Introduction To Cassandra

pandaforme

Published with GitBook

Table of Contents

This book is a basic introduction to Cassandra. The major information and knowledge came from http://datastax.com/.

I will also comment some concepts based on realistic examples and can understand those concepts clearly.

CAP Theorem

In theoretical computer science, the CAP theorem, also named Brewer's theorem after computer scientist Eric Brewer, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- Consistency: all nodes see the same data at the same time
- Availability: a guarantee that every request receives a response about whether it succeeded or failed
- Partition tolerance: the system continues to operate despite arbitrary partitioning due to network failures

因為在分散式系統中，網路是不穩定的，P是一定要選擇的，要選擇C或A就要看需求.

- 選擇CP：response有可能會timeout.
- 選擇AP：放棄C，退而要求Eventually Consistency．選擇A，response會在一定時間內回應，不管成功或失敗.

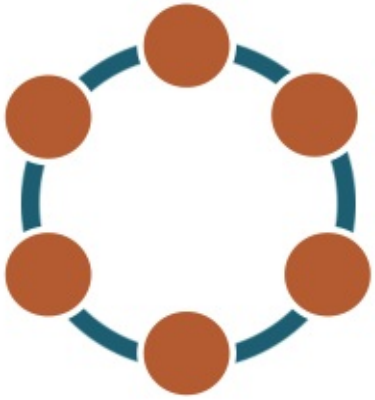
Cassandra可以動態切換成CP/AP.

References:

- CAP Theorem
- CAP Theorem: Revisited

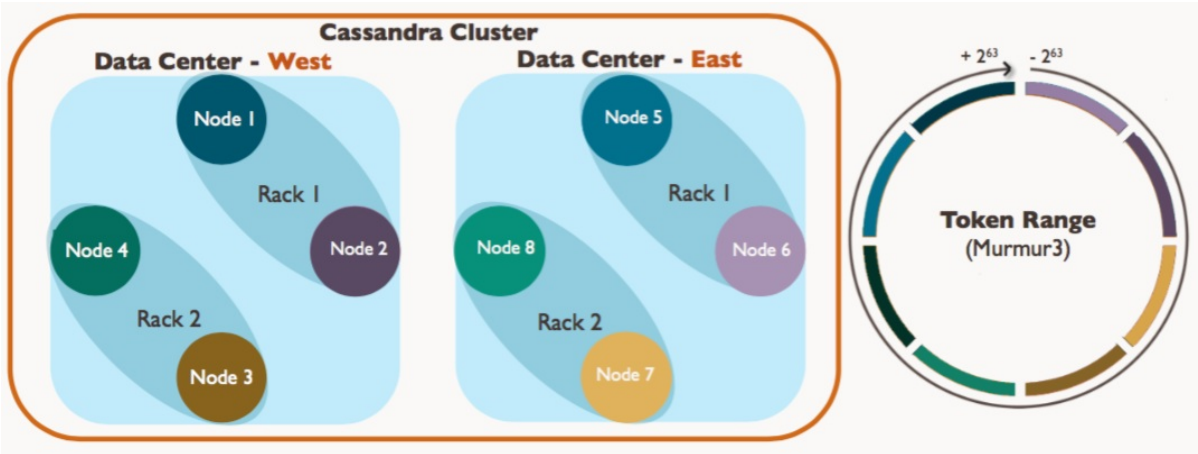
Understand how requests are coordinated

Cassandra has a masterless 「ring」 architecture that is elegant, easy to set up, and easy to maintain.



What is a cluster?

- Node: one Cassandra instance
- Rack: a logical set of nodes
- Data Center: a logical set of racks
- Cluster: the full set of nodes which map to a single complete token ring

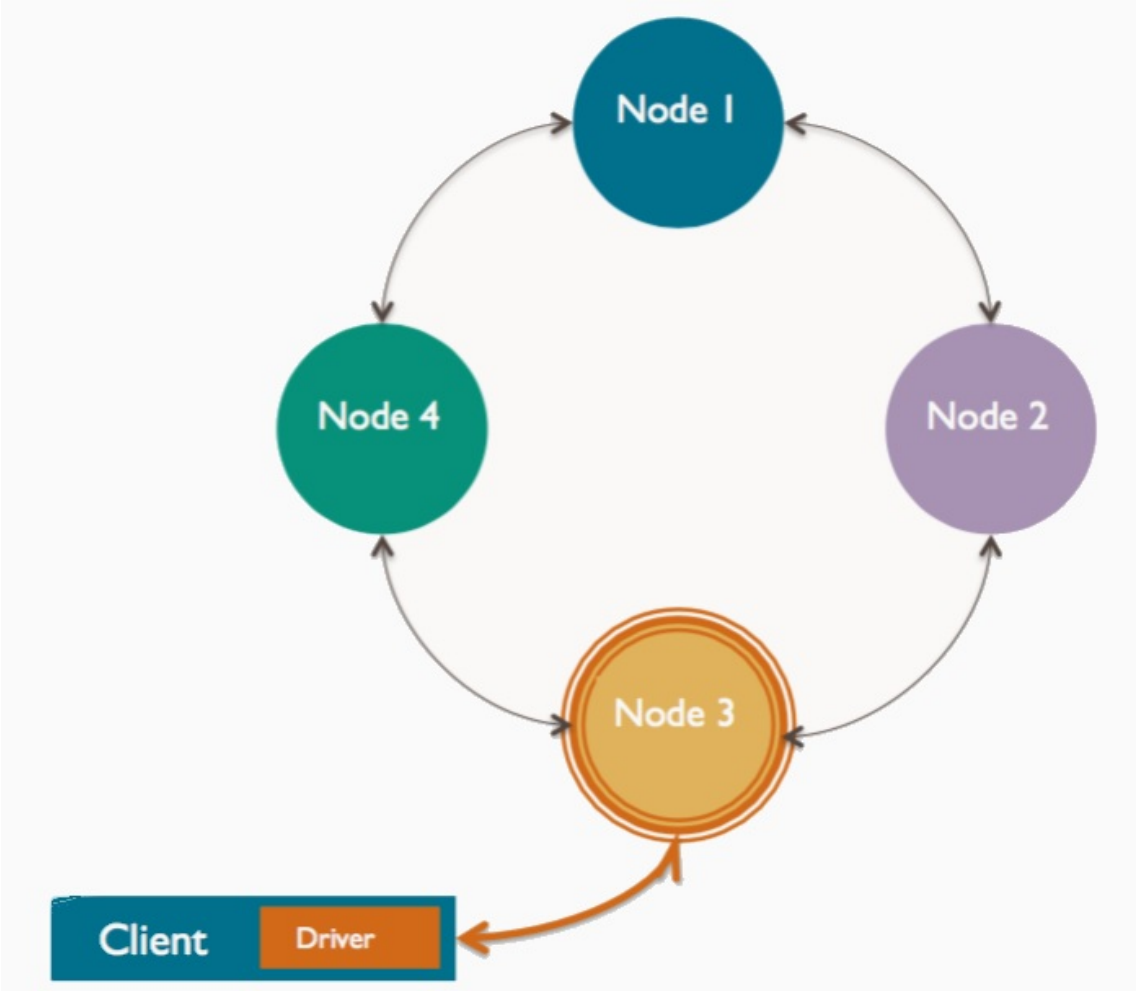


What is a coordinator?

The node chosen by the client to receive a particular read or write request to its cluster

- Any node can coordinate any request
- Each client request may be coordinated by a a different node

- The coordinator manages the Replication Factor (RF)
  - Replication factor (RF) – onto how many nodes should a write be copied?
- The coordinator also applies the Consistency Level (CL)
  - Consistency level (CL) – how many nodes must acknowledge a read or write request



What is the partitioner?

A system on each node which hashes tokens from designated values in rows being added

How does a partitioner work?

A node's partitioner hashes a token from the partition key value of a write request

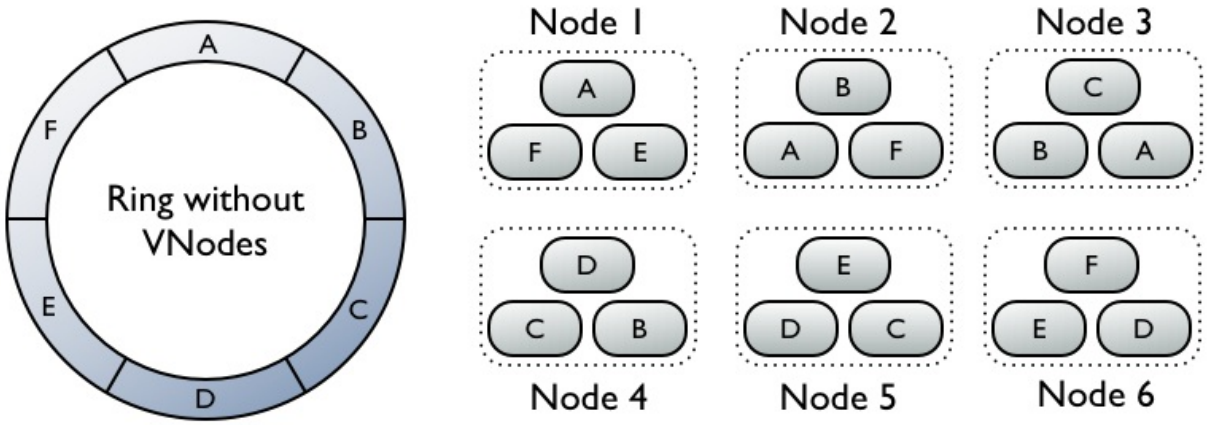
What partitioners does Cassandra offer?

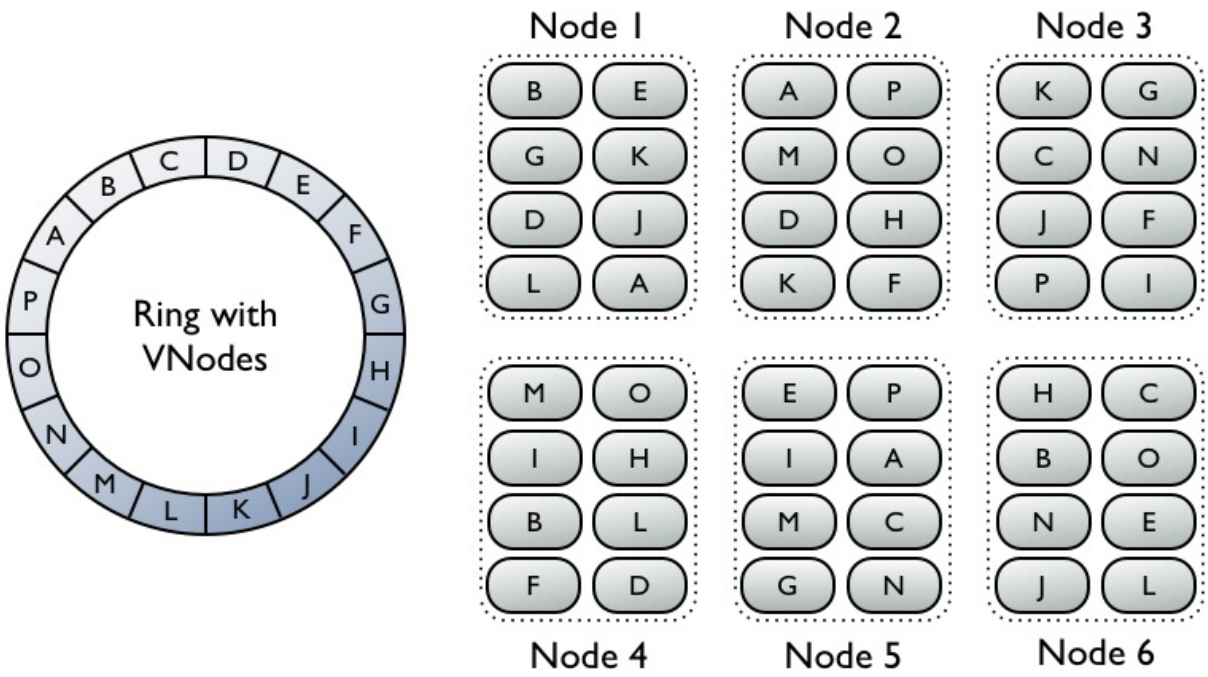
Cassandra offers three partitioners

- Murmur3Partitioner (default): uniform distribution based on Murmur3 hash
- RandomPartitioner: uniform distribution based on MD5 hash
- ByteOrderedPartitioner (legacy only): lexical distribution based on key bytes

What are virtual nodes?

There's one token per node, and thusly a node owns exactly one contiguous range in the ringspace.Vnodes change this paradigm from one token or range per node, to many per node. Within a cluster these can be randomly selected and be non-contiguous, giving us many smaller ranges that belong to each node.





How are virtual nodes helpful?

- token ranges are distributed, so machines bootstrap faster

- impact of virtual node failure is spread across entire cluster
- token range assignment automated
-

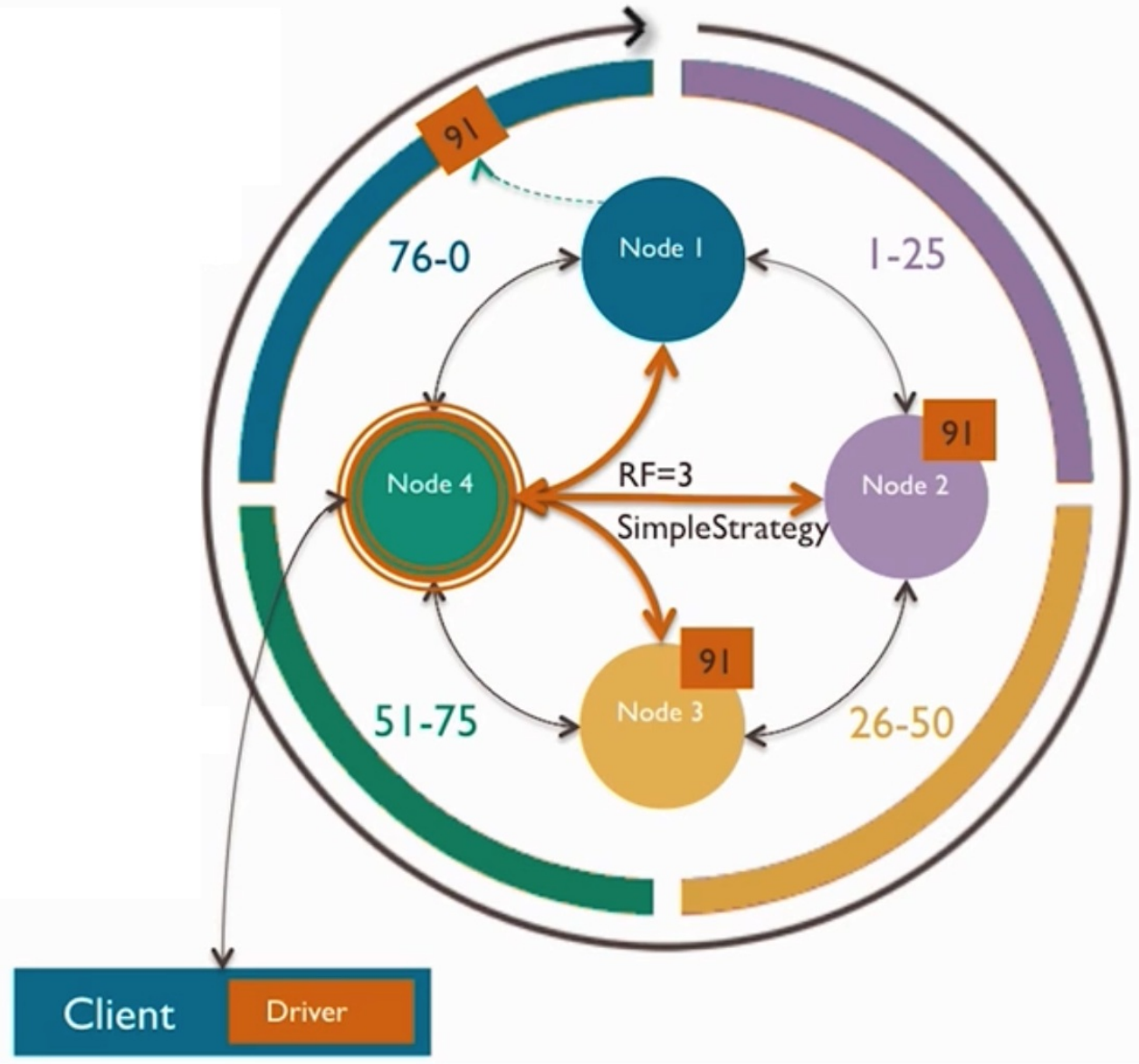
References:

- [Virtual nodes in Cassandra 1.2] (http://www.datastax.com/dev/blog/virtual-nodes-in-cassandra-1-2)

Understand replication

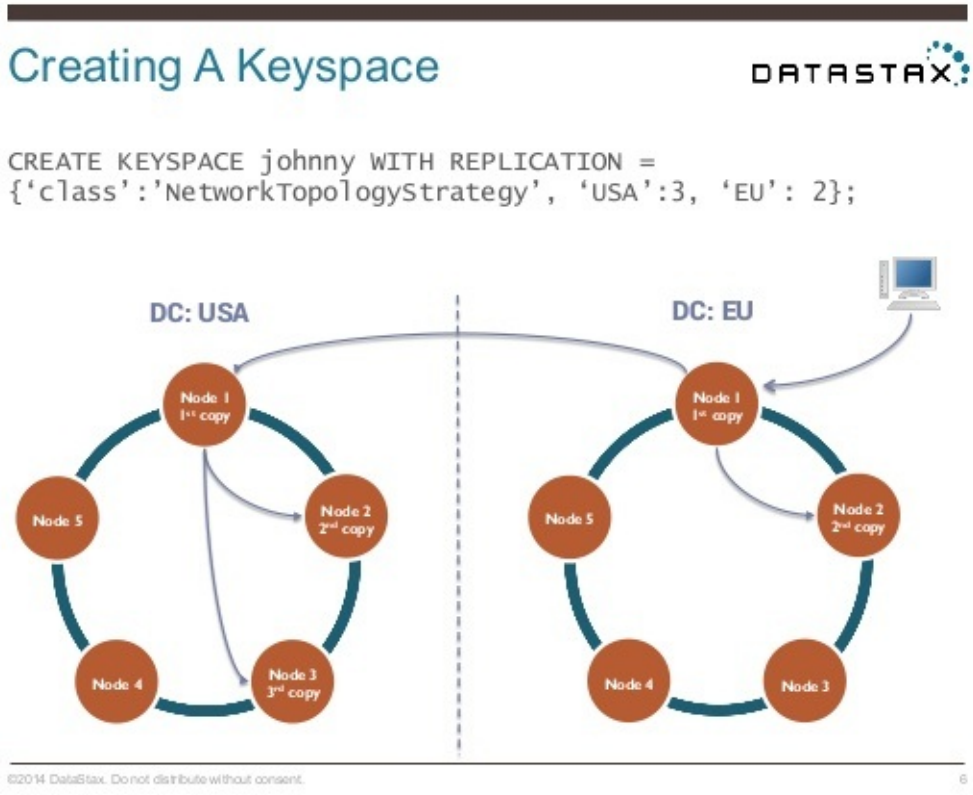
How is data replicated among nodes?

SimpleStrategy: create replicas on nodes subsequent to the primary range node



How is data replicated between data centers?

NetworkTopologyStrategy: distribute replicas across racks and data centers



What is a hinted handoff?

A recovery mechanism for writes targeting offline nodes. Coordinator can store a hinted handoff if target node for a write

- is known to be down, or
- fails to acknowledge

Coordinator stores the hint in its `system.hints` table. The write is replayed when the target node comes online

Understand and tune consistency

What is consistency?

Consistency Level: sets how many of the nodes to be sent a given request must acknowledge that request, for a response to be returned to the client

- Write request: how many nodes must acknowledge they received and wrote the write request?
- Read request: how many nodes must acknowledge by sending their most recent copy of the data?

Write consistency levels

Level	Description	Usage
ALL	A write must be written to the commit log and memtable on all replica nodes in the cluster for that partition.	Provides the highest consistency and the lowest availability of any other level.
EACH_QUORUM	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in *all* data centers.	Used in multiple data center clusters to strictly maintain consistency at the same level in each data center. For example, choose this level if you want a read to fail when a data center is down and the QUORUM cannot be reached on that data center.
QUORUM	A write must be written to the commit log and memtable on a quorum of replica nodes.	Provides strong consistency if you can tolerate some level of failure.
LOCAL_QUORUM	Strong consistency. A write must be written to the commit log and memtable on a quorum of replica nodes in the same data center as the coordinator node. Avoids latency of inter-data center communication.	Used in multiple data center clusters with a rack-aware replica placement strategy, such as NetworkTopologyStrategy, and a properly configured snitch. Use to maintain consistency locally (within the single data center). Can be used with SimpleStrategy.

ONE	A write must be written to the commit log and memtable of at least one replica node.	Satisfies the needs of most users because consistency requirements are not stringent.
TWO	A write must be written to the commit log and memtable of at least two replica nodes.	Similar to ONE.
THREE	A write must be written to the commit log and memtable of at least three replica nodes.	Similar to TWO.
LOCAL_ONE	A write must be sent to, and successfully acknowledged by, at least one replica node in the local datacenter.	In a multiple data center clusters, a consistency level of ONE is often desirable, but cross-DC traffic is not. LOCAL_ONE accomplishes this. For security and quality reasons, you can use this consistency level in an offline datacenter to prevent automatic connection to online nodes in other data centers if an offline node goes down.
ANY	A write must be written to at least one node. If all replica nodes for the given partition key are down, the write can still succeed after a hinted handoff has been written. If all replica nodes are down at write time, an ANY write is not readable until the replica nodes for that partition have recovered.	Provides low latency and a guarantee that a write never fails. Delivers the lowest consistency and highest availability.
SERIAL	Achieves linearizable consistency for lightweight transactions by preventing unconditional updates.	You cannot configure this level as a normal consistency level, configured at the driver level using the consistency level field. You configure this level using the serial consistency field as part of the native protocol operation. See failure scenarios.
LOCAL_SERIAL	Same as SERIAL but confined to the data center. A write must be written conditionally to the commit log and memtable on a quorum of replica nodes in	Same as SERIAL. Used for disaster recovery. See failure scenarios.

	the same data center.	

Read consistency levels

Level	Description	Usage
ALL	Returns the record after all replicas have responded. The read operation will fail if a replica does not respond.	Provides the highest consistency of all levels and the lowest availability of all levels.
EACH_QUORUM	Not supported for reads.	Not supported for reads.
QUORUM	Returns the record after a quorum of replicas has responded from any data center.	Ensures strong consistency if you can tolerate some level of failure.
LOCAL_QUORUM	Returns the record after a quorum of replicas in the current data center as the coordinator node has reported. Avoids latency of inter-data center communication.	Used in multiple data center clusters with a rack-aware replica placement strategy (NetworkTopologyStrategy) and a properly configured snitch. Fails when using SimpleStrategy.
ONE	Returns a response from the closest replica, as determined by the snitch. By default, a read repair runs in the background to make the other replicas consistent.	Provides the highest availability of all the levels if you can tolerate a comparatively high probability of stale data being read. The replicas contacted for reads may not always have the most recent write.
TWO	Returns the most recent data from two of the closest replicas.	Similar to ONE.
THREE	Returns the most recent data from three of the closest replicas.	Similar to TWO.
LOCAL_ONE	Returns a response from the closest replica in the local data center.	Same usage as described in the table about write consistency levels.
SERIAL	Allows reading the current (and possibly uncommitted) state of data without proposing a new addition or update. If a SERIAL read finds an uncommitted transaction in progress, it will commit the transaction as part of the read. Similar to QUORUM.	To read the latest value of a column after a user has invoked a lightweight transaction to write to the column, use SERIAL. Cassandra then checks the inflight lightweight transaction for updates and, if found, returns the latest data.
LOCAL_SERIAL	Same as SERIAL, but confined to the data center. Similar to LOCAL_QUORUM.	Used to achieve linearizable consistency for lightweight transactions.

About the QUORUM levels

$$quorum = (sum of replication_factors / 2) + 1$$

The sum of all the replication_factor settings for each data center is the sum of replication factors.

If consistency is a top priority, you can ensure that a read always reflects the most recent write by using the following formula:

$$(nodes written + nodes read) > replication factor $$

Understand the System keyspace

What is the System keyspace?

Cassandra stores its state in System keyspace tables

Some system keyspace tables:

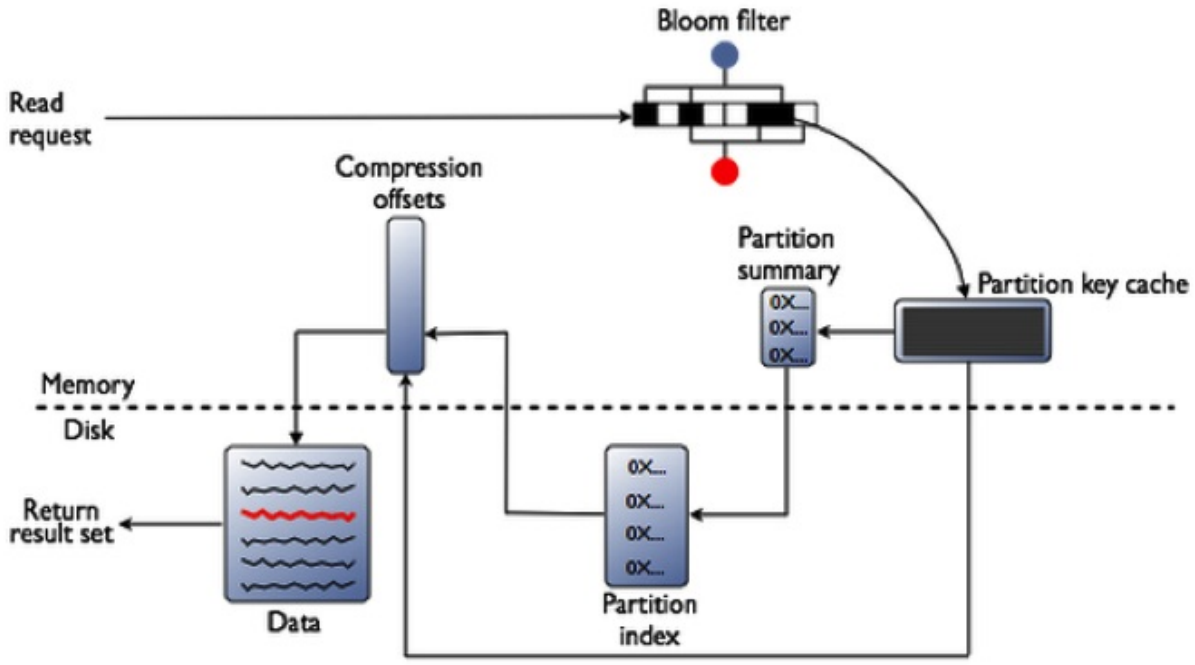
Table	Purpose
schema_keyspaces	Keyspaces available within this cluster, along with their assigned replication strategy and replication factor
schema_columns	Details of cluster compound primary key columns
schema_columns	Details of cluster tables and their configuration
peers	Local tracking of cluster-wide gossip for this node
local	Details of the local node's own state
hints	Stores information for hinted handoffs

How is data read

For a read request, Cassandra consults an in-memory data structure called a Bloom filter that checks the probability of an SSTable having the needed data. The Bloom filter can tell very quickly whether the file probably has the needed data, or certainly does not have it. If answer is a tenative yes, Cassandra consults another layer of in-memory caches, then fetches the compressed data on disk. If the answer is no, Cassandra doesn't trouble with reading that SSTable at all, and moves on to the next.

To satisfy a read, Cassandra must combine results from the active memtable and potentially multiple SSTables. Cassandra processes data at several stages on the read path to discover where the data is stored, starting with the data in the memtable and finishing with SSTables:

- Check the memtable
- Check row cache, if enabled
- Checks Bloom filter
- Checks partition key cache, if enabled
- Goes directly to the compression offset map if a partition key is found in the partition key cache, or checks the partition summary if not If the partition summary is checked, then the partition index is accessed
- Locates the data on disk using the compression offset map
- Fetches the data from the SSTable on disk

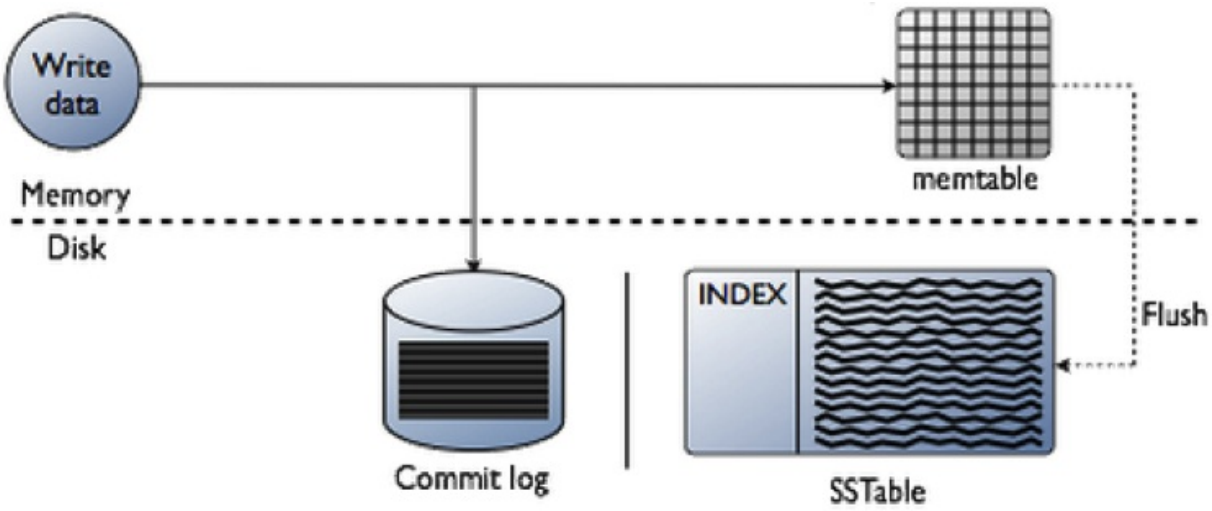


How is data written

Data written to a Cassandra node is first recorded in an on-disk commit log and then written to a memory-based structure called a memtable. When a memtable's size exceeds a configurable threshold, the data is written to an immutable file on disk called an SSTable. Buffering writes in memory in this way allows writes always to be a fully sequential operation, with many megabytes of disk I/O happening at the same time, rather than one at a time over a long period. This architecture gives Cassandra its legendary write performance.

Cassandra processes data at several stages on the write path, starting with the immediate logging of a write and ending in with a write of data to disk:

- Logging data in the commit log
- Writing data to the memtable
- Flushing data from the memtable
- Storing data on disk in SSTables



Cassandra data model

Understand the Cassandra data model

The Cassandra data model defines

- Column family as a way to store and organize data
- Table as a two-dimensional view of a multi-dimensional column family
- Operations on tables using the Cassandra Query Language (CQL)

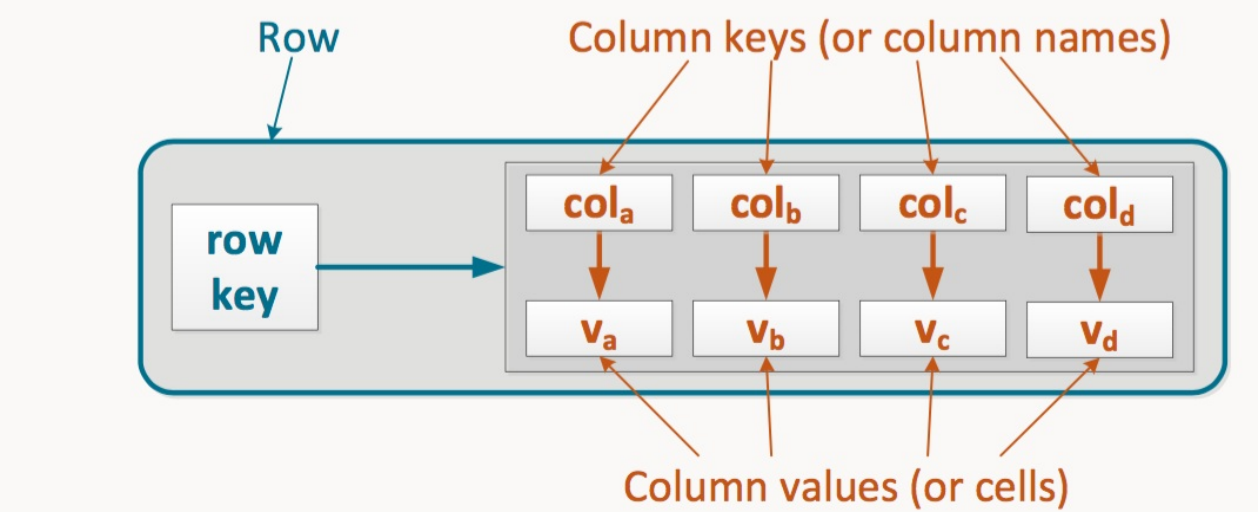
Cassandra1.2+reliesonCQLschema,concepts,andterminology, though the older Thrift API remains available

Table (CQL API terms)	Column Family (Thrift API terms)
Table is a set of partitions	Column family is a set of rows
Partition may be single or multiple row	Row may be skinny or wide
Partition key uniquely identifies a partition, and may be simple or composite	Row key uniquely identifies a row, and may be simple or composite
Column uniquely identifies a cell in a partition, and may be regular or clustering	Column key uniquely identies a cell in a row, and may be simple or composite
Primary key is comprised of a partition key plus clustering columns, if any, and uniquely identifies a row in both its partition and table	

Row (Partition)

Row is the smallest unit that stores related data in Cassandra

- Rows: individual rows constitute a column family
- Row key: uniquely identifies a row in a column family
- Row: stores pairs of column keys and column values
- Column key: uniquely identifies a column value in a row
- Column value: stores one value or a collection of values



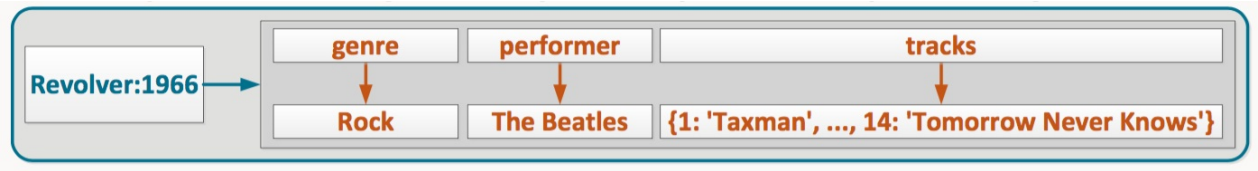
Rows may be described as `skinny` or `wide`

- Skinny row: has a fixed, relatively small number of column keys
- Wide row: has a relatively large number of column keys (hundreds or thousands); this number may increase as new data values are inserted

Key (Partition Key)

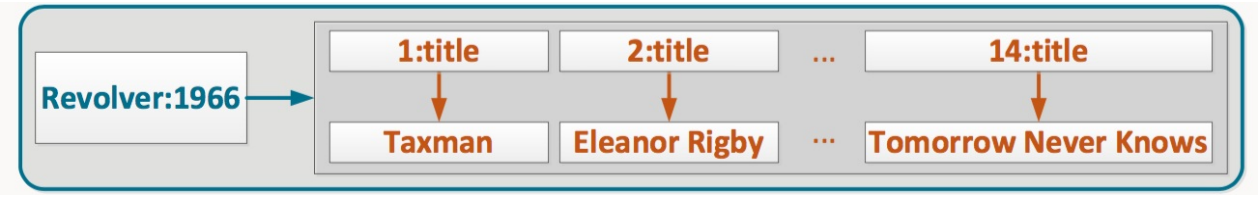
Composite row key

multiple components separated by colon



Composite column key

multiple components separated by colon



Column family (Table)

set of rows with a similar structure

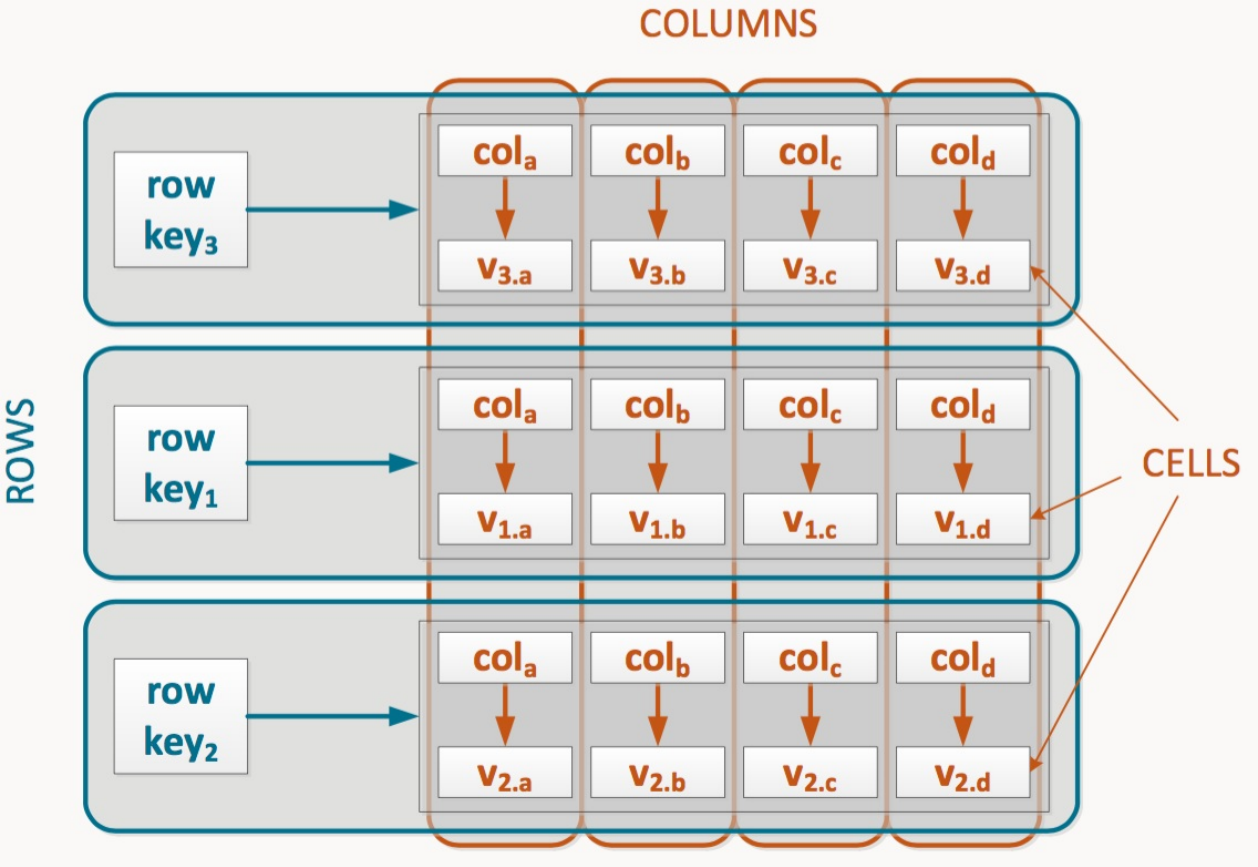


Table with single-row partitions

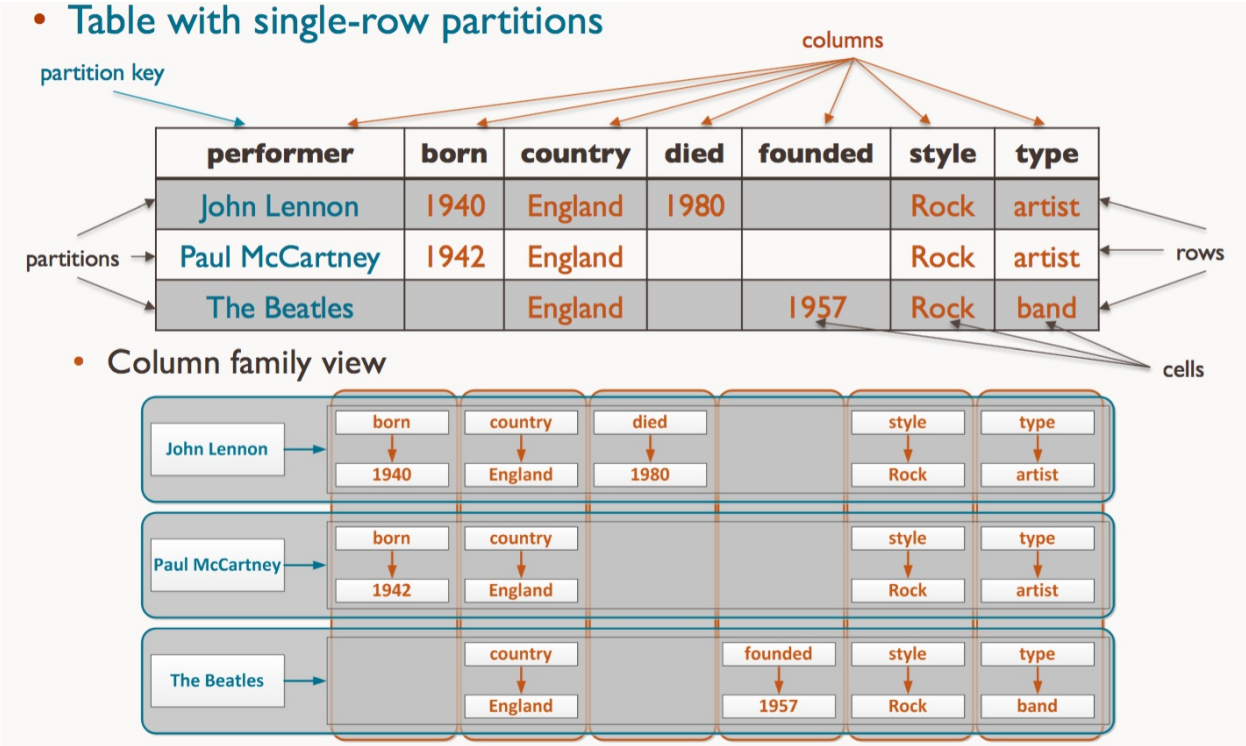
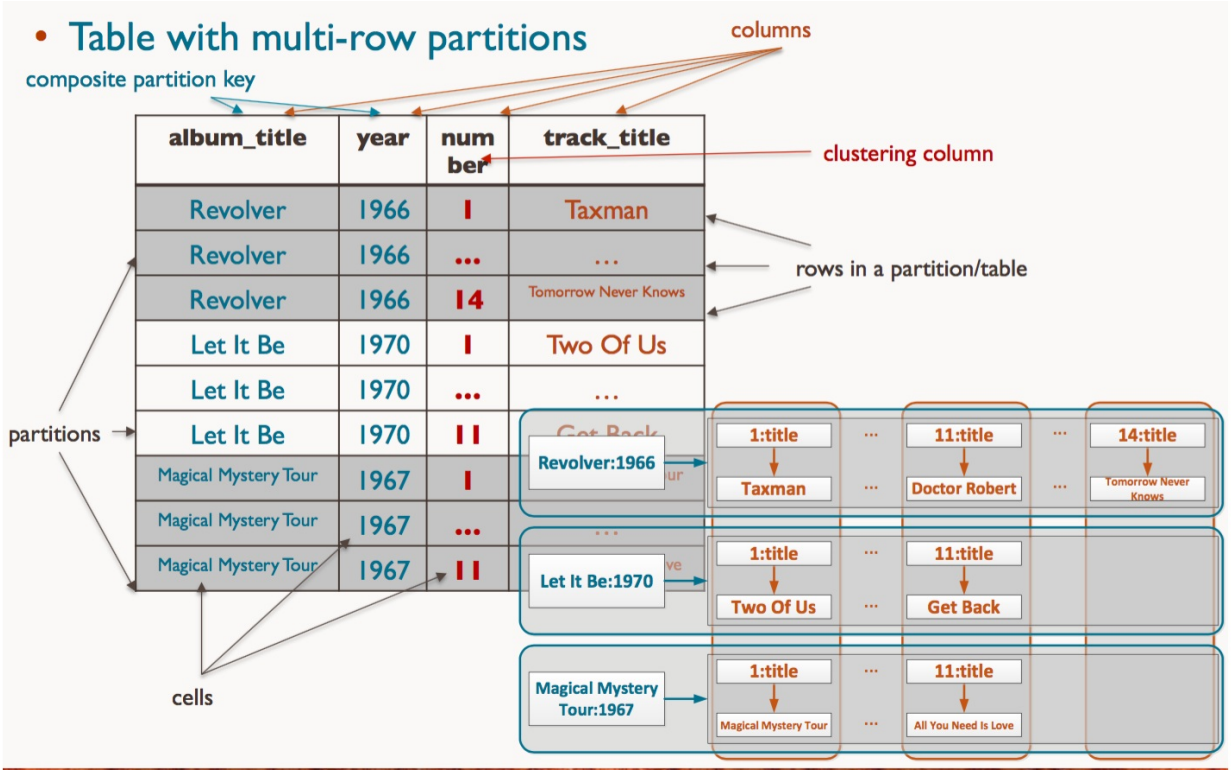


Table with multi-row partitions



Understand and use the DDL subset of CQL

Keyspace

a top-level namespace for a CQL table schema

- Defines the replication strategy for a set of tables
  - Keyspace per application is a good idea
- Data objects (e.g., tables) belong to a single keyspace

How are primary key, partition key, and clustering columns defined?

- Simple partition key, no clustering columns

  ```
  PRIMARY KEY ( partition_key_column )
  ```

- Composite partition key, no clustering columns

  ```
  PRIMARY KEY ( ( partition_key_col1, ..., partition_key_colN ) )
  ```

- Simple partition key and clustering columns

  ```
  PRIMARY KEY ( partition_key_column,clustering_column1, ..., clustering_columnM )
  ```

- Composite partition key and clustering columns

  ```
  PRIMARY KEY ( ( partition_key_col1, ..., partition_key_colN ), clustering_column1, ...,clustering_columnM ) )
  ```

UUID

universally unique identifiers

Format: `hex{8}-hex{4}-hex{4}-hex{4}-hex{12}`

TIMEUUID

- Embeds a time value within a UUID

- CQL function now() generates a new TIMEUUID
- CQL function dateOf() extracts the embedded timestamp as a date
- TIMEUUID values in clustering columns or in column names are ordered based on time

TIMESTAMP

64-bit integer representing a number of milliseconds since January 1 1970 at 00:00:00 GMT

COUNTER

- Cassandra supports distributed counters
- Useful for tracking a count
- Counter column stores a number that can only be updated
  - Incremented or decremented
  - Cannot assign an initial value to a counter (initial value is 0)
- Counter column cannot be part of a primary key
- If a table has a counter column, all non-counter columns must be part of a primary key

What is a secondary index?

- Can index additional columns to enable searching by those columns
- Cannot be created for
  - counter columns
  - static columns

When do you want to use a secondary index?

- Use with low-cardinality columns

  Columns that may contain a relatively small set of distinct values

Do not use:

- On high-cardinality columns
- On counter column tables
- On a frequently updated or deleted columns
- To look for a row in a large partition unless narrowly queried

Understand and use the DML subset of CQL

What is the syntax of the INSERT statement?

```
INSERT INTO table_name (column1, column2 ...) VALUES (value1, value2 ...)
```

What is the syntax of the UPDATE statement?

```
 UPDATE <keyspace>.<table> SET column_name1 = value, column_name2 = value WHERE primary_key_column = value;
```

- Row must be identified by values in primary key columns

What is an "upsert"?

- Both UPDATE and INSERT are write operations
- No reading before writing

What are lightweight transactions or ‘compare and set’?

Introduces a new clause IF NOT EXISTS for inserts

- Insert operation executes if a row with the same primary key does not exist
- Uses a consensus algorithm called Paxos to ensure inserts are done serially
- Multiple messages are passed between coordinator and replicas with a large performance penalty
- [applied] column returns true if row does not exist and insert executes
- [applied] column is false if row exists and the existing row will be returned

Update uses IF to verify the value for column(s) before execution

- [applied] column returns true if condition(s) matches and update written
- [applied] column is false if condition(s) do not match and the current row will be returned

What is the purpose of the BATCH statement?

BATCH statement combines multiple INSERT, UPDATE, and DELETE statements into a single logical operation

- Saves on client-server and coordinator-replica communication
- Atomic operation
  - If any statement in the batch succeeds, all will
- No batch isolation
  - Other “transactions” can read and write data being affected by a partially executed batch

Example:

```
BEGIN BATCH
  DELETE FROM albums_by_performer WHERE performer = 'The Beatles' AND year = 1966 AND tit
  INSERT INTO albums_by_performer (performer, year, title, genre) VALUES ('The Beatles',
APPLY BATCH;
```

Lightweight transactions in batch

- Batch will execute only if conditions for all lightweight transactions are met
- All operations in batch will execute serially with the increased performance overhead

Example:

```
BATCH
  UPDATE user SET lock = true IF lock = false; WHERE performer = 'The Beatles' AND year =
  INSERT INTO albums_by_performer (performer, year, title, genre) VALUES ('The Beatles',
  UPDATE user SET lock = false;
APPLY BATCH;
```

Resources

DS201 Course Slides Datastax Academy