Data Modeling in the New World with Apache Cassandra^{TM}

Jonathan Ellis
CTO, DataStax
Project chair, Apache Cassandra

Download & install

Cassandra

- http://planetcassandra.org/cassandra/

CQL Basics

CQL Basics

Cassandra **Q**uery **L**anguage

Keyspace – analogous to a schema.

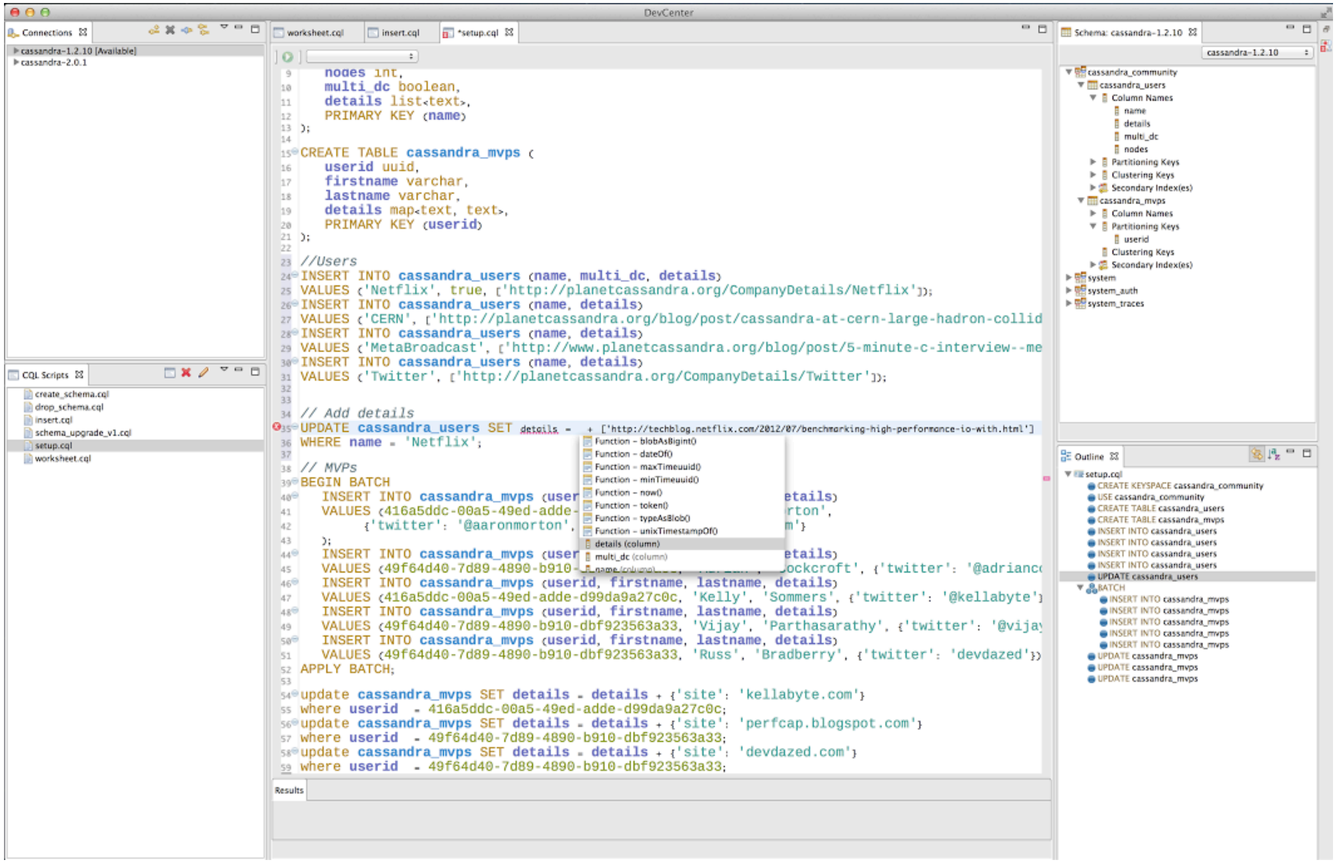
- Has various storage attributes.
- The keyspace determines the RF (replication factor).

Table – looks like a SQL Table.

- A table must have a Primary Key.
- We can fully qualify a table as <keyspace>.<table>

DevCenter

- DataStax DevCenter – a free, visual query tool for creating and running CQL statements against Cassandra and DataStax Enterprise.



```
    nodes int,
    multi_dc boolean,
    details list<text>,
    PRIMARY KEY (name)
);

CREATE TABLE cassandra_mvps (
    userid uuid,
    firstname varchar,
    lastname varchar,
    details map<text, text>,
    PRIMARY KEY (userid)
);

//Users
INSERT INTO cassandra_users (name, multi_dc, details)
VALUES ('Netflix', true, ['http://planetcassandra.org/CompanyDetails/Netflix']);
INSERT INTO cassandra_users (name, details)
VALUES ('CERN', ['http://planetcassandra.org/blog/post/cassandra-at-cern-large-hadron-collid
INSERT INTO cassandra_users (name, details)
VALUES ('MetaBroadcast', ['http://www.planetcassandra.org/blog/post/5-minute-c-interview--me
INSERT INTO cassandra_users (name, details)
VALUES ('Twitter', ['http://planetcassandra.org/CompanyDetails/Twitter']);


// Add details
UPDATE cassandra_users SET details =  + ['http://techblog.netflix.com/2012/07/benchmarking-high-performance-io-with.html']
WHERE name = 'Netflix';

// MVPs
BEGIN BATCH
   INSERT INTO cassandra_mvps (user                                      etails)
   VALUES (416a5ddc-00a5-49ed-adde-                                      rton',
         {'twitter': '@aaronmorton',                                     m'}
   );
   INSERT INTO cassandra_mvps (user                                      etails)
   VALUES (49f64d40-7d89-4890-b910-                                      ockcroft', {'twitter': '@adriancc
   INSERT INTO cassandra_mvps (userid, firstname, lastname, details)
   VALUES (416a5ddc-00a5-49ed-adde-d99da9a27c0c, 'Kelly', 'Sommers', {'twitter': '@kellabyte'}
   INSERT INTO cassandra_mvps (userid, firstname, lastname, details)
   VALUES (49f64d40-7d89-4890-b910-dbf923563a33, 'Vijay', 'Parthasarathy', {'twitter': '@vijay
   INSERT INTO cassandra_mvps (userid, firstname, lastname, details)
   VALUES (49f64d40-7d89-4890-b910-dbf923563a33, 'Russ', 'Bradberry', {'twitter': 'devdazed'})
APPLY BATCH;

update cassandra_mvps SET details = details + {'site': 'kellabyte.com'}
where userid  = 416a5ddc-00a5-49ed-adde-d99da9a27c0c;
update cassandra_mvps SET details = details + {'site': 'perfcap.blogspot.com'}
where userid  = 49f64d40-7d89-4890-b910-dbf923563a33;
update cassandra_mvps SET details = details + {'site': 'devdazed.com'}
where userid  = 49f64d40-7d89-4890-b910-dbf923563a33;
```

CQLSH

- Command line interface comes with Cassandra

- Launching on Linux

```
$ cqlsh [options] [host [port]]
```

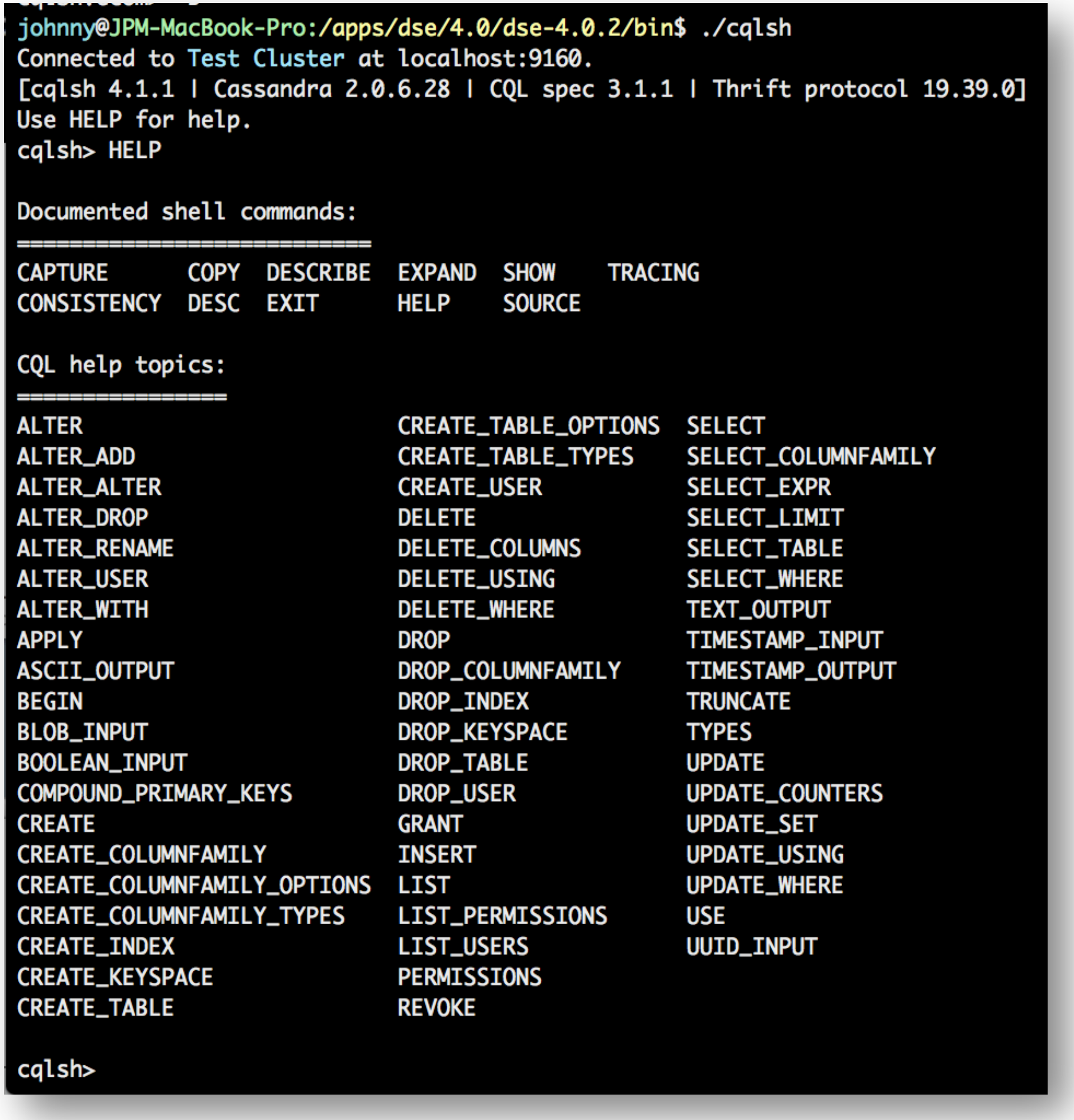
- Launching on Windows

```
python cqlsh [options] [host [port]]
```

- Example

```
$ cqlsh
$ cqlsh -u student -p cassandra 127.0.0.1 9160
```

CQLSH



```
johnny@JPM-MacBook-Pro:/apps/dse/4.0/dse-4.0.2/bin$ ./cqlsh
Connected to Test Cluster at localhost:9160.
[cqlsh 4.1.1 | Cassandra 2.0.6.28 | CQL spec 3.1.1 | Thrift protocol 19.39.0]
Use HELP for help.
cqlsh> HELP

Documented shell commands:
==========================
CAPTURE      COPY  DESCRIBE  EXPAND  SHOW    TRACING
CONSISTENCY  DESC  EXIT      HELP    SOURCE

CQL help topics:
================
ALTER                        CREATE_TABLE_OPTIONS  SELECT
ALTER_ADD                    CREATE_TABLE_TYPES    SELECT_COLUMNFAMILY
ALTER_ALTER                  CREATE_USER           SELECT_EXPR
ALTER_DROP                   DELETE                SELECT_LIMIT
ALTER_RENAME                 DELETE_COLUMNS        SELECT_TABLE
ALTER_USER                   DELETE_USING          SELECT_WHERE
ALTER_WITH                   DELETE_WHERE          TEXT_OUTPUT
APPLY                        DROP                  TIMESTAMP_INPUT
ASCII_OUTPUT                 DROP_COLUMNFAMILY     TIMESTAMP_OUTPUT
BEGIN                        DROP_INDEX            TRUNCATE
BLOB_INPUT                   DROP_KEYSPACE         TYPES
BOOLEAN_INPUT                DROP_TABLE            UPDATE
COMPOUND_PRIMARY_KEYS        DROP_USER             UPDATE_COUNTERS
CREATE                       GRANT                 UPDATE_SET
CREATE_COLUMNFAMILY          INSERT                UPDATE_USING
CREATE_COLUMNFAMILY_OPTIONS  LIST                  UPDATE_WHERE
CREATE_COLUMNFAMILY_TYPES    LIST_PERMISSIONS      USE
CREATE_INDEX                 LIST_USERS            UUID_INPUT
CREATE_KEYSPACE              PERMISSIONS
CREATE_TABLE                 REVOKE

cqlsh>
```

Non-CQL commands in cqlsh

Command	Description
CAPTURE	Captures command output and appends it to a file
CONSISTENCY	Shows the current consistency level, or given a level, sets it
COPY	Imports and exports CSV (comma-separated values) data
DESCRIBE	Provides information about a Cassandra cluster or data objects
EXIT	Terminates cqlsh
SHOW	Shows the Cassandra version, host, or data type assumptions
SOURCE	Executes a file containing CQL statements
TRACING	Enables or disables request tracing

What is keyspace or schema?

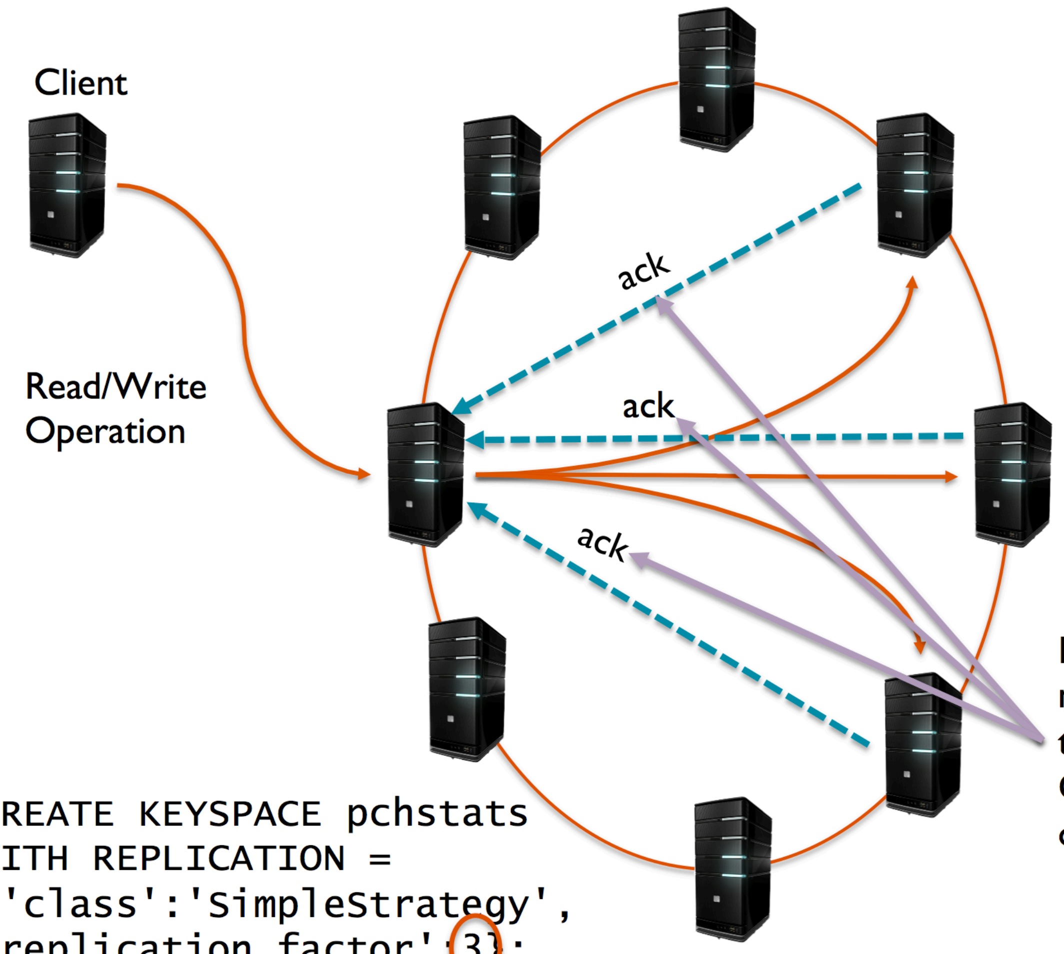
Keyspace or schema is a top-level namespace

- All data objects (e.g., tables) must belong to some keyspace
- Defines how data is replicated on nodes
- Keyspace per application is a good idea

Replica placement strategy

- SimpleStrategy (prototyping)
- NetworkTopologyStrategy (production)

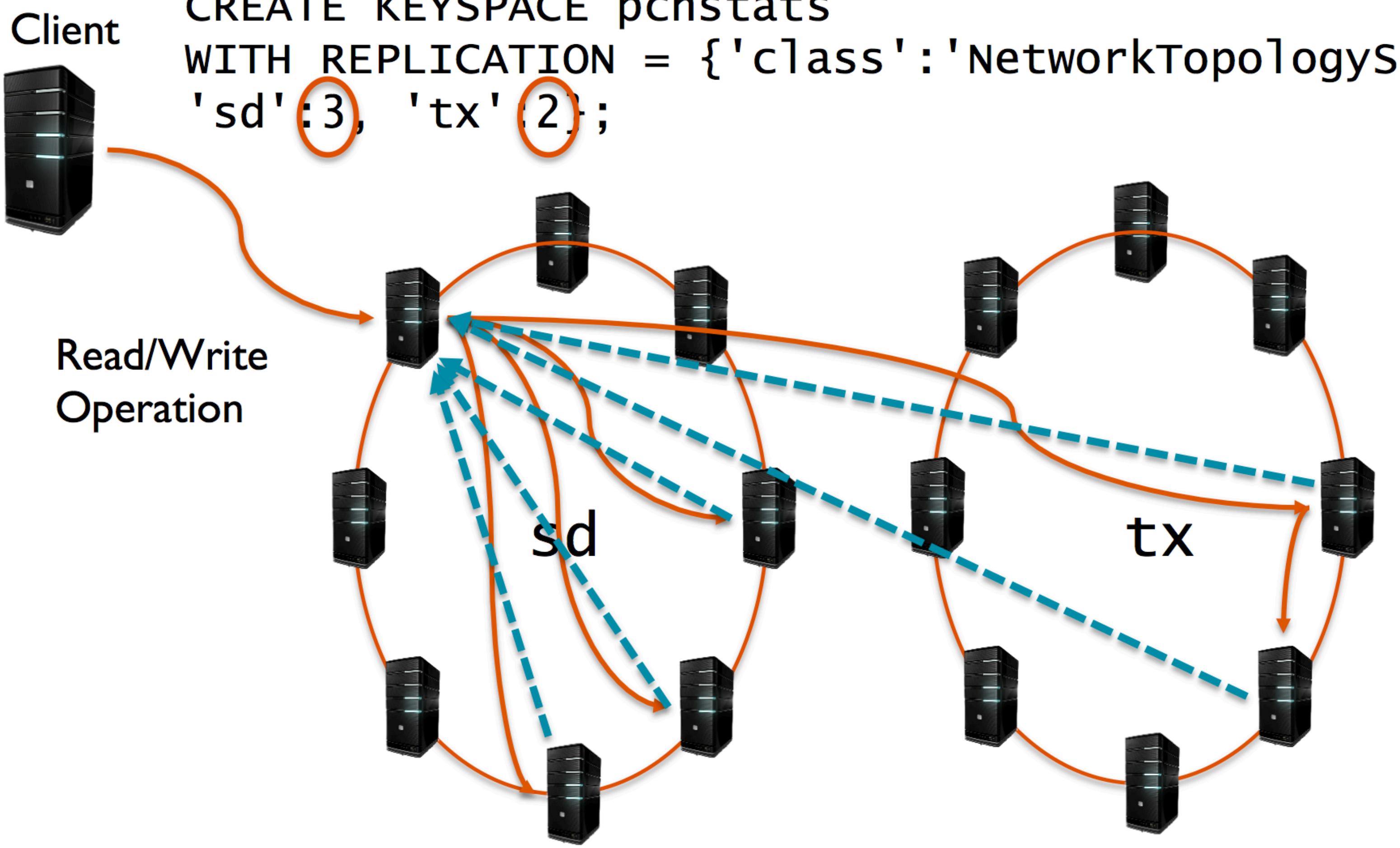
Creating a keyspace
Single Data Centre Consistency



```
CREATE KEYSPACE pchstats
WITH REPLICATION =
{'class':'SimpleStrategy',
'replication_factor':3};
```

Number of nodes that must acknowledge is tuned by setting the CONSISTENCY_LEVEL on any given operation

Creating a keyspace
Multiple Data Centre Consistency



```
CREATE KEYSPACE pchstats
WITH REPLICATION = {'class':'NetworkTopologyStrategy',
'sd':3, 'tx':2};
```

Use and Drop a keyspace

To work with data objects (e.g., tables) in a keyspace:

```
USE pchstats;
```

To delete a keyspace and all internal data objects

```
DROP KEYSPACE pchstats;
```

CQL Basics – creating a table

```
 CREATE TABLE cities (
    city_name   varchar,
    elevation   int,
    population  int,
    latitude    float,
    longitude   float,
    PRIMARY KEY (city_name)
);
```

In this example, the partition key = primary key

Compound Primary Key

The Primary Key

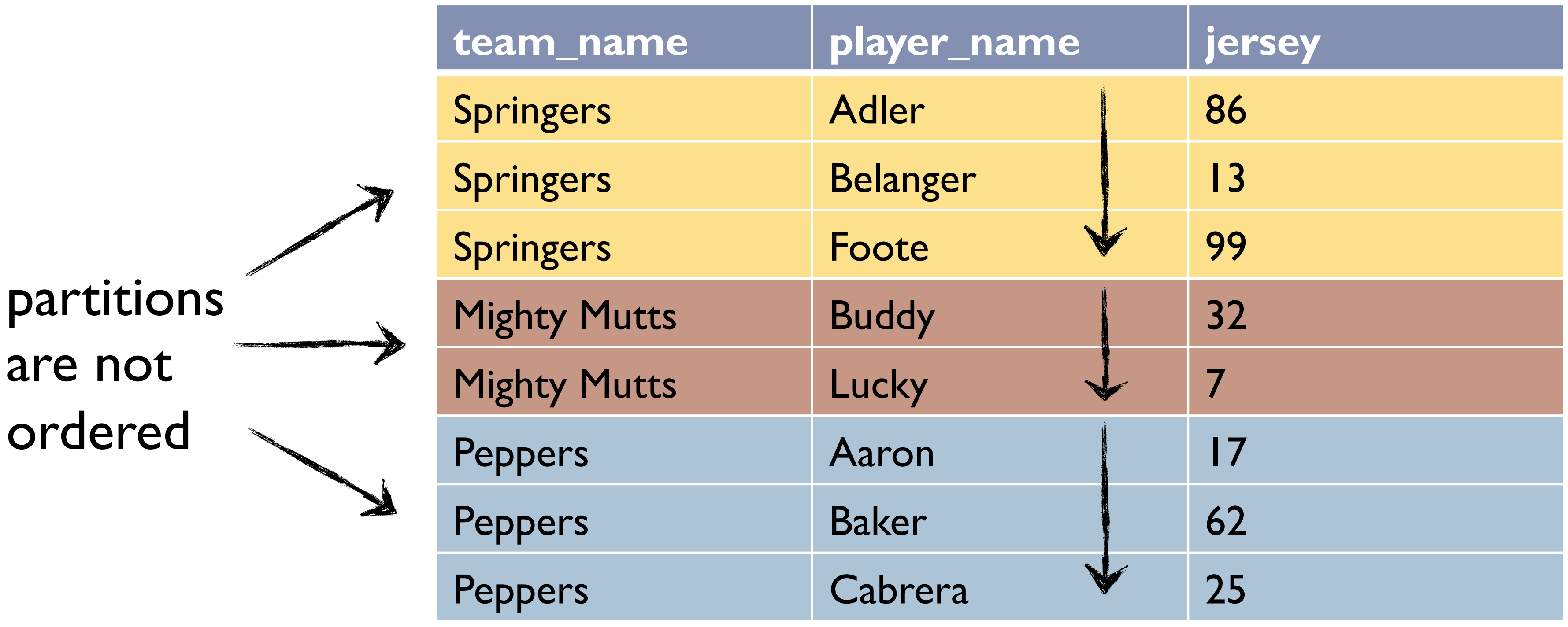
- The key uniquely identifies a row.
- A compound primary key consists of:
  - A **partition key**
  - One or more **clustering columns**

e.g. `PRIMARY KEY (partition key, cluster columns, ...)`

- The **partition key** determines on which node the partition resides
- Data is ordered in **cluster column** order within the partition

Compound Primary Key

```
CREATE TABLE sporty_league (
   team_name   varchar,
   player_name varchar,
   jersey      int,
   PRIMARY KEY (team_name, player_name)
);
```



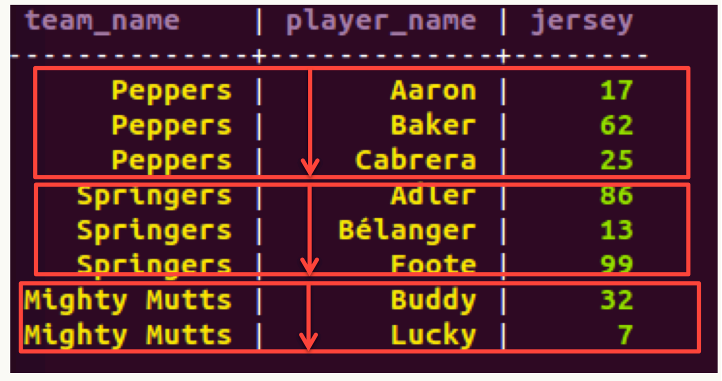
partitions are not ordered

team_name	player_name	jersey
Springers	Adler	86
Springers	Belanger	13
Springers	Foote	99
Mighty Mutts	Buddy	32
Mighty Mutts	Lucky	7
Peppers	Aaron	17
Peppers	Baker	62
Peppers	Cabrera	25

Rows within partition clustered by player_name

Simple Select

```
SELECT * FROM sporty_league;
```

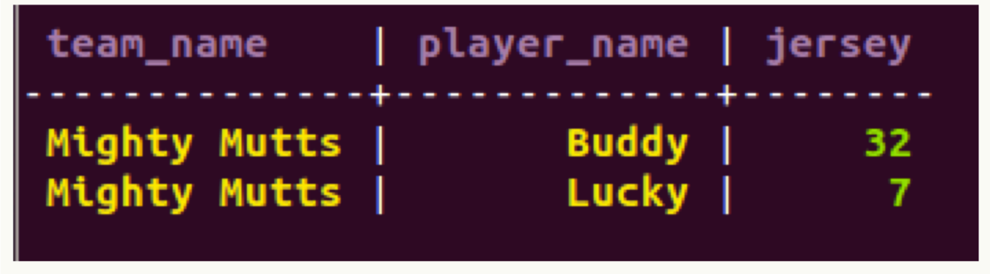


```
 team_name     | player_name | jersey
---------------+-------------+--------
       Peppers |       Aaron |     17
       Peppers |       Baker |     62
       Peppers |     Cabrera |     25
     Springers |       Adler |     86
     Springers |    Bélanger |     13
     Springers |       Foote |     99
  Mighty Mutts |       Buddy |     32
  Mighty Mutts |       Lucky |      7
```

- More that a few rows can be slow.
- Use **LIMIT** keyword to choose fewer or more rows

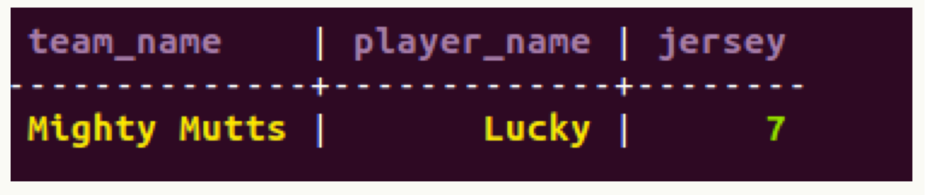
Simple Select on
Partition Key and Cluster Colum

```
SELECT * FROM sporty_league
WHERE team_name = 'Mighty Mutts';
```



```
team_name     | player_name | jersey
--------------+-------------+--------
Mighty Mutts  |       Buddy |     32
Mighty Mutts  |       Lucky |      7
```

```
SELECT * FROM sporty_league
WHERE team_name = 'Mighty Mutts'
and player_name = 'Lucky';
```



```
team_name     | player_name | jersey
--------------+-------------+--------
Mighty Mutts  |       Lucky |      7
```

ORDER BY

- Only allowed for single-partition queries
- Only allowed against clustering columns
- Data will returned by default in the order of the clustering column
- ASC or DESC can override the default

```
SELECT * FROM sporty_league
WHERE team_name = ‘Mighty Mutts’
ORDER BY player_name DESC;
```

CLUSTERING ORDER BY clause

Defines on-disk ordering of rows in a partition

```
CREATE TABLE albums_by_genre (
  genre VARCHAR,
  performer VARCHAR,
  year INT,
  title VARCHAR,
  PRIMARY KEY (genre, performer, year, title)
) WITH CLUSTERING ORDER BY
       (performer ASC, year DESC, title ASC);
```

Predicates

- **On the partition key:** = and IN
- **On the cluster columns:** <, <=, =, >=, >, IN

Insert/Update

```
INSERT INTO sporty_league (team_name, player_name, jersey)
VALUES ('Mighty Mutts',’Felix’,90);

UPDATE sporty_league SET jersey = 77
WHERE team_name = 'Mighty Mutts’ AND player_name = ‘Felix’;
```

Primary key columns uniquely identify the row and are mandatory

- No multi-row update predicates

Writes isolated from reads

- No updated columns are visible until entire row is finished
  - (technically, entire partition)

What is an upsert?

UPdate + inSERT

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

Term “upsert” denotes the following behavior

- INSERT updates or overwrites an existing row
  - When inserting a row in a table that already has another row with the same values in primary key columns
- UPDATE inserts a new row
  - When a to-be-updated row, identified by values in primary key columns, does not exist
- **Upserts are legal and do not result in error or warning messages**

How to avoid UPSERTS

Guarantee that your primary keys are unique from one another

- Use an appropriate natural key based on your data
- Use a surrogate key for partition key

Use lightweight transactions

- INSERT … IF NOT EXISTS

Surrogate keys in Cassandra

RDBMS typically use sequences

- MS SQL IDENTITY, MYSQL AUTO_INCREMENT

-
  ```
  INSERT INTO user (id, firstName, LastName)
  VALUES (seq.nextVal(), ‘Ted’, ‘Codd’)
  ```

Cassandra has no sequences!

- Requires a lock (performance killer)
- Requires coordination (availability killer)

What to do?

- Use part of the data to create a unique key
- Use a UUID

UUID

- Universal Unique ID
- 128 bits
  - 99051fe9-6a9c-46c2-b949-38ef78858dd0
- Easily generated on the client
- Version 1 has a timestamp component (TIMEUUID)
- Version 4 has no timestamp component
  - Faster to generate

TIMEUUID

TIMEUUID data type supports Version 1 UUIDs

- Generated using time (60 bits), a clock sequence number (14 bits), and MAC* address (48 bits)
  - **CQL function 'now()' generates a new TIMEUUID**
- 1be43390-9fe4-11e3-8d05-425861b86ab6
- Time can be extracted from TIMEUUID
  - **CQL function dateOf() extracts the timestamp as a date**
- TIMEUUID values in clustering columns or in column names are ordered based on time
  - **DESC order on TIMEUUID lists most recent data first**

UUID Example

Example

- Users are identified by UUID
- User activities (i.e., rating a track) are identified by TIMEUUID
  - A user may rate the same track multiple times
  - Activities are ordered by the time component of TIMEUUID

```
CREATE TABLE track_ratings_by_user (
  user UUID,
  activity TIMEUUID,
  rating INT,
  album_title VARCHAR,
  album_year INT,
  track_title VARCHAR,
  PRIMARY KEY (user, activity)
) WITH CLUSTERING ORDER BY (activity DESC);
```

Exercise 1

Creating a keyspace and table

Exercise 1

- Install Cassandra
- CREATE KEYSPACE demo
- CREATE TABLE users
  - id
  - email
  - Password
- CREATE TABLE tweets
  - author
  - created_at
  - body
  - id?

cqlsh tab completion is your friend!

Exercise 1

Who used a uuid for the primary key?

Benefits? Drawbacks?

Performance considerations

- The best queries are in a single partition.
  i.e. WHERE partition key = <something>
- Each new partition requires a new disk seek.
- Queries that span multiple partitions are **s-l-o-w**
- Queries that span multiple clustered rows are **fast**

ALTER TABLE

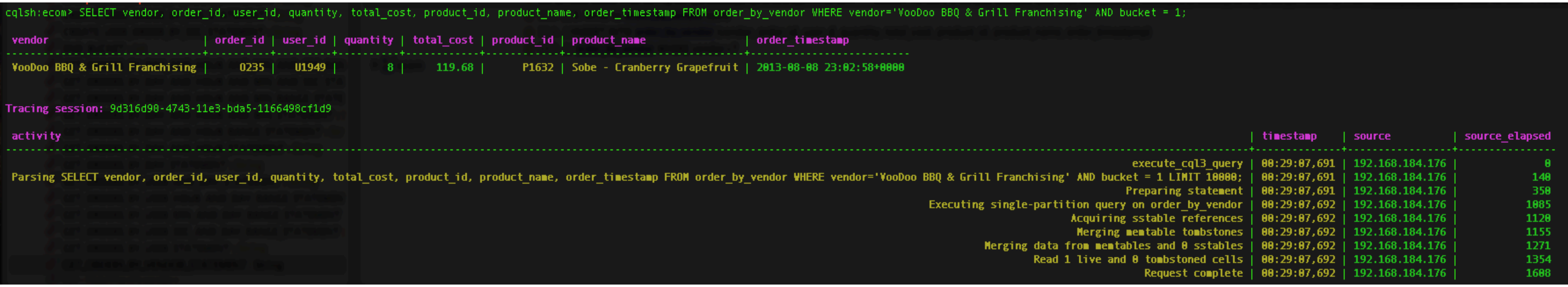
- ALTER TABLE x ADD y <type>;
- ALTER TABLE x DROP y;

Authentication and Authorisation

- CQL supports creating users and granting them access to tables etc..
- You need to enable authentication in the cassandra.yaml config file.
- You can create, alter, drop and list users
- You can then GRANT permissions to users accordingly – ALTER, AUTHORIZE, DROP, MODIFY, SELECT.

Query Tracing

- You can turn on tracing on or off for queries with the TRACING ON | OFF command.
- This can help you understand what Cassandra is doing and identify any performance problems.



```
cqlsh:ecom> SELECT vendor, order_id, user_id, quantity, total_cost, product_id, product_name, order_timestamp FROM order_by_vendor WHERE vendor='VooDoo BBQ & Grill Franchising' AND bucket = 1;

 vendor                         | order_id | user_id | quantity | total_cost | product_id | product_name                | order_timestamp
--------------------------------+----------+---------+----------+------------+------------+-----------------------------+--------------------------
 VooDoo BBQ & Grill Franchising |     0235 |   U1949 |        8 |     119.68 |      P1632 | Sobe - Cranberry Grapefruit | 2013-08-08 23:02:58+0000


Tracing session: 9d316d90-4743-11e3-bda5-1166498cf1d9

 activity                                                                                                                                                                                                 | timestamp    | source          | source_elapsed
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------+--------------+-----------------+----------------
                                                                                                                                                                                       execute_cql3_query | 00:29:07,691 | 192.168.184.176 |              0
 Parsing SELECT vendor, order_id, user_id, quantity, total_cost, product_id, product_name, order_timestamp FROM order_by_vendor WHERE vendor='VooDoo BBQ & Grill Franchising' AND bucket = 1 LIMIT 10000; | 00:29:07,691 | 192.168.184.176 |            140
                                                                                                                                                                                      Preparing statement | 00:29:07,691 | 192.168.184.176 |            350
                                                                                                                                                      Executing single-partition query on order_by_vendor | 00:29:07,692 | 192.168.184.176 |           1085
                                                                                                                                                                             Acquiring sstable references | 00:29:07,692 | 192.168.184.176 |           1120
                                                                                                                                                                              Merging memtable tombstones | 00:29:07,692 | 192.168.184.176 |           1155
                                                                                                                                                               Merging data from memtables and 0 sstables | 00:29:07,692 | 192.168.184.176 |           1271
                                                                                                                                                                       Read 1 live and 0 tombstoned cells | 00:29:07,692 | 192.168.184.176 |           1354
                                                                                                                                                                                         Request complete | 00:29:07,692 | 192.168.184.176 |           1608
```

- http://www.datastax.com/dev/blog/tracing-in-cassandra-1-2

What CQL data types are available?

CQL Type	Constants	Description
ASCII	strings	US-ASCII character string
BIGINT	integers	64-bit signed long
BLOB	blobs	Arbitrary bytes (no validation), expressed as hexadecimal
BOOLEAN	booleans	true or false
COUNTER	integers	Distributed counter value (64-bit long)
DECIMAL	integers, floats	Variable-precision decimal
DOUBLE	integers	64-bit IEEE-754 floating point
FLOAT	integers, floats	32-bit IEEE-754 floating point
INET	strings	IP address string in IPv4 or IPv6 format*
INT	integers	32-bit signed integer
LIST	n/a	A collection of one or more ordered elements
MAP	n/a	A JSON-style array of literals: { literal : literal, literal : literal ... }
SET	n/a	A collection of one or more elements
TEXT	strings	UTF-8 encoded string
TIMESTAMP	integers, strings	Date plus time, encoded as 8 bytes since epoch
UUID	uuids	A UUID in standard UUID format
TIMEUUID	uuids	Type 1 UUID only (CQL 3)
VARCHAR	strings	UTF-8 encoded string
VARINT	integers	Arbitrary-precision integer

Collection Data Type

CQL supports having columns that contain collections of data.

The collection types include:

- Set, List and Map.

```
CREATE TABLE collections_example (
     id int PRIMARY KEY,
     set_example set<text>,
     list_example list<text>,
     map_example map<int, text>
);
```

These data types are intended to support the type of 1-to-many relationships that can be modeled in a relational DB e.g. a user has many email addresses.

Some performance considerations around collections.

- Requires serialization so don’t go crazy!
- Often more efficient to denormalise further rather than use collections if intending to store lots of data.
- **Favour sets over list – lists not as performant**

Watch out for collection indexing in Cassandra 2.1!

Collection considerations

- Designed to store a small amount of data
- A collection is retrieved in its entirety
- Maximum number of elements in a collection is 64 thousands
    - In practice – hundreds
- Maximum size of element values is 64 KB
- Collection columns cannot be part of a primary key
    - No collections in a partition key
    - No collections in clustering columns
- Cannot nest a collection inside of another collection

Counters

- Stores a number that incrementally counts the occurrences of a particular event or process.
- **Note: If a table has a counter column, all non-counter columns must be part of a primary key**

```
CREATE TABLE UserActions (
  user VARCHAR,
  action VARCHAR,
  total COUNTER,
  PRIMARY KEY (user, action)
);
```

```
UPDATE UserActions SET total = total + 2
  WHERE user = 123 AND action = ’xyz';
```

Counter Considerations

Performance considerations

- Read is as efficient as for non-counter columns
- Update is fast but slightly slower than an update for non-counter columns
  - A read is required before a write can be performed

Accuracy considerations

- If a counter update is timed out, a client application cannot simply retry a “failed” counter update as the timed-out update may have been persisted
  - Counter update is not an idempotent operation

Static columns

```
CREATE TABLE bills (
    user text,
    balance int static,
    expense_id int,
    amount int,
    description text,
    paid boolean,
    PRIMARY KEY (user, expense_id)
);
```

Lightweight Transactions (LWT)

Why?

- Solve a class of race conditions in Cassandra that you would otherwise need to install an external locking manager to solve.

Syntax:

```
INSERT INTO customer_account (customerID, customer_email)
VALUES (‘Johnny’, ‘jmiller@datastax.com’)
IF NOT EXISTS;


UPDATE  customer_account
SET customer_email=’jmiller@datastax.com’
IF customer_email=’jmiller@datastax.com’;
```

Example Use Case:

- Registering a user

Not Will Ferrell @itsWillyFerrell · Apr 5
In about 20 years, the hardest thing our kids will have to do is find a username that isn't taken.

Lightweight Transactions

- **Uses Paxos algorthim**
  - All operations are quorum-based i.e. we can loose nodes and its still going to work!
  - See *Paxos Made Simple* - http://bit.ly/paxosmadesimple
- **Consequences of Lightweight Transactions**
  - 4 round trips vs. 1 for normal updates
  - Operations are done on a per-partition basis
  - Will be going across data centres to obtain consensus (unless you use LOCAL_SERIAL consistency)
  - Cassandra user will need read and write access i.e. you get back the row!

Great for 1% your app, but eventual consistency is still your friend!

Find out more:

- http://www.datastax.com/dev/blog/lightweight-transactions-in-cassandra-2-0
- **Eventual Consistency != Hopeful Consistency** http://www.youtube.com/watch?v=A6qzx_HE3EU

Batch Statements

```
BEGIN BATCH
  INSERT INTO users (userID, password, name) VALUES ('user2', 'ch@ngem3b', 'second user')
  UPDATE users SET password = 'ps22dhds' WHERE userID = 'user2'
  INSERT INTO users (userID, password) VALUES ('user3', 'ch@ngem3c')
  DELETE name FROM users WHERE userID = 'user2’
APPLY BATCH;
```

- 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

No semicolon after BEGIN BATCH! Fixed in 2.0.9

Batch Statements

BEGIN UNLOGGED BATCH

- Does not write to the batchlog
- More performant, but **no longer atomic**

BEGIN COUNTER BATCH

- Only for counter mutations

Batch Statements

All conditions are applied to all changes to that partition

```
CREATE TABLE log (
    log_name text,
    seq int static,
    logged_at timeuuid,
    entry text,
    primary key (log_name, logged_at)
);

INSERT INTO log (log_name, seq)
VALUES ('foo', 0);
```

Atomic log appends

```
BEGIN BATCH

UPDATE log SET seq = 1
WHERE log_name = 'foo'
IF seq = 0;

INSERT INTO log (log_name, logged_at, entry)
VALUES ('foo', now(), 'test');

APPLY BATCH;
```

Secondary Indexes

- This gives you fast access to data
- If we want to do a query on a column that is not part of your PK, you can create an index:

```
        CREATE INDEX ON <table>(<column>);
```

- Can be created on any column except counter, static and collection columns
- Than you can do a select:

```
    SELECT * FROM product WHERE type= ’PC';
```

- **Avoid doing this for high volume queries!**
  - Scatter/gather required
- **Much more efficient to model your data around the query i.e. roll your own indexes!!**

When do you want to use a secondary index?

- **Secondary indexes are for searching convenience**
  - Use with low-cardinality columns
  - Columns that may contain a relatively small set of distinct values
  - Use when prototyping, ad-hoc querying or with smaller datasets

- **Do not use**
  - On high-cardinality columns
  - In tables that use a counter column
  - On a frequently updated or deleted column
  - To look for a row in a large partition
    - **unless narrowly queried a search on both a partition key and an indexed column**

Keyword index example

Video table defined as:

```
CREATE TABLE videos (
  videoid uuid,
  videoname varchar,
  username varchar,
  description varchar,
  tags varchar,
  upload_date timestamp,
  PRIMARY KEY(videoid)
);
```

Now we can define an index for tagging videos

```
CREATE TABLE video_tag_index (
  tag varchar,
  videoid uuid,
  timestamp timestamp
  PRIMARY KEY(tag, videoid)
);
```

Partial word index example

Table:

```
CREATE TABLE email_index (
        domain varchar,
        user varchar,
        username varchar,
        PRIMARY KEY (domain, user)
)
```

User: jmiller, Email: jmiller@datastax.com

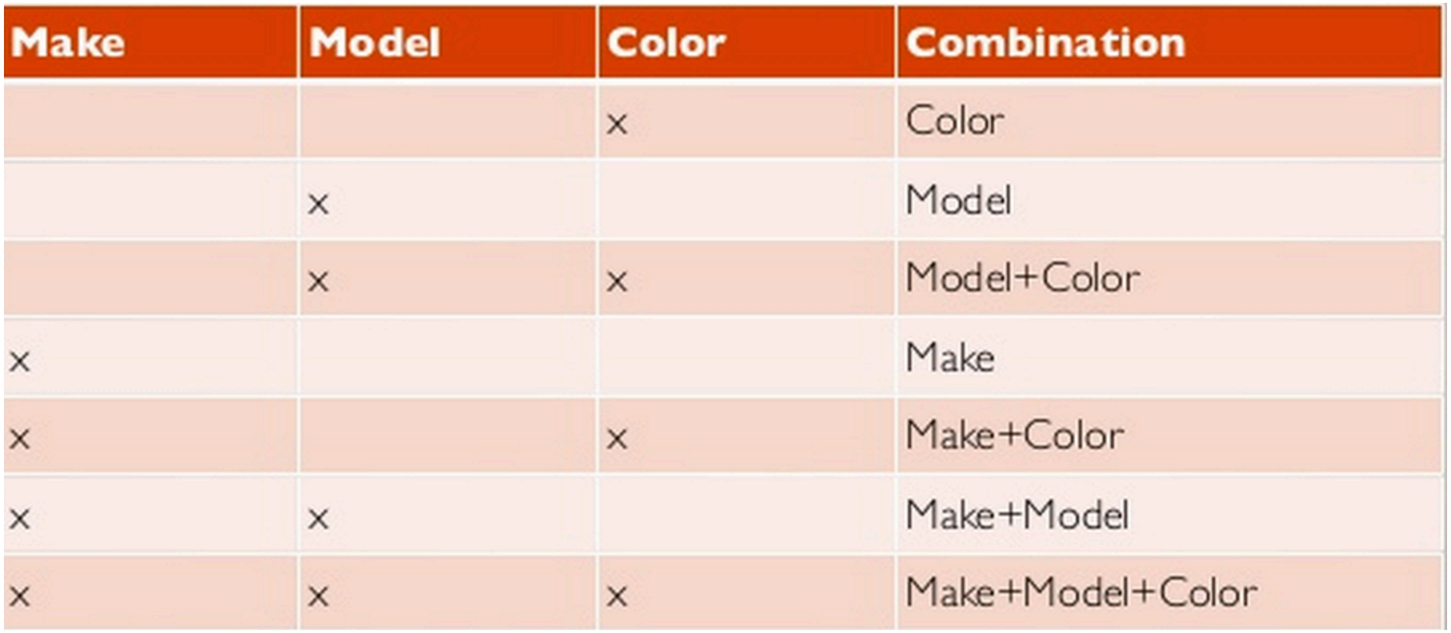
```
INSERT INTO email_index (domain, user, username)
VALUES (‘@datastax.com’, ‘jmiller’, ‘jmiller’)
```

Bitmap(ish) Index Example

- Multiple parts to a key
- Create a truth table of the various combinations
- However, inserts == the number of combinations

Bitmap(ish) Index Example

Find a car in a car park by variable combinations



Make	Model	Color	Combination
		x	Color
	x		Model
	x	x	Model+Color
x			Make
x		x	Make+Color
x	x		Make+Model
x	x	x	Make+Model+Color

Bitmap(ish) index example

Make a table with three different key combinations

```
CREATE TABLE car_location_index (
      make varchar,
      model varchar,
      colour varchar,
      vehicle_id int,
      lot_id int,
      PRIMARY KEY ((make, mode, colour), vehicle_id)
);
```

Bitmap(ish) Index Example

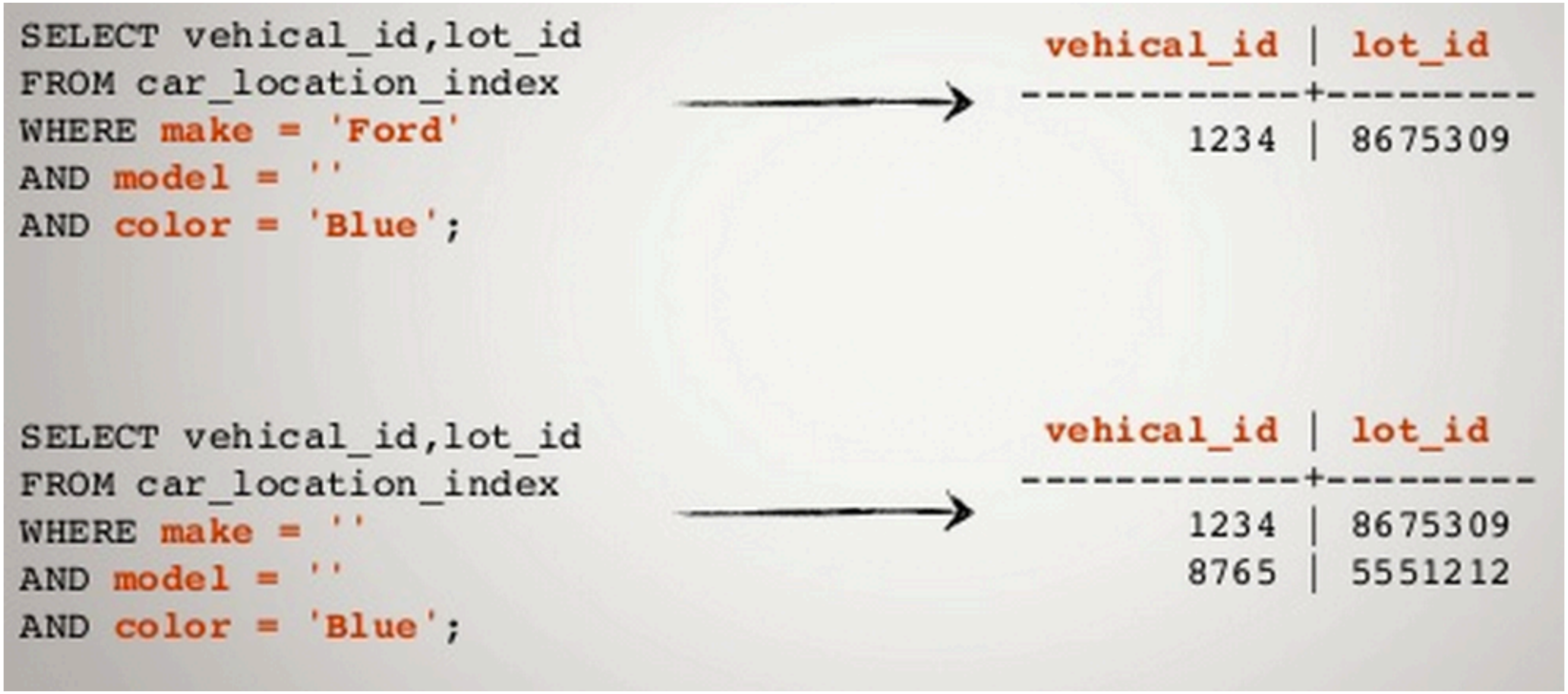
We are pre-optimizing for 7 possible queries of the index on insert.

```
1.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘Ford’, ‘Mustang’, ‘Blue’, 1234, 8675309);
2.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘Ford’, ‘Mustang’, ‘’, 1234, 8675309);
3.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘Ford’, ‘’, ‘Blue’, 1234, 8675309);
4.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘Ford’, ‘’, ‘’, 1234, 8675309);
5.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘’, ‘Mustang’, ‘Blue’, 1234, 8675309);
6.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘’, ‘Mustang’, ‘’, 1234, 8675309);
7.    INSERT INTO car_location_index (make, model, colour,
      vehicle_id, lot_id)
      VALUES (‘’, ‘’, ‘Blue’, 1234, 8675309);
```

(Batched)

```
BEGIN BATCH
INSERT INTO CARS (…) VALUES (…);
INSERT INTO car_location_index (…)
 VALUES (…);
INSERT INTO car_location_index (…)
 VALUES (…);
…
APPLY BATCH;
```

Different Queries are now possible!



```
SELECT vehical_id,lot_id
FROM car_location_index
WHERE make = 'Ford'
AND model = ''
AND color = 'Blue';
```

→

```
 vehical_id | lot_id
------------+---------
       1234 | 8675309
```

```
SELECT vehical_id,lot_id
FROM car_location_index
WHERE make = ''
AND model = ''
AND color = 'Blue';
```

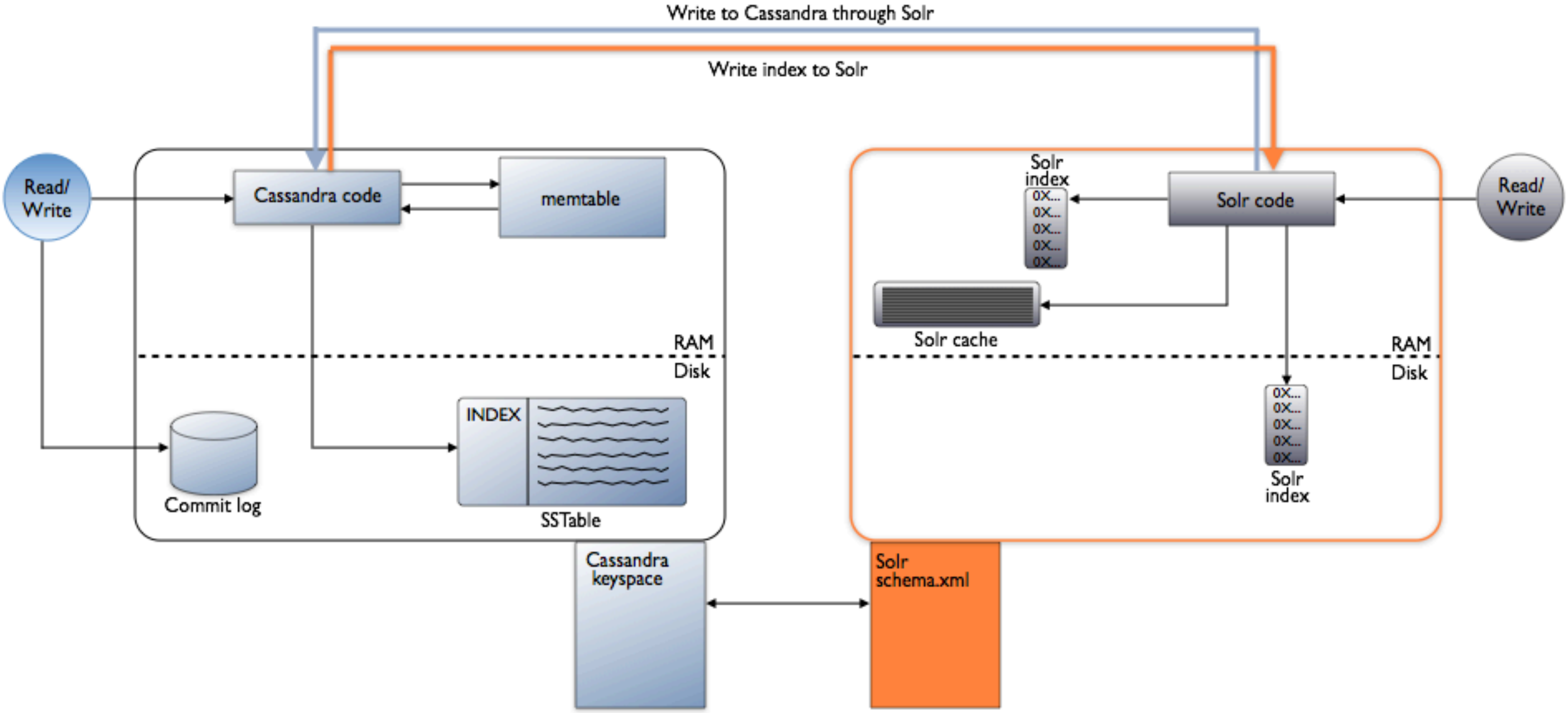
→

```
 vehical_id | lot_id
------------+---------
       1234 | 8675309
       8765 | 5551212
```

Don’t fear the writes

- 3 column index = 7 index rows per entry
- 4 columns = 15
- 5 columns = 31
- 6 columns = 63

DSE solr indexes



What is data modeling?

- **Data modeling is a process that involves**
  - Collection and analysis of data requirements in an information system
  - Identification of participating entities and relationships among them
  - Identification of data access patterns
  - A particular way of organizing and structuring data
  - Design and specification of a database schema
  - Schema optimization and data indexing techniques

- **Data modeling = Science + Art**

Key steps of data modeling for Cassandra

- Understand data and application queries
  - Data may or may not exist in some format (RDBMS, XML, CSV, …)
- Design tables
  - Design is based on access patterns or queries over data
- Implement the design using CQL
  - Optimizations concerning data types, keys, partition sizes, ordering

Cassandra modeling vs relational

Cassandra

Precompute queries at write time

- Optimizing for writes means we get optimized reads for free

All data required to answer a query must be nested in a table

- Referential integrity is a non-issue

Data modeling methodology is driven by queries and data

- Data duplication is considered normal (side effect of data nesting)

Relational

Recompute queries when read

- Expensive JOIN and ORDER BY

Data from many relations is combined to answer a query

- Referential integrity is important

Data modeling is driven by data only

- Data duplication is considered a problem (normalization theory)

Exercise 2

Twissandra

Exercise 2

- Users follow other users
- Users read the tweets of the users they follow
- [OPTIONAL] add tags to tweets table

```
CREATE TABLE friends (
  follower text references users (username),
  followed text references users (username)
);

SELECT * FROM tweets
WHERE author IN
    (SELECT followed FROM friends
     WHERE follower = ?);
```

Time Series/Sensor Data

What is time series data?

- Sensors
  - CPU, Network Card, Electronic Power Meter, Resource Utilization, Weather
- Clickstream data
- Historical trends
- Stock Ticker
- Anything that varies on a temporal basis
- Top Ten Most Popular Videos

Table Definition

- Data partitioned by source ID and time
  - **Timestamp** goes in the clustered column
  - Store the **measurement** as the non-clustered column(s)

```
CREATE TABLE temperature (
      weatherstation_id text,
      event_time timestamp,
      temperature text
      PRIMARY KEY (weatherstation_id, event_time)
);
```

Insert and Query Data

Simple to insert:

```
INSERT INTO temperature (weatherstation_id, event_time, temperature)
VALUES (‘1234abcd’, ‘2013-12-11 07:01:00’, ‘72F’);
```

Simple to query

```
SELECT temperature from temperature WHERE weatherstation_id=‘1234abcd’
AND event_time > ‘2013-04-03 07:01:00’ AND event_time < ‘2013-04-03
07:04:00’
```

Time Series Partitioning

- With the previous table, you can end up with a very large row on 1 partition i.e. `PRIMARY KEY (weatherstation_id, event_time)`
- This would have to fit on 1 node.
- Cassandra can store 2 billion columns per storage row.

- The solution is to have **a composite partition key** to split things up:

```
CREATE TABLE temperature (
      weatherstation_id text,
      date text,
      event_time timestamp,
      temperature text
      PRIMARY KEY ((weatherstation_id, date),event_time)
);
```

Reverse Ordering

```
CREATE TABLE temperature (
      weatherstation_id text,
      date text,
      event_time timestamp,
      temperature text
      PRIMARY KEY ((weatherstation_id, date),
event_time)
) WITH CLUSTERING ORDER BY (event_time DESC);
```

As part of the table definition, `WITH CLUSTERING ORDER BY (event_time DESC)`, is used to order the data by the most recent first i.e. the data will be returned in this order.

Rolling Storage

- Common pattern for time series data is rolling storage.
- For example, we only want to show the last 10 temperature readings and older data is no longer needed
- On most DBs you would need some background job to purge the old data.
- **With Cassandra you can set a time-to-live and forget it**
- *Combine that with the ordering of your data…….*

Time Series TTL’ing

```
INSERT INTO temperature (weatherstation_id, date, event_time,
temperature) VALUES (‘1234abcd’, ‘2013-12-11’, ‘2013-12-11
07:01:00’, ‘72F’) USING TTL 20;
```

- This data point will automatically be deleted after 20 seconds.
- Eventually you will see all the data disappear.

Exercise 3

Time series in Twissandra

Exercise 3

- Suppose I follow 100,000 people on Twitter who make 10 tweets per day
- How would you change the timeline table to avoid the large partition problem?
- What changes in my queries would this require?

Example code

http://www.datastax.com/dev/blog/python-driver-overview-using-twissandra

https://github.com/OpenNMS/newts

For more on data modeling…

Data modeling video series by Patrick McFadin

Part 1: The Data Model is Dead, Long Live the Data Model
http://www.youtube.com/watch?v=px6U2n74q3g

Part 2: Become a Super Modeler
http://www.youtube.com/watch?v=qphhxujn5Es

Part 3: The World's Next Top Data Model
http://www.youtube.com/watch?v=HdJlsOZVGwM