Cassandra for Developers

Module 2

Data Model

Whole picture

- Column-oriented store
- Multidimensional map
- Map<RowKey, SortedMap<Name, Value, TimeStamp, TTL>>
- Efficient key lookup
- RowKey, Name, Value – byte arrays

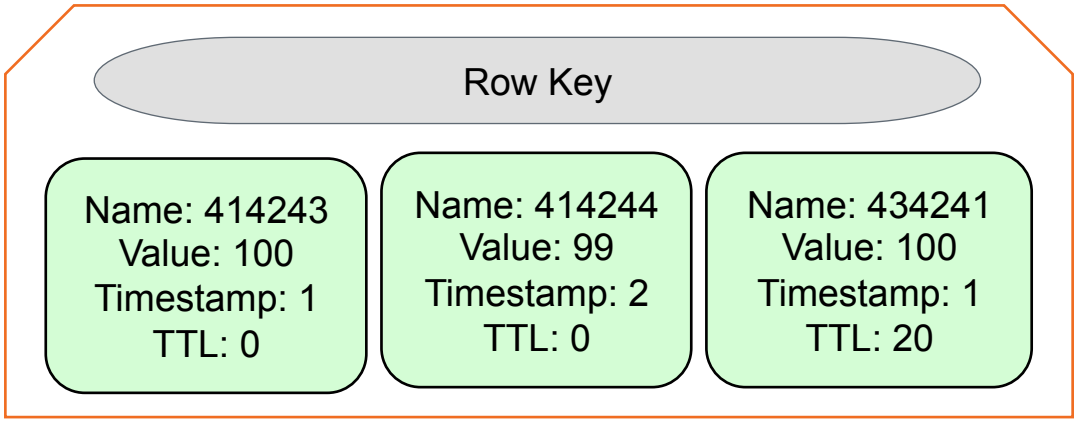
Column

- Column is like struct
  - Name
    - Mandatory, any type and value, unique
    - Stored with every column!
    - Data type is called a comparator
    - Max 64Kb
  - Value
    - Any type and value (effectively byte array)
    - Can be empty
    - Data type is called a validator
    - Max 2 Gb
  - Timestamp
  - Time to live (TTL) – optional (def: 0)

Name: 414243
Value: 100
Timestamp: 10000004
TTL: 0

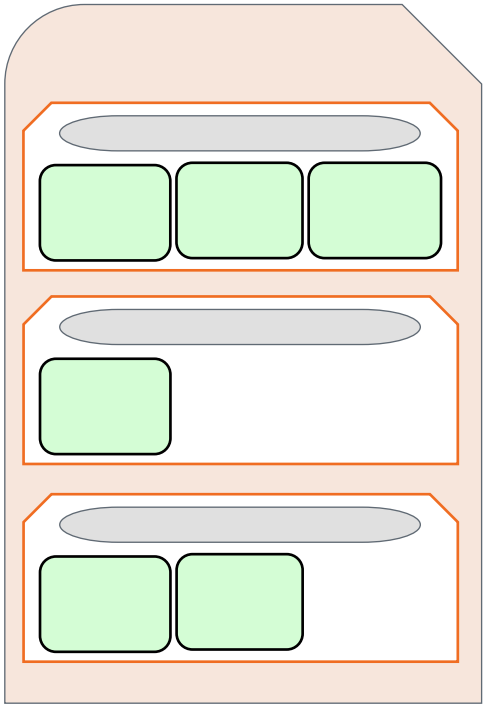
Rows

- Row is like named hash table
  - Name = row key
    - Unique in a column family
  - Collection of columns (indexed by name)
    - Orderable
- Like RDBMS tuple
- Each row replica is stored on one node (no splitting)
  - Must fit on disk

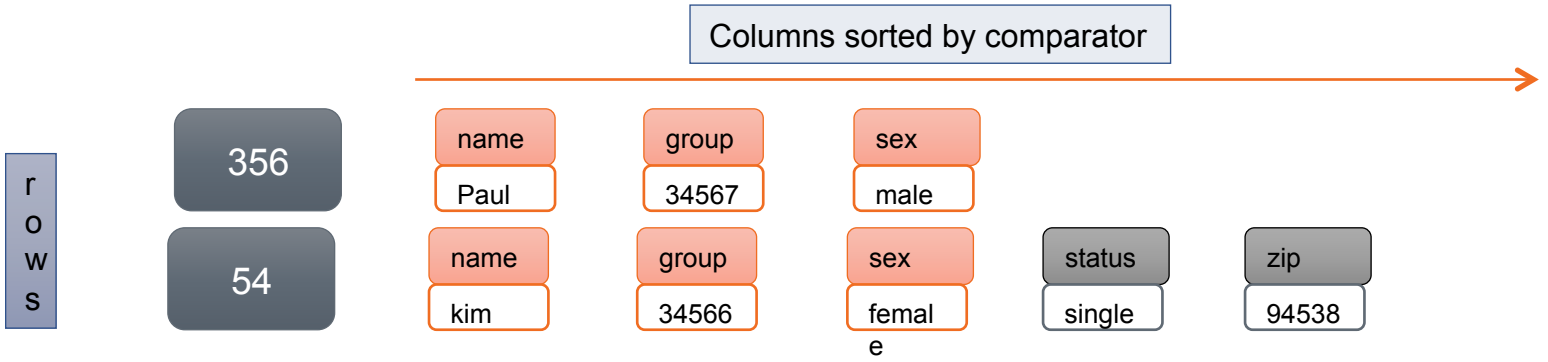


Column Families

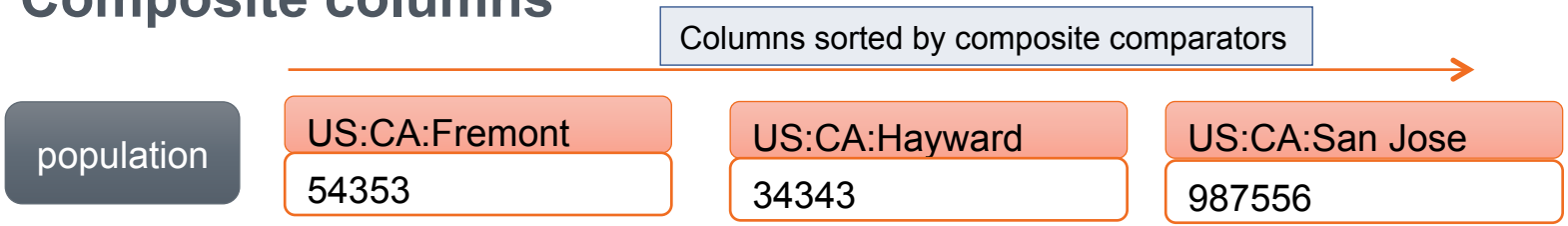
- Column family is like
  named hash table of hash tables
  - Name
  - Collection of rows (indexed by row keys)
    - Unique row keys
    - Columns of different rows may differ
      - i.e. CF is sparse nested hash table
    - Default ordering: by clustering columns – second part of row key
      - Compare row keys as bytes, long, UTF8, ASCII, UUID, custom
- Loose analog to RDBMS tables



Data Model



Composite columns



Static Column Family

- Most similar to RDBMS tables
- Most rows have the same column names
- But columns in rows can be different

Dynamic Column Family

- “Wide” rows
- Highly denormalized
- All columns needed to answer a query to a row

Super column

- Super column is like named map of columns
  - Name
  - Collection of columns (indexed by name)
- Deprecated since C* 2.0
- No access from CQL (only Thrift)
- Use composite columns instead

Super column family

- Like a hash table of hash tables of hash tables
- Super column family is a column family where rows consist of super columns instead of columns

Keyspace

- Keyspace is a collection of column families (and super column families)
- Has specific replication configuration
  - Replication strategy
  - Replication factor
- Like schema in Oracle Database
- Like database in MySQL, Microsoft SQL Server

Cluster

- Collection of keyspaces
- At least
  - system keyspace
  - user keyspaces

Partitioners

- Token = Partitioner(Key)
- Murmur3Partitioner (RandomPartitioner)
  - Ordering is random
- ByteOrderedPartitioner
  - Ordering is lexical for each node and the ring

Column families properties that drives data modeling

- Are not tables
- Can be very wide
- Can be very narrow
- There’s no joins
  - While foreign keys are there
    - Value (or name, or name part) of a column might be an another row key
- There’s only one index by default (row key)
  - Actually, used for distributing rows (=partitioning =sharding)

Secondary indexes

- Any column (or group of columns) can be indexed
- Performant for low-cardinality columns
- Local for node (not for keyspace or cluster)
- The only possibility to use a column in a search expression

Some basic modeling patterns

- Entities in column family
- (Another) column family as index
- Materialized views
- Time series
- Event sourcing

Entities modeling

- Single primary key
- Fairly consistent columns names
- Fairly narrow rows
- Feels like schema-less relational data model

Column family as index

- Rows are indexed by row key
- Secondary indexes prefer low cardinality
- We have to build our own
- Query-first data modeling
  - Column family per query
  - Column values become entity row keys

Materialized views

- Based on entity data
- A separate column family
- Entity data organized by indexed key

Event sourcing

- Martin Fowler's Enterprise Application Architecture pattern: "Capture all changes to an application state as a sequence of events"
- Persist state change events
- Do not persist current state
- All writes are immutable
- At any time events sequence can be replayed and current state will be formed

Time series

- Row key is a time identifier
- Column names are events
- Column values are measurements
  - Complex measured values can be serialized (JSON/XML/Thrift/…)
- Rows can be very wide

Indexing with Tables

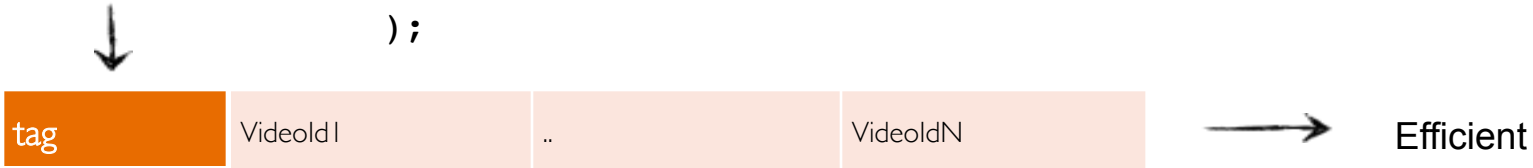
- Indexing expresses application intent
- Fast access to specific queries
- Secondary indexes != relational indexes
- Use information you have. No pre-reads.

Keyword index

- Use a word as a key
- Columns are the occurrence
- Ex: Index of tag words about videos

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

Fast



Partial word index

- Where row size will be large
- Take one part for key, rest for columns name

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

User: tcodd Email: tcodd@relational.com

```
INSERT INTO email_index (domain, user, username)
VALUES ('@relational.com','tcodd', 'tcodd');
```

Partial word index

- Create partitions + partial indexes

```
CREATE TABLE product_index (
   store int,
   part_number0_3 int,
   part_number4_9 int,
   count int,
   PRIMARY KEY ((store,part_number0_3), part_number4_9)
);
```

Compound row key!

- Store #8675309 has 3 of part# 7079748575

```
INSERT INTO product_index (store,part_number0_3,part_number4_9,count)
VALUES (8675309,7079,48575,3);

SELECT count
FROM product_index
WHERE store = 8675309
AND part_number0_3 = 7079
AND part_number4_9 = 48575;
```

Fast and efficient!

Bit map index – supports ad hoc queries

- Multiple parts to a key
- Create a truth table of the different combinations
- Inserts == the number of combinations
  - 3 fields? 7 options (Not going to use null choice)
  - 4 fields? 15 options
  - $2^n - 1$, where n = # of dynamic query fields

Bit map index

- Find a car in a lot 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

Bit map index - Table create

- Make a table with three different key combos

```
CREATE TABLE car_location_index (
   make varchar,
   model varchar,
   color varchar,
   vehical_id int,
   lot_id int,
   PRIMARY KEY ((make,model,color),vehical_id)
);
```

Compound row key with three different options

Bit map index - Adding records

- Pre-optimize for 7 possible questions on insert

```
INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('Ford','Mustang','Blue',1234,8675309);

INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('Ford','Mustang','',1234,8675309);

INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('Ford','','Blue',1234,8675309);

INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('Ford','','',1234,8675309);

INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('','Mustang','Blue',1234,8675309);

INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('','Mustang','',1234,8675309);

INSERT INTO car_location_index (make,model,color,vehical_id,lot_id)
VALUES ('','','Blue',1234,8675309);
```

Bit map index - Selecting records

- Different combinations 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
```

Requirements

Find orders by customer, supplier, product, and employee

- Optional date range

Need to display order information – header/details

List all employees for a manager

Orders that have not shipped

- By shipper
- By date

Thank you!

Questions?