Chapter 10. Column-Family Stores

Column-family stores, such as Cassandra [Cassandra], HBase [Hbase], Hypertable [Hypertable], and Amazon SimpleDB [Amazon SimpleDB], allow you to store data with keys mapped to values and the values grouped into multiple column families, each column family being a map of data.

RDBMS	Cassandra
database instance	cluster
database	keyspace
table	column family
row	row
column (same for all rows)	column (can be different per row)

10.1. What Is a Column-Family Data Store?

There are many column-family databases. In this chapter, we will talk about Cassandra but also reference other column-family databases to discuss features that may be of interest in particular scenarios.

Column-family databases store data in column families as rows that have many columns associated with a row key (Figure 10.1). Column families are groups of related data that is often accessed together. For a `Customer`, we would often access their `Profile` information at the same time, but not their `Orders`.

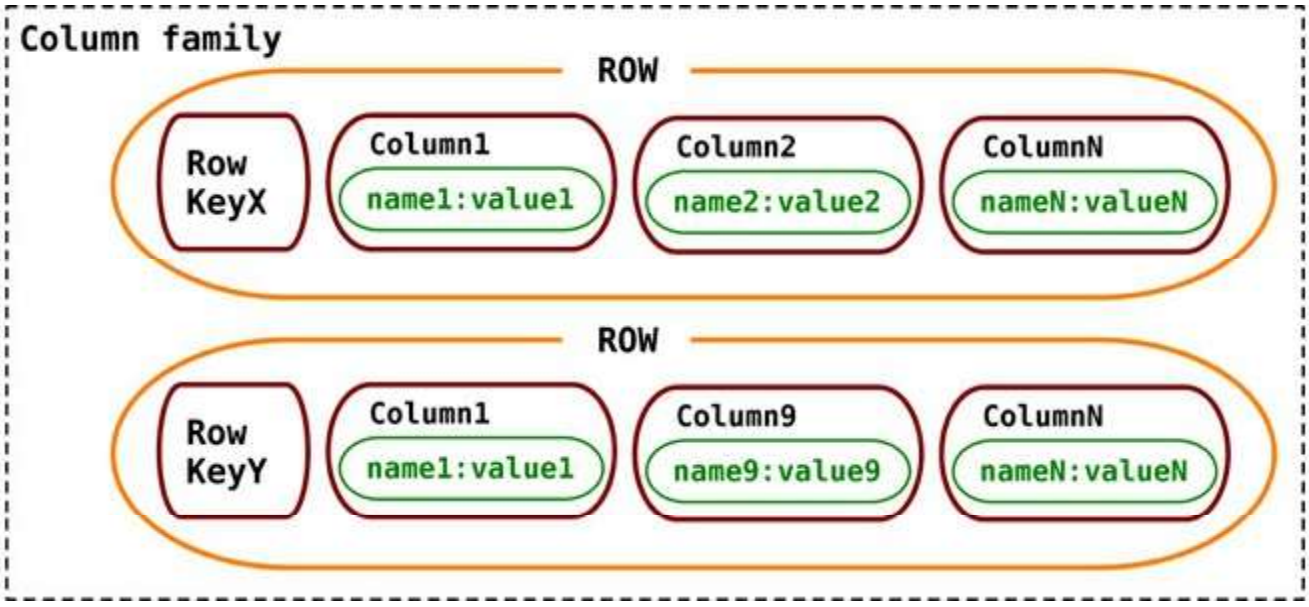

Figure 10.1. Cassandra's data model with column families

Cassandra is one of the popular column-family databases; there are others, such as HBase, Hypertable, and Amazon DynamoDB [Amazon DynamoDB]. Cassandra can be described as fast and easily scalable with write operations spread across the cluster. The cluster does not have a master node, so any read and write can be handled by any node in the cluster.

10.2. Features

Let's start by looking at how data is structured in Cassandra. The basic unit of storage in Cassandra is a column. A Cassandra column consists of a name-value pair where the name also behaves as the key. Each of these key-value pairs is a single column and is always stored with a timestamp value. The timestamp is used to expire data, resolve write conflicts, deal with stale data, and do other things. Once the column data is no longer used, the space can be reclaimed later during a compaction phase.

Click here to view code image

```
{
  name: "fullName",
  value: "Martin Fowler",
  timestamp: 12345667890
}
```

The `column` has a key of `firstName` and the value of `Martin` and has a timestamp attached to it. A row is a collection of columns attached or linked to a key; a collection of similar rows makes a column family. When the columns in a column family are simple columns, the column family is known as **standard column family**.

Click here to view code image

```
//column family
{
//row
  "pramod-sadalage" : {
     firstName: "Pramod",
     lastName: "Sadalage",
     lastVisit: "2012/12/12"
  }
//row
  "martin-fowler" : {
     firstName: "Martin",
     lastName: "Fowler",
     location: "Boston"
  }
}
```

Each column family can be compared to a container of rows in an RDBMS table where the key identifies the row and the row consists on multiple columns. The difference is that various rows do not have to have the same columns, and columns can be added to any row at any time without having to add it to other rows. We have the `pramod-sadalage` row and the `martin-fowler` row with different columns; both rows are part of the column family.

When a column consists of a map of columns, then we have a **super column**. A super column consists of a name and a value which is a map of columns. Think of a super column as a container of columns.

Click here to view code image

```
{
  name: "book:978-0767905923",
  value: {
    author: "Mitch Albon",
    title: "Tuesdays with Morrie",
    isbn: "978-0767905923"
  }
```

```
}
```

When we use super columns to create a column family, we get a **super column family**.

[Click here to view code image](#)

```
//super column family
{
//row
name: "billing:martin-fowler",
value: {
  address: {
    name: "address:default",
    value: {
      fullName: "Martin Fowler",
      street:"100 N. Main Street",
      zip: "20145"
    }
  },
  billing: {
    name: "billing:default",
    value: {
      creditcard: "8888-8888-8888-8888",
      expDate: "12/2016"
      }
    }
  }
//row
name: "billing:pramod-sadalage",
value: {
  address: {
    name: "address:default",
    value: {
      fullName: "Pramod Sadalage",
      street:"100 E. State Parkway",
      zip: "54130"
    }
   },
   billing: {
      name: "billing:default",
      value: {
        creditcard: "9999-8888-7777-4444",
        expDate: "01/2016"
        }
      }
   }
}
```

Super column families are good to keep related data together, but when some of the columns are not needed most of the time, the columns are still fetched and deserialized by Cassandra, which may not be optimal.

Cassandra puts the standard and super column families into **keyspaces**. A keyspace is similar to a database in RDBMS where all column families related to the application are stored. Keyspaces have to be created so that column families can be assigned to them:

```
create keyspace ecommerce
```

10.2.1. Consistency

When a write is received by Cassandra, the data is first recorded in a commit log, then written to an

in-memory structure known as **memtable**. A write operation is considered successful once it's written to the commit log and the memtable. Writes are batched in memory and periodically written out to structures known as **SSTable**. SSTables are not written to again after they are flushed; if there are changes to the data, a new SSTable is written. Unused SSTables are reclaimed by **compactation**.

Let's look at the read operation to see how consistency settings affect it. If we have a consistency setting of `ONE` as the default for all read operations, then when a read request is made, Cassandra returns the data from the first replica, even if the data is stale. If the data is stale, subsequent reads will get the latest (newest) data; this process is known as **read repair**. The low consistency level is good to use when you do not care if you get stale data and/or if you have high read performance requirements.

Similarly, if you are doing writes, Cassandra would write to one node's commit log and return a response to the client. The consistency of `ONE` is good if you have very high write performance requirements and also do not mind if some writes are lost, which may happen if the node goes down before the write is replicated to other nodes.

Click here to view code image

```
quorum = new ConfigurableConsistencyLevel();
quorum.setDefaultReadConsistencyLevel(HConsistencyLevel.QUORUM);
quorum.setDefaultWriteConsistencyLevel(HConsistencyLevel.QUORUM);
```

Using the `QUORUM` consistency setting for both read and write operations ensures that majority of the nodes respond to the read and the column with the newest timestamp is returned back to the client, while the replicas that do not have the newest data are repaired via the read repair operations. During write operations, the `QUORUM` consistency setting means that the write has to propagate to the majority of the nodes before it is considered successful and the client is notified.

Using `ALL` as consistency level means that all nodes will have to respond to reads or writes, which will make the cluster not tolerant to faults—even when one node is down, the write or read is blocked and reported as a failure. It's therefore upon the system designers to tune the consistency levels as the application requirements change. Within the same application, there may be different requirements of consistency; they can also change based on each operation, for example showing review comments for a product has different consistency requirements compared to reading the status of the last order placed by the customer.

During **keyspace** creation, we can configure how many replicas of the data we need to store. This number determines the replication factor of the data. If you have a replication factor of 3, the data copied on to three nodes. When writing and reading data with Cassandra, if you specify the consistency values of 2, you get that `R + W` is greater than the replication factor `(2 + 2 > 3)` which gives you better consistency during writes and reads.

We can run the node repair command for the keyspace and force Cassandra to compare every key it's responsible for with the rest of the replicas. As this operation is expensive, we can also just repair a specific column family or a list of column families:

```
repair ecommerce

repair ecommerce customerInfo
```

While a node is down, the data that was supposed to be stored by that node is handed off to other nodes. As the node comes back online, the changes made to the data are handed back to the node. This

technique is known as **hinted handoff**. Hinted handoff allows for faster restore of failed nodes.

10.2.2. Transactions

Cassandra does not have transactions in the traditional sense—where we could start multiple writes and then decide if we want to commit the changes or not. In Cassandra, a write is atomic at the row level, which means inserting or updating columns for a given row key will be treated as a single write and will either succeed or fail. Writes are first written to commit logs and memtables, and are only considered good when the write to commit log and memtable was successful. If a node goes down, the commit log is used to apply changes to the node, just like the **redo log** in Oracle.

You can use external transaction libraries, such as ZooKeeper [ZooKeeper], to synchronize your writes and reads. There are also libraries such as Cages [Cages] that allow you to wrap your transactions over ZooKeeper.

10.2.3. Availability

Cassandra is by design highly available, since there is no master in the cluster and every node is a peer in the cluster. The availability of a cluster can be increased by reducing the consistency level of the requests. Availability is governed by the `(R + W) > N` formula ("Quorums," p. 57) where `W` is the minimum number of nodes where the write must be successfully written, `R` is the minimum number of nodes that must respond successfully to a read, and `N` is the number of nodes participating in the replication of data. You can tune the availability by changing the `R` and `W` values for a fixed value of `N`.

In a 10-node Cassandra cluster with a replication factor for the keyspace set to 3 (`N = 3`), if we set `R = 2` and `W = 2`, then we have `(2 + 2) > 3`. In this scenario, when one node goes down, availability is not affected much, as the data can be retrieved from the other two nodes. If `W = 2` and `R = 1`, when two nodes are down the cluster is not available for write but we can still read. Similarly, if `R = 2` and `W = 1`, we can write but the cluster is not available for read. With the `R + W > N` equation, you are making conscious decisions about consistency tradeoffs.

You should set up your keyspaces and read/write operations based on your needs—higher availability for write or higher availability for read.

10.2.4. Query Features

When designing the data model in Cassandra, it is advised to make the columns and column families optimized for reading the data, as it does not have a rich query language; as data is inserted in the column families, data in each row is sorted by column names. If we have a column that is retrieved much more often than other columns, it's better performance-wise to use that value for the row key instead.

10.2.4.1. Basic Queries

Basic queries that can be run using a Cassandra client include the `GET`, `SET`, and `DEL`. Before starting to query for data, we have to issue the keyspace command `use ecommerce;`. This ensures that all of our queries are run against the keyspace that we put our data into. Before starting to use the column family in the keyspace, we have to define the column family.

Click here to view code image

```
CREATE COLUMN FAMILY Customer
WITH comparator = UTF8Type
AND key_validation_class=UTF8Type
```

```
AND column_metadata = [
{column_name: city, validation_class: UTF8Type}
{column_name: name, validation_class: UTF8Type}
{column_name: web, validation_class: UTF8Type}
];
```

We have a column family named `Customer` with `name`, `city`, and `web` columns, and we are inserting data in the column family with a Cassandra client.

Click here to view code image

```
SET Customer['mfowler']['city']='Boston';
SET Customer['mfowler']['name']='Martin Fowler';
SET Customer['mfowler']['web']='www.martinfowler.com';
```

Using the Hector [Hector] Java client, we can insert the same data in the column family.

Click here to view code image

```
ColumnFamilyTemplate<String, String> template =
        cassandra.getColumnFamilyTemplate();
ColumnFamilyUpdater<String, String> updater =
        template.createUpdater(key);
for (String name : values.keySet()) {
    updater.setString(name, values.get(name));
}
try {
    template.update(updater);
} catch (HectorException e) {
    handleException(e);
}
```

We can read the data back using the `GET` command. There are multiple ways to get the data; we can get the whole column family.

```
GET Customer['mfowler'];
```

We can even get just the column we are interested in from the column family.

```
GET Customer['mfowler']['web'];
```

Getting the specific column we need is more efficient, as only the data we care about is returned—which saves lots of data movement, especially when the column family has a large number of columns. Updating the data is the same as using the `SET` command for the column that needs to be set to the new value. Using `DEL` command, we can delete either a column or the entire column family.

Click here to view code image

```
DEL Customer['mfowler']['city'];

DEL Customer['mfowler'];
```

10.2.4.2. Advanced Queries and Indexing

Cassandra allows you to index columns other than the keys for the column family. We can define an index on the `city` column.

Click here to view code image

```
UPDATE COLUMN FAMILY Customer
WITH comparator = UTF8Type
AND column_metadata = [{column_name: city,
```

```
                    validation_class: UTF8Type,
                    index_type: KEYS}];
```

We can now query directly against the indexed column.

```
GET Customer WHERE city = 'Boston';
```

These indexes are implemented as *bit-mapped* indexes and perform well for low-cardinality column values.

10.2.4.3. Cassandra Query Language (CQL)

Cassandra has a query language that supports SQL-like commands, known as Cassandra Query Language (CQL). We can use the CQL commands to create a column family.

Click here to view code image

```
CREATE COLUMNFAMILY Customer (
  KEY varchar PRIMARY KEY,
  name varchar,
  city varchar,
  web  varchar);
```

We insert the same data using CQL.

Click here to view code image

```
INSERT INTO Customer (KEY,name,city,web)
  VALUES ('mfowler',
            'Martin Fowler',
            'Boston',
            'www.martinfowler.com');
```

We can read data using the `SELECT` command. Here we read all the columns:

```
SELECT * FROM Customer
```

Or, we could just `SELECT` the columns we need.

```
SELECT name,web FROM Customer
```

Indexing columns are created using the `CREATE INDEX` command, and then can be used to query the data.

Click here to view code image

```
SELECT name,web FROM Customer WHERE city='Boston'
```

CQL has many more features for querying data, but it does not have all the features that SQL has. CQL does not allow joins or subqueries, and its `where` clauses are typically simple.

10.2.5. Scaling

Scaling an existing Cassandra cluster is a matter of adding more nodes. As no single node is a master, when we add nodes to the cluster we are improving the capacity of the cluster to support more writes and reads. This type of horizontal scaling allows you to have maximum uptime, as the cluster keeps serving requests from the clients while new nodes are being added to the cluster.

10.3. Suitable Use Cases

Let’s discuss some of the problems where column-family databases are a good fit.

10.3.1. Event Logging

Column-family databases with their ability to store any data structures are a great choice to store event information, such as application state or errors encountered by the application. Within the enterprise, all applications can write their events to Cassandra with their own columns and the rowkey of the form `appname:timestamp`. Since we can scale writes, Cassandra would work ideally for an event logging system (Figure 10.2).

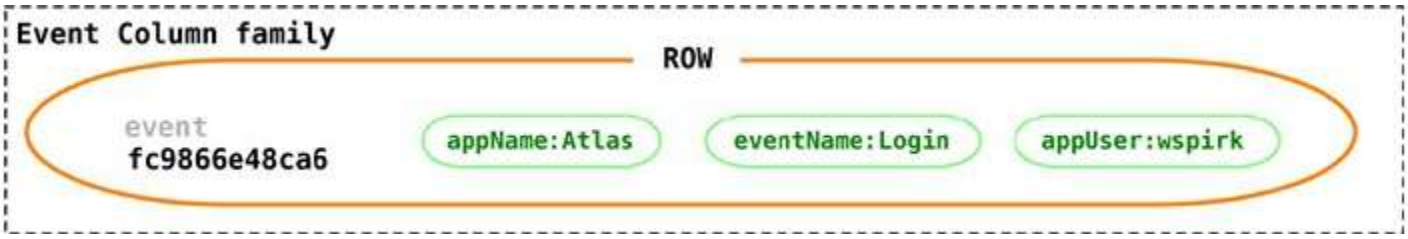

Figure 10.2. Event logging with Cassandra

10.3.2. Content Management Systems, Blogging Platforms

Using column families, you can store blog entries with tags, categories, links, and trackbacks in different columns. Comments can be either stored in the same row or moved to a different keyspace; similarly, blog users and the actual blogs can be put into different column families.

10.3.3. Counters

Often, in web applications you need to count and categorize visitors of a page to calculate analytics. You can use the `CounterColumnType` during creation of a column family.

Click here to view code image

```
CREATE COLUMN FAMILY visit_counter
WITH default_validation_class=CounterColumnType
AND key_validation_class=UTF8Type AND comparator=UTF8Type;
```

Once a column family is created, you can have arbitrary columns for each page visited within the web application for every user.

Click here to view code image

```
INCR visit_counter['mfowler'][home] BY 1;
INCR visit_counter['mfowler'][products] BY 1;
INCR visit_counter['mfowler'][contactus] BY 1;
```

Incrementing counters using CQL:

Click here to view code image

```
UPDATE visit_counter SET home = home + 1 WHERE KEY='mfowler'
```

10.3.4. Expiring Usage

You may provide demo access to users, or may want to show ad banners on a website for a specific time. You can do this by using **expiring columns**: Cassandra allows you to have columns which, after a given time, are deleted automatically. This time is known as TTL (Time To Live) and is defined in seconds. The column is deleted after the TTL has elapsed; when the column does not exist, the access can be revoked or the banner can be removed.

Click here to view code image

```
SET Customer['mfowler']['demo_access'] = 'allowed' WITH ttl=2592000;
```

10.4. When Not to Use

There are problems for which column-family databases are not the best solutions, such as systems that require ACID transactions for writes and reads. If you need the database to aggregate the data using queries (such as `SUM` or `AVG`), you have to do this on the client side using data retrieved by the client from all the rows.

Cassandra is not great for early prototypes or initial tech spikes: During the early stages, we are not sure how the query patterns may change, and as the query patterns change, we have to change the column family design. This causes friction for the product innovation team and slows down developer productivity. RDBMS impose high cost on schema change, which is traded off for a low cost of query change; in Cassandra, the cost may be higher for query change as compared to schema change.