Structures, Processes, and Protocols

06016414 and 06026207: NoSQL Database Systems

Instructor: Asst. Prof. Dr. Praphan Pavarangkoon

Outline

- Column Family Database Terminology
  - Basic Components of Column Family Databases
  - Structures and Processes: Implementing Column Family Databases
  - Processes and Protocols
- Lab Session

Column Family Database Terminology

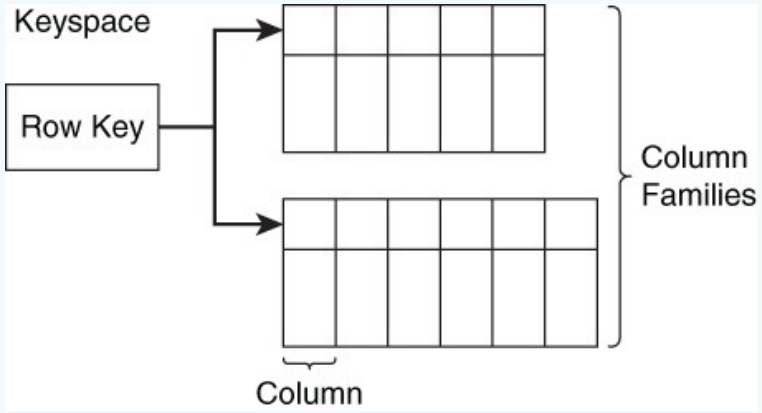
- When you read documentation and books about column family databases, you will see many familiar terms.
- Columns, partitions, and keyspaces are just a few of the commonly used terms you will see.
- When you are trying to understand a new technology, it often helps when the new technology uses the same terms used in existing technology—that is, unless they mean something else.

Basic Components of Column Family Databases

- The basic components of a column family database are the data structures developers deal with the most.
- The terms include
  - Keyspace
  - Row key
  - Column
  - Column families

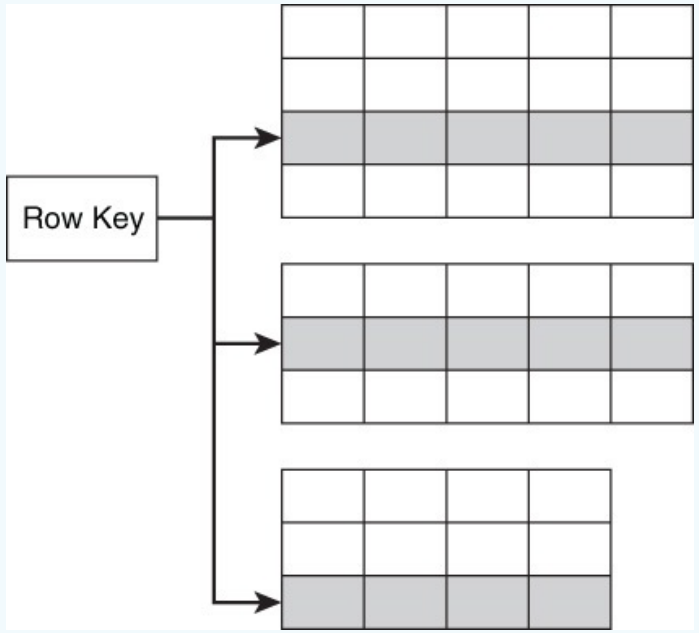
Keyspace

- A *keyspace* is the top-level data structure in a column family database.
- Typically, you will have one keyspace for each of your applications.



Row Key

- A *row key* uniquely identifies a row in a column family.
- It serves some of the same purposes as a primary key in a relational database.

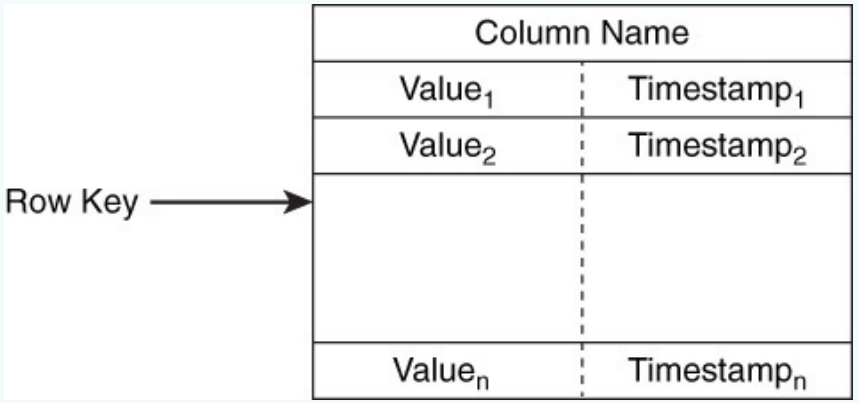


Row Key (cont.)

- In addition to uniquely identifying rows, row keys are also used to partition and order data.
- In HBase, rows are stored in lexicographic order of row keys.
- In Cassandra, rows are stored in an order determined by an object known as a partitioner.

Column

- A *column* is the data structure for storing a single value in a database.
- Depending on the type of column family database you are using, you might find values are represented simply as a string of bytes.
- HBase takes this approach.



Column (cont.)

- In other cases, you might be able to specify data types ranging from integers and strings to lists and maps.
- Cassandra's Query Language (CQL) offers almost 20 different data types.
- Columns are members of column family databases.
- Columns have three parts:
  - A column name
  - A time stamp or other version stamp
  - A value

Column Families

- Column families are collections of related columns.
- Columns that are frequently used together should be grouped into the same column family.
- Column families are stored in a keyspace.

Street	City	State	Province	Zip	Postal Code	Country
178 Main St.	Boise	ID		83701		U.S.
89 Woodridge	Baltimore	MD		21218		U.S.
293 Archer St.	Ottawa		ON		K1A 2C5	Canada
8713 Alberta DR	Vancouver		BC		VSK 0AI	Canada

Structures and Processes: Implementing Column Family Databases

- Column family databases are complicated.
- There are many processes that continually run in order to ensure the database functions as expected.
- There are also sophisticated data structures that significantly improve performance over more naive implementations.

Internal Structures and Configuration Parameters of Column Family Databases

- Internal structures and configuration parameters of column family databases span the full range of the database.
- Several are particularly important for database application designers and developers to understand:
  - Cluster
  - Partition
  - Commit log
  - Bloom filter
  - Replication count
  - Consistency level

Internal Structures and Configuration Parameters of Column Family Databases (cont.)

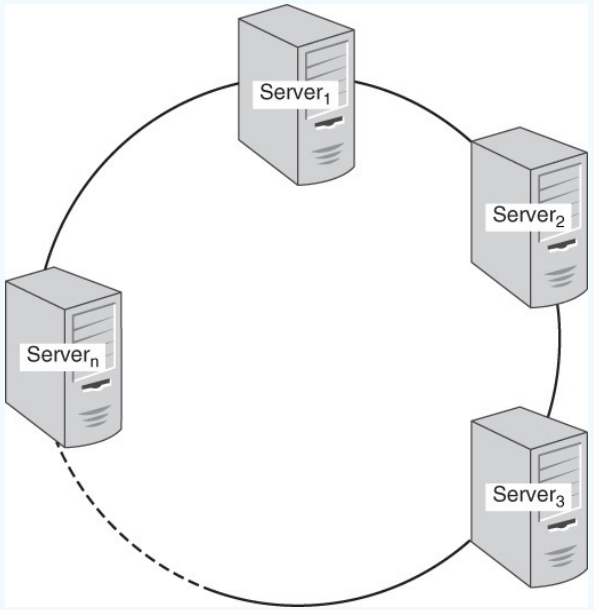
- Clusters and partitions are commonly used in distributed databases and are probably familiar topics by now.
- The commit log and Bloom filter are supporting data structures that improve integrity and availability of data as well as the performance of read operations.
- Replication count and consistency level are configuration parameters that allow database administrators to customize functionality of the column family database according to the needs of applications using it.

Old Friends: Clusters and Partitions

- Clusters and partitions enable distributed databases to coordinate processing and data storage across a set of servers.

Cluster

- A *cluster* is a set of servers configured to function together. Servers sometimes have differentiated functions and sometimes they do not.



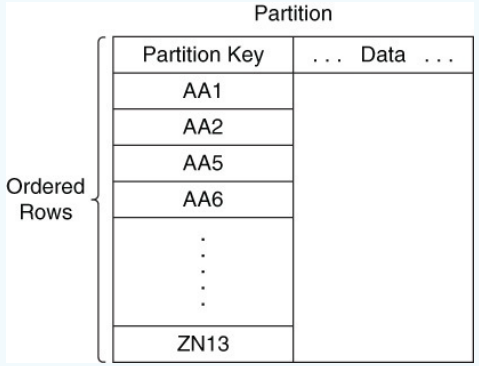
Cluster (cont.)

- Cassandra uses a single type of node. There are no master or slave nodes.
- Each node shares similar responsibilities, including
  - Accepting read and write requests
  - Forwarding read and write requests to servers able to fulfill the requests
  - Gathering and sharing information about the state of servers in the clusters
  - Helping compensate for failed nodes by storing write requests for the failed node until it is restored

Partition

- A *partition* is a logical subset of a database. Partitions are usually used to store a set of data based on some attribute of the data.
- For example, a database might assign data to a particular partition based on one of the following:
  - A range of values, such as the value of a row ID
  - A hash value, such as the hash value of a column name
  - A list of values, such as the names of states or provinces
  - A composite of two or more of the above options
- Each node or server within a column family cluster maintains one or more partitions.

Partition (cont.)



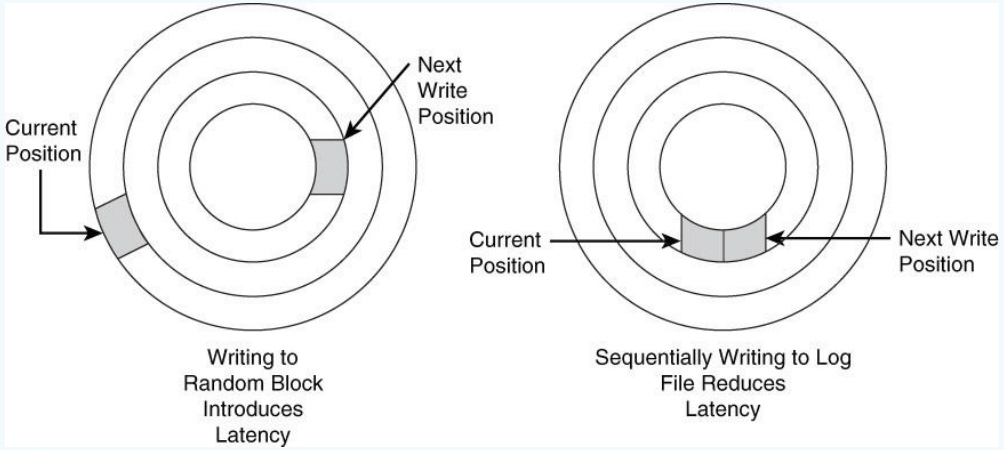
- When a client application requests data, the request is routed to a server with the partition containing the requested data.
- A request could go to a central server in a master-slave architecture or to any server in a peer-to-peer architecture.

Taking a Look Under the Hood: More Column Family Database Components

- In addition to the structures and procedures you will routinely work with, there are a few less visible components of column family databases worth understanding:
  - Commit logs
  - Bloom filter
  - Consistency level
- These components are not obvious to most developers, but they play crucial roles in achieving availability and performance.

Commit Log

- If your application writes data to a database and receives a successful status response, you reasonably expect the data to be stored on persistent storage.
- Even if a server fails immediately after sending a write success response, you should be able to retrieve your data once the server restarts.



Commit Log (cont.)

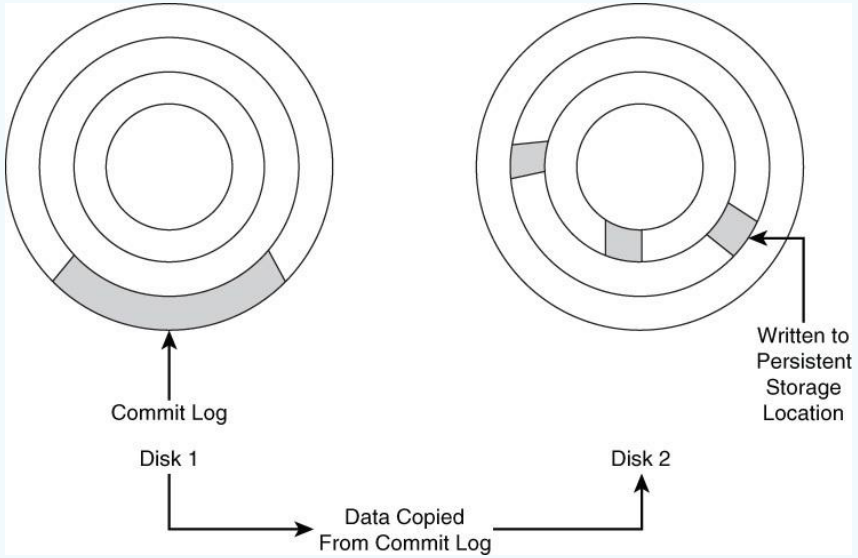
- One way to ensure this is to have the database write data to disk (or other persistent storage) before sending the success response.
- The database could do this, but it would have to wait for the read/write heads to be in the correct position on the disk before writing the data.
- If the database did this for every write, it could significantly cut down on write performance.

Commit Log (cont.)

- Instead of writing data immediately to its partition and disk block, column family databases can employ commit logs.
- These are append only files that always write data to the end of the file.

Commit Log (cont.)

- In the event of a failure, the database management system reads the commit log on recovery.
- Any entries in the commit log that have not been saved to partitions are then written to appropriate partitions



Bloom Filter

- A *Bloom filter* tests whether or not an element is a member of a set.
- Unlike a typical member function, the Bloom filter sometimes returns an incorrect answer.
- It could return a positive response in cases where the tested element is not a member of the set.
- This is known as a false-positive. จะไม่มีการreturn negative
- Bloom filters never return a negative response unless the element is not in the set.

Bloom Filter (cont.)

Member Function

Input Set	Test Element	In Set
{a,b,c}	a	Yes
{a,b,c}	c	Yes
{a,b,c}	e	No

Bloom Filter

Input Set	Test Element	In Set
{a,b,c}	a	Yes
{a,b,c}	c	Yes
{a,b,c}	e	Yes
{a,b,c}	f	No
{a,b,c}	g	No

Low Probability but Possible

ดูจากความน่าจะเป็นแทนที่จะหาเป็นตัว ๆ (ซ้า)
ถ้าหาที่ไม่มีในเซตอาจจะบอกผิด แต่ถ้าหาที่มีในเซตจะไม่มีทางบอกผิด

- Bloom filters help reduce the number of read operations by avoiding reading partitions or other data structures that definitely do not contain a sought-after piece of data.

ปรับค่า error ได้
- ตั้งค่า error น้อย ใช้ memory เยอะ ช้าลง
-

Bloom Filter (cont.)

- Another way to achieve the same benefit is to use a hash function.
- The application would only need to read that one partition.
- Bloom filters use less memory than typical hash functions, and the savings can be significant for the large-scale databases typically deployed in column family databases.
- The more memory you allocate for the Bloom filter, the smaller your error rate.
- Both HBase and Cassandra make use of Bloom filters to avoid unnecessary disk seeks.

Consistency Level

- *Consistency level* refers to the consistency between copies of data on different replicas.
- In the strictest sense, data is consistent only if all replicas have the same data.
- At the other end of the spectrum, you could consider the data “consistent” as long as it is persistently written to at least one replica.
- There are several intermediate levels as well.

Consistency Level (cont.)

- Consistency level is set according to several, sometimes competing, requirements:
  - How fast should write operations return a success status after saving data to persistent storage?
  - Is it acceptable for two users to look up a set of columns by the same row ID and receive different data?
  - If your application runs across multiple data centers and one of the data centers fails, must the remaining functioning data centers have the latest data?
  - Can you tolerate some inconsistency in reads but need updates saved to two or more replicas?

Consistency Level (cont.)

- In many cases such as sensor data, a low consistency level can satisfy requirements.
- Other situations such as online game call for a moderate consistency level.
- The highest levels of consistency, such as writing replicas to multiple replicas in multiple data centers, should be saved for the most demanding fault-tolerant applications.

Processes and Protocols

- In addition to the data structures described above, a number of important background processes are responsible for maintaining a functional column family database.

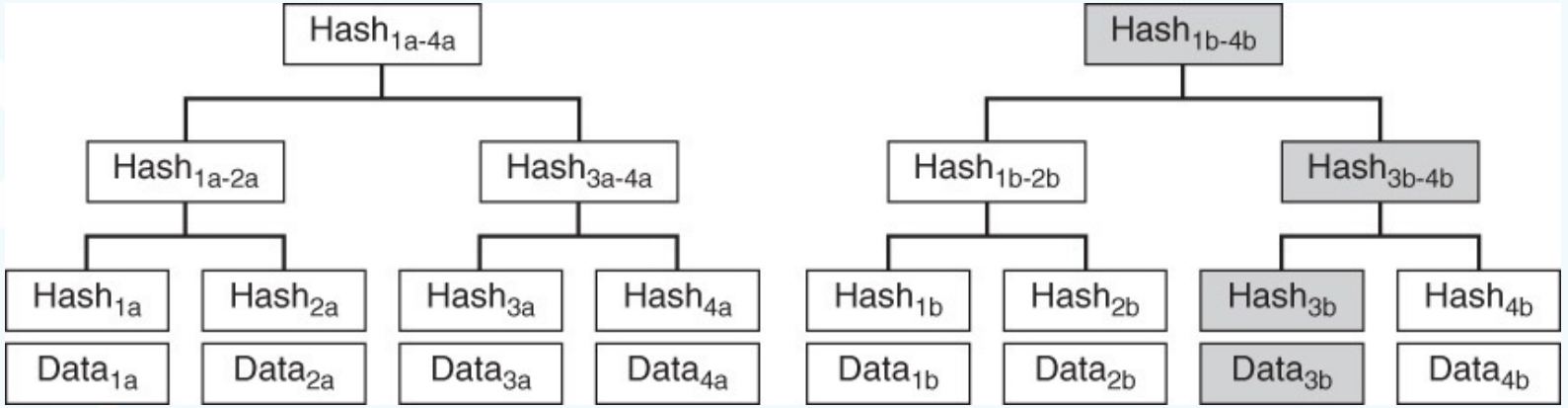
Replication

- Replication is a process closely related to consistency level.
- Whereas the consistency level determines how many replicas to keep, the replication process determines where to place replicas and how to keep them up to date.
- In the simplest case, the server for the first replica is determined by hash function, and additional replicas are placed according to the relative position of other servers.
- Column family databases can also use network topology to determine where to place replicas.

Anti-Entropy

- *Anti-entropy* is the process of detecting differences in replicas.
- The naive way to compare replicas is to send a copy of one replica to the node storing another replica and compare the two.
- Column family databases can exploit the fact that much of replica data may not change between anti-entropy checks.
- One method employs a tree of hashes, also known as a Merkle tree.

Anti-Entropy (cont.)



- Anti-entropy processes can calculate hash trees on all replicas.
- One replica sends its hash tree to another node.
- That node compares the hash values in the two root nodes.
- If there is a difference, then the anti-entropy process can compare the hash values at the next level down.

Anti-Entropy (cont.)

- At least one pair of these hash values will differ between replicas.
- The process of traversing the tree continues until the process reaches one or more leaf nodes that have changed.
- Only the data associated with those leaf nodes needs to be exchanged.

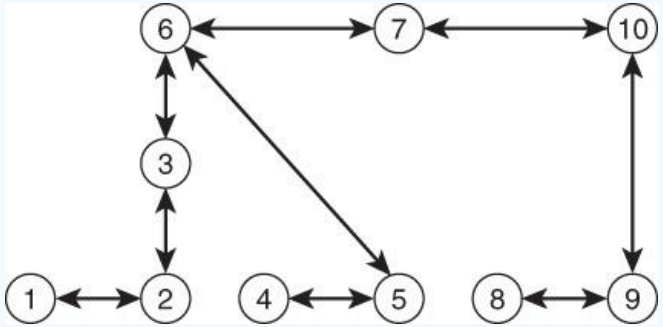
Gossip Protocol

- A fundamental challenge in any distributed system is keeping member nodes up to date on information about other members of the system.

Number of Nodes	Number of Messages for Complete Communication
2	2
3	6
4	12
5	20
10	90
15	210
20	380
25	600
50	2,450
100	9,900

Gossip Protocol (cont.)

- It is more efficient to have nodes share information about themselves as well as other nodes from which they have received updates.



- When using a gossip protocol—in which each node sends information about itself and all information it has received from other nodes—all nodes can be updated with a fraction of the number of messages required for complete communication.

Hinted Handoff

- Replicas enable read availability even if some nodes have failed.
- They do not address how to handle a write operation that is directed to a node that is down.
- The hinted handoff mechanism is designed to solve this problem.
- If a write operation is directed to a node that is unavailable, the operation can be redirected to another node, such as another replica node or a node designated to receive write operations when the target node is down.

Hinted Handoff (cont.)

- The node receiving the redirected write message creates a data structure to store information about the write operation and where it should ultimately be sent.
- The hinted handoff periodically checks the status of the target server and sends the write operation when the target is available.

Lab Session

- Cassandra Query Language:
  - Learn about the most essential data definition and data manipulation statements in Cassandra Query Language (CQL)

Cassandra Query Language

- Learn about Cassandra Query Language (CQL)
- Use the CQL shell
- Execute statements CREATE KEYSPACE, USE and CREATE TABLE
- Practice using statements INSERT, SELECT, UPDATE and DELETE

Cassandra Query Language (cont.)

- *Cassandra Query Language* (*CQL*) is the primary language for interacting with Apache Cassandra™ databases. CQL data definition and data manipulation statements include:
- *CQL Data Definition*
  - CREATE | ALTER | DROP KEYSPACE
  - USE
  - CREATE | ALTER | DROP | TRUNCATE TABLE
- *CQL Data Manipulation*
  - INSERT (**C**reate)
  - SELECT (**R**ead)
  - UPDATE (**U**pdate)
  - DELETE (**D**elete)
- Let's use some of these statements and see how they work.

Cassandra Query Language (cont.)

- **Start the CQL shell**
  - The CQL shell is a command-line client for executing CQL statements over a Cassandra database interactively.
  - Get the CQL shell usage help:

```
cqlsh -h
```

  - Start the CQL shell:

```
cqlsh
```

Cassandra Query Language (cont.)

- **Create a keyspace**
  - A keyspace is a namespace for a set of tables sharing a data replication strategy and some options. It is conceptually similar to a "database" in a relational database management system.
  - Create the keyspace:

```
CREATE KEYSPACE killr_video
WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': 1 };
```

```
CREATE  KEYSPACE [IF NOT EXISTS] keyspace_name
   WITH REPLICATION = {
      'class' : 'SimpleStrategy', 'replication_factor' : N }
     | 'class' : 'NetworkTopologyStrategy',
      'dc1_name' : N [, ...]
  }
  [AND DURABLE_WRITES =  true|false] ;
```

  - We create killr_video keyspace on a single node evaluation cluster.

Cassandra Query Language (cont.)

- **Set a working keyspace**
  - Many CQL statements work with tables, indexes and other objects defined within a specific keyspace. For example, to refer to a table, we have to either use a fully-qualified name consisting of a keyspace name and a table name, or set a working keyspace and simply refer to the table by its name. For convenience, we go with the second option.
  - Set the current working keyspace:

```
USE killr_video;
```

```
USE keyspace_name
```

Cassandra Query Language (cont.)

- **Create a table**
  - A Cassandra table has named columns with data types, rows with values, and a primary key to uniquely identify each row.
  - A primary key in Cassandra consists of one or more partition keys and zero or more clustering key components. The order of these components always puts the partition key first and then the clustering key.
  - Apart from making data unique, the partition key component of a primary key plays an additional significant role in the placement of the data. As a result, it improves the performance of reads and writes of data spread across multiple nodes in a cluster.

Cassandra Query Language (cont.)

- **Create a table**
  - As an example, let's create table users with four columns and primary key email.
  - Create the table:

```
CREATE TABLE users (
  email TEXT PRIMARY KEY,
  name TEXT,
  age INT,
  date_joined DATE
);
```

```
CREATE TABLE [IF NOT EXISTS] [keyspace_name.]table_name (
   column_definition [, ...]
   PRIMARY KEY (column_name [, column_name ...])
[WITH table_options
   | CLUSTERING ORDER BY (clustering_column_name order])
   | ID = 'table_hash_tag'
   | COMPACT STORAGE]
```

Cassandra Query Language (cont.)

- **Insert a row**
  - Add the row into our table using the CQL INSERT statement:

```
INSERT INTO users (email, name, age, date_joined)
VALUES ('joe@datastax.com', 'Joe', 25, '2020-01-01');
```

  - Insert another row into the table:

```
INSERT INTO users (email, name, age, date_joined)
VALUES ('jen@datastax.com', 'Jen', 27, '2020-01-01');
```

```
INSERT INTO [keyspace_name.] table_name (column_list)
VALUES (column_values)
[IF NOT EXISTS]
[USING TTL seconds | TIMESTAMP epoch_in_microseconds]
```

Cassandra Query Language (cont.)

- **Retrieve a row**
  - Now, retrieve the row using the CQL SELECT statement:

```
SELECT * FROM users
WHERE email = 'joe@datastax.com';
```

  - Retrieve a different row from the table:

```
SELECT * FROM users
WHERE email = 'jen@datastax.com';
```

```
SELECT * | select_expression | DISTINCT partition
FROM [keyspace_name.] table_name
[WHERE partition_value
   [AND clustering_filters
   [AND static_filters]]]
[ORDER BY PK_column_name ASC|DESC]
[LIMIT N]
[ALLOW FILTERING]
```

Cassandra Query Language (cont.)

- **Update a row**
  - Next, update the row using the CQL UPDATE statement:

```
UPDATE users SET name = 'Joseph'
WHERE email = 'joe@datastax.com';

SELECT * FROM users;
```

  - Update another row in the table:

```
UPDATE users SET name = 'Jennifer'
WHERE email = 'jen@datastax.com';

SELECT * FROM users;
```

```
UPDATE [keyspace_name.] table_name
[USING TTL time_value | USING TIMESTAMP timestamp_value]
SET assignment [, assignment] . . .
WHERE row_specification
[IF EXISTS | IF condition [AND condition] . . .] ;
```

Cassandra Query Language (cont.)

- **Delete a row**
  - Finally, delete the row using the CQL DELETE statement:

```
DELETE FROM users
WHERE email = 'joe@datastax.com';

SELECT * FROM users;
```

  - Deleting another row from the table:

```
DELETE FROM users
WHERE email = 'jen@datastax.com';

SELECT * FROM users;
```

```
DELETE [column_name (term)][, ...]
FROM [keyspace_name.] table_name
[USING TIMESTAMP timestamp_value]
WHERE PK_column_conditions
[IF EXISTS | IF static_column_conditions]
```

Cassandra Query Language (cont.)

- **CQL vs. SQL**
  - If you are familiar with SQL, CQL may look quite similar. Indeed, there are many syntactic similarities between the two languages, but there are also many important differences. Here are just a few facts about CQL that highlight some of the differences:
    - CQL supports tables with single-row and multi-row partitions
    - CQL table primary key consists of a mandatory partition key and an optional clustering key
    - CQL does not support referential integrity constraints
    - CQL updates or inserts may result in upserts
    - CQL queries cannot retrieve data based on an arbitrary table column
    - CQL supports no joins or other binary operations
    - CQL CRUD operations are executed with a tunable consistency level
    - CQL supports lightweight transactions but not ACID transactions
  - If some of the above facts do not sound familiar, you know that there are more about CQL to learn!

Q & A