Lab 2. Cassandra Data Modeling

In this lab, we will open the door to the world of Cassandra data modeling. We will briefly go through its building blocks, the main differences to the relational data model, and examples of constructing queries on a Cassandra data model.

Cassandra describes its data model components by using the terms that are inherited from the Google BigTable parent, for example, column family, column, row, and so on. Some of these terms also exist in a relational data model. They, however, have completely different meanings. It often confuses developers and administrators who have a background in the relational world. At first sight, the Cassandra data model is counterintuitive and very difficult to grasp and understand.

In the relational world, you model the data by creating entities and associating them with relationships according to the guidelines governed by the relational theories. It means that you can solely concentrate on the logical view or structure of the data without any considerations of how the application accesses and manipulates the data. The objective is to have a stable data model complying with the relational guidelines. The design of the application can be done separately. For instance, you can answer different queries by constructing different SQL statements, which is not of your concern during data modeling. In short, relational data modeling is process oriented, based on a clear separation of concerns.

On the contrary, in Cassandra, you reverse the above steps and always start from what you want to answer in the queries of the application. The queries exert a considerable amount of influence on the underlying data model. You also need to take the physical storage and the cluster topology into account. Therefore, the query and the data model are twins, as they were born together. Cassandra data modeling is result oriented based on a clear understanding of how a query works internally in Cassandra.

Owing to the unique architecture of Cassandra, many simple things in a relational database, such as sequence and sorting, cannot be presumed. They require your special handling in implementing the same. Furthermore, they are usually design decisions that you need to make upfront in the process of data modeling. Perhaps it is the cost of the trade-off for the attainment of superb scalability, performance, and fault tolerance.

To enjoy reading this course, you are advised to temporarily think in both relational and NoSQL ways. Although you may not become a friend of Cassandra, you will have an eye-opening experience in realizing the fact that there exists a different way of working in the world.

What is unique to the Cassandra data model?

If you want me to use just one sentence to describe Cassandra's data model, I will say it is a non-relational data model, period. It implies that you need to forget the way you do data modeling in a relational database.

You focus on modeling the data according to relational theories. However, in Cassandra and even in other NoSQL databases, you need to focus on the application in addition to the data itself. This means you need to think about how you will query the data in the application. It is a paradigm shift for those of you coming from the relational world. Examples are given in the subsequent sections to make sure that you understand why you cannot apply relational theories to model data in Cassandra.

Another important consideration in Cassandra data modeling is that you need to take the physical topology of a Cassandra cluster into account. In a relational database, the primary goal is to remove data duplication through normalization to have a single source of data. It makes a relational database ACID compliant very easily. The related storage space required is also optimized. Conversely, Cassandra is designed to work in a massive-scale, distributed environment in which ACID compliance is difficult to achieve, and replication is a must. You must be aware of such differences in the process of data modeling in Cassandra.

Map and SortedMap

In *Bird's Eye View of Cassandra*, you learned that Cassandra's storage model is based on BigTable, a column-oriented store. A column-oriented store is a multidimensional map. Specifically, it is a data structure known as **Map**. An example of the declaration of map data structure is as follows:

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```

The `Map` data structure gives efficient key lookup, and the sorted nature provides efficient scans. `RowKey` is a unique key and can hold a value. The inner `SortedMap` data structure allows a variable number of `ColumnKey` values. This is the trick that Cassandra uses to be schemaless and to allow the data model to evolve organically over time. It should be noted that each column has a client-supplied timestamp associated, but it can be ignored during data modeling. Cassandra uses the timestamp internally to resolve transaction conflicts.

In a relational database, column names can be only strings and be stored in the table metadata. In Cassandra, both `RowKey` and `ColumnKey` can be strings, long integers, Universal Unique IDs, or any kind of byte arrays. In addition, `ColumnKey` is stored in each column. You may opine that it wastes storage space to repeatedly store the `ColumnKey` values. However, it brings us a very powerful feature of Cassandra. `RowKey` and `ColumnKey` can store data themselves and not just in `ColumnValue`. We will not go too deep into this at the moment; we will revisit it in later labs.

Note

Universal Unique ID

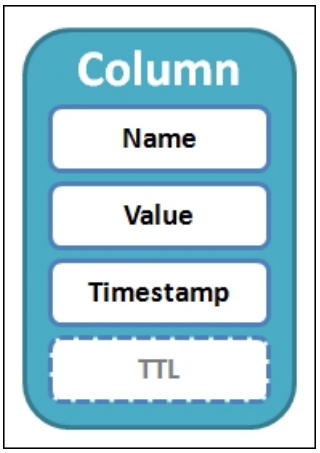
Universal Unique ID (**UUID**) is an **Internet Engineering Task Force** (**IETF**) standard, **Request for Comments** (**RFC**) 4122, with the intent of enabling distributed systems to uniquely identify information without significant central coordination. It is a 128-bit number represented by 32 lowercase hexadecimal digits, displayed in five groups separated by hyphens, for example: `0a317b38-53bf-4cad-a2c9-4c5b8e7806a2` :::

Logical data structure

There are a few logical building blocks to come up with a Cassandra data model. Each of them is introduced as follows.

Column

Column is the smallest data model element and storage unit in Cassandra. Though it also exists in a relational database, it is a different thing in Cassandra. As shown in the following figure, a column is a name-value pair with a timestamp and an optional **Time-To-Live** (**TTL**) value:

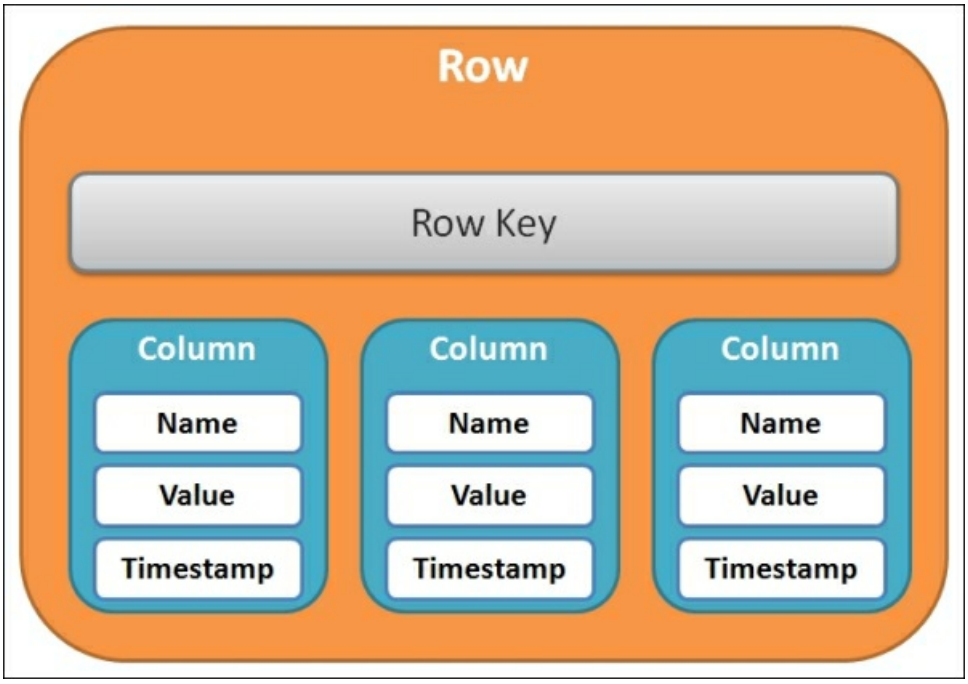


::: The elements of a column :::

The name and the value (`ColumnKey` and `ColumnValue` in `SortedMap` respectively) are byte arrays, and Cassandra provides a bunch of built-in data types that influence the sort order of the values. The timestamp here is for conflict resolution and is supplied by the client application during a write operation. Time-To-Live is an optional expiration value used to mark the column deleted after expiration. The column is then physically removed during compaction.

Row

One level up is a row, as depicted in the following figure. It is a set of orderable columns with a unique row key, also known as a primary key:



::: The structure of a row :::

The row key can be any one of the same built-in data types as those for columns. What orderable means is that columns are stored in sorted order by their column names.

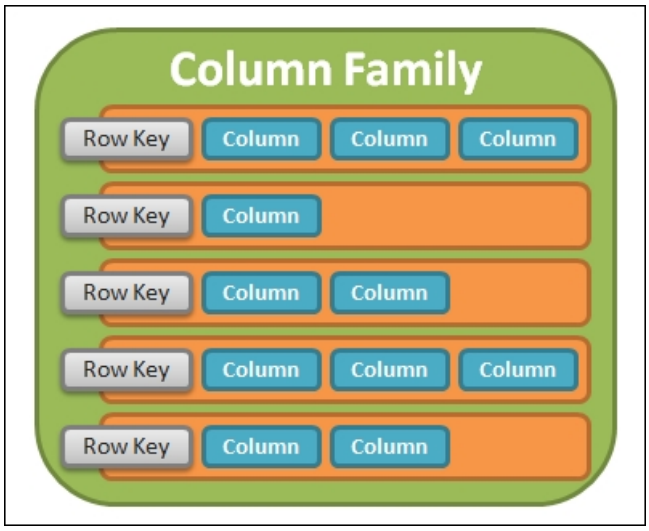
Note

Sort order is extremely important because Cassandra cannot sort by value as we do in a relational database.

Different names in columns are possible in different rows. That is why Cassandra is both row oriented and column oriented. It should be remarked that there is no timestamp for rows. Moreover, a row cannot be split to store across two nodes in the cluster. It means that if a row exists on a node, the entire row exists on that node.

Column family

The next level column family. As shown in the following figure, it is a container for a set of rows with a name:



::: The structure of a column family :::

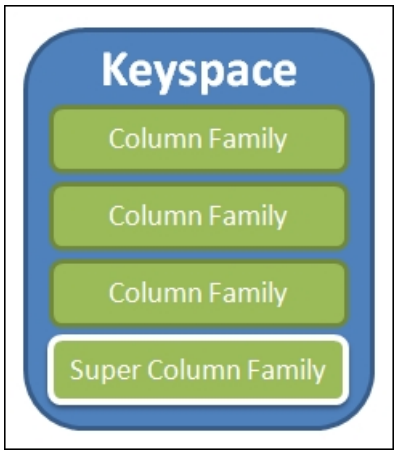
The row keys in a column family must be unique and are used to order rows. A column family is analogous to a table in a relational database, but you should not go too far with this idea. A column family provides greater flexibility by allowing different columns in different rows. Any column can be freely added to any column family at any time. Once again, it helps Cassandra be schemaless.

Columns in a column family are sorted by a comparator. The comparator determines how columns are sorted and ordered when Cassandra returns the columns in a query. It accepts long, byte and UTF8 for the data type of the column name, and the sort order in which columns are stored within a row.

Physically, column families are stored in individual files on a disk. Therefore, it is important to keep related columns in the same column family to save disk I/O and improve performance.

Keyspace

The outermost data model element is keyspace, as illustrated in the following figure:



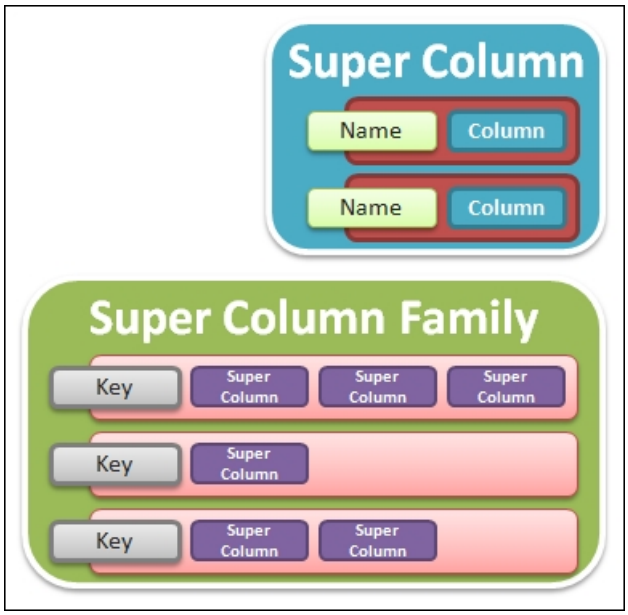
::: The structure of a keyspace :::

Keyspace is a set of column families and super column families, which will be introduced in the following section. It is analogous to a schema or database in the relational world. Each Cassandra instance has a system keyspace to keep system-wide metadata.

Keyspace contains replication settings controlling how data is distributed and replicated in the cluster. Very often, one cluster contains just one keyspace.

Super column and super column family

As shown in the following figure, a super column is a named map of columns and a super column family collection of super columns:



::: The structure of a super column and a super column family :::

Super columns were popular in the earlier versions of Cassandra but are not recommended anymore since they are not supported by the Cassandra Query Language (CQL), a SQL-like language to manipulate and query Cassandra, and must be accessed by using the low-level Thrift API. A column family is enough in most cases.

Note

Thrift

Thrift is a software framework for the development of scalable cross-language services. It combines a software stack with a code generation engine to build services that work efficiently and seamlessly with numerous programming languages. It is used as a **remote procedure call** (**RPC**) framework and was developed at Facebook Inc. It is now an open source project in the Apache Software Foundation.

There are other alternatives, for example, Protocol Buffers, Avro, MessagePack, JSON, and so on.

Collections

Cassandracollections, namely sets, lists, and maps, as parts of the data model. Collections are a complex type that can provide flexibility in querying.

Cassandra allows the following collections:

- **Sets**: These provide a way of keeping a unique set of values. It means that one can easily solve the problem of tracking unique values.
- **Lists**: These are suitable for maintaining the order of the values in the collection. Lists are ordered by the natural order of the type selected.
- **Maps**: These are similar to a store of key-value pairs. They are useful for storing table-like data within a single row. They can be a workaround of not having joins.

Here we only provided a brief introduction, and we will revisit the collections in subsequent labs.

No foreign key

Foreign keys are used in a relational database to maintain referential integrity that defines the relationship between two tables. They are used to enforce relationships in a relational data model such that the data in different but related tables can be joined to answer a query. Cassandra does not have the concept of referential integrity and hence, joins are not allowed either.

No join

Foreign keys and joins are the product of normalization in a relational data model. Cassandra has neither foreign keys nor joins. Instead, it encourages and performs best when the data model is denormalized.

Indeed, denormalization is not completely disallowed in the relational world, for example, a data warehouse built on a relational database. In practice, denormalization is a solution to the problem of poor performance of highly complex relational queries involving a large number of table joins.

Note

In Cassandra, denormalization is normal.

Foreign keys and joins can be avoided in Cassandra with proper data modeling.

No sequence

In a relational database, sequences are usually used to generate unique values for a surrogate key. Cassandra has no sequences because it is extremely difficult to implement in a peer-to-peer distributed system. There are however workarounds, which are as follows:

- Using part of the data to generate a unique key

- Using a UUID :::

In most cases, the best practice is to select the second workaround.

Counter

A counter columna special column used to store a number that keeps counting values. Counting can be either increment or decrement and timestamp is not required.

The counter column should not be used to generate surrogate keys. It is just designed to hold a distributed counter appropriate for distributed counting. Also bear in mind that updating a counter is not idempotent.

Note

Idempotent

Idempotent was originally a term in mathematics. But in computer science, idempotent is used more comprehensively to describe an operation that will produce the same results if executed once or multiple times.

Time-To-Live

Time-To-Live (**TTL**) is set on columns only. The unit is in seconds. When set on a column, it automatically counts down and will then be expired on the server side without any intervention of the client application.

Typical use cases are for the generation of security token and one-time token, automatic purging of outdated columns, and so on.

Secondary index

One important thing you need to remember is that the secondary index in Cassandra is not identical to that in a relational database. The secondary index in Cassandra can be created to query a column that is not a part of the primary key. A column family can have more than one secondary index. Behind the scenes, it is implemented as a separate hidden table which is maintained automatically by Cassandra's internal process.

The secondary index does not support collections and cannot be created on the primary key itself. The major difference between a primary key and a secondary index is that the former is a distributed index while the latter is a local index. The primary key is used to determine the node location and so, for a given row key, its node location can be found immediately. However, the secondary index is used just to index data on the local node, and it might not be possible to know immediately the locations of all matched rows without having examined all the nodes in the cluster. Hence, the performance is unpredictable.

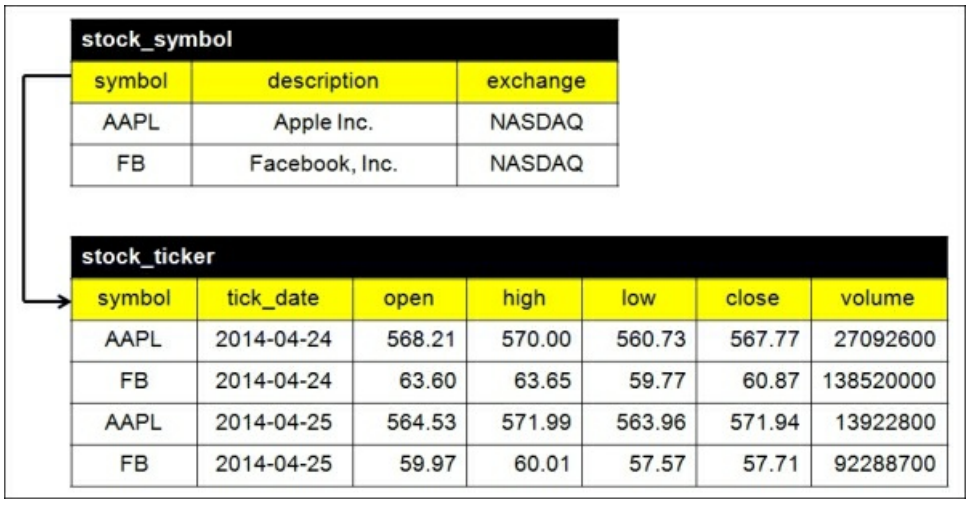
More information on secondary keys will be provided as we go through the later labs.

Modeling by query

In the previous section, we gained a basic understanding of the differences between a relational database and Cassandra. The most important difference is that a relational database models data by relationships whereas Cassandra models data by query. Now let us start with a simple example to look into what modeling by query means.

Relational version

The following figure shows a simple relational data model of a stock quote application:



stock_symbol

symbol	description	exchange
AAPL	Apple Inc.	NASDAQ
FB	Facebook, Inc.	NASDAQ

stock_ticker

symbol	tick_date	open	high	low	close	volume
AAPL	2014-04-24	568.21	570.00	560.73	567.77	27092600
FB	2014-04-24	63.60	63.65	59.77	60.87	138520000
AAPL	2014-04-25	564.53	571.99	563.96	571.94	13922800
FB	2014-04-25	59.97	60.01	57.57	57.71	92288700

::: The relational data model of a stock quote application (Source: Yahoo! Finance) :::

The `stock_symbol` table is an entity representing the stock master information such as the symbol of a stock, the description of the stock, and the exchange that the stock is traded. The `stock_ticker` table is another entity storing the prices of `open` , `high` , `low` , `close` , and the transacted `volume` of a stock on a trading day. Obviously the two tables have a relationship based on the `symbol` column. It is a well-known one-to-many relationship.

The following is the **Data Definition Language** (**DDL**) of the two tables:

```
CREATE TABLE stock_symbol (
symbol varchar PRIMARY KEY,
description varchar,
exchange varchar
);

CREATE TABLE stock_ticker (
symbol varchar references stock_symbol(symbol),
tick_date varchar,
open decimal,
high decimal,
low decimal,
close decimal,
volume bigint,
PRIMARY KEY (symbol, tick_date)
);
```

Consider the following three cases: first, we want to list out all stocks and their description in all exchanges. The SQL query for this is very simple:

```
// Query A
SELECT symbol, description, exchange
FROM stock_symbol;
```

Second, if we want to know all the daily close prices and descriptions of the stocks listed in the `NASDAQ` exchange, we can write a SQL query as:

```
// Query B
SELECT stock_symbol.symbol, stock_symbol.description,
stock_ticker.tick_date, stock_ticker.close
FROM stock_symbol, stock_ticker
WHERE stock_symbol.symbol = stock_ticker.symbol
AND stock_symbol.exchange = ''NASDAQ'';
```

Furthermore, if we want to know all the day close prices and descriptions of the stocks listed in the `NASDAQ` exchange on April 24, 2014, we can use the following SQL query:

```
// Query C
SELECT stock_symbol.symbol, stock_symbol.description,
stock_ticker.tick_date, stock_ticker.open,
stock_ticker.high, stock_ticker.low, stock_ticker_close,
stock_ticker.volume
FROM stock_symbol, stock_ticker
WHERE stock_symbol.symbol = stock_ticker.symbol
AND stock_symbol.exchange = ''NASDAQ''
AND stock_ticker.tick_date = ''2014-04-24'';
```

By virtue of the relational data model, we can simply write different SQL queries to return different results with no changes to the underlying data model at all.

Cassandra version

Now let us turn to Cassandra. The DDL statements in the last section can be slightly modified to create column families, or tables, in Cassandra, which are as follows:

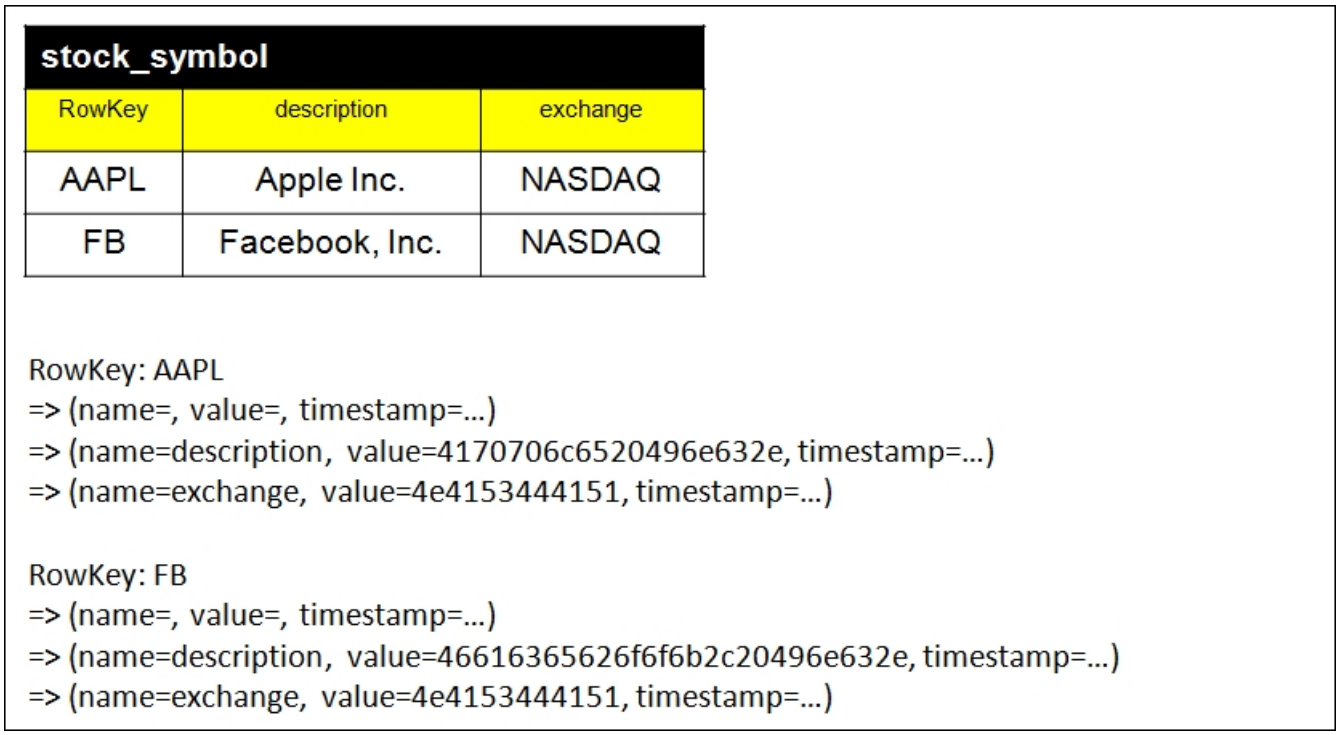
```
CREATE TABLE stock_symbol (
symbol varchar PRIMARY KEY,
description varchar,
exchange varchar
);

CREATE TABLE stock_ticker (
symbol varchar,
tick_date varchar,
open decimal,
high decimal,
low decimal,
close decimal,
volume bigint,
PRIMARY KEY (symbol, tick_date)
);
```

They seem to be correct at first sight.

As for `Query A` , we can query the Cassandra `stock_symbol` table exactly the same way:

```
// Query A
SELECT symbol, description, exchange
FROM stock_symbol;
```

The following figure depicts the logical and physical storage views of the `stock_symbol` table:



::: The Cassandra data model for Query A :::

The primary key of the `stock_symbol` table involves only one single column, `symbol` , which is also used as the row key and partition key of the column family. We can consider the `stock_symbol` table in terms of the SortedMap data structure mentioned in the previous section:

```
Map<RowKey, SortedMap<ColumnKey, ColumnValue>>
```

The assigned values are as follows:

```
RowKey=AAPL
ColumnKey=description
ColumnValue=Apple Inc.
ColumnKey=exchange
ColumnValue=NASDAQ
```

So far so good, right?

However, without foreign keys and joins, how can we obtain the same results for `Query B` and `Query C` in Cassandra? It indeed highlights that we need another way to do so. The short answer is to use denormalization.

For `Query B`, what we want is all the day close prices and descriptions of the stocks listed in the `NASDAQ` exchange. The columns involved are `symbol`, `description`, `tick_date`, `close`, and `exchange`. The first four columns are obvious, but why do we need the `exchange` column? The `exchange` column is necessary because it is used as a filter for the query. Another implication is that the `exchange` column is required to be the row key, or at least part of the row key.

Remember two rules:

1. A row key is regarded as a partition key to locate the nodes storing that row
2. A row cannot be split across two nodes :::

In a distributed system backed by Cassandra, we should minimize unnecessary network traffic as much as possible. In other words, the lesser the number of nodes the query needs to work with, the better the performance of the data model. We must cater to the cluster topology as well as the physical storage of the data model.

Therefore we should create a column family for `Query B` similar to the previous one:

```
// Query B
CREATE TABLE stock_ticker_by_exchange (
exchange varchar,
symbol varchar,
description varchar,
tick_date varchar,
close decimal,
PRIMARY KEY (exchange, symbol, tick_date)
);
```

The logical and physical storage views of `stock_ticker_by_exchange` are shown as follows:

stock_ticker_by_exchange

RowKey	AAPL:2014-04-24:	AAPL:2014-04-24:close	AAPL:2014-04-24:description	AAPL:2014-04-25:	AAPL:2014-04-25:close	AAPL:2014-04-25:description
NASDAQ	0	568.21	Apple Inc.	0	571.94	Apple Inc.
	FB:2014-04-24:	FB:2014-04-24:close	FB:2014-04-24:description	FB:2014-04-25:	FB:2014-04-25:close	FB:2014-04-25:description
	0	60.87	Facebook, Inc.	0	57.71	Facebook, Inc.

RowKey: NASDAQ
=> (name=AAPL:2014-04-24:, value=, timestamp=...)
=> (name=AAPL:2014-04-24:close, value=0000000200ddc9, timestamp=...)
=> (name=AAPL:2014-04-24:description, value=4170706c6520496e632e, timestamp=...)
=> (name=AAPL:2014-04-25:, value=, timestamp=...)
=> (name=AAPL:2014-04-25:close, value=0000000200df6a, timestamp=...)
=> (name=AAPL:2014-04-25:description, value=4170706c6520496e632e, timestamp=...)
=> (name=FB:2014-04-24:, value=, timestamp=...)
=> (name=FB:2014-04-24:close, value=0000000217c7, timestamp=...)
=> (name=FB:2014-04-24:description, value=46616365626f6f6b2c20496e632e, timestamp=...)
=> (name=FB:2014-04-25:, value=, timestamp=...)
=> (name=FB:2014-04-25:close, value=00000002168b, timestamp=...)
=> (name=FB:2014-04-25:description, value=46616365626f6f6b2c20496e632e, timestamp=...)

::: The Cassandra data model for Query B :::

The row key is the `exchange` column. However, this time, it is very strange that the column keys are no longer `symbol`, `tick_date`, `close`, and `description`. There are now 12 columns including `APPL:2014-04-24:`, `APPL:2014-04-24:close`, `APPL:2014-04-24:description`, `APPL:2014-04-25:`, `APPL:2014-04-25:close`, `APPL:2014-04-25:description`, `FB:2014-04-24:`, `FB:2014-04-24:close`, `FB:2014-04-24:description`, `FB:2014-04-25:`, `FB:2014-04-

`25:close` , and `FB:2014-04-25:description` , respectively. Most importantly, the column keys are now dynamic and are able to store data in just a single row. The row of this dynamic usage is called a wide row, in contrast to the row containing static columns of the `stock_symbol` table---termed as a skinny row.

Whether a column family stores a skinny row or a wide row depends on how the primary key is defined.

Note

If the primary key contains only one column, the row is a skinny row.

If the primary key contains more than one column, it is called a compound primary key and the row is a wide row.

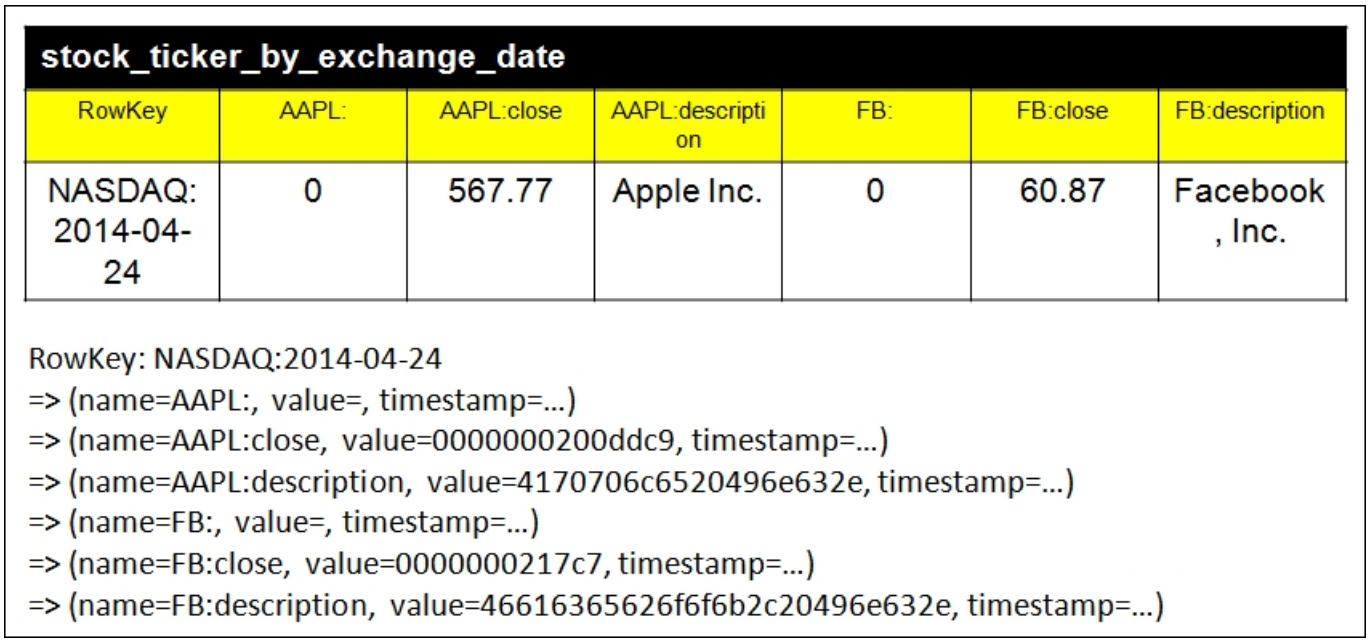
In either case, the first column in the primary key definition is the row key.

Finally, we come to `Query C` . Similarly, we make use of denormalization. `Query C` differs from `Query B` by an additional date filter on April 24, 2014. You might think of reusing the `stock_ticker_by_exchange` table for `Query C` . The answer is wrong. Why? The clue is the primary key which is composed of three columns, `exchange` , `symbol` , and `tick_date` , respectively. If you look carefully at the column keys of the `stock_ticker_by_exchange` table, you find that the column keys are dynamic as a result of the `symbol` and `tick_date` columns. Hence, is it possible for Cassandra to determine the column keys without knowing exactly which symbols you want? Negative.

A suitable column family for `Query C` should resemble the following code:

```
// Query C
CREATE TABLE stock_ticker_by_exchange_date (
exchange varchar,
symbol varchar,
description varchar,
tick_date varchar,
close decimal,
PRIMARY KEY ((exchange, tick_date), symbol)
);
```

This time you should be aware of the definition of the primary key. It is interesting that there is an additional pair of parentheses for the `exchange` and `tick_date` columns. Let's look at the logical and physical storage views of `stock_ticker_by_exchange_date` , as shown in the following figure:



stock_ticker_by_exchange_date

RowKey	AAPL:	AAPL:close	AAPL:description	FB:	FB:close	FB:description
NASDAQ:2014-04-24	0	567.77	Apple Inc.	0	60.87	Facebook, Inc.

RowKey: NASDAQ:2014-04-24
=> (name=AAPL:, value=, timestamp=...)
=> (name=AAPL:close, value=0000000200ddc9, timestamp=...)
=> (name=AAPL:description, value=4170706c6520496e632e, timestamp=...)
=> (name=FB:, value=, timestamp=...)
=> (name=FB:close, value=0000000217c7, timestamp=...)
=> (name=FB:description, value=46616365626f6f6b2c20496e632e, timestamp=...)

::: The Cassandra data model for Query C :::

You should pay attention to the number of column keys here. It is only six instead of 12 as in `stock_ticker_by_exchange` for `Query B` . The column keys are still dynamic according to the `symbol` column but the row key is now `NASDAQ:2014-04-24` instead of just `NASDAQ` in `Query B` . Do you remember the previously mentioned additional pair of parentheses? If you define a primary key in that way, you intend to use more than one column to be the row key and the partition key. It is called a composite partition key. For the time being, it is enough for you to know the terminology only. Further information will be given in later labs.

Until now, you might have felt dizzy and uncomfortable, especially for those of you having so many years of expertise in the relational data model. I also found the Cassandra data model very difficult to comprehend at the first time. However, you should be aware of the subtle differences between a relational data model and Cassandra data model. You must also be very cautious of the query that you handle. A query is always the starting point of designing a Cassandra data model. As an analogy, a query is a question and the data model is the answer. You merely use the data model to answer the query. It is exactly what modeling by query means.

Data modeling considerations

Apart from modeling by query, we need to bear in mind a few important points when designing a Cassandra data model. We can also consider a few good patterns that will be introduced in this section.

Data duplication

Denormalization is an evil in a relational data model, but not in Cassandra. Indeed, it is a good and common practice. It is solely based on the fact that Cassandra does not use high-end disk storage subsystem. Cassandra loves commodity-grade hard drives, and hence disk space is cheap. Data duplication as a result of denormalization is by no means a problem anymore; Cassandra welcomes it.

Sorting

In a relational database, sorting can be easily controlled using the `ORDER BY` clause in a SQL query. Alternatively, a secondary index can be created to further speed up the sorting operations.

In Cassandra, however, sorting is by design because you must determine how to compare data for a column family at the time of its creation. The comparator of the column family dictates how the rows are ordered on reads. Additionally, columns are ordered by their column names, also by a comparator.

Wide row

It is common to use wide rows for ordering, grouping and efficient filtering. Besides, you can use skinny rows. All you have to consider is the number of columns the row contains.

It is worth noting that for a column family storing skinny rows, the column key is repeatedly stored in each column. Although it wastes some storage space, it is not a problem on inexpensive commodity hard disks.

Bucketing

Even though a wide row can accommodate up to 2 billion variable columns, it is still a hard limit that cannot prevent voluminous data from filling up a node. In order to break through the 2 billion column limit, we can use a workaround technique called bucketing to split the data across multiple nodes.

Bucketing requires the client application to generate a bucket ID, which is often a random number. By including the bucket ID into a composite partition key, you can break up and distribute segments of the data to different nodes. However, it should not be abused. Breaking up the data across multiple nodes causes reading operations to consume extra resources to merge and reorder data. Thus, it is expensive and not a favorable method, and therefore should only be a last resort.

Valueless column

Column keys can store values as shown in the *Modeling by query* section. There is no \"Not Null\" concept in Cassandra such that column values can store empty values without any problem. Simply storing data in column keys while leaving empty values in the column, known as a valueless column, is sometimes used purposely. It's a common practice with Cassandra.

One motivation for valueless columns is the sort-by-column-key feature of Cassandra. Nonetheless, there are some limitations and caveats. The maximum size of a column key is 64 KB, in contrast to 2 GB for a column value. Therefore, space in a column key is limited. Furthermore, using timestamp alone as a column key can result in timestamp collision.

Time-series data

What is time-series data? It is anything that varies on a temporal basis such as processor utilization, sensor data, clickstream, and stock ticker. The stock quote data model introduced earlier is one such example. Cassandra is a perfect fit for storing time-series data. Why? Because one row can hold as many as 2 billion variable columns. It is a single layout on disk, based on the storage model. Therefore, Cassandra can handle voluminous time-series data in a blazing fast fashion. TTL is another excellent feature to simplify data housekeeping.

In the second half of this course, a complete stock quote technical analysis application will be developed to further explain the details of using Cassandra to handle time-series data.

Cassandra Query Language

It is quite common for other authors to start introducing the Cassandra data model from CQL. I use a different approach in this lab. I try to avoid diving too deep in CQL before we have a firm understanding of how Cassandra handles its physical storage.

The syntax of CQL is designed to be very similar to that of SQL. This intent is good for someone who is used to writing SQL statements in the relational world, to migrate to Cassandra. However, because of the high degree of similarity between CQL and SQL, it is even more difficult for us to throw away the relational mindsets if CQL is used to explain how to model data in Cassandra. It might cause more confusion in the end. I prefer the approach of a microscopic view of how the data model relates to the physical storage. By doing so, you can grasp the key points more quickly and understand the inner working mechanism more clearly. CQL is covered extensively in the next lab.

Summary

In this lab, we looked at the basics of a Cassandra data model and are now familiar with the column, row, column family, keyspace, counter, and other related terms. A comparison of the main differences between a relational data model and the Cassandra data model was also given to explain the concept of modeling by query that may seem shocking and counterintuitive at first sight. Then a few important considerations on data modeling and typical usage patterns were introduced. Finally, the reason why the introduction of CQL is deliberately postponed was expressed.

This lab is only the first part on Cassandra data modeling. In the next lab, we will continue the second part of the tour, Cassandra Query Language.