SEATTLE CASSANDRA USERS JUNE 2015

INTRODUCTION TO DATA MODELLING

Aaron Morton
@aaronmorton
Co-Founder & Team Member

THE LAST PICKLE

Example 1

Example 2

The other stuff.

“It’s a like SQL, you just need to think about things more. But it stores massive amounts of data. Which is nice.”

Example 1

Stock Market Data

exchange Table

```
CREATE TABLE exchange (
  exchange_id   text,
  name          text,
  city          text,
  PRIMARY KEY (exchange_id)
);
```

Tables.

Primary Key

Strong Typing

Pre defined Column names

All non Primary Key Columns optional

Tables, But.

Sparse Layout

No Foreign Keys

No Joins

No Constraints

No ALTERTABLE locks

No Type Casting on ALTERTABLE

exchange

```
CREATE TABLE exchange (
  exchange_id   text,
  name          text,
  city          text,
  PRIMARY KEY (exchange_id)
);
```

Data Types

ascii, text, varchar

int, bigint, varint

blob

boolean

counter

decimal, double, float

inet

list, map, set

timestamp

timeuuid, uuid

tuple

exchange Table

```
CREATE TABLE exchange (
  exchange_id   text,
  name          text,
  city          text,
  PRIMARY KEY (exchange_id)
);
```

Primary Key

```
…
PRIMARY KEY (PARTITION_KEY,
CLUSTERING_KEY, CLUSTERING_KEY,…)
…
```

Partition

A Partition is a storage engine row.

Rows with the same Partition Key are in the same Partition.

cqlsh

```
$ bin/cqlsh

Connected to Test Cluster at 127.0.0.1:9042.
[cqlsh 5.0.1 | Cassandra 2.1.2 | CQL spec 3.2.0 |
Native protocol v3]
Use HELP for help.
cqlsh>
```

exchange Table In Action

```
INSERT INTO
  exchange
  (exchange_id, name, city)
VALUES
  ('nyse', 'New York Stock Exchange', 'New York');
```

exchange Table In Action

```
SELECT
  *
FROM
  exchange
WHERE
  exchange_id = ‘nyse';
```

```
 exchange_id | city     | name
-------------+----------+-------------------------
        nyse | New York | New York Stock Exchange
```

exchange Table In Action

```
DELETE FROM
  exchange
WHERE
  exchange_id = 'nyse';
```

Table Scan Anti Pattern

```
SELECT
  *
FROM
  exchange;

 exchange_id | city     | name
-------------+----------+-------------------------
        nyse | New York | New York Stock Exchange
```

stock Table

```
CREATE TABLE stock (
  exchange_id   text
  ticker        text,
  name          text,
  sector        text,
  PRIMARY KEY ( (exchange_id, ticker) )
);
```

Compound Partition Key

stock Table In Action

```
INSERT INTO
  stock
  (exchange_id, ticker, name, sector)
VALUES
  ('nyse', 'tlp', 'The Last Pickle', 'awesomeness');
```

stock Table In Action

```
SELECT
  *
FROM
  stock
WHERE
  exchange_id = ‘nyse’ AND ticker = 'tlp';

 exchange_id | ticker | name            | sector
-------------+--------+-----------------+---------------
        nyse |    tlp | The Last Pickle | awesomeness
```

Primary Key Restrictions

```
SELECT
  *
FROM
  stock
WHERE
  exchange_id = 'nyse';
```

```
code=2200 [Invalid query] message="Partition key part
ticker must be restricted since preceding part is"
```

stock_ticker Table

```
CREATE TABLE stock_ticker (
  exchange_id   text,
  ticker        text,
  date          int,  // YYYYMMDD
  eod_price     int,
 PRIMARY KEY ( (exchange_id, ticker), date)
);
```

Standard comments

Clustering Key

Multiple Rows Per Partition.

Rows with the same Partition Key are in the same Partition.

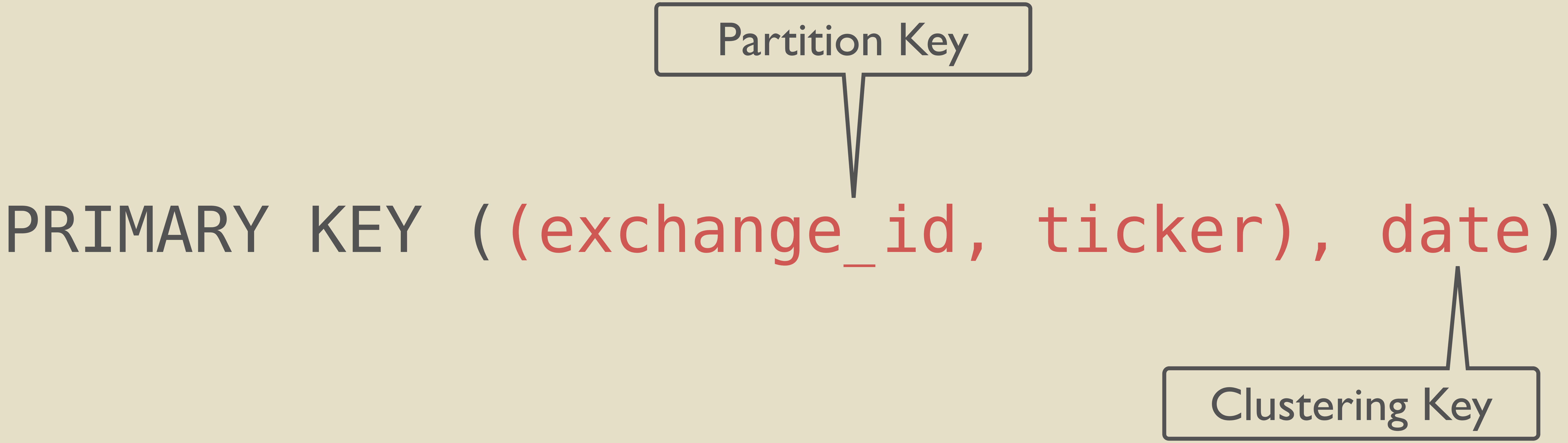
Multiple Rows Per Partition.

Rows in the same Partition are identified by their Clustering Key(s).

Multiple Rows Per Partition.

Together the Partition Key and Clustering Key(s) form the Primary Key that identifies a Row.

Primary Key



```
PRIMARY KEY ((exchange_id, ticker), date)
```

Partition Key

Clustering Key

stock_ticker Table In Action

```
INSERT INTO
  stock_ticker
  (exchange_id, ticker, date, eod_price)
VALUES
  ('nyse', 'tlp', 20150110, 100);
INSERT INTO
  stock_ticker
  (exchange_id, ticker, date, eod_price)
VALUES
  ('nyse', 'tlp', 20150111, 110);
INSERT INTO
  stock_ticker
  (exchange_id, ticker, date, eod_price)
VALUES
  ('nyse', 'tlp', 20150112, 80);
```

stock_ticker Table In Action

```
SELECT
  *
FROM
  stock_ticker
WHERE
  exchange_id = 'nyse' AND ticker = 'tlp' and date =
20150110;

 exchange_id | ticker | date       | eod_price
-------------+--------+------------+-----------
        nyse |    tlp | 20150110 |       100
```

stock_ticker Table In Action

```
SELECT
  *
FROM
  stock_ticker
WHERE
  exchange_id = 'nyse' AND ticker = ‘tlp';

 exchange_id | ticker | date     | eod_price
-------------+--------+----------+-----------
        nyse |    tlp | 20150110 |       100
        nyse |    tlp | 20150111 |       110
        nyse |    tlp | 20150112 |        80
```

stock_ticker Table In Action

```
SELECT
  *
FROM
  stock_ticker
WHERE
  exchange_id = 'nyse' AND ticker = 'tlp'
ORDER BY
  date desc;

 exchange_id | ticker | date     | eod_price
-------------+--------+----------+-----------
        nyse |    tlp | 20150112 |        80
        nyse |    tlp | 20150111 |       110
        nyse |    tlp | 20150110 |       100
```

Reversing The Stock Ticker Table

```
CREATE TABLE stock_ticker (
  exchange_id   text,
  ticker        text,
  date          int,  // YYYYMMDD
  number_traded int,
  PRIMARY KEY ( (exchange_id, ticker), date)
)
WITH CLUSTERING ORDER BY (date DESC);
```

stock_ticker Table In Action

```
SELECT
  *
FROM
  stock_ticker
WHERE
  exchange_id = 'nyse' AND ticker = ‘tlp' AND date > 20150110;

 exchange_id | ticker | date     | eod_price
-------------+--------+----------+-----------
        nyse |    tlp | 20150112 |        80
        nyse |    tlp | 20150111 |       110
```

So Far:

Tables with Columns

Data Types

Partitions and Clustering

Clustering Order

Table Properties

Example 1

Example 2

The other stuff.

Data Modelling Guidelines.

Denormalise by creating materialised views that support the read paths of the application.

Data Modelling Guidelines.

Constrain the Partition Size by time or space.

Data Modelling Guidelines.

Solve problems with the read path of your application in the write path.

Example 2

Vehicle Tracking

A “Black Box” on Vehicles sends position, speed, etc every 30 seconds via mobile networks.

Requirements

1. Lookup vehicle details by `vehicle_id`.

2. Get data points for a time slice by `vehicle_id`.

3. Get distinct days a vehicle has been active by `vehicle_id`.

Data Model Planning - Requirement 1

Vehicle sounds like a simple entity identified by `vehicle_id`.

Data Model Planning - Requirement 2

Sounds like a (potentially infinite) *Time Series* of data per `vehicle_id`.

Data Model Planning - Requirement 3

Is a summary of Time Series data per `vehicle_id`.

Keyspace == Database

```
create keyspace
  trak_u_like
WITH REPLICATION =
{
  'class':'NetworkTopologyStrategy',
  'datacenter1' : 3
};

use trak_u_like;
```

vehicle Table

```
CREATE TABLE vehicle (
  vehicle_id        text,
  make              text,
  model             text,
  accidents         list<text>,
  drivers           set<text>,
  modifications     map<text, text>,
  PRIMARY KEY (vehicle_id)
);
```

Collection Types

CQL 3 Spec…

"Collections are meant for storing/denormalizing relatively small amount of data."

vehicle Table In Action

```
INSERT INTO
  vehicle
  (vehicle_id, make, model, drivers, modifications)
VALUES
  ('wig123', 'Big Red', 'Car',
    {'jeff', 'anthony'},
    {'wheels' : 'mag wheels'});
```

vehicle Table In Action

```
SELECT
  *
FROM
  vehicle
WHERE
  vehicle_id = ‘wig123';
```

```
 vehicle_id | accidents | drivers             | make    | model | modifications
------------+-----------+---------------------+---------+-------+--------------------------
     wig123 |      null | {'anthony', 'jeff'} | Big Red |   Car | {'wheels': 'mag wheels'}
```

vehicle Table In Action

```
UPDATE
  vehicle
SET
  accidents = accidents + ['jeff crashed into dorothy 2015/01/21']
where
  vehicle_id = 'wig123';
```

vehicle Table In Action

```
SELECT
  vehicle_id, accidents, drivers
FROM
  vehicle
WHERE
  vehicle_id = ‘wig123';
```

```
 vehicle_id | accidents                                | drivers
------------+------------------------------------------+---------------------
     wig123 | ['jeff crashed into dorothy 2015/01/21'] | {'anthony', 'jeff'}
```

vehicle Table In Action

```
UPDATE
  vehicle
SET
  drivers = drivers - {'jeff'}
where
  vehicle_id = 'wig123';
```

vehicle Table In Action

```
SELECT
  vehicle_id, accidents, drivers
FROM
  vehicle
WHERE
  vehicle_id = 'wig123';
```

```
 vehicle_id | accidents                                | drivers
------------+------------------------------------------+-------------
     wig123 | ['jeff crashed into dorothy 2015/01/21'] | {'anthony'}
```

data_point Table

```
CREATE TABLE data_point (
  vehicle_id        text,
  day               int,
  sequence          timestamp,
  latitude          double,
  longitude         double,
  heading           double,
  speed             double,
  distance          double,
  PRIMARY KEY ( (vehicle_id, day), sequence)
)
WITH CLUSTERING ORDER BY (sequence DESC);
```

Bucketing the data_point Table

```
PRIMARY KEY ( (vehicle_id, day), sequence)
WITH CLUSTERING ORDER BY (sequence DESC);
```

All data points for one day are stored in the same partition.

Each Partition will have up to 2,880 rows.

data_point Table In Action

```
INSERT INTO
    data_point
(vehicle_id, day, sequence, latitude, longitude, heading, speed, distance)
VALUES
('wig123', 20150120, '2015-01-20 09:01:00', -41, 174, 270, 10, 500);

INSERT INTO
    data_point
(vehicle_id, day, sequence, latitude, longitude, heading, speed, distance)
VALUES
('wig123', 20150120, '2015-01-20 09:01:30', -42, 174, 270, 10, 500);

…
```

data_point Table In Action

```
SELECT
  vehicle_id, day, sequence, latitude, longitude
FROM
  data_point
WHERE
  vehicle_id = 'wig123' AND day = 20150120;

 vehicle_id | day      | sequence                 | latitude | longitude
------------+----------+--------------------------+----------+-----------
     wig123 | 20150120 | 2015-01-20 09:02:30+1300 |      -44 |       174
     wig123 | 20150120 | 2015-01-20 09:02:00+1300 |      -43 |       174
     wig123 | 20150120 | 2015-01-20 09:01:30+1300 |      -42 |       174
     wig123 | 20150120 | 2015-01-20 09:01:00+1300 |      -41 |       174
```

data_point Table In Action

```
SELECT
  vehicle_id, day, sequence, latitude, longitude
FROM
  data_point
WHERE
  vehicle_id = 'wig123' AND day in (20150120, 20150121);
```

```
 vehicle_id | day      | sequence                  | latitude | longitude
------------+----------+---------------------------+----------+-----------
     wig123 | 20150120 | 2015-01-20 09:02:30+1300  |      -44 |       174
     wig123 | 20150120 | 2015-01-20 09:02:00+1300  |      -43 |       174
     wig123 | 20150120 | 2015-01-20 09:01:30+1300  |      -42 |       174
     wig123 | 20150120 | 2015-01-20 09:01:00+1300  |      -41 |       174
     wig123 | 20150121 | 2015-01-21 08:02:30+1300  |      -44 |       176
     wig123 | 20150121 | 2015-01-21 08:02:00+1300  |      -44 |       175
```

active_day Table

```
CREATE TABLE active_day (
  vehicle_id        text,
  day               int,
  distance          counter,
  PRIMARY KEY (vehicle_id, day)
)
WITH
    CLUSTERING ORDER BY (day DESC)
AND
    COMPACTION =
{
    'class' : 'LeveledCompactionStrategy'
};
```

active_day Table In Action

```
UPDATE
  active_day
SET
  distance = distance + 500
WHERE
  vehicle_id = 'wig123' and day = 20150120;

UPDATE
  active_day
SET
  distance = distance + 500
WHERE
  vehicle_id = 'wig123' and day = 20150120;
```

active_day Table In Action

```
SELECT
  *
FROM
  active_day
WHERE
  vehicle_id = 'wig123';

 vehicle_id | day       | distance
------------+----------+-----------
     wig123 | 20150121 |     1000
     wig123 | 20150120 |     2000
```

active_day Table In Action

```
SELECT
  *
FROM
  active_day
WHERE
  vehicle_id = 'wig123'
LIMIT 1;
```

```
 vehicle_id | day      | distance
------------+----------+----------
     wig123 | 20150121 |     1000
```

“It’s a like SQL, you just need to think about things more. But it stores massive amounts of data. Which is nice.”

Example 1

Example 2

And now the other stuff.

Light Weight Transactions

Static Columns

Indexes

Light Weight Transactions

Uses Paxos

Added in 2.0.

Light Weight Transactions

Provide linearizable consistency, similar to SERIAL Transaction Isolation.

Light Weight Transactions

Use Sparingly.

Impacts Performance and Availability.

Light Weight Transactions

```
CREATE TABLE user (
  user_name     text,
  password      text,
  PRIMARY KEY (user_name)
);
```

Insert If Not Exists

```
INSERT INTO
    user
    (user_name, password)
VALUES
    ('aaron', 'pwd')
IF
    NOT EXISTS;
```

Failing Insert

```
INSERT INTO
    user
    (user_name, password)
VALUES
    ('aaron', 'pwd')
IF
    NOT EXISTS;

 [applied] | user_name | password
-----------+-----------+-----------
     False |     aaron |    newpwd
```

Update If No Change

```
UPDATE
    user
SET
    password = 'newpwd'
WHERE
    user_name = 'aaron'
IF
    password = 'pwd';
```

Failing Update

```
UPDATE
    user
SET
    password = 'newpwd'
WHERE
    user_name = 'aaron'
IF
    password = ‘pwd’;
```

```
 [applied] | password
-----------+-----------
     False |   newpwd
```

Light Weight Transactions

Static Columns

Indexes

Static Columns

Column value stored at the partition level.

All rows in the partition have the same value.

Static Columns

```
CREATE TABLE web_order (
  order_id      text,
  order_total   int static,
  order_item    text,
  item_cost     int,
  PRIMARY KEY (order_id, order_item)
);
```

Static Columns - Simple Example

```
INSERT INTO
    web_order
    (order_id, order_total, order_item, item_cost)
VALUES
    ('ord1', 5, 'foo', 5);
INSERT INTO
    web_order
    (order_id, order_total, order_item, item_cost)
VALUES
    ('ord1', 10, 'bar', 5);
INSERT INTO
    web_order
    (order_id, order_total, order_item, item_cost)
VALUES
    ('ord1', 20, 'baz', 10);
```

Static Columns - Simple Example

```
select * from web_order;

 order_id | order_item | order_total | item_cost
----------+------------+-------------+-----------
     ord1 |        bar |          20 |         5
     ord1 |        baz |          20 |        10
     ord1 |        foo |          20 |         5
```

Static Columns With LWT

Static Columns may be used in a conditional UPDATE.

All updates to the Partition in the BATCH will be included.

Static Columns With LWT

```
BEGIN BATCH

UPDATE web_order
SET order_total = 50
WHERE order_id='ord1'
IF order_total = 20;

INSERT INTO web_order
    (order_id, order_item, item_cost)
VALUES
    ('ord1', 'monkey', 30);

APPLY BATCH;
```

Light Weight Transactions

Static Columns

Indexes

Secondary Indexes

Use non Primary Key fields in the WHERE clause.

Secondary Indexes

Use Sparingly.
Impacts Performance and Availability.

Secondary Indexes

```
CREATE TABLE user (
  user_name     text,
  state         text,
  password      text,
  PRIMARY KEY (user_name)
);

CREATE INDEX on user(state);
```

Secondary Indexes

```
INSERT INTO user
    (user_name, state, password)
VALUES
    ('aaron', 'ca', ‘pwd');

INSERT INTO user
    (user_name, state, password)
VALUES
    ('nate', 'tx', ‘pwd');

INSERT INTO user
    (user_name, state, password)
VALUES
    ('kareem', 'wa', 'pwd');
```

Secondary Indexes

```
SELECT * FROM user WHERE state = ‘ca';
```

```
 user_name | password | state
-----------+----------+-------
     aaron |      pwd |    ca
```

Thanks.

Aaron Morton
@aaronmorton

Co-Founder & Team Member
www.thelastpickle.com

THE LAST PICKLE