Making the Most of Cassandra

Paul O'Fallon

@paulofallon

Summary

Materialized views

Secondary indexes

Batches

Transactions

User defined functions & aggregates

Materialized Views

```
CREATE TABLE users (
    id varchar,
    first_name varchar,
    last_name varchar,
    company varchar,
    // ...
    PRIMARY KEY (id)
);
```

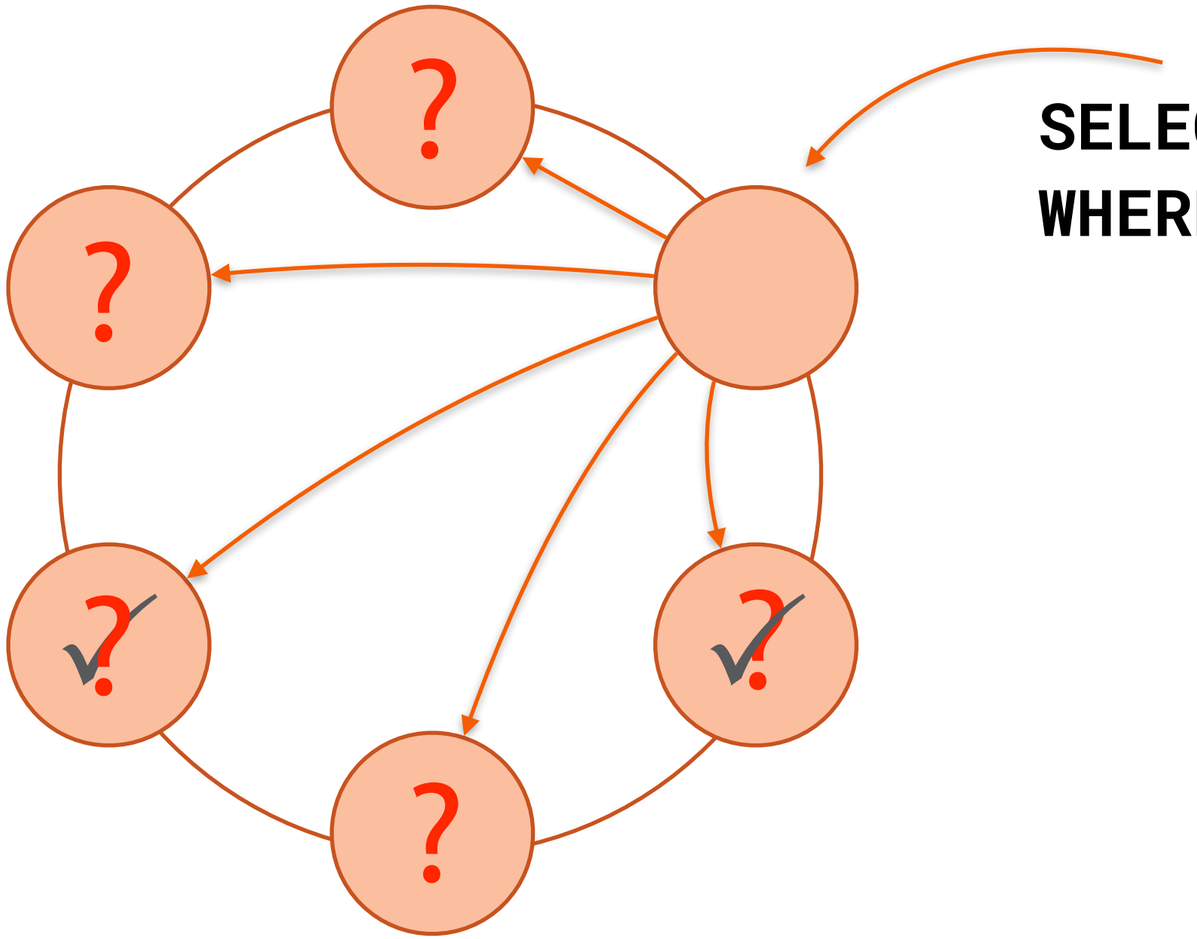
```
CREATE MATERIALIZED VIEW
users_by_company AS
SELECT * FROM users
WHERE company IS NOT null
PRIMARY KEY (company, id);
```

```
SELECT first_name, last_name
FROM users_by_company
WHERE company = ‘Acme Corp’
```

Secondary Indexes

```
CREATE INDEX ON users(company);

SELECT * FROM users WHERE company = ‘Acme Corp’;
```

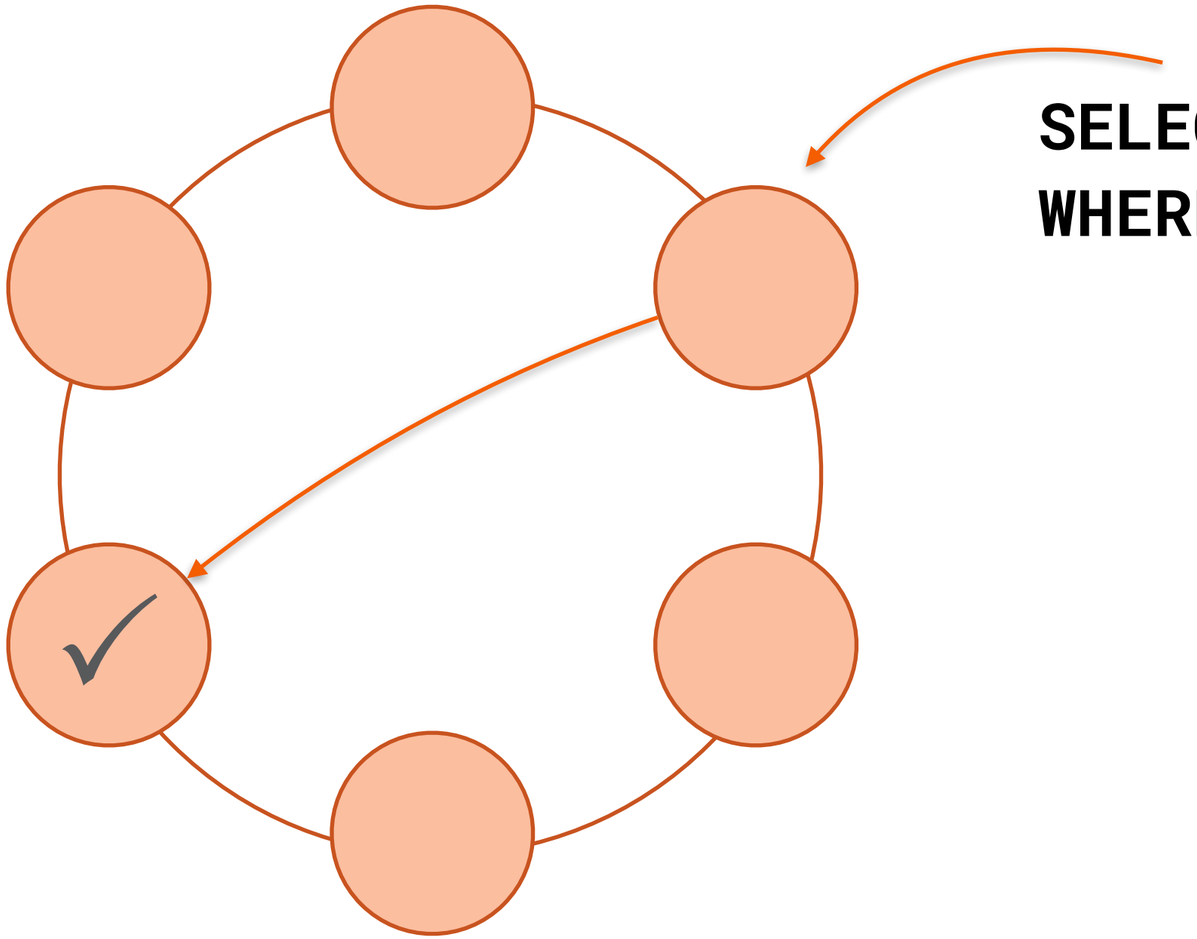
What's the Difference?



```
SELECT * FROM users
WHERE company = 'Acme Corp'
```

https://www.datastax.com/blog/2016/05/materialized-view-performance-cassandra-3x

What's the Difference?



```
SELECT * FROM users_by_company
WHERE company = 'Acme Corp'
```

https://www.datastax.com/blog/2016/05/materialized-view-performance-cassandra-3x

Secondary Indexes on Collections

Tags

developer (1334)

open-source (183)

javascript (174)

node.js (35)

```
CREATE TABLE users (
    id varchar,
    first_name varchar,
    last_name varchar,
    company varchar,
    tags set<varchar>,
    // ...
    PRIMARY KEY (id)
);

CREATE INDEX ON users(tags);
```

Secondary Indexes on Collections

```
INSERT INTO users (id, first_name, last_name, company, tags)
VALUES ('john-doe','John','Doe','acme-corp',{'java'});
```

```
SELECT * FROM users WHERE tags CONTAINS 'java';
```

Secondary Indexes on Collections

✓ Set
✓ List
✓ Map

```
CREATE INDEX ON <table>(<collection_column>)

WHERE <collection_column> CONTAINS <value>
```

✓ Map

```
CREATE INDEX ON <table>(KEYS(<map_column>))

WHERE <map_column> CONTAINS KEY <key>
```

Demo

Add a materialized view for users

Add a secondary index to users

Add a tags set to users, with an index

Extra Credit: Tracing Views and Indexes

- Launch a three node, single DC cluster
- Create “pluralsight” keyspace with SimpleStrategy and replication factor of 1
- Create a “users” table (including a company column)
- Create a “users_by_company” materialized view
- Add a secondary index on the users table for the “company” column
- Add some sample users (with companies)
- Turn tracing on
- Select from the view and the table with “company” in the where clause
- Compare the tracing output, including which nodes are involved

Manually Maintained Indexes

Tags

developer (1334)

open-source (183)

javascript (174)

node.js (35)

```
CREATE TABLE courses (
    id varchar,
    // ...
    tags set<varchar> static,
    module_id int,
    // ...
    PRIMARY KEY (id, module_id)
);
```

Manually Maintained Indexes

```
CREATE TABLE course_tags (
  tag varchar,
  course_id varchar,
  PRIMARY KEY (tag, course_id)
);
```

Manually Maintained Indexes

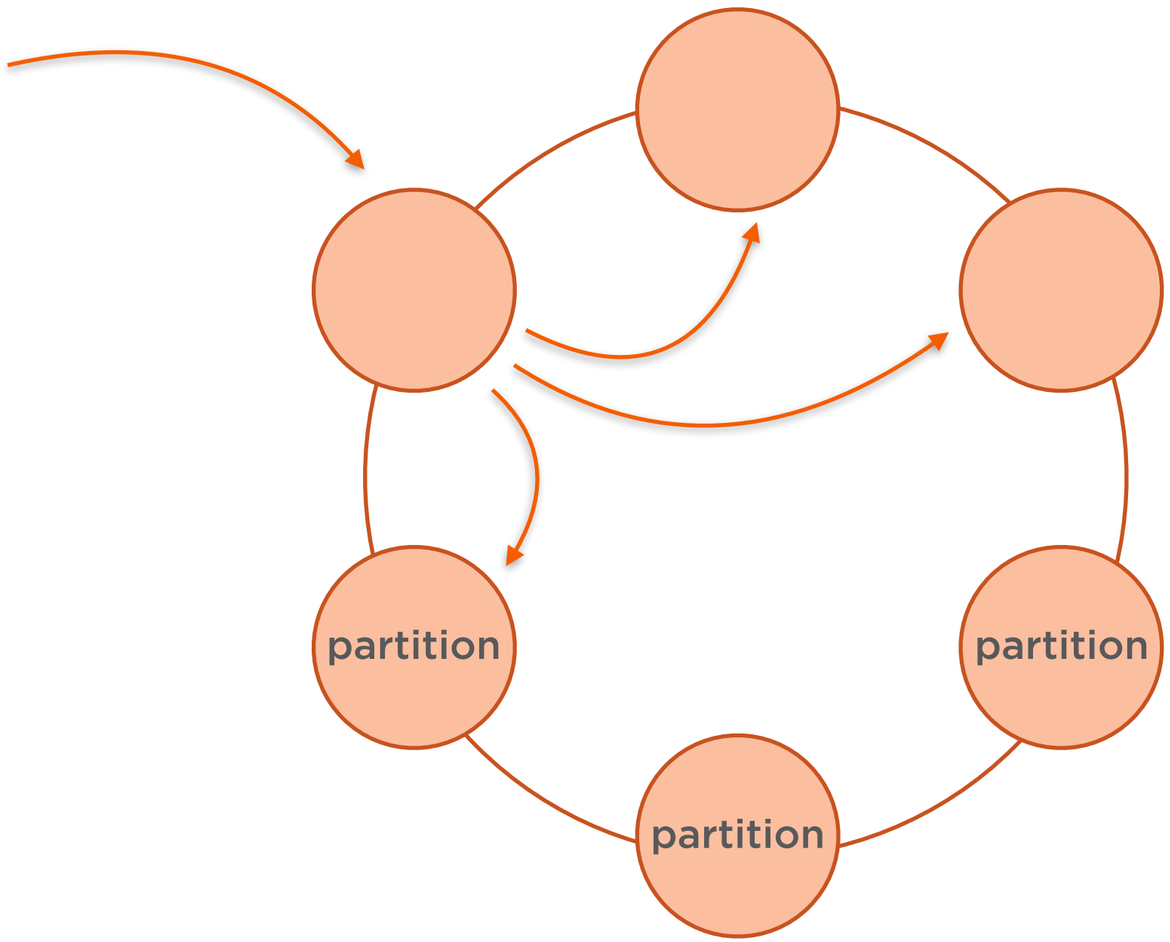
```
INSERT INTO courses (id, tags)
VALUES (‘nodejs-big-picture‘,
  {‘developer','open-source','javascript','node.js'});

INSERT INTO course_tags (tag, course_id)
VALUES (‘developer’,'nodejs-big-picture');

INSERT INTO course_tags (tag, course_id)
VALUES (‘open-source’,’nodejs-big-picture');

// ... etc.
```

Batches

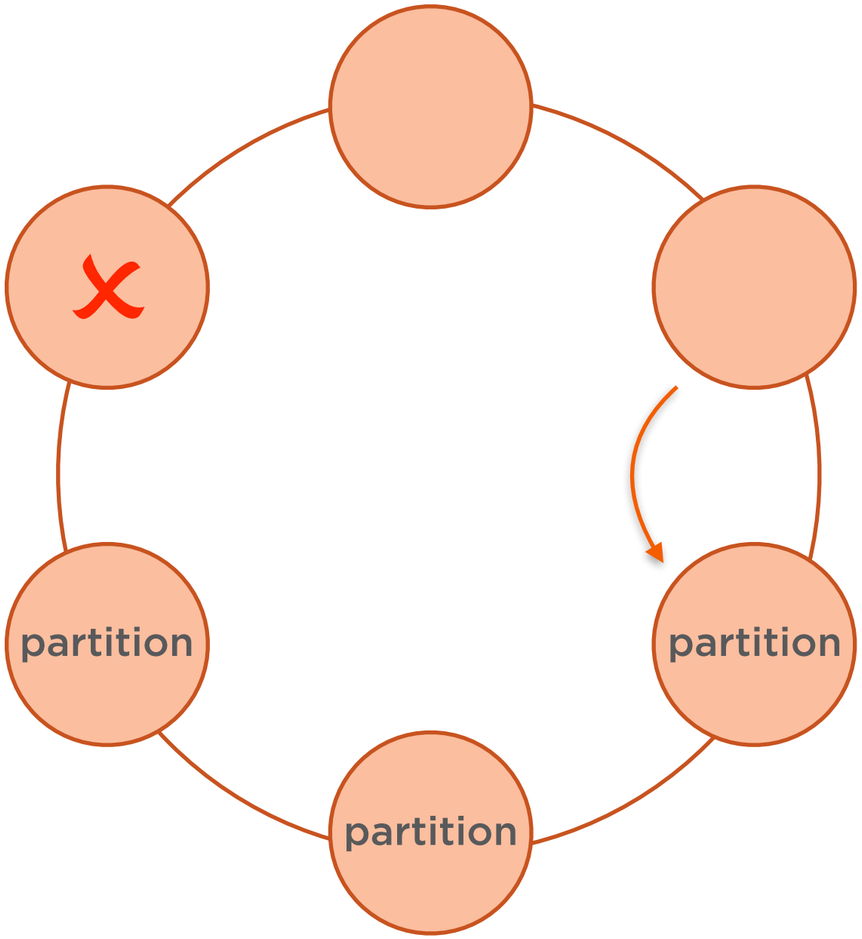
Intended for:
Keeping tables in sync

Not intended for:
Fast loading of data

Batches



Batches

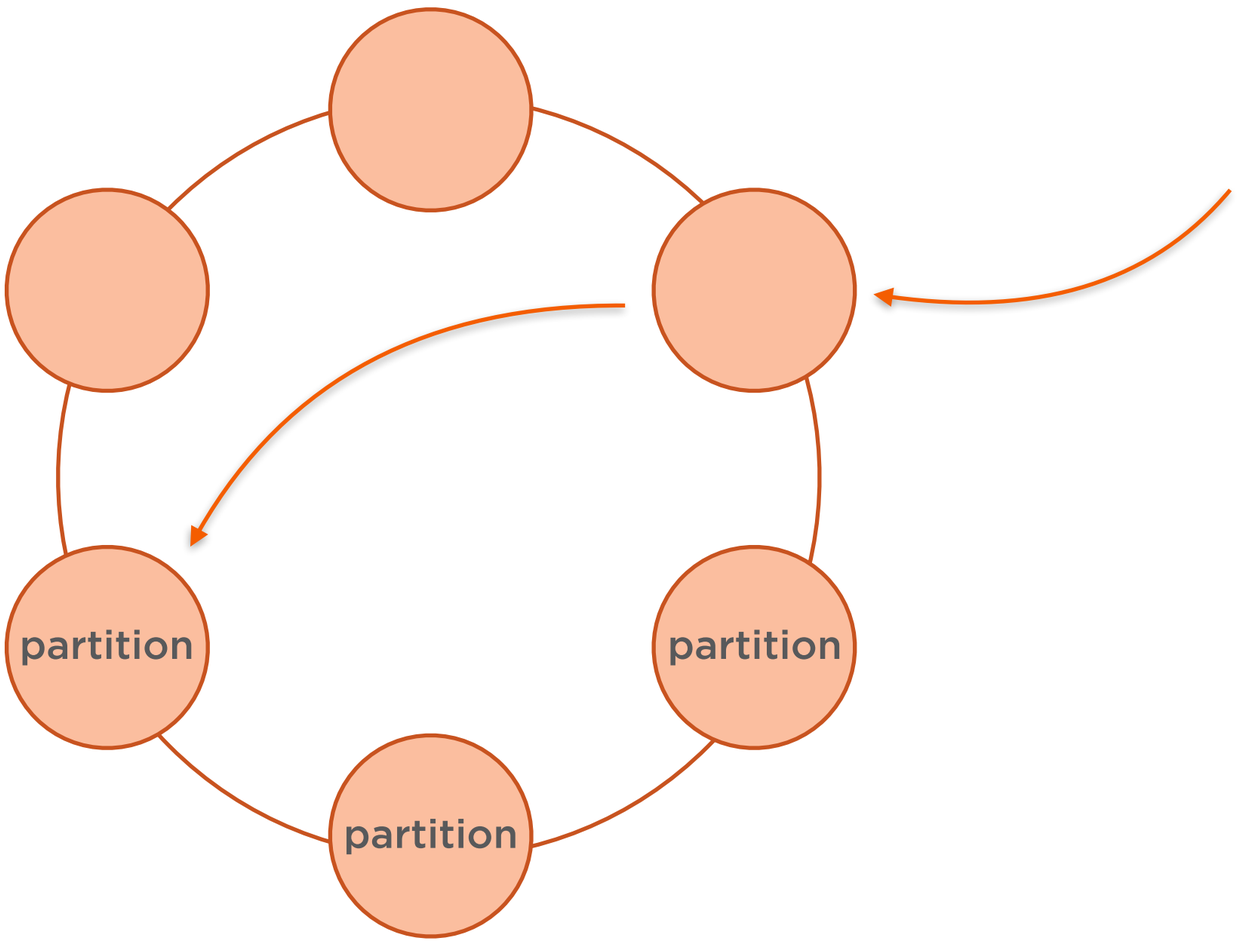


Batches

```
BEGIN BATCH

INSERT INTO courses (id, tags)
VALUES (‘nodejs-big-picture‘,
  {‘developer', 'javascript', 'node.js', 'open-source'});

INSERT INTO course_tags (tag, course_id)
VALUES (‘developer’,'nodejs-big-picture');

// ... etc.

APPLY BATCH;
```

Unlogged Batches

ONE WRITE!



Unlogged Batches

```
BEGIN UNLOGGED BATCH

INSERT INTO courses (id, name)
VALUES (‘nodejs-big-picture‘, 'Node.js: The Big Picture');

INSERT INTO courses (id, module_id, module_name)
VALUES (‘nodejs-big-picture‘, 1, 'Course Overview');

// ... etc.

APPLY BATCH;
```

Alternative to Batches

Client pseudocode

```
PREPARE <insert_statement>
LOOP_OVER_DATA
  EXECUTE(<prepared_statement>,<row_of_data>)
END_LOOP
```

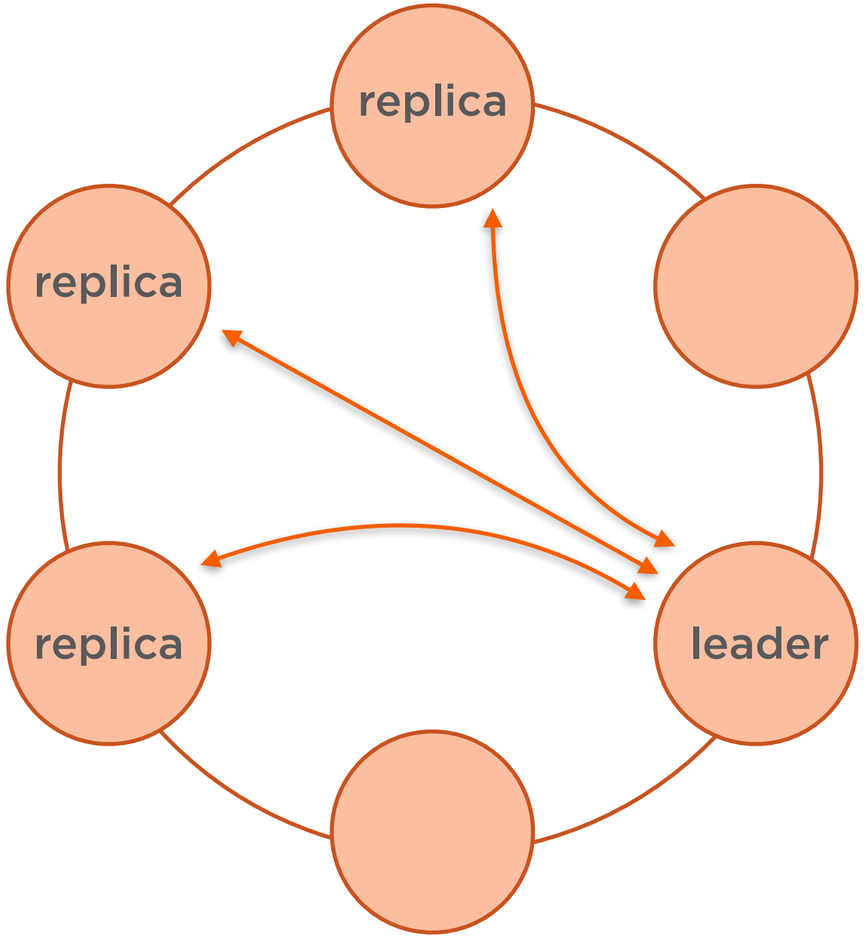
Let the client optimize for you!

Demo

Use batches to manually maintain tags

Examine our use of unlogged batches

Lightweight Transactions



1. **Prepare ↔ Promise**
2. **Read ↔ Results**
3. **Propose ↔ Accept**
4. **Commit ↔ Ack**

Compare-and-Set Operations

Insert

```
INSERT INTO users (id, first_name, last_name)
VALUES ('john-doe', 'John','Doe')
IF NOT EXISTS;
```

Update

```
UPDATE users SET password = 'mypass', reset_token = null
WHERE id = 'john-doe'
IF reset_token = '1GRhEs1';
```

Works with Batches Too!

```
BEGIN BATCH

  INSERT INTO <table> ... IF NOT EXISTS;
  INSERT INTO <table> ...
  INSERT INTO <table> ...

APPLY BATCH;
```

Demo

Conditional insert into the users table

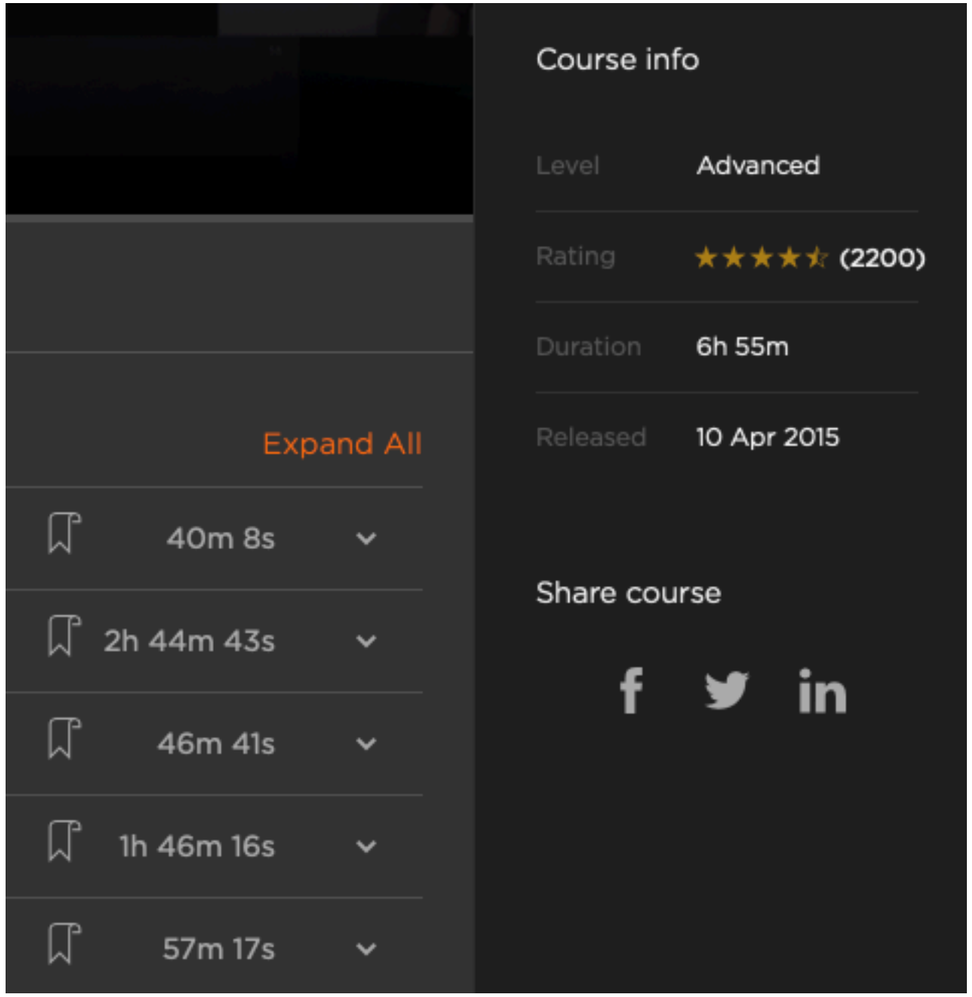
Conditional update with a reset token

User Defined Functions

- Can be used in SELECT, INSERT and UPDATE statements
- Java and JavaScript languages are supported by default
- Support for other JSR 223 languages can be added
- UDFs are part of the schema
- Your code is executed inside the Cassandra daemon

USE WITH CARE!

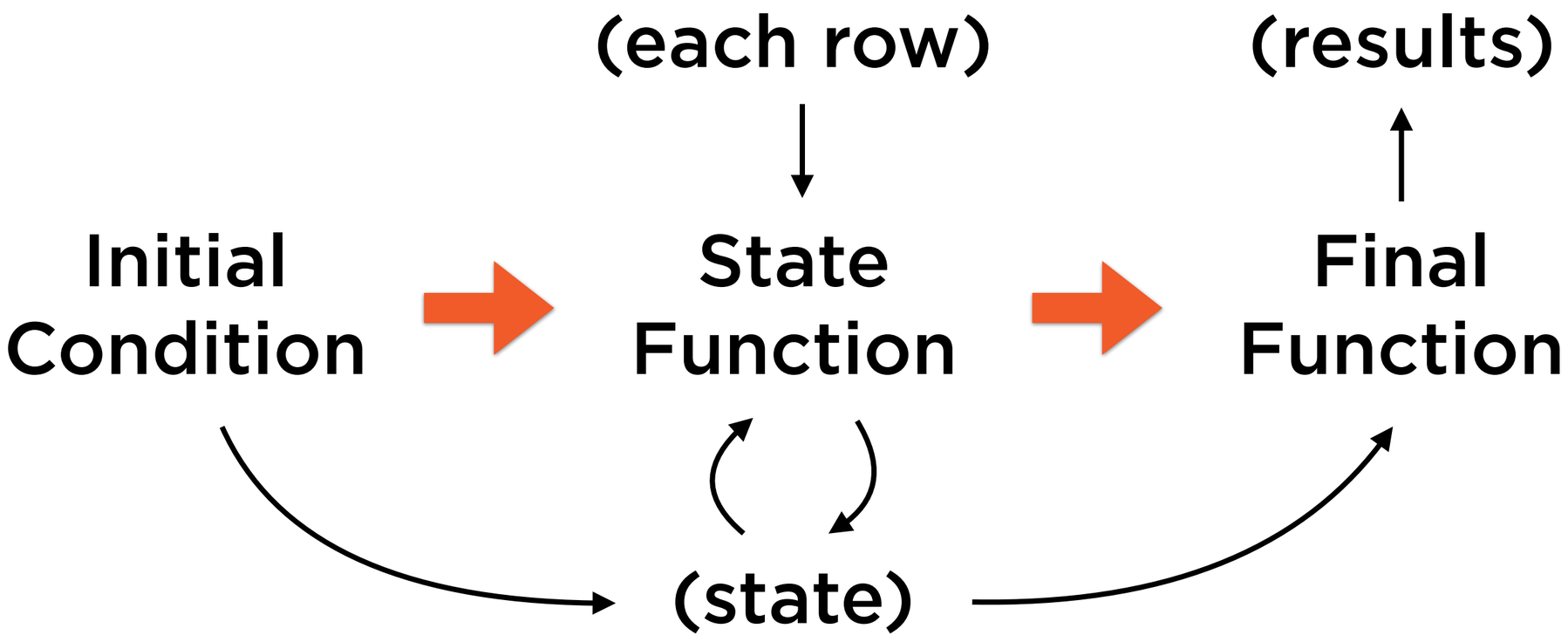
User Defined Functions



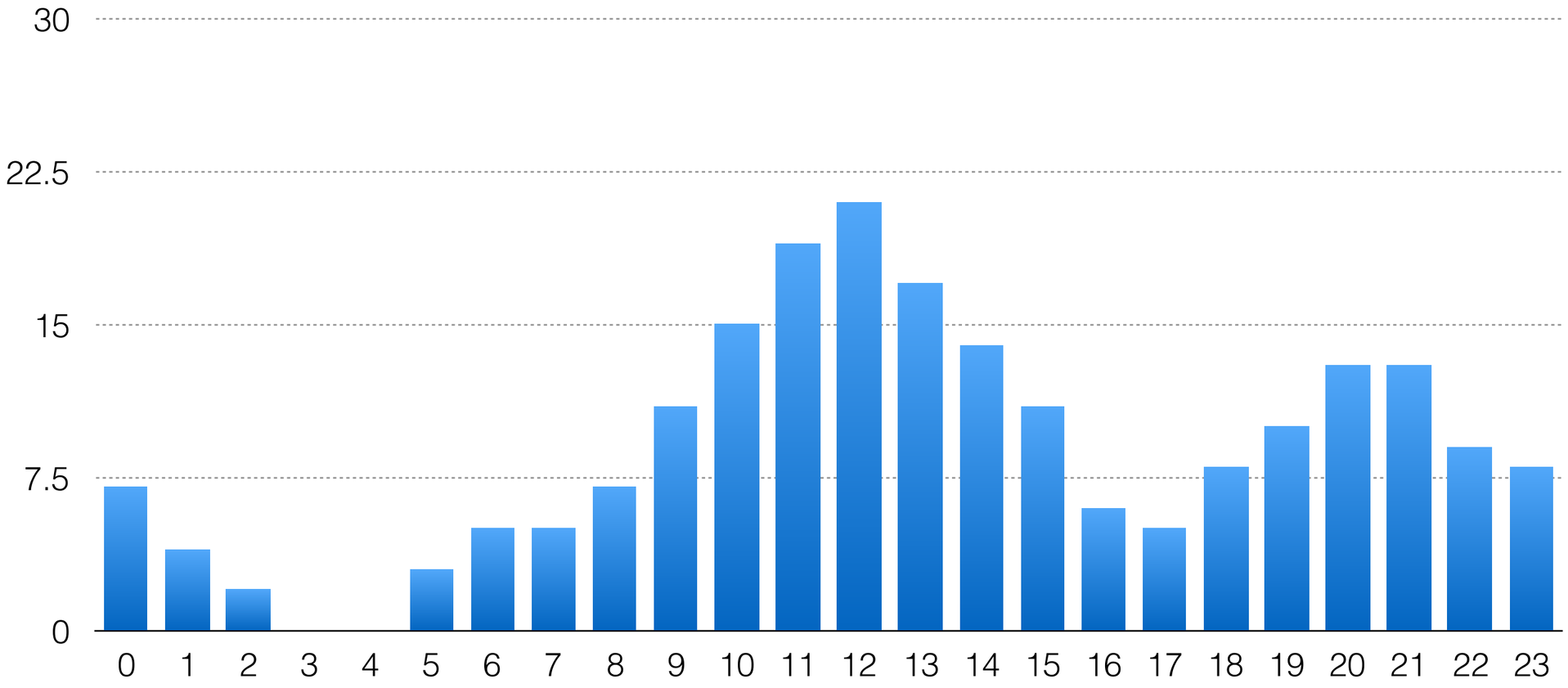

```
SELECT asTimeStr(duration, false)
FROM courses
WHERE id = 'advanced-javascript'
```

24936 → **6 h 55 m**

User Defined Aggregates



User Defined Aggregates



Course page views by hour

User Defined Aggregates

```
SELECT hourly(view_id) FROM course_page_views
   WHERE course_id = ‘advanced-javascript'
```

```
{3: 4, 10: 2}
```

Four page views at 3:00 a.m. and two more at 10:00 a.m.

Demo

Create a time formatting UDF

Create an aggregate for hourly views

Conclusion

Materialized views

Secondary indexes

Indexed collections

Manually maintained indexes

Batches (logged & unlogged)

Lightweight transactions

User-defined functions & aggregates