customer_id", Text). // basic primary key: one partition key
       property("name", Text).
       create();
```

The `partitionBy()` method indicates the value that will be included in the label's partition key. In this case, we have only one value, `customer_id`. This means that `customer_id` is the full primary key and partition key for the `Customer` vertex.

From a developer's perspective, this decision has three consequences for your application. First, the value for `customer_id` uniquely identifies the data. Second, your application will need the value for `customer_id` to read the data about the customer. We will cover the third point in a moment.

These two consequences govern how you, the user, design your data's primary and partition keys. Let's take a look at an example. Previously, you used your primary key to look up this data in Gremlin via:

```
g.V().has("Customer", "customer_id", "customer_0").
      elementMap()
```

This returns:

```
{
  "id": "dseg:/Customer/customer_0",
  "label": "Customer",
  "name": "Michael",
  "customer_id": "customer_0"
}
```

Looking up vertices or edges by their full primary key is the fastest way to read data in DataStax Graph. This is one of the main reasons that selecting a good partition and primary key for your data is so important.

There is a third consequence of the partition key in Apache Cassandra. A vertex label's partition key assigns your graph data to a host within a distributed environment. Partition keys also give you different ways you can colocate your graph data. Let's dig into the details.

Partition Keys and Data Locality in a Distributed Environment

We recommend this section if you like getting deep in the weeds.

This section aims to synthesize topics across the Cassandra and graph communities. We will explore some hypothetical alternatives to graph partitioning by examining different partition key choices to colocate graph data. We will conclude with the partition strategy we started with for our example, but you will have gained a better understanding of the effects of partition key design and the graph partitioning problem.

And we need to be pedantic about what we really mean for a brief moment.

The word *partition* means two very different things to two different groups of people. The Cassandra community's understanding of *partition* answers the question,

"Where is my data in my cluster?" The graph community's understanding of the term answers the question, "How can I organize my graph data into a smaller group to minimize an invariant?"

This book applies the Cassandra community's definition of partitioning to working with graph data. When we refer to a partition, we are referencing data locality, or on which server your data is written to disk across your distributed system.

To illustrate how we will be using the idea of partitioning, let's recall some data for our current example, as shown in Figure 5-2.

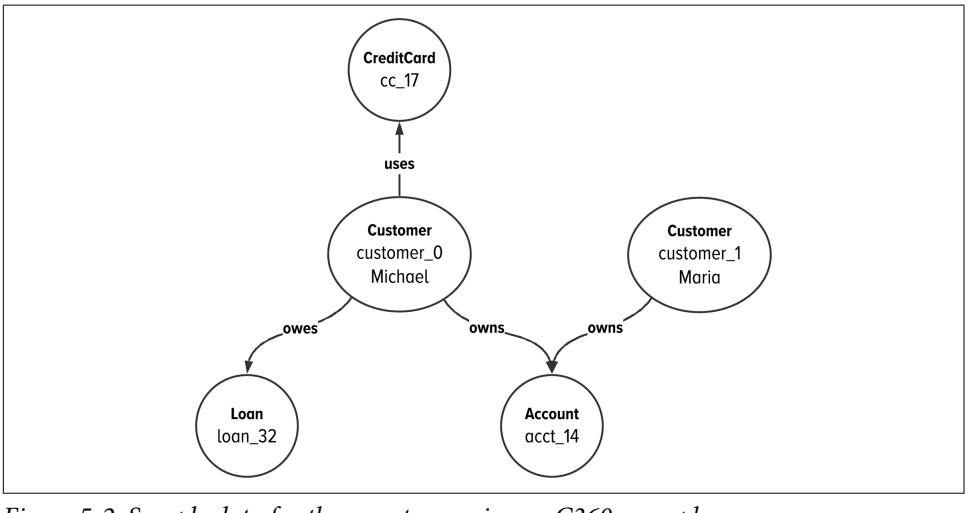


Figure 5-2. Sample data for three customers in our C360 example

To visualize data assignment to a server (also referred to as an instance or a node) in a cluster, imagine you are working with a cluster of four servers running DataStax Graph in Apache Cassandra. In Figure 5-3, we represent a distributed cluster with a circle that has four servers running Cassandra. (Each eye in Figure 5-3 represents DataStax Graph in Cassandra.) Then, we show where your graph data is written to disk by illustrating the graph data next to the server around your cluster, as we do in Figure 5-3.

The largest circle in Figure 5-3 represents a cluster of four servers, each indicated with the Cassandra eye logo, running DataStax Graph in Apache Cassandra. The sample data from Figure 5-2 is shown next to the server in which the data is physically stored. In Apache Cassandra, data is mapped to a specific server in your cluster according to its partition key.

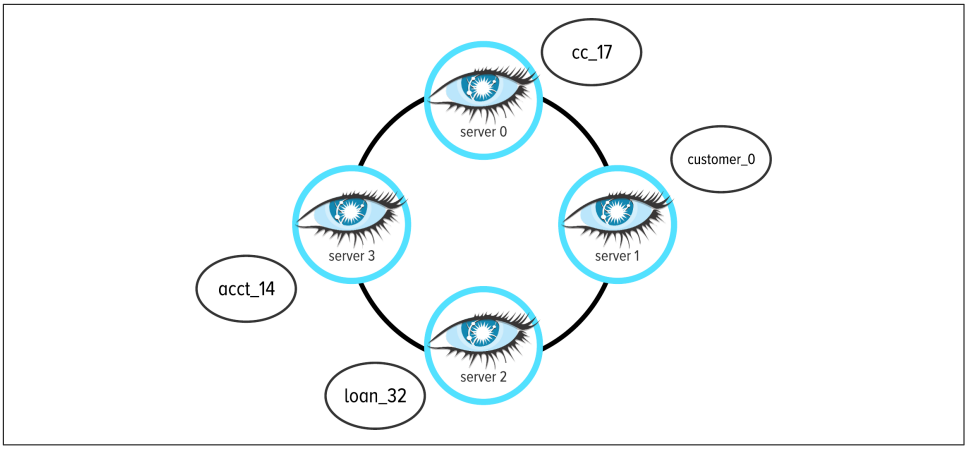


Figure 5-3. Illustrating which server (node) each vertex is stored in a distributed cluster; the circle with four eyes represents a distributed cluster

In Figure 5-3, you see that the data for `customer_0` is mapped to four different machines. The customer vertex is written to server 1, the loan vertex is on server 2, the account vertex is written to machine 3, and the credit card vertex is on machine 0.

You can think of partition keys and their association to data locality in a distributed environment as follows: data with the same partition key is stored on the same machine, and data with different keys may be stored on different machines.

Partitioning graph data according to access pattern

With graph data, there are strategies for designing your partition keys to minimize the latency of your graph traversals. Different partitioning strategies affect the colocation of your data and, therefore, the latency of your query.

To minimize jumping around machines in your cluster when you are processing your graph data, you may consider a partitioning strategy that keeps all the data related to your query within the same partition. To illustrate this idea, Figure 5-4 shows a partitioning strategy optimized for the expected access pattern of a C360 application. The partitions are defined according to the individual customer and their data because a C360 query will typically be looking for an individual and their associated data. For our sample data, we would create a partition for each individual.

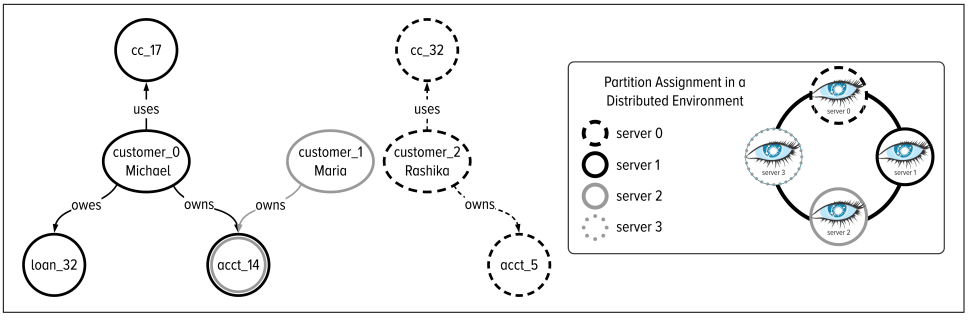


Figure 5-4. Partitioning graph data in a distributed system according to access pattern

If you have a background in graph theory, the partitioning strategy illustrated in Figure 5-4 is similar to partitioning according to connected components.

If you have a background in working with Apache Cassandra, the partitioning strategy illustrated in Figure 5-4 follows the same practice of partitioning according to access pattern.

To implement the partitioning strategy illustrated in Figure 5-4, you would need to add the customer's unique identifier as the partition key for every vertex label. In your schema code, we implement the partitioning strategy with:

```
schema.vertexLabel("Account").
       ifNotExists().
       partitionBy("customer_id", Text).
       clusterBy("acct_id", Text).  // to be defined in a coming section
       property("name", Text).
       create();
```

It is useful to consider this partitioning strategy because it minimizes the latency of your query. All the data for your customer-centric query is colocated on the same node in your environment. This is an optimization at the physical data layer that will be advantageous when you query your data.

However, there are two reasons why this type of partition strategy will not be recommended for the queries we are exploring for our example. Recall that the full primary key is required to start your graph query. The first downside to the design shown in Figure 5-4 is that you will need to know the customer's identifier to start your graph traversal at an account.

Applying this to our example of a shared account, `acct_14`, brings us to another drawback to using this partition strategy. This schema design will create two vertices about `acct_14` that are adjacent to two different people. This means that you won't be able to start at `acct_14` and find all customers who own that account. This has implications for your graph query.

For the C360 queries we are exploring in this example, the partition strategy from Figure 5-4 doesn't make sense. When we talk about trees in an upcoming example, however, it makes sense to consider data model optimizations to minimize query latency.

Partitioning according to unique key

Let's look at a second strategy and compare it to colocating data according to your application's access pattern.

Think back to the full schema for our example and recall that each vertex label had a single, unique partition key. You can think of this as separating your graph data via the most granular division possible: the data's most unique value.

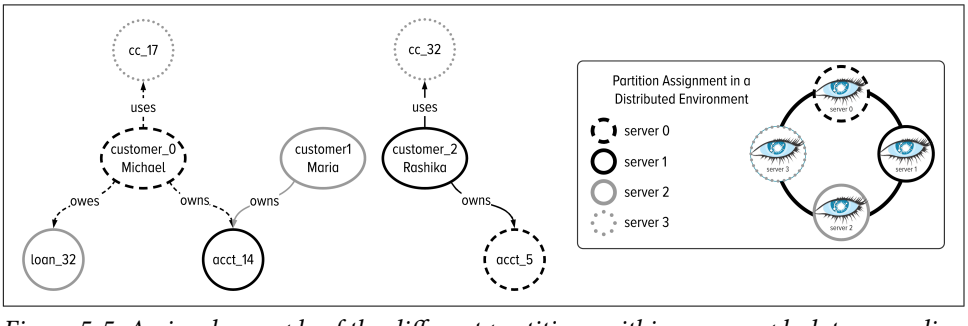


Figure 5-5. A visual example of the different partitions within your graph data according to the vertex label's partition key

The graph data in Figure 5-5 would distribute the vertices across your cluster according to the partition key's value. Essentially, each vertex will be mapped to a different partition because each partition key's value is unique.

One of the drawbacks to partitioning your vertices according to a unique key is that any time you need to walk through your data, you will be jumping between machines across your distributed environment. The purpose of using graph data in your application is to use the connections and relationships in your data. If you structure your graph data across a distributed environment according to unique identifiers, that also means that you will (likely) be switching servers each time you need to access connected data.

Final thoughts on partitioning strategies

There are benefits and drawbacks to the different strategies for partitioning graph data in a distributed environment. Partitioning your graph data according to access pattern creates limitations on how you can walk through connected data. On the other hand, this strategy minimizes traversal latency by colocating large components of data onto the same node.

The most common way to partition your graph data is by the data's unique identifier. This makes it easiest to plan for query flexibility but does also introduce latency to your queries due to the distributed nature of your graph. This is the approach we will use for our C360 example.

The only way to understand the implications of any partition strategy is to calculate what it would look like for your data and queries. This requires a balance between understanding the distributions of the data for the application you need to build today and considering the future scope toward which you are building.

Selecting a good partitioning strategy is more complicated when we are working with graph data. Partitioning graph data around a distributed environment is synonymous with breaking up your graph data into different sections. Optimizing which data belongs to a particular section is classified as one of the hardest types of problems in computer science: an NP-complete problem. While maybe not the best news, this helps to explain why using graph technologies in a distributed environment isn't as simple as translating an entity-relationship diagram into a graph data model.

On the topic of partitioning, there are two main takeaways to restate here: uniqueness and locality. In DataStax Graph, your data's primary key is its unique identifier. For the fastest performance, you start your graph queries by data via its full primary key.

The second thing to note is that your data's partition key determines its locality in your cluster. This governs which machines in your cluster will store the data and the colocation of other data alongside it.

Given uniqueness and locality with partition keys, let's take a look at how edges are represented in Apache Cassandra.

Understanding Edges, Part 1: Edges in Adjacency Lists

Diving into the world of graph modeling brings a large wave of terms, concepts, and thinking patterns. Now that we have the basics covered, let's take a look at how graph data, namely the edges, can be represented on disk or in memory.

There are three main data structures for representing edges in data:

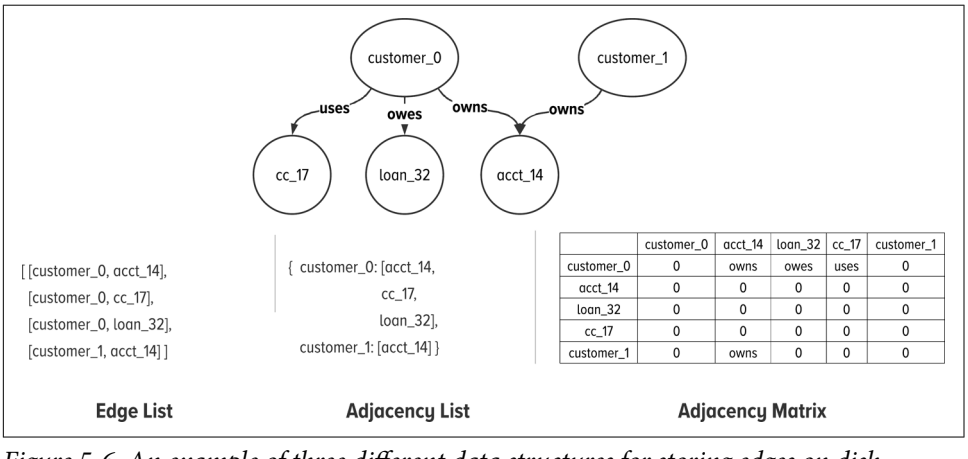
Edge list
: An edge list is a list of pairs in which every pair contains two adjacent vertices. The first element in the pair is the source (from) vertex, and the second element is the destination (to) vertex.

Adjacency list
: An adjacency list is an object that stores keys and values. Each key is a vertex, and the value is a list of the vertices that are adjacent to the key.

Adjacency matrix
: An adjacency matrix represents the full graph as a table. There is a row and column for each vertex in the graph. An entry in the matrix indicates whether there is an edge between the vertices represented by the row and column.

To understand these data structures, let's look at how we would map a small example of graph data into each structure.

There is a significant amount of detail illustrated in Figure 5-6. At the top, we show an example of five vertices that are connected by four edges. Direction matters when you map the data into each of the graph data structures below.



	customer_0	acct_14	loan_32	cc_17	customer_1
customer_0	0	owns	owes	uses	0
acct_14	0	0	0	0	0
loan_32	0	0	0	0	0
cc_17	0	0	0	0	0
customer_1	0	owns	0	0	0

Figure 5-6. An example of three different data structures for storing edges on disk

Let's walk through each data structure.

On the lower left in Figure 5-6, we have written out how the example data would be stored in an edge list. The edge list contains four entries: one entry per edge in our example data. In the center, we represent how the example data maps to an adjacency list. The adjacency list has two keys: one key per vertex with outgoing edges. The value for each key is a list of the incoming vertices the edges point to. The last data structure, shown on the far right, is an adjacency matrix. There are five rows and five columns: one row or column for each vertex in the graph. Each entry in the matrix indicates whether there is an edge going from the row vertex to the column vertex.

There are space and time trade-offs for each data structure. Skipping over optimizations that can be made for each individual data structure, let's consider the complexities of each at a basic level. Edge lists are the most compressed version of representing

your graph, but you have to scan the entire data structure to process all edges about a specific vertex. Adjacency matrices are the fastest way to walk through your data, but they take up an inordinate amount of space. Adjacency lists combine the benefits of the other two models by providing an indexed way to access a vertex and limit the list scans to only the individual vertex's outgoing edges.

In DataStax Graph, we use Apache Cassandra as a distributed adjacency list to store and traverse your graph data. Let's dig into how we optimize the storage of edges on disk so you can get the most benefit out of the sorting order of your edges during your graph traversals.

Understanding Edges, Part 2: Clustering Columns

You used the concept of clustering columns when you added edge labels to your graph in Chapter 4.

Clustering column
: A clustering column determines a sorting order of your data in tables on disk.

Clustering columns make up the final components of a table's primary key in Cassandra. Clustering columns inform the database how to store the rows in a sorted order on disk, which makes data retrieval more efficient.

We want to dig into the details of clustering columns because they explain two concepts at the same time. First, the technical implications of clustering columns detail exactly why the query at the beginning of this chapter returned an error. Second, clustering columns illustrate how we sort your edges on disk in an adjacency list structure to provide the fastest access possible.

Example 5-2 illustrates the use of a clustering column in creating an edge label.

Example 5-2.

```
schema.edgeLabel("owns").
       ifNotExists().
       from("Customer").   // the edge label's partition key
       to("Account").      // the edge label's clustering column
       property("role", Text).
       create()
```

Following the example in Example 5-2, we can pick out the partition key and clustering columns for our edge label:

1. The `from(Customer)` step means that the full primary key of the `Customer` vertex will be the partition key for the edge label `owns`: `(customer_id)`.
2. The full primary key for `Account` will be the clustering column for the `owns` edge: `(acct_id)`.

Putting this together, we can lay out Cassandra's table structures alongside the graph schema, as see in Figure 5-7.

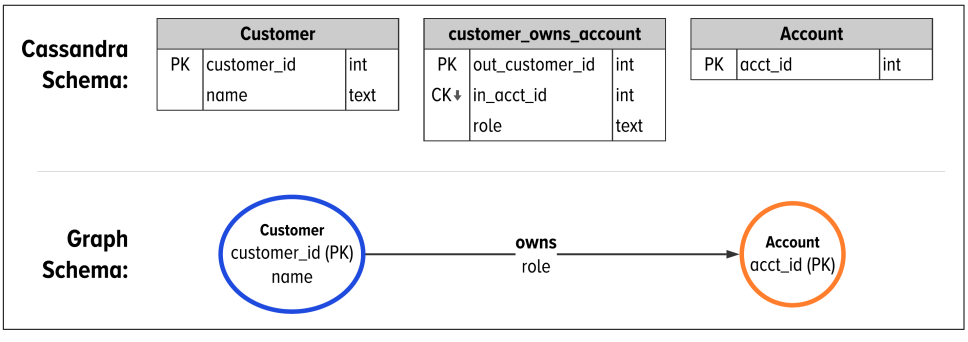


Figure 5-7. The default table structure in Cassandra for an edge between two vertices

Figure 5-7 shows the table structures in Cassandra as they map to graph schema using the Graph Schema Language (GSL). The `Customer` vertex creates a table with a partition key, `customer_id`. The `owns` edge connects `Customer` to `Account`. The partition key of the `owns` edge is the `customer_id`. The `owns` edge also has a clustering key, `in_acct_id`, which is the partition key of the account vertex. There is a third column in the `customer_owns_account` table: `role`. This is a simple property and is not a part of the primary key. As a result, the value for `role` will come from the most recent write of an edge between a customer and an account.

To make this concrete, Figure 5-8 shows an example of data that follows the schema from Figure 5-7.

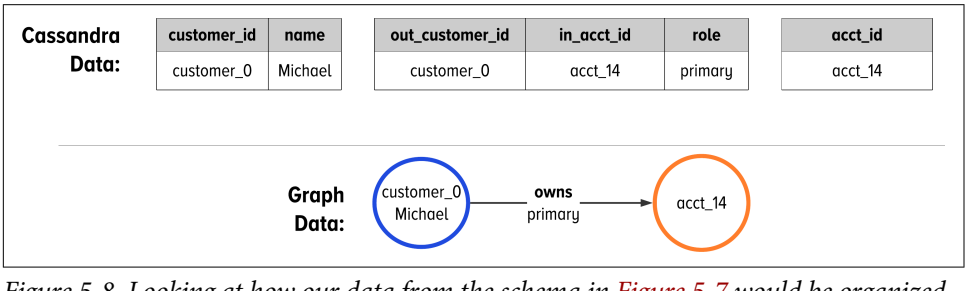


Figure 5-8. Looking at how our data from the schema in Figure 5-7 would be organized on disk and how it would be represented in DataStax Graph

Before we move on to a different topic, there is one last idea to synthesize about clustering keys and edges in DataStax Graph. In Chapter 2, we outlined cases in which you want to have many edges between two vertices. We denote this in the GSL with a double-lined edge. In Cassandra, we would make that property a clustering key. Figure 5-9 shows the Cassandra schema alongside a graph schema that models adjacent vertices as a collection.

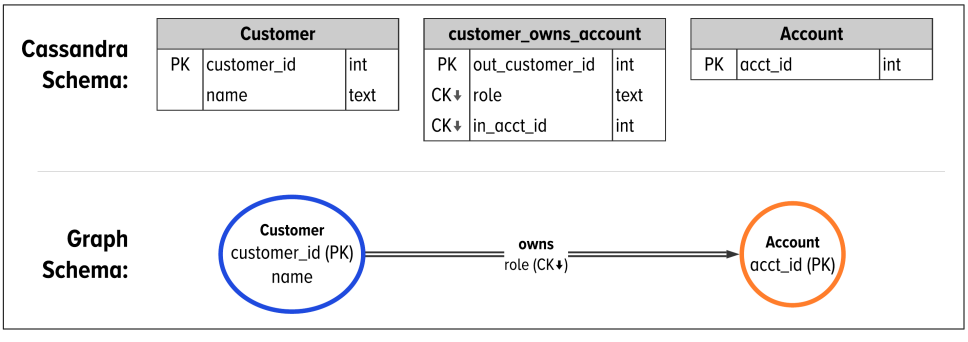


Figure 5-9. The table structure in Cassandra when we model clustering keys on edges

Figure 5-10 shows the table structure in Cassandra as they map to graph schema using the GSL when we model the multiplicity of a graph with many edges between instance vertices. The difference is in the table for the `owns` edge. We now have the `role` as a clustering key for this edge, before the clustering key for the `acct_id`. The schema from Figure 5-9 allows there to be multiple edges between vertices, as we show in Figure 5-10.

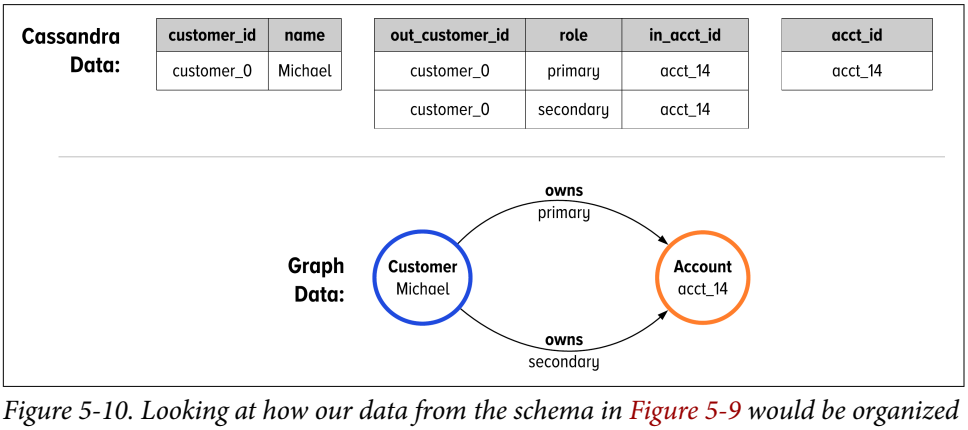


Figure 5-10. Looking at how our data from the schema in Figure 5-9 would be organized on disk and how it would be represented in DataStax Graph

Now that we understand the structure of edges on disk, let's visit where they will be stored within a distributed environment.

Synthesizing concepts: Edge location in a distributed cluster

Recall that the partition key identifies where the data will be written within the cluster. This means that the outgoing edges for a vertex will be stored on the same machine as the vertex itself. We previewed this in Figure 5-5 because the edges have the same color as the customer vertices; they are all orange. To illustrate what we mean, let's look at the locality of edges in our cluster, as shown in Figure 5-11.

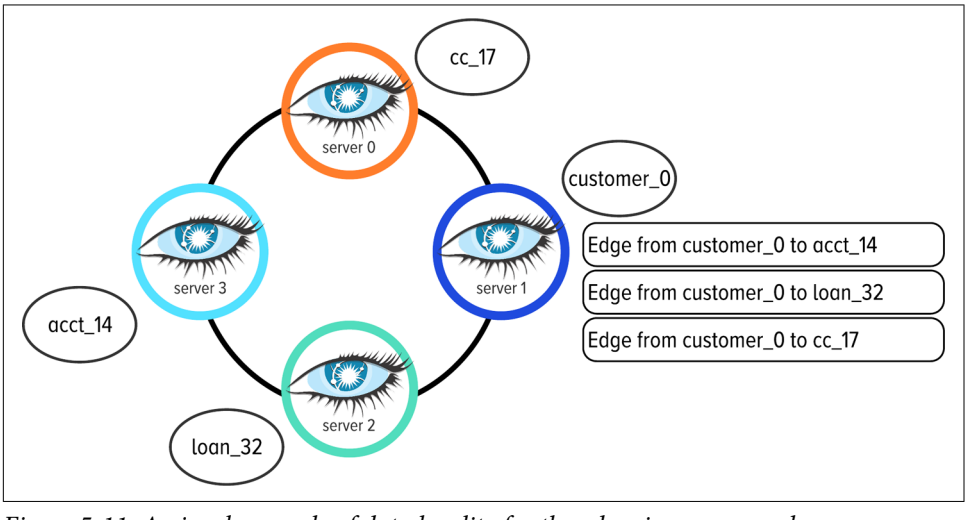


Figure 5-11. A visual example of data locality for the edges in our example

The image in Figure 5-11 illustrates where the edges for `customer_0` will be stored within a distributed environment. Each of the edges will be colocated on the same machine as the vertex for `customer_0` because each edge has the same partition key: the `customer_id`.

The next thing to understand is how the edges are sorted within their partition. The full primary key of the *adjacent* vertex label becomes the clustering column(s) of the edge label. This means that the edges are sorted on disk according to their incoming vertex's primary key, as visualized in Figure 5-12.

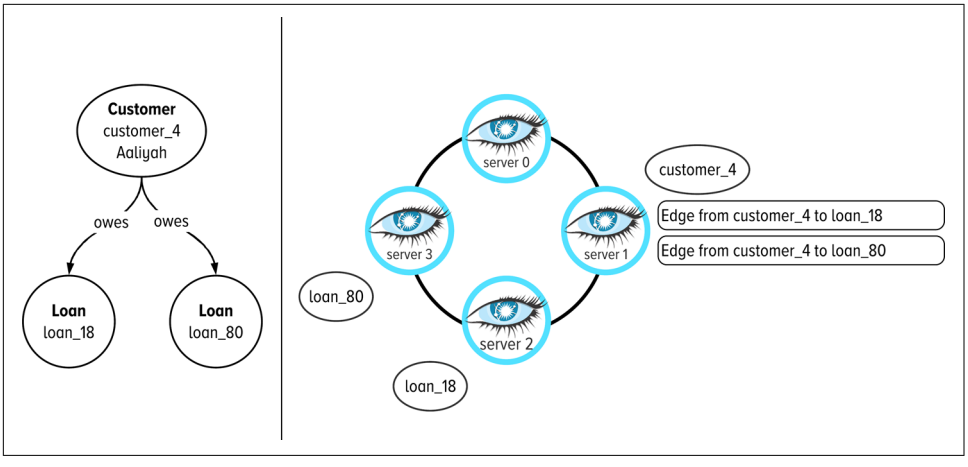


Figure 5-12. A visual example of mapping graph data, on the left, to its storage location in a distributed cluster, on the right

The main concept to understand from Figure 5-12 is illustrated on the right. We are showing that the vertex for `customer_4`, Aaliyah, is written to disk on machine 1 in our cluster. Also on machine 1, we will find the outgoing edges from Aaliyah sorted according to their incoming vertex. Aaliyah has two loans connected to her with an `owes` edge. We see that on disk, these edges will be sorted according to the incoming vertex's partition key, the `loan_id`. We see `loan_18` is the first entry and `loan_80` is the second entry.

To check whether you are synthesizing concepts: where would the customer vertices for Michael, Maria, Rashika, and Jamie be in Figure 5-12? Answer: The partition key for each of those vertices is their `customer_id`, which would be hashed and mapped to any one of the servers. Because we are working with five customers in total, there will be at least one server with two customer vertices. This logic is referred to as the "pigeonhole principle" in mathematics.

You might be asking yourself: why are we getting into all this? It all comes down to the minimum requirement for accessing a piece of data in Apache Cassandra: the partition key.

Understanding Edges, Part 3: Materialized Views for Traversals

The main area in which you are going to feel the effects of an edge's primary key design comes into how you access your edges. To use an edge, you have to know its partition key.

Because of this, we cannot yet traverse our edges in the reverse direction! This is because there are no edges in the system that start with the partition key from the incoming vertex labels in our examples.

Remember our query from Example 5-1?

```
g.V().has("Customer", "customer_id", "customer_0"). // the customer
      out("owns").                                  // walk to their account(s)
      in("withdraw_from", "deposit_to") // walk to all transactions
```

Recalling the schema we built in the previous chapter, the `deposit_to` edges point from a `Transaction` to an `Account`. However, this query is trying to walk that edge in the reverse direction: from the `Account` to the `Transaction`.

Applying what we just learned about edges in DataStax Graph, we know that this error happens because the edge does not exist on disk. The edge was written from the transaction to the account, but not the reverse.

If we want to walk from accounts to transactions, then we need to store the edge in the other direction as well. This is not done by default in DataStax Graph because of performance implications, similar to how indexing every column in a relational data model is an antipattern.

What we need are bidirectional edges, or edges that go in both directions. This option brings us to the last technical topic in this chapter.

Materialized views for bidirectional edges

One of the main reasons engineers love Apache Cassandra is they are willing to trade data duplication for faster data access. This is where materialized views come into play with DataStax Graph. From the user's perspective, you can think of a materialized view as follows:

Materialized view
: A materialized view creates and maintains a copy of the data in a separate table with a different primary key structure, rather than requiring your application to manually write the same data multiple times to create the access patterns you need.

Under the hood, DataStax Graph uses materialized views to be able to walk an edge in its reverse direction.

To demonstrate, Example 5-3 shows how to create a materialized view on the existing edge label for `deposit_to`.

Example 5-3.

```
schema.edgeLabel("deposit_to").
       from("Transaction").
       to("Account").
       materializedView("Transaction_Account_inv").
       ifNotExists().
       inverse().
       create()
```

Example 5-3 creates a table in Apache Cassandra called `"Transaction_Account_inv"`. The partition key for this table is the `acct_id`. The clustering column is `transaction_id`.

The full primary key from Example 5-3 is written as `(acct_id, transaction_id)`. This notation means that the full primary key contains two pieces of data: `acct_id` and `transaction_id`. The first value, `acct_id`, is the partition key, and the second value, `transaction_id`, is the clustering column.

From the user's perspective, this gives us the ability to walk through the `deposit_to` edge from accounts to transactions. To convince ourselves of this, let's see the edges that are stored between these two data structures by inspecting the data on disk.

We can inspect the edges on disk for the `deposit_to` edge label by querying the underlying data structures in Apache Cassandra. There are two tables to inspect. First, let's look at the original table for `Transaction_deposit_to_Account`; you can do this from DataStax Studio with the following (the results are shown in Table 5-2):

```
select * from "Transaction_deposit_to_Account";
```

Table 5-2. The data layout from the table `Transaction_deposit_to_Account`

Transaction_transaction_id	Account_acct_id
220	acct_14
221	acct_14
222	acct_0
223	acct_5
224	acct_0

The following query shows how to list all edges on disk for the materialized view of the `deposit_to` edge label, and Table 5-3 displays the results:

```
select * from "Transaction_Account_inv";
```

Table 5-3. The data layout from the table `Transaction_Account_inv`

Account_acct_id	Transaction_transaction_id
acct_0	222
acct_0	224
acct_5	223
acct_14	220
acct_14	221

Let's look very closely at the differences between Table 5-2 and Table 5-3. The easiest one to spot is the transaction involving `acct_5`. In Table 5-2, we see that the partition key for this edge is `out_transaction_id`, which is `223`. The clustering column is `in_acct_id`, which is `acct_5`.

Examine how this same edge is stored in Table 5-3, the materialized view of Table 5-2. We can see that the edge's keys are flipped; the partition key for this edge is `in_acct_id`, which is `acct_5`, and the clustering column is `out_transaction_id`, which is `223`. We now have bidirectional edges to use in our example.

How far down do you want to go?

We just walked through all of the technical explanations for topics in Apache Cassandra that we have planned for this book. Our explanations of the technical concepts are intentionally only a surface-level introduction to the internals of Apache Cassandra, presented from the perspective of a graph application engineer. There is much more to understand about partition keys, clustering columns, materialized views, and more within distributed systems.

We encourage you to go deeper and can recommend two other resources to get you there.

First, for a deep dive on the internals of Apache Cassandra, consider picking up a different O'Reilly book: *Cassandra: The Definitive Guide*, Third Edition by Jeff Carpenter and Eben Hewitt (O'Reilly).

Or for a complete examination of the internals of distributed systems, check out Alex Petrov's *Database Internals: A Deep Dive into How Distributed Data Systems Work* (O'Reilly).

Where are we going from here?

We are coming back up from the internals of distributed graph data for one last pass of our C360 example. Applying the concepts we have discussed can give us more data modeling recommendations, schema optimizations, and a few new ways to implement our Gremlin queries. So that is exactly where we are going.

The upcoming section applies our knowledge of keys and views in Apache Cassandra to data modeling best practices with DataStax Graph.

Graph Data Modeling 201

The new knowledge of the layout of vertices and edges in DataStax Graph opens up more data modeling optimizations. Let's apply our understanding of partition keys, clustering columns, and materialized views and visit our second set of data modeling recommendations (picking up from the six recommendations provided in Chapter 4).

To begin with, let's recall where our graph schema left off—see Figure 5-13.

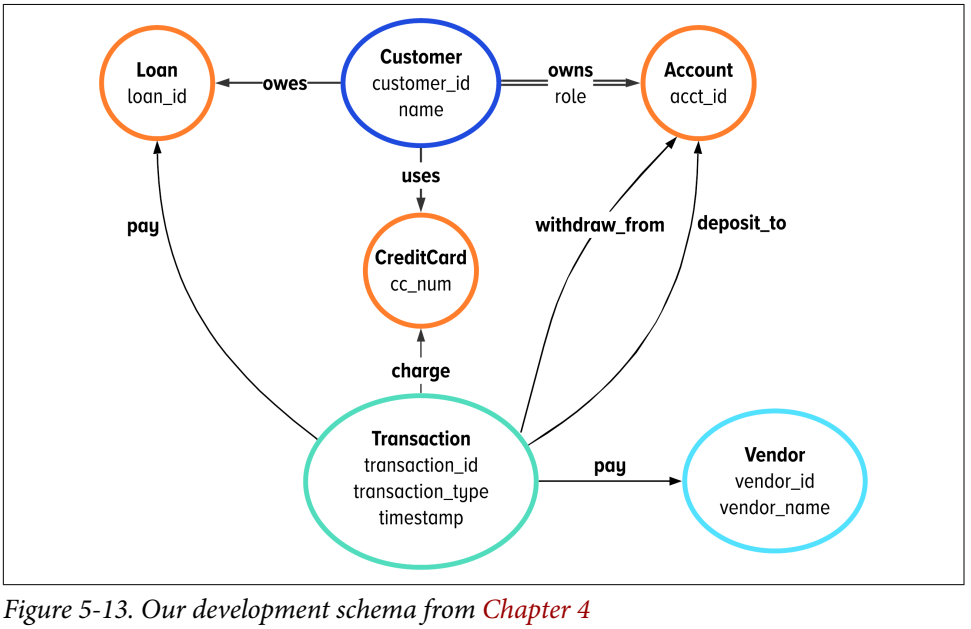


Figure 5-13. Our development schema from Chapter 4

This brings us to our next data modeling recommendation.

Rule of Thumb #7

Properties can be duplicated onto edges or vertices; use denormalization to reduce the number of elements you have to process in a query.

To apply this tip, consider a case in which an account has thousands of transactions. When we want to find the most recent 20 transactions, we need to access the account vertex by walking through all transactions *before* we can subselect the vertices by time. It is pretty expensive to traverse all of the edges to access all transactions and *then* sort the transaction vertices.

Can we be smarter and reduce the amount of data we have to process?

We can. Specifically, we can store a transaction's time in two places: on the transaction vertex and on the edges. This way, we can subselect the edges to limit our traversal to only the most recent 20 edges. Figure 5-14 illustrates duplicating time onto an edge label.

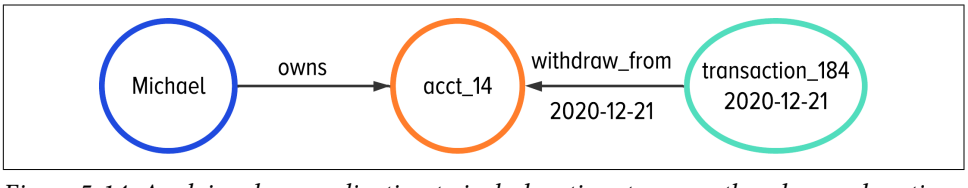


Figure 5-14. Applying denormalization to include a timestamp on the edges and vertices as an optimization to improve read performance

For simplicity's sake, in Figure 5-14 we show only the addition of a timestamp to the `withdraw_from` edge; we will apply the same technique for the `deposit_to` and `charge` edge labels.

This type of optimization requires your application to write the same timestamp onto both the edge and the vertex. This is called *denormalization*.

Denormalization
: Denormalization is the strategy of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data grouped differently.

Duplicating properties, or denormalization, is a very popular strategy that balances the dualities between unlimited query flexibility and query performance. On one hand, modeling your data in a graph database allows for more flexibility and easier integration of data sources. This flexibility is one of the main reasons teams are picking up graph technologies; graph technology inherently integrates more expressive modeling and query languages.

On the other hand, poor planning during development has left many teams before you with unrealistic expectations for their production graph model. They focused more on data model flexibility at the expense of query performance. Your queries will be more performant if you take advantage of modeling tricks like denormalization.

Before you start adding properties and materialized views to all of your edges, consider our next recommendation.

Rule of Thumb #8

Let the direction you want to walk through your edge labels determine the indexes you need on an edge label in your graph schema.

With this tip, we are asking you to do a few things. First, we are advising you to work out your Gremlin queries in development mode first, just like we did in Chapter 4. Then we can apply those final queries to determine *only* the materialized views that you need. You don't need indexes for everything.

There are two ways to do this in DataStax Graph: you can do it yourself, or you can tell the system to do it.

Let's start with what it would look like if you were to figure out indexes on your own.

To recognize when you need an index, you have to map your Gremlin query onto your graph schema. Mapping a query onto schema is something we've been mentally practicing throughout this book, but let's see what this looks like drawn out in Figure 5-15. We will draw out our first query's steps in our schema from start to end. Then, we use the query steps overlaid on our schema to identify where we will need an edge index. Figure 5-15 depicts a query's steps drawn over schema followed by Example 5-4, which shows the Gremlin query.

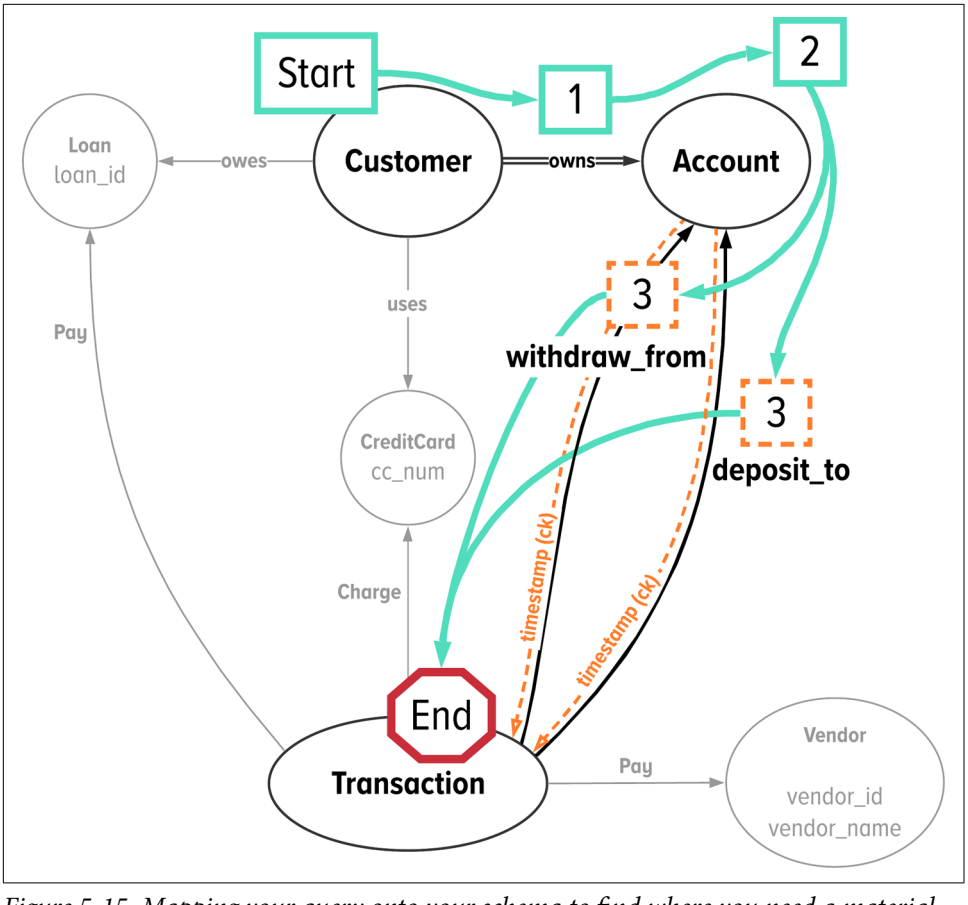


Figure 5-15. Mapping your query onto your schema to find where you need a materialized view on an edge label

Example 5-4.

```
dev.V().has("Customer", "customer_id", "customer_0").  // [START]
        out("owns").                                   // [1 & 2]
        in("withdraw_from", "deposit_to"). // [3]
        order().                           // [3]
          by(values("timestamp"), desc).   // [3]
        limit(20).                         // [3]
        values("transaction_id")           // [END]
```

Let's break down what we are showing in Figure 5-15 alongside Example 5-4. We mapped each step of the query to the schema that you walk through during the query. The boxes labeled from `Start` to `End` map a green path through the schema elements to match the query's steps to where we are walking throughout our schema.

The walk through our schema can be thought of as follows. We begin the traversal by uniquely identifying a customer, shown in the query and schema with the `Start` box. This is line 1 in our query. Then we use the `owns` edge to access that customer's account; this is shown in the boxes labeled `1` and `2`. This is line 2 in our query. Box `3` maps together the processing and sorting of transactions. This maps to lines 3, 4, 5, and 6 in our query. `End` labels where the traversal stops, on line 7 of our query.

The most important concept in Figure 5-15 is at step `3`. The query walks through the incoming `withdraw_from` and `deposit_to` edge labels to access the `Transaction` vertex label. However, we are walking *against* the direction of these edge labels in our schema. We highlighted this in Figure 5-15 with orange dotted lines.

Being able to mentally see that we are walking against the direction of an edge label identifies where you need a materialized view in your graph. This is a very important concept that we hope you followed from Figure 5-15 alongside Example 5-4. We think of this last example as one of the most fundamental aha moments for understanding graph data in Apache Cassandra, and we hope you got there.

Finding Indexes with an Intelligent Index Recommendation System

If juggling all of this in your head is new or does not feel natural, there is another way: you can let DataStax Graph do it for you.

DataStax Graph has an intelligent index recommendation system called `indexFor`. To let the index analyzer figure out what indexes a particular traversal requires, all you need to do is execute `schema.indexFor(<your_traversal>).analyze()` using the query we walked through in Figure 5-15:

```
schema.indexFor(g.V().has("Customer", "customer_id", "customer_0").
                      out("owns").
                      in("withdraw_from", "deposit_to").
                      order().
                        by(values("timestamp"), desc).
                      limit(20).
                      values("transaction_id")).
       analyze()
```

Because we already created a materialized view for `deposit_to`, this command will output only one recommendation. The output contains the following information, reformatted here to make it easier to read:

```
Traversal requires that the following indexes are created:
schema.edgeLabel("withdraw_from").
    from("Transaction").
    to("Account").
    materializedView("Transaction__withdraw_from__Account_by_Account_acct_id").
    ifNotExists().
    inverse().
    create()
```

Essentially, Figure 5-15 and `indexFor(<your_traversal>).analyze()` are doing the same thing. They are mapping your traversal onto your schema to see where you need a materialized view.

After you develop all of your queries, as we did in Chapter 4, you can use either technique to figure out where you will need indexes in your production schema. The manual approach can be useful for figuring out the default direction you should use for an edge label. If you only use `indexFor(…).analyze()`, you could end up with a bunch of indexes that may not be needed if some of the edges are simply turned around.

The next recommendation is for when you are first setting up your production database.

Rule of Thumb #9

Load your data; then apply your indexes.

We recommend loading data before applying indexes because this will significantly speed up your data loading process. The application of this recommendation depends on your team's deployment strategy.

This is a common loading strategy because of the popularity of blue-green deployment patterns for production graph databases. If this is the type of pattern you would like to use, we recommend loading data and then applying indexes. For a resource on deployment strategies to minimize system downtime, like the blue-green pattern, we recommend *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation* by Jez Humble and David Farley (Addison-Wesley).

There is one last tip to recommend.

Rule of Thumb #10

Keep only the edges and indexes that you need for your production queries.

Between development and production, you may find edge labels that you do not need for your traversals. That is expected. When you move your schema into production, get rid of the edge labels you are not going to use. Save some space on disk and the time spent persisting it.

Let's apply the new data modeling recommendations we just covered to the development schema we built up in Chapter 4. This will be the last time we use this example and sample data before we move into different graph models in future chapters.

Production Implementation Details

The remaining implementation details in this section represent the final production version of our C360 example.

First, we will add the required materialized views to the schema for our C360 example. Then we will go through an introduction of how to load data with DataStax Bulk Loader. Last, we will revisit and update our Gremlin queries to use the new optimizations.

Materialized Views and Adding Time onto Edges

We have a few changes to make to our development schema. First, we want to find areas where adding time onto our edges will reduce the amount of data we need to process in a query.

Let's visualize this in Figure 5-16 for the second query of our example. Figure 5-16 steps through the Gremlin query.

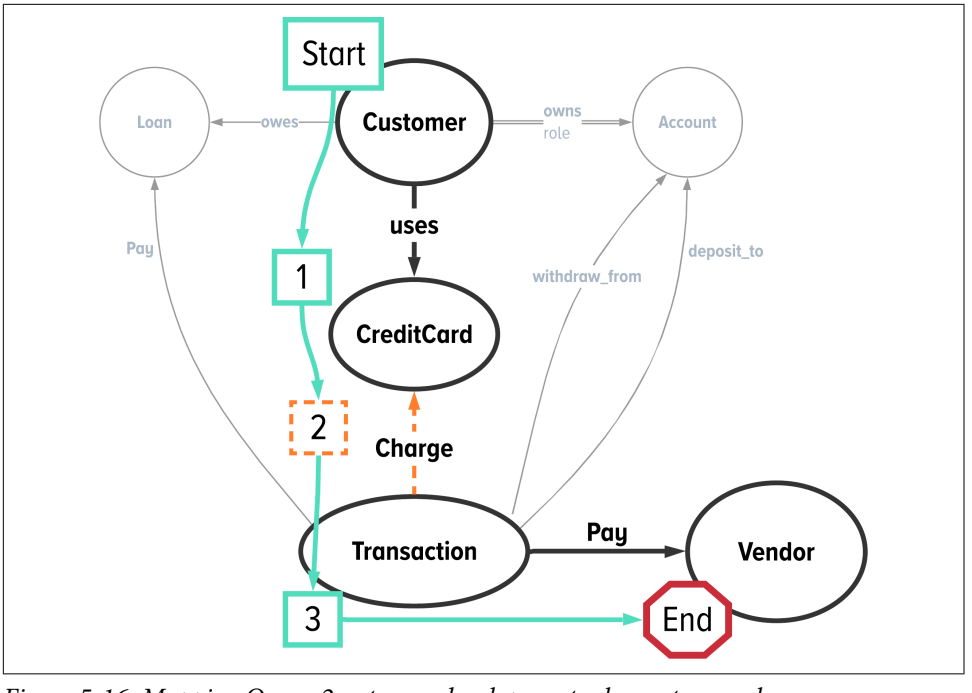


Figure 5-16. Mapping Query 2 onto our development schema to see where we can use denormalization to minimize the amount of data we need to process

Example 5-5.

```
dev.V().has("Customer", "customer_id", "customer_0").  // Start
        out("uses").                                   // 1
        in("charge").                                  // 2
        has("timestamp",                               // 2
            between("2020-12-01T00:00:00Z", // 2
            "2021-01-01T00:00:00Z")).                  // 2
        out("Pay").                                    // 3
        groupCount().                                  // End
        by("vendor_name").                             // End
        order(local).                                  // End
        by(values, decr)                               // End
```

Comparing Figure 5-16 with Example 5-5 illustrates two production schema strategies. First, we can apply denormalization to optimize this query. Currently, time is stored only on the `Transaction` vertex. We can reduce the number of edges required in this traversal if we denormalize the `timestamp` property and store it on the `charge` edge. This is illustrated in Figure 5-16 and Example 5-5 with the label `2`.

We also see in Figure 5-16 that our query walks against the direction of the `charge` edge. This means we need another materialized view on this edge label. The schema code is:

```
schema.edgeLabel("charge").
       from("Transaction").
       to("CreditCard").
       materializedView("Transaction_charge_CreditCard_inv").
       ifNotExists().
       inverse().
       create()
```

Following this same style of mapping, we can find three edge labels where denormalization can optimize our queries. This optimization minimizes the amount of data a traversal has to process by sorting the edges on disk. Specifically, we can minimize the amount of data required to process our traversals if we also add the `timestamp` property to the `withdraw_from`, `deposit_to`, and `charge` edge labels.

Our Final C360 Production Schema

We have been exploring through schema, queries, and data integration to iteratively introduce and build up our C360 example. Together, the technical concepts and previous discussions bring us to the final production schema for our C360 example shown in Figure 5-17.

The adjustment we applied here is to denormalize and add `timestamp` onto the edge labels that we use in our traversals.

The final version of the schema code for our edge labels is shown in Example 5-6.

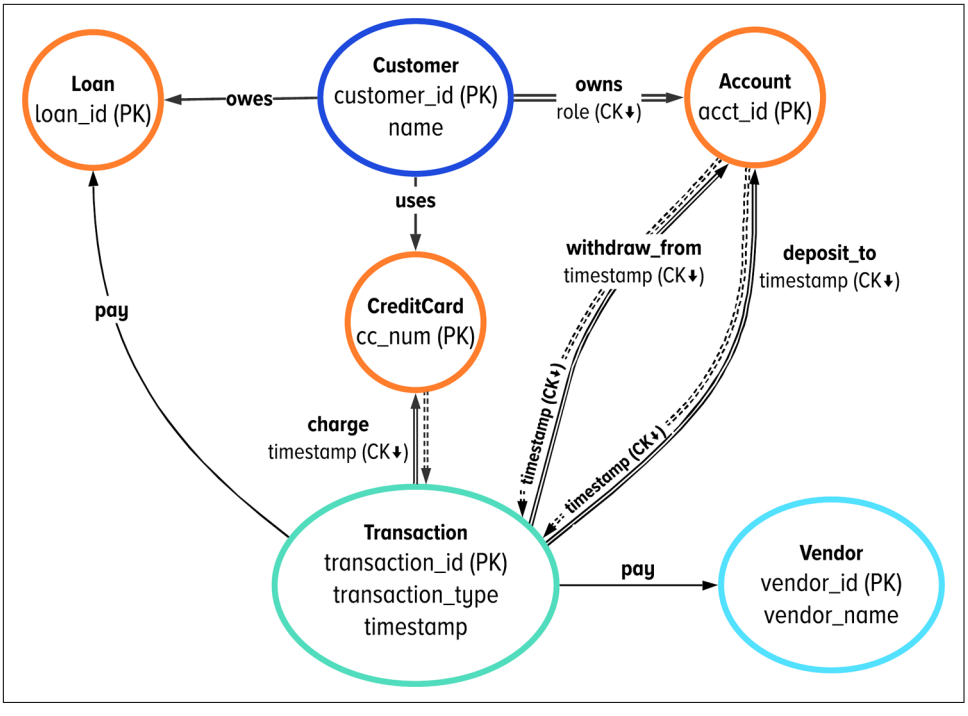


Figure 5-17. The starting data model for a graph-based implementation of a C360 application from the previous chapter

Example 5-6.

```
schema.edgeLabel("withdraw_from").
       ifNotExists().
       from("Transaction").
       to("Account").
       clusterBy("timestamp", Text). // sort the edges by time
       create();

schema.edgeLabel("deposit_to").
       ifNotExists().
       from("Transaction").
       to("Account").
       clusterBy("timestamp", Text). // sort the edges by time
       create();

schema.edgeLabel("charge").
       ifNotExists().
       from("Transaction").
       to("CreditCard").
       clusterBy("timestamp", Text). // sort the edges by time
       create();
```

To make the examples easier to follow in this book, we use `Text` to represent time and then query with strings such as `2020-12-01T00:00:00Z`. The `timestamp` property type uses less space on disk than `Text` and may be the best option for your final application.

Altogether, we need only the following changes from our development schema to our production schema:

1. Denormalize a property onto five edge labels
2. Add three materialized views to walk three edges in reverse

Let's detail how to use a bulk loading tool to insert the data into your graph database.

Bulk Loading Graph Data

We created a script that loads all of the data into DataStax Graph from CSV files. DataStax Bulk Loader is the fastest way to load data in production. We provided a CSV file for each vertex and edge label from our data model. Let's walk through the general process for loading vertices and then show the same for edges.

Loading vertex data with DataStax Bulk Loader

Let's look at all of the included vertex datafiles and a brief description for each file in Table 5-4.

Table 5-4. The full list of CSV files for the vertex data used in this chapter's examples

Vertex file	Description
Accounts.csv	The account IDs, one per line
CreditCards.csv	The credit card IDs, one per line
Customers.csv	Customer details, one per line
Loans.csv	The loan IDs, one per line
Transactions.csv	Transaction details, one per line
Vendors.csv	Vendor details, one per line

Let's see an example of how to load vertex data with DataStax Bulk Loader by examining `Transactions.csv`. The first five lines of `Transactions.csv` are shown in Table 5-5. Each line contains three pieces of information about the transaction that map to our expected schema. You also see in Table 5-5 that all transactions are loaded with an `unknown` type because one of our traversals is to mutate this property according to the graph's structure.

Table 5-5. The first five lines of data from the file `Transactions.csv`

transaction_id	timestamp	transaction_type
219	2020-11-10T01:00:00Z	unknown
23	2020-12-02T01:00:00Z	unknown
114	2019-06-16T01:00:00Z	unknown
53	2020-06-05T01:00:00Z	unknown

The most important line in Table 5-5 is the header. In the accompanying loading scripts, the header doubles as the mapping configuration between the file and the database. The header and the property names in DataStax Graph must match.

We can load the CSV file using the command-line bulk loading utility, as shown in Example 5-7.

Example 5-7.

```
1  dsbulk load -url /path/to/Transactions.csv
2              -g neighborhoods_prod
3              -v Transaction
4              -header true
```

Example 5-7 shows the most basic way to load vertex data on your localhost. The first part of line 1, `dsbulk load`, invokes the loading tool from the command line. The next four parameters, which can come in any order, are `-url`, `-g`, `-v`, and `-header`:

1. The `-url` parameter indicates where the CSV is stored.
2. `-g` is the name of the graph.
3. `-v` is the vertex label.
4. `-header` specifies that the data should be mapped according to the file's header.

The DataStax dsbulk documentation contains all the details for other loading options, including loading into a distributed cluster, configuration files, and much more.

Next, let's take a look at the edge data and loading process.

Loading edge data with DataStax Bulk Loader

All of the included edge datafiles and a brief description for each are listed in Table 5-6.

Table 5-6. The full list of CSV files for the edge data used in this chapter's examples

Edge file	Description
charge.csv	The `charge` edges from a `Transaction` to a `CreditCard`
deposit_to.csv	The `deposit_to` edges from a `Transaction` to an `Account`
owes.csv	The `owes` edges from a `Customer` to a `Loan`
owns.csv	The `owns` edges from a `Customer` to an `Account`
pay_loan.csv	The `pay` edges from a `Transaction` to a `Loan`
pay_vendor.csv	The `pay` edges from a `Transaction` to a `Vendor`
uses.csv	The `uses` edges from a `Customer` to a `CreditCard`
withdraw_from.csv	The `withdraw_from` edges from a `Transaction` to an `Account`

Let's see an example of how to load edge data with DataStax Bulk Loader by examining `deposit_to.csv`. The first five lines of `deposit_to.csv` are shown in Table 5-7. Each line contains three pieces of information about the deposit that map to our schema: the `transaction_id`, the `acct_id`, and a `timestamp`.

Table 5-7. The first five lines of data from the file `deposit_to.csv`

Transaction_transaction_id	Account_acct_id	timestamp
185	acct_5	2020-01-19T01:00:00Z
251	acct_5	2020-07-25T01:00:00Z
247	acct_5	2020-03-06T01:00:00Z
214	acct_14	2020-06-11T01:00:00Z

The most important line in Table 5-7 is the header; the header has to match the table schema in DataStax Graph. DataStax Graph autogenerates different column names for the edge properties that are part of the table's primary key. The generated name appends the vertex label to the front of the property name, such as `Transaction_` in front of `transaction_id` and `Account_` in front of `acct_id`.

We can load the edge CSV file using the command-line bulk loading utility, as shown in Example 5-8.

Example 5-8.

```
1  dsbulk load -url /path/to/Transactions.csv
2              -g neighborhoods_prod
3              -e deposit_to
4              -from Transaction
5              -to Account
6              -header true
```

Example 5-8 shows the most basic way to load edge data on your localhost. The first part of line 1, `dsbulk load`, invokes the loading tool from the command line, as we saw in the previous example. The next six parameters can come in any order: `-url`, `-g`, `-e`, `-to`, `-from`, and `-header`. The `-url` parameter indicates where the CSV is stored, `-g` is the name of the graph, `-e` is the edge label, `-from` is the outgoing vertex label, `-to` is the incoming vertex label, and `-header` says to map the data according to the file's header.

The accompanying scripts show how to load all vertex and edge labels for this chapter and all examples in this book. Please head to the data directory within book's GitHub repository for the data and loading scripts for each chapter.

You will see many more examples of bulk loading data into DataStax Graph throughout the rest of the book. For now, let's move on to the next stage of our implementation details: querying our graph with Gremlin.

Updating Our Gremlin Queries to Use Time on Edges

Now that we have updated our edge labels and indexes, let's revisit the queries and the results for each query. These are the same queries we walked through in Chapter 4, but there are two changes. First, we now can use the production traversal source `g`. We have moved out of development mode into writing queries against a production application. Second, we are going to update each query to use our new production schema. We will be using time on edges in addition to the materialized view.

Let's start by revisiting Query 1.

Query 1: What are the most recent 20 transactions involving Michael's account?

All of the work we did to set up the schema and graph data empowers the simplicity of the query in Example 5-9 to answer our first question.

Example 5-9.

```
g.V().has("Customer", "customer_id", "customer_0").
      out("owns").
      inE("withdraw_from", "deposit_to"). // uses materialized view on deposit_to
      order().                            // sort the edges
        by("timestamp",desc).             // by time
      limit(20).                          // walk through the 20 most recent edges
      outV().                             // walk to the transaction vertices
      values("transaction_id")            // get the transaction_ids
```

The results remain the same, but the query processed less data by sorting the edges:

```
    "184", "244", "268", ...
```

The main change from the query in Chapter 4 to this example can be seen with the addition of a single character: `E`. The query changed from using `in()` to `inE()`. This one character change uses a materialized view and the sorted order of edges.

To dig into the details, let's recall how we walked through this data in development mode. In Chapter 4, the `in()` step walked directly through edges, to the vertices, ignoring the edges' direction, and then sorted the vertex objects. That was simple enough for figuring out how to walk through our graph data.

In a production environment, we would need to ensure that this query processes only the data it needs. In Example 5-9, we optimized this query by using `inE()`, sorting all edges by time, and traversing only the 20 most recent edges.

The sorting of all edges requires three concepts from our schema. First, we use the materialized views we built on the `deposit_to` and `withdraw_from` edge labels. Second, we use the clustering key for `deposit_to` because the edges are ordered on disk by time. And last, we use the clustering key for the `withdraw_from` edge label because these edges are also ordered on disk by time.

That is a significant amount of optimization from just a small change: from `in()` to `inE()`. Let's look at what we need to do to our next query to take advantage of our new schema.

Query 2: In December, at which vendors did Michael shop, and with what frequency?

We are going to apply the same pattern to optimize our next query. We want to take advantage of the denormalization of time on the `charge` edge to minimize the amount of data we need to process. In Gremlin, this looks like Example 5-10.

Example 5-10.

```
g.V().has("Customer", "customer_id", "customer_0").
      out("uses").
      inE("charge").                          // access edges
          has("timestamp",                    // sort edges
              between("2020-12-01T00:00:00Z", // beginning of December 2020
              "2021-01-01T00:00:00Z")).       // end of December 2020
      outV().                                 // traverse to transactions
      out("pay").hasLabel("Vendor").          // traverse to vendors
      groupCount().
        by("vendor_name").
      order(local).
        by(values, desc)
```

The results are the same as before:

```
{
  "Target": "3",
  "Nike": "2",
  "Amazon": "1"
}
```

The change and optimization we applied in Example 5-10 follow the same pattern as Example 5-9. This time, we used `inE()` to access only incoming edges. We used the clustering key `timestamp` to apply a range function to the edges. Once we found all edges in a certain range, we moved to the transaction vertices and continued our traversal, as in Chapter 4.

This brings us to our last query from Chapter 4.

Query 3: Find and update the transactions that Jamie and Aaliyah most value: their payments from their account to their mortgage loan.

Let's think about the data this query is processing before we look at the final version of the query. In this query, we are starting from Aaliyah and finding all withdrawals from her accounts. There are no limits or time requests for this query; we want to find them all. This means that we will not be using any time ranges on the edges.

Further, every step along this query uses an existing outgoing edge label. Because of this, we do not need any materialized views and can walk out the existing edges to satisfy this query. Therefore, we need only to switch to our production traversal source, and this query will be ready to go—see Example 5-11.

Example 5-11.

```
g.V().has("Customer", "customer_id", "customer_4").    // accessing Aaliyah's vertex
      out("owns").                                     // walking to the account
      in("withdraw_from").                             // Only consider withdraws
      filter(
             out("pay").              // walking out to loans or vendors
             has("Loan", "loan_id", "loan_18")).       // only keep loan_18
      property("transaction_type",    // mutating step: set the "transaction_type"
               "mortgage_payment").  // to "mortgage_payment"
      values("transaction_id", "transaction_type")     // return the id and type
```

The results look exactly the same as those in Chapter 4:

```
    "144", "mortgage_payment",
    "153", "mortgage_payment",
    "132", "mortgage_payment",
    ...
```

With Example 5-11, we have concluded the transformation from development to our production schema and queries. We encourage you to apply the thought process of shaping query results from "Advanced Gremlin: Shaping Your Query Results" on page 106 to create more robust payloads and data structures to share within your application.

Moving On to More Complex, Distributed Graph Problems

We consider the transition from Chapter 4 to the topics and production optimizations presented in this chapter to be the final stage of learning how to work with graph data in Apache Cassandra. Along the way, you experienced limitations, followed by their resolutions. We will see more of that as we go along but in shorter iterations.

Our First 10 Tips to Get from Development to Production

Throughout Chapter 4, we presented data modeling tips for mapping your data into a distributed graph database. In this chapter, we augmented those tips with specific ways to optimize your production graph database. Let's revisit all 10 tips to recall the journey we went through from development to production (Figure 5-18).

These 10 tips are foundational to starting over with a new dataset and use case. We will be applying them repeatedly in the coming chapters. And we will find more recommendations to add to this list as we explore different common structures for distributed graph applications.

No.	Tip
1	If you want to start your traversal on some piece of data, make that data a vertex.
2	If you need the data to connect concepts, make that data an edge.
3	Vertex-Edge-Vertex should read like a sentence or phrase from your queries.
4	Nouns and concepts should be vertex labels. Verbs should be edge labels.
5	When in development, let the direction of your edges reflect how you would think about the data in your domain.
6	If you need to use data to sub-select a group, make it a property.
7	Properties can be duplicated onto edges or vertices use denormalization to reduce the number of elements you have to process in a query.
8	Let the direction you want to walk through your edge labels determine the indexes you need on an edge label in your graph schema.
9	Load your data; then apply your indexes.
10	Only keep the edges and indexes that you need for production queries.

Figure 5-18. Our top 10 graph data modeling tips

From here, we think you are ready to tackle deeper and more complex graph problems such as paths, recursive walks, collaborative filtering, and more.

The most advanced graph users today are those who are willing to learn through trial and error. We have collected what they have learned so far and will be walking you through those details within the context of new use cases in the coming chapters.

As we see it, gaining traction with new technology and new ways of thinking is a journey. We have presented the major foundational milestones others have reached so far. Now, you are ready to come along with us and apply graph thinking in production applications to solve complex problems.

In Chapter 6, we'll look at one of the most popular ways for people to extend graph thinking into their data. We will solve a complex problem found at the intersection of edge computing and hierarchical graph data in a self-organizing communication network of sensors.

About the Authors

As Chief Data Officer of DataStax, **Dr. Denise Koessler Gosnell** applies the thought processes in this book to make more informed decisions with data. Prior to this role, Dr. Gosnell joined DataStax to create and lead the Global Graph Practice, a team that builds some of the largest distributed graph applications in the world. Dr. Gosnell earned her Ph.D. in computer science from the University of Tennessee as an NSF fellow. Her research coined the concept social fingerprinting by applying graph algorithms to predict user identity from social media interactions.

Like this book, Dr. Gosnell's career centers on her passion for examining, applying, and advocating the applications of graph data. She has patented, built, published, and spoken on dozens of topics related to graph theory, graph algorithms, graph databases, and applications of graph data across all industry verticals. Prior to her role with DataStax, Dr. Gosnell worked in the healthcare industry, where she contributed to software solutions for permissioned blockchains, machine learning applications of graph analytics, and data science.

Dr. Matthias Broecheler is the Chief Technologist of DataStax and an entrepreneur with substantial research and development experience. Dr. Broecheler's work focuses on disruptive software technologies and understanding complex systems. He is known as an industry expert in graph databases, relational machine learning, and big data analysis in general. He is a practitioner of lean methodologies and experimentation to drive continuous improvement. Dr. Broecheler is the inventor of the Titan graph database and a founder of Aurelius.

Colophon

The animal on the cover of *The Practitioner's Guide to Graph Data* is the Mediterranean rainbow wrasse (*Coris julis*). This colorful fish inhabits the Northeastern Atlantic from Sweden to Senegal and into the Mediterranean. It lives near the shoreline and favors rocky, grassy areas. It feeds on small crustaceans such as shrimp and sea urchins and gastropods such as sea slugs. To eat its crusty prey, the rainbow wrasse has evolved sharp teeth and a protractile jaw.

A sequential hermaphrodite, the rainbow wrasse changes in color and size over its lifespan. These fish may be born either male or female, and in the primary phase they are colored brown with a white belly and a yellow-orange band running down either side of the body. Secondary-phase females reach a length of up to about seven inches, or they may change into secondary-phase males, and increase in size up to about 10 inches. Secondary-phase males are much more colorful—green or blue with a bright orange zig-zag stripe along either side.

The population of the Mediterranean rainbow wrasse is stable and not threatened. Many of the animals on O'Reilly covers are endangered; all of them are important to the world.

The cover fonts are Gilroy Semibold and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.