

Intro to Cassandra and CQL

Erik Davidson



@aphistic



aphistic

Who Am I?

- Engineer on ShoreTel's Summit Platform team
- Implementing a scalable unified communications platform
- ShoreTel uses neat tech such as Cassandra, Docker, Mesos, Go
- I LOVE learning new tech and languages!

Why Am I Here?

- Learned the ins and outs of Cassandra over the last couple years while using it in production
- Came across a few “gotchas” that are good to know about
- Cassandra is extremely powerful if you know how to use it
- This is a “quick start” to introduce you to Cassandra and give you an idea of where to look for more information


What is Cassandra?

- Created by Facebook and open sourced in 2008
- Became an Apache Incubator project in 2009 and a top-level project in 2010
- Highly-available, decentralized database
- Scalable and fault-tolerant



“Old” Cassandra Usage

- Thrift protocol
- Column Families – Data grows sideways to billions of columns



	housewares	electronics	movies	...
Employees:Department	10	5	3	

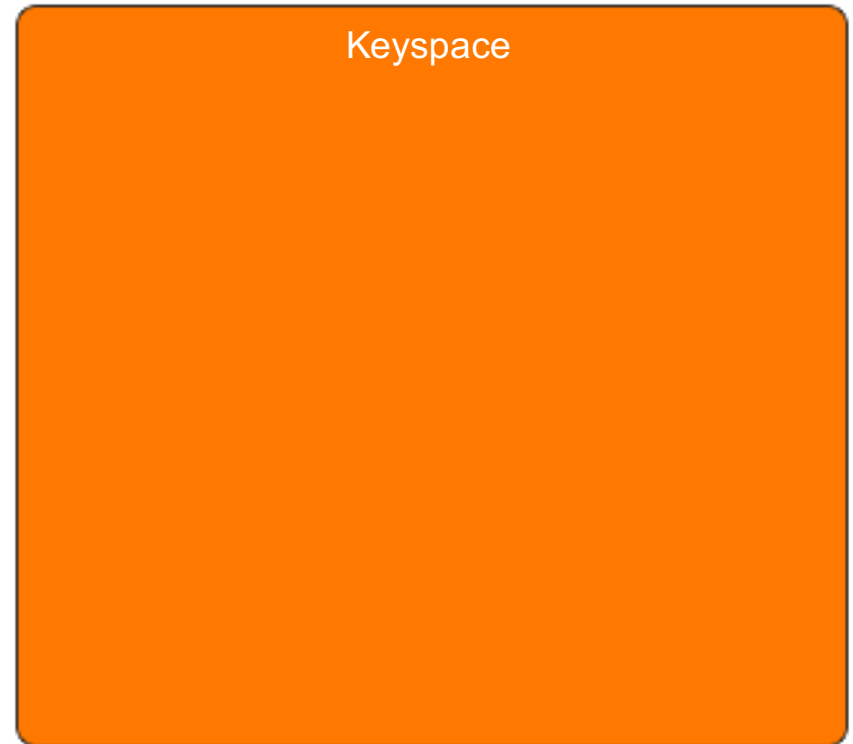
- No enforced schema, do whatever!
- This is no longer the recommended usage

“New” Cassandra Usage

- Concepts closer to a traditional relational database with tables, columns and rows
- New native protocol to replace Thrift
- Cassandra Query Language (CQL) – Best-practices language similar to SQL
- Still the same data underneath, just abstracted away
 - Great presentation on how data maps between the two can be found at <http://www.slideshare.net/DataStax/understanding-how-cql3-maps-to-cassandras-internal-data-structure>

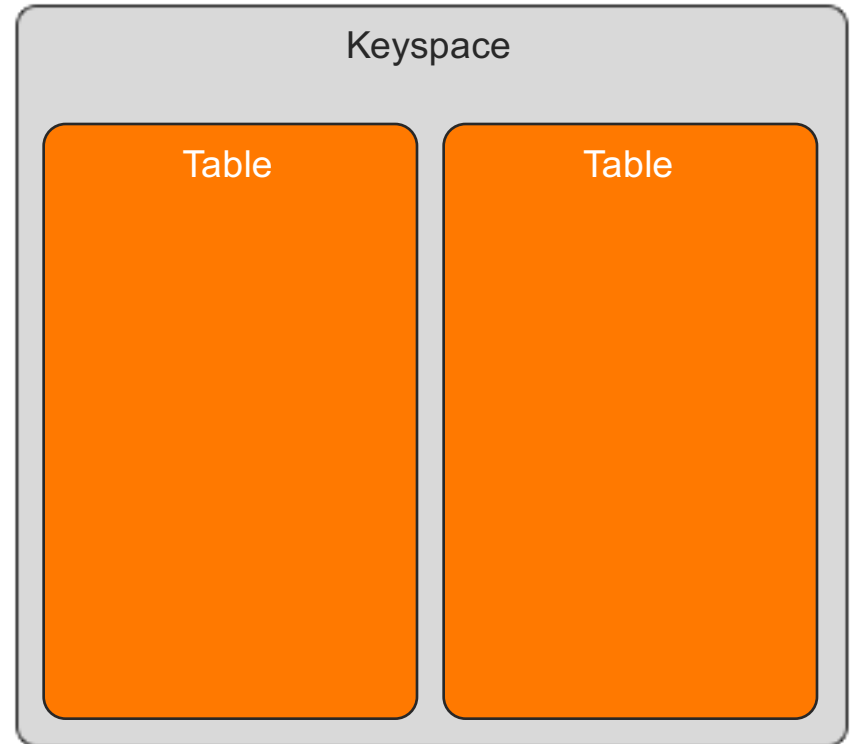
Cassandra / CQL Terminology

- Keyspace – Database in an RDBMS



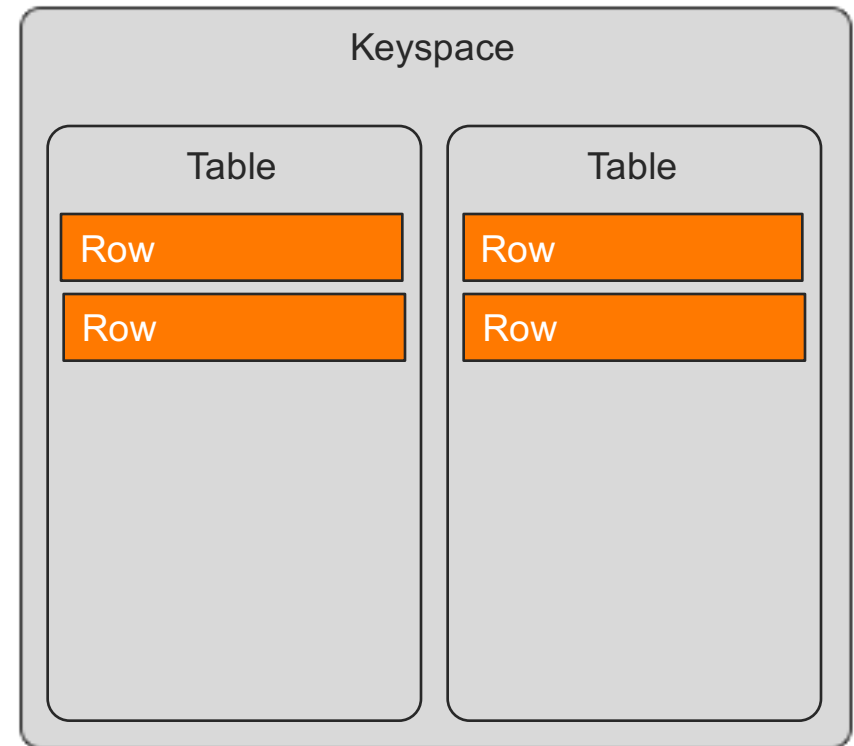
Cassandra / CQL Terminology

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS



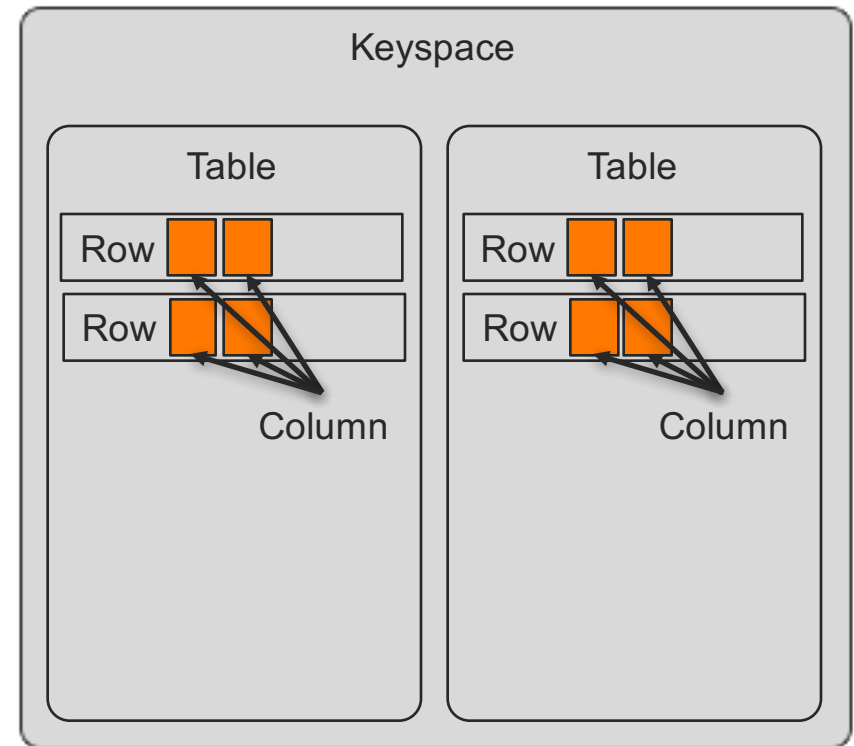
Cassandra / CQL Terminology

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS
- Row – Row in an RDBMS



Cassandra / CQL Terminology

- Keyspace – Database in an RDBMS
- Table – Table in an RDBMS
- Row – Row in an RDBMS
- Column – Column in an RDBMS



Data Types

- Strings
 - ascii, varchar
- Numbers
 - bigint, decimal, double, float, int
- Timestamp (includes date and time)
- Collections
 - list, map, set
- UUID
- ... And More!

CQL - SELECT

- Used to retrieve data from a table and can use a wildcard match:

SELECT * FROM my_table;

key1	col1	col2
1	My data 1	My data 2
2	Second data 1	Second data 2

- ... Or specific columns:

SELECT key1, col2 FROM my_table;

key1	col2
1	My data 2
2	Second data 2

- ... And can also be filtered by a WHERE clause:

SELECT * FROM my_table WHERE key1 = 1;

key1	col1	col2
1	My data 1	My data 2

CQL - INSERT

- Should look familiar if you've worked with SQL:

```
INSERT INTO my_table (key1, col1, col2)
VALUES (1, 'My data 1', 'My data 2');
```

- Would insert the following row into table **my_table**:

key1	col1	col2
1	My data 1	My data 2

- An INSERT query with the same primary key as existing data would result in updating the row

CQL - UPDATE

- Our first Cassandra-ism...
- UPDATE is the same as an INSERT!
- The following CQL UPDATE query:

```
UPDATE my_table  
SET col1 = 'my update 1', col2 = 'my update 2'  
WHERE key1 = 1;
```
- Is identical to the following CQL INSERT query:

```
INSERT INTO my_table (key1, col1, col2)  
VALUES (1, 'my update 1', 'my update2');
```

CQL - DELETE

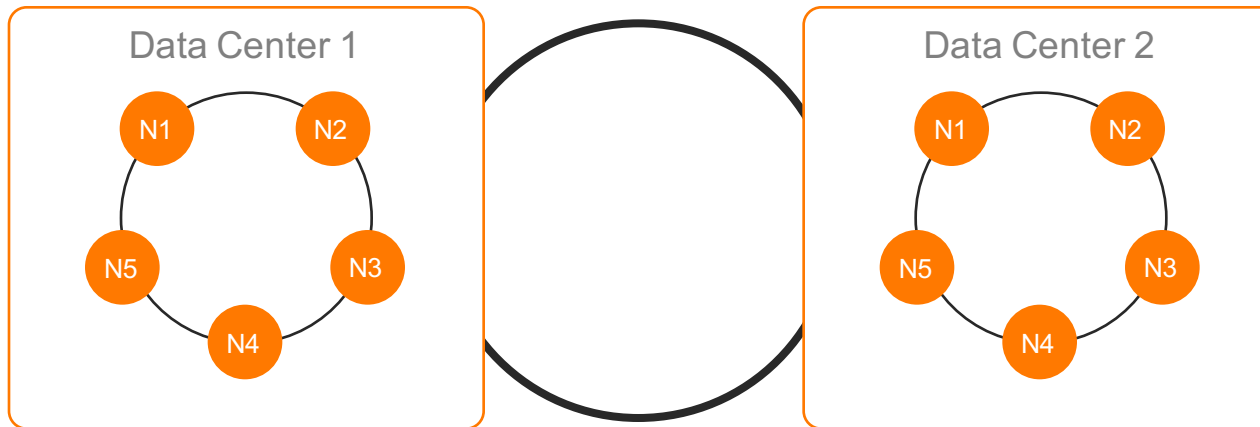
- Marks data for removal from a table. Unlike SQL, it MUST contain a WHERE clause:

```
DELETE FROM my_table WHERE key1 = 2;
```

- Wait, what do you mean “marks data for removal”?
- Data isn’t removed from disk right away, it’s removed later through a process called **compaction**
- More on **compaction** later!

Distributing Data in the Cluster

- Data is stored across multiple nodes and multiple data centers



- How does Cassandra know which nodes to put data on, though?

Table Primary Keys

- Cassandra uses the primary key of a row to determine where to store data in the cluster
- The primary key of a table can be a simple key with one column:

key1	col1	col2
1	My data 1	My data 2

- Or a composite key with multiple columns:

state	county	city	playgrounds	golf_courses
Wisconsin	Waukesha	Brookfield	26	2

- Sounds like any other RDBMS so far, right?

Partition Key

- A portion (or all) of the primary key used to partition different rows on to different nodes

Primary Key: (state, city)
Partition Key: state

Node 1

state	city	theaters
WI	Brookfield	5
WI	Milwaukee	15
MN	Minneapolis	20

Node 2

state	city	theaters
OR	Portland	21
NY	New York	1000000
NY	Buffalo	10

Primary Key: ((state, city))
Partition Key: (state, city)

Node 1

state	city	theaters
WI	Brookfield	5
NY	New York	1000000
MN	Minneapolis	20

Node 2

state	city	theaters
OR	Portland	21
WI	Milwaukee	15
NY	Buffalo	10

CQL – CREATE TABLE

- Looks similar to SQL CREATE:

```
CREATE TABLE state_theater (  
  state VARCHAR,  
  city VARCHAR,  
  theaters INT,  
  PRIMARY KEY ((state), city)  
);
```

- PRIMARY KEY looks a little different, though...

Partition Key
←→
PRIMARY KEY ((part1, part2), pk3)

- By default the partition key is the first column of a composite key:

```
PRIMARY KEY (state, city) = PRIMARY KEY ((state), city)
```

Beware of the “Gotchas”!

Partitioning Hot Nodes

- A “hot node” is a node that is working harder than the others
- Be careful of using partition keys that have an abnormally high amount of data compared to others
- Example, partition key of (city):

Node 1 - Table: shows

city	show	actors
New York	Fiddler on the Roof	30
New York	Les Miserables	40
New York	The Lion King	45
New York	School of Rock	35
Milwaukee	The Lion King	45


Node 2 - Table: shows

city	show	actors
Madison	School of Rock	35
Portland	Fiddler on the Roof	30
Denver	Les Miserables	40

- New York has an abnormally high amount of shows leading to Node 1 handling more queries than Node 2.

Query Limitations

- WHERE clauses can only contain primary key or indexed columns and the partition key must be included in full
- Example:



The diagram shows two horizontal double-headed arrows above the table. The top arrow, labeled 'Primary Key', spans from the 'state' column to the 'voters' column. The bottom arrow, labeled 'Partition Key', spans from the 'state' column to the 'section' column.

state	city	area	section	voters
WI	Milwaukee	1	2	25000
WI	Milwaukee	2	2	50000
NY	New York	3	1	100000
MN	Minneapolis	2	1	35000

`SELECT * FROM census WHERE state = 'WI';` -- Invalid, missing 'city' from partition key

`SELECT * FROM census WHERE state = 'WI' AND city = 'Milwaukee';` -- Returns 2 rows

`SELECT * FROM census WHERE state = 'WI' AND city = 'Milwaukee' AND section = 2;` -- Invalid, must include 'area'

`SELECT * FROM census WHERE state = 'WI' AND city = 'Milwaukee' AND area = 1 AND section = 2;` -- Returns 1 row

Consider Queries When Designing Schemas

- Due to query limitations you should consider how you will use your data when designing the schema
- Denormalization isn't bad!
- Can't decide on a good single table schema? Add the data you need to an additional table!

user

user_id	username
00000000-0000-0000-0000-000000000001	erikd
00000000-0000-0000-0000-000000000002	bq

user_id_lookup

username	user_id
erikd	00000000-0000-0000-0000-000000000001
bq	00000000-0000-0000-0000-000000000002

Be Careful of Collections

- Collections are really neat and powerful!
- If they grow too large they can degrade performance.
- Older versions of the protocol only allow access to 64k elements.
- Collections are fetched as a whole when retrieving them so the larger they are the more data needs to be fetched.

Be Careful of too Many DELETES

- When data is deleted a **tombstone** is placed
- During **compaction**, files on disk are optimized and space is reclaimed
- The more DELETES happen the more data needs to be compacted.
- Tombstones can only be deleted if all values for a specific row are all being compacted
- There are **major** and **minor** compactions, **minor** compactions may not remove all tombstoned data.
- **Major** compactions are started manually so they aren't run often

Eventual Consistency

- Data is not stored on all replication nodes atomically, it will **eventually** be replicated to other nodes
- Replication is typically on the order of milliseconds but in the case of a write collision, **last write wins**
- A **replication factor** can be specified when creating a Keyspace to define how many times data will be replicated
- A **replication strategy** determines how data will be replicated to the replication factor
- Use **NetworkTopologyStrategy** if you have more than one data center

Configurable Read/Write Consistency

- Each read and write query can specify a **consistency level** to allow better control over availability of data
- Common consistency levels:
 - ALL** – All nodes in the cluster must confirm (even non-local nodes)
 - QUORUM** – A quorum of nodes (half the replication factor plus one) in the cluster must confirm
 - ONE / TWO / THREE** – One, two or three nodes in the cluster must confirm
 - LOCAL_QUORUM** – A quorum of nodes in the local data center must confirm
 - LOCAL_ONE** – One node in the local data center must confirm

A Practical Example

- Since Cassandra doesn't have foreign keys how would I model a parent / child relationship between data?
- As an example, this is how you could model a number of companies and their employees:

Table: company, Primary Key: (company_id)

company_id	name
1	ShoreTel
2	Erik's Company

Table: employee, Primary Key: ((company_id), employee_id)

company_id	employee_id	first_name	last_name
1	1	John	Doe
1	2	Jane	Doe
2	3	Erik	Davidson
2	4	Grace	Hopper

- This schema would allow you to query the employee table using the **company_id** to retrieve all the employees for a specific company.

Thanks for Coming!

- Slides and a few example CQL files from this presentation are available on my GitHub:

<https://github.com/aphistic/intro-to-cassandra-and-cql>

- I'll tweet the URL above and also try to answer questions via Twitter:

[@aphistic](https://twitter.com/aphistic)

Questions?