# C* Data Modeling

Travis Price, Solution Architect

# Agenda

- C* Schema Overview
  - Columns and their components
  - Column Families
  - Keyspaces
- Designing a Data Model
  - Partitioning
  - Indexing
  - Keys
- Example Models
  - Shopping Cart
  - User Activity
  - Logging
  - Form Versioning
- Advanced Modeling
  - LWT
  - Indexing by Table

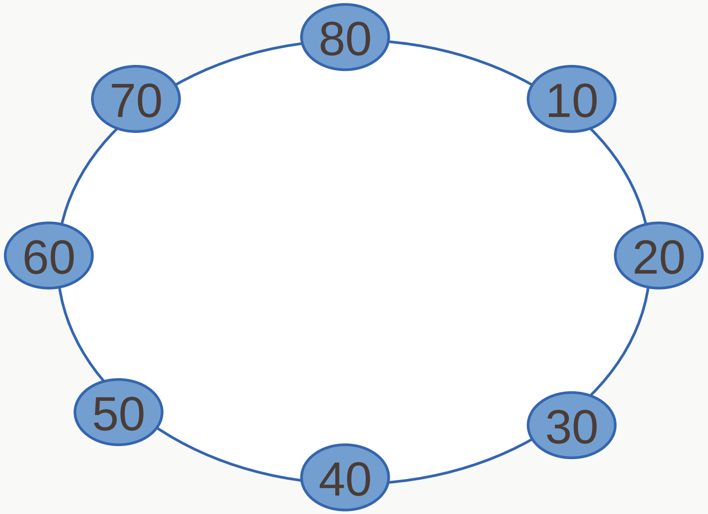
# Cassandra Architecture Review

# Terms

**Cluster**
**Datacenter**
**Node**
**Keyspace**
**Column Family**
**Partition**
Storage row vs. CQL row

**Snitch**
**Replication Factor**
**Consistency**

# Cassandra Architecture



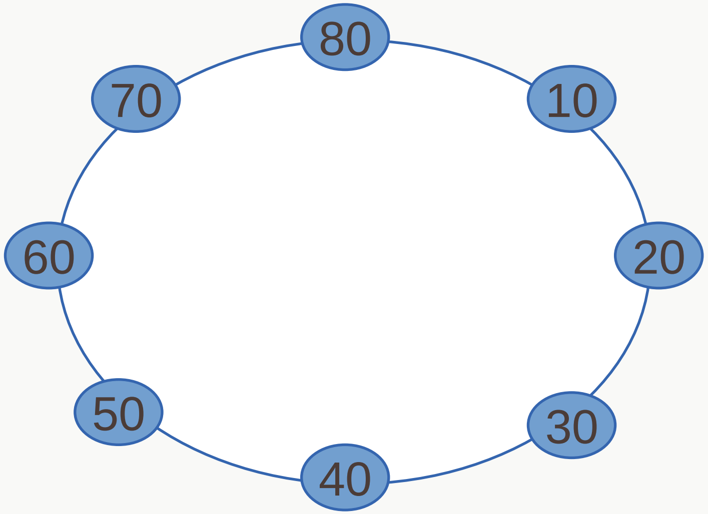

System and hardware failures can and do happen

Peer-to-peer, distributed system
All nodes identical – *Gossip* status and state

Data partitioned among nodes
Data replication ensures fault tolerance

Read/write anywhere design

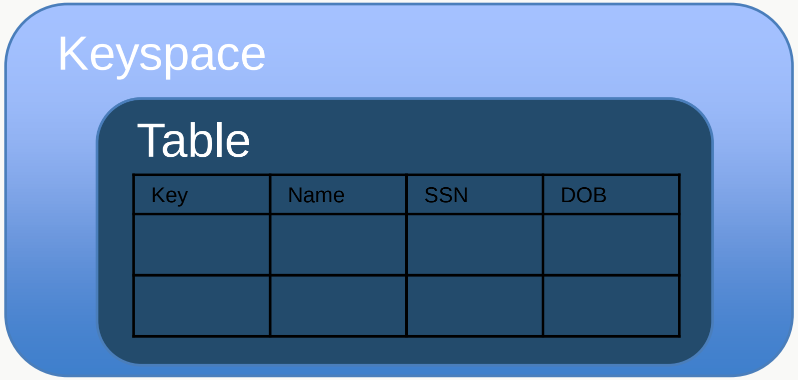
# Keyspaces and Tables



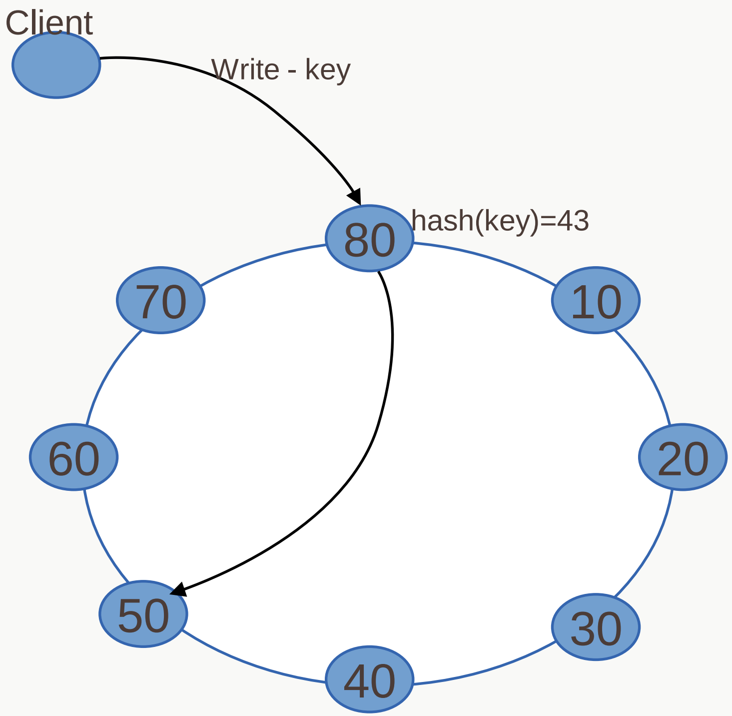

Schema is a row-oriented, column structure

A keyspace is akin to a database
Tables are more flexible/dynamic

A row in a table is indexed by its key
Other columns may be indexed as well



# Writing Data



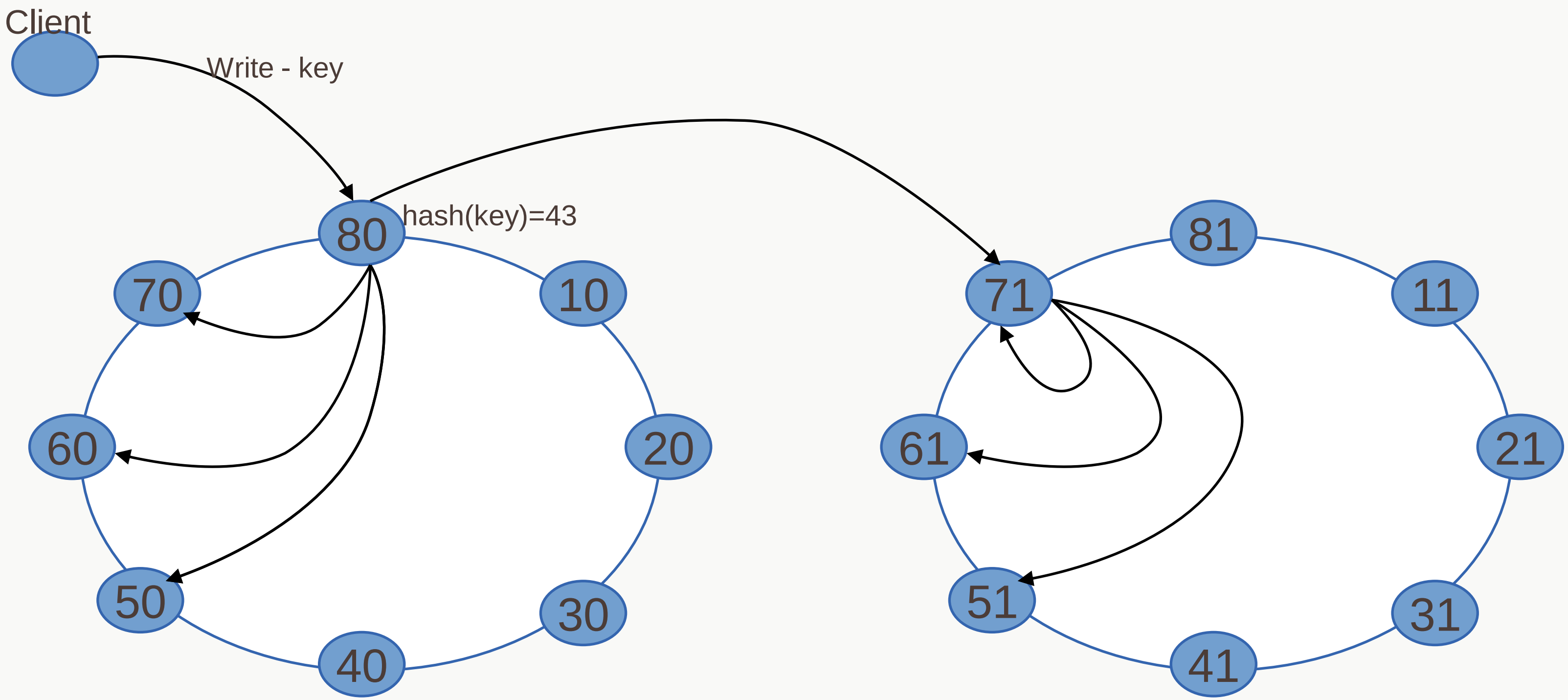

Data is partitioned by token
Nodes own token and range back to previous token
Nodes may own multiple tokens (rf, vnodes)

Client connects to ***any*** node
Node takes on role of ***coordinator*** during transaction
hash(key) => token, rd/wr node(s)

Cluster owns all data – typically, each DC does, too
Nodes own slices of data - one, or more

Replicas can be rack-aware

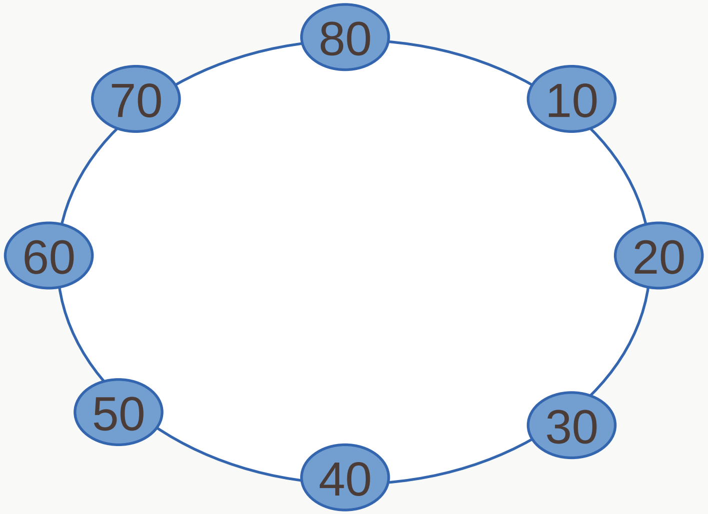
# Replication and DCs




Data Center 1; rf=3

Data Center 2; rf=3

# Tunable Consistency



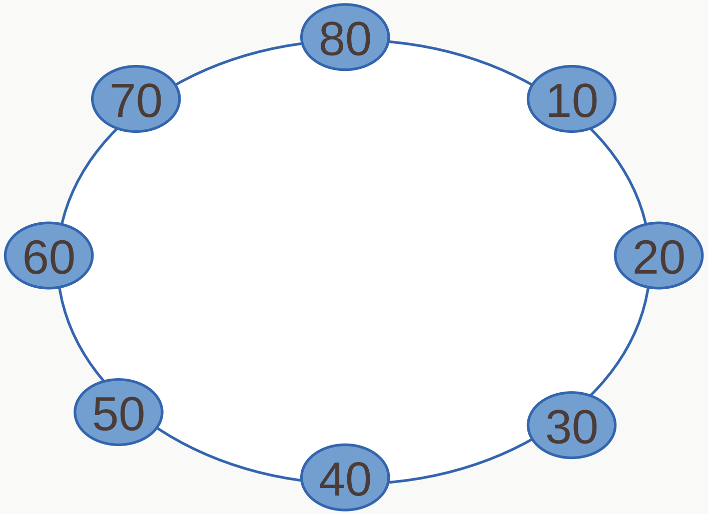

Choose between strong and eventual consistency (one to all nodes acknowledging) depending upon the need

Can be configured on a per-operation basis, and for both reads and writes

Handles multi-data center transparently

CLwr + CLrd > RF  yields consistency

# CQL Language



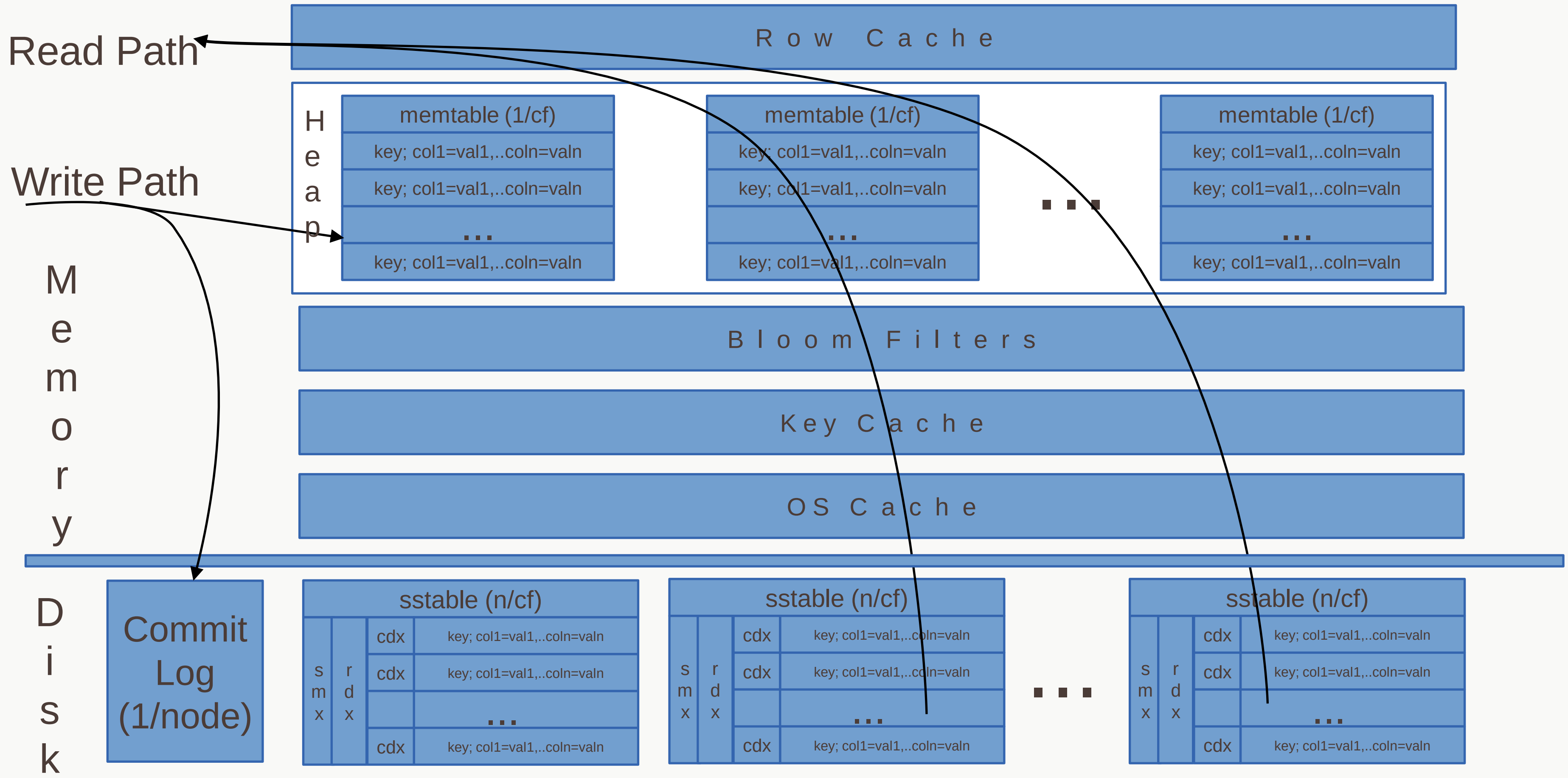

Familiar SQL syntax without joins, group by, etc

Create objects via DDL (e.g. CREATE...)

Core DML commands supported (e.g. CREATE, UPDATE, DELETE)

Query data with SELECT

# Read / Write Path

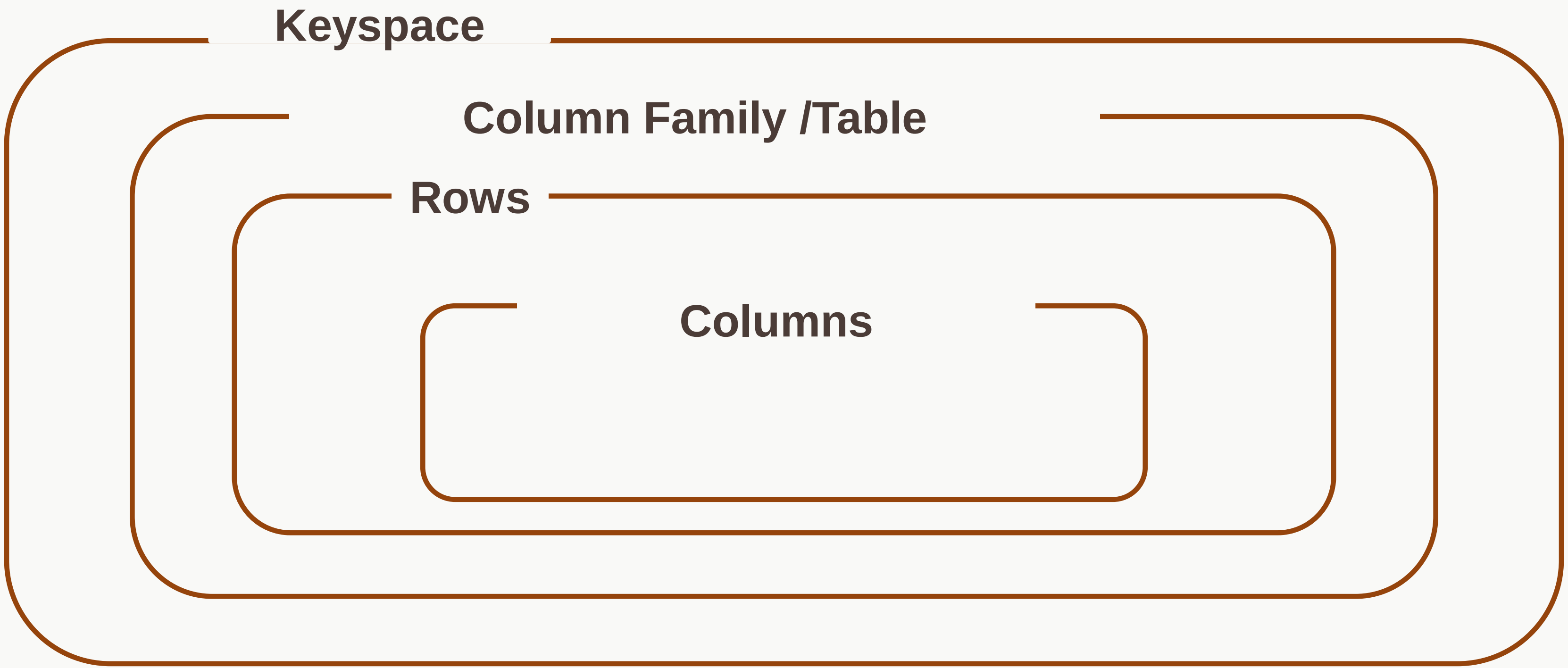


# The Cassandra Schema

Consists of:

- Column
- Column Family
- Keyspace
- Cluster

# High Level Overview



# Components of the Column

The column is the fundamental data type in Cassandra and includes:

- Column name
- Column value
- Timestamp
- TTL (Optional)

# The Column

| Name |
|---|
| Value |
| Timestamp |

```
(Name: “firstName”, Value: “Travis”, Timestamp: 1363106500)
```

# Column Name

- Can be any value
- Can be any type
- Not optional
- Must be unique
- Stored with every value

# Column Value

- Any value
- Any type
- Can be empty – but is required

# Column Names and Values

- the data type for a column *value* is called a *validator*.
- The data type for a column *name* is called a *comparator*.
- Cassandra validates that data type of the keys of rows.
- Columns are sorted, and stored in sorted order on disk, so you have to specify a comparator for columns. This can be reversed… more on this later

# Data Types

| Internal Type | CQL Name | Description |
|---|---|---|
| BytesType | blob | Arbitrary hexadecimal bytes (no validation) |
| AsciiType | ascii | US-ASCII character string |
| UTF8Type | text, varchar | UTF-8 encoded string |
| IntegerType | varint | Arbitrary-precision integer |
| Int32Type | int | 4-byte integer |
| InetAddressType | inet | IP address string in xxx.xxx.xxx.xxx form |
| LongType | bigint | 8-byte long |
| UUIDType | uuid | Type 1 or type 4 UUID |
| TimeUUIDType | timeuuid | Type 1 UUID only (CQL3) |
| DateType | timestamp | Date plus time, encoded as 8 bytes since epoch |
| BooleanType | boolean | true or false |
| FloatType | float | 4-byte floating point |
| DoubleType | double | 8-byte floating point |
| DecimalType | decimal | Variable-precision decimal |
| CounterColumnType | counter | Distributed counter value (8-byte long) |

# Column TimeStamp

- 64-bit integer
- Best Practice
  - Should be created in a consistent manner by all your clients
- Required

```
SELECT WRITETIME (title)
  FROM songs
  WHERE id = 8a172618-b121-4136-bb10-f665cfc469eb;

 writetime(title)
------------------
 1353890782373000
```

# Column TTL

- Defined on INSERT
- Positive delay (in seconds)
- After time expires it is marked for deletion

```
INSERT INTO excelsior.clicks (
  userid, url, date, name)
  VALUES (
    3715e600-2eb0-11e2-81c1-0800200c9a66,
    'http://apache.org',
    '2013-10-09', 'Mary')
    USING TTL 86400;
```

```
SELECT TTL (name) from excelsior.clicks
  WHERE url = 'http://apache.org';
```

Output is, for example, 85908 seconds:

```
 ttl(name)
-----------
 85908
```

# Special Types of Columns

- Counter
- Collections

# Counters

- Allows for addition / subtraction
- 64-bit value
- No timestamp

```
update recommendation_summary
set num_products = num_products + 1
where recommendation = 'highly recommend';
```

# Cassandra feature - Collections

- Collections give you three types:
  - Set
  - List
  - Map
- Each allow for dynamic updates
- Fully supported in CQL 3
- Requires serialization

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

# Cassandra Collections - Set

- Set is sorted by CQL type comparator

set_example set<text>

Collection name

Collection type

CQL Type

INSERT INTO collections_example (id, set_example)
VALUES(1, **{'1-one', '2-two'}**);

# Cassandra Collections - Set Operations

- Adding an element to the set

```
UPDATE collections_example
SET set_example = set_example + {'3-three'} WHERE id = 1;
```

- After adding this element, it will sort to the beginning.

```
UPDATE collections_example
SET set_example = set_example + {'0-zero'} WHERE id = 1;
```

- Removing an element from the set

```
UPDATE collections_example
SET set_example = set_example - {'3-three'} WHERE id = 1;
```

# Cassandra Collections - List

- Ordered by insertion

**list_example list<text>**

Collection name

Collection type

CQL Type

INSERT INTO collections_example (id, list_example)
VALUES(1, **['1-one', '2-two']**);

# Cassandra Collections - List Operations

- Adding an element to the end of a list

```
UPDATE collections_example
SET list_example = list_example + ['3-three'] WHERE id = 1;
```

- Adding an element to the beginning of a list

```
UPDATE collections_example
SET list_example = ['0-zero'] + list_example WHERE id = 1;
```

- Deleting an element from a list

```
UPDATE collections_example
SET list_example = list_example - ['3-three'] WHERE id = 1;
```

# Cassandra Collections - Map

- Key and value
- Key is sorted by CQL type comparator

map_example **map**<int,text>

Collection name

Collection type

Key CQL Type

Value CQL Type

INSERT INTO collections_example (id, map_example)
VALUES(1, **{ 1 : 'one', 2 : 'two' }**);

# Cassandra Collections - Map Operations

- Add an element to the map

UPDATE collections_example
SET map_example[3] = 'three' WHERE id = 1;

- Update an existing element in the map

UPDATE collections_example
SET map_example[3] = 'tres' WHERE id = 1;

- Delete an element in the map

DELETE map_example[3]
FROM collections_example WHERE id = 1;

# User model

- Enhance a simple user table
- Great for static + some dynamic
- Takes advantage of row level isolation

```
CREATE TABLE user_with_location (
	username text PRIMARY KEY,
	first_name text,
	last_name text,
	address1 text,
	city text,
	postal_code text,
	last_login timestamp,
	location_by_date map<timeuuid,text>
);
```

# User profile - Operations

- Adding new login locations to the map

```
UPDATE user_with_location
SET last_login = now(), location_by_date = {now() : '123.123.123.1'}
WHERE username='PatrickMcFadin';
```

- Adding new login locations to the map + TTL!

```
UPDATE user_with_location
USING TTL 2592000  // 30 Days
SET last_login = now(), location_by_date = {now() : '123.123.123.1'}
WHERE username='PatrickMcFadin';
```

# The Cassandra Schema

Consists of:

- Column
- Column Family
- Keyspace
- Cluster

# Column Families / Tables

- Similar to tables
  - Groupings of Rows
  - Tunable Consistency
- De-Normalization
  - To avoid I/O
  - Simplify the Read Path
- Static or Dynamic

# Static Column Families

- Are the most similar to a relational table
- Most rows have the same column names
- Columns in rows can be different

| Row Key | Columns | | | |
|---|---|---|---|---|
| jbellis | **Name** | **Email** | **Address** | **State** |
| | Jonathan | jb@ds.com | 123 main | TX |
| dhutch | **Name** | **Email** | **Address** | **State** |
| | Daria | dh@ds.com | 45 2nd St. | CA |
| egilmore | **Name** | **Email** | | |
| | eric | eg@ds.com | | |

# Dynamic Column Families

- Also called “wide rows”
- Structured so a query into the row will answer a question

| Row Key | Columns | | | |
|---|---|---|---|---|
| jbellis | dhutch | egilmore | datastax | mzcassie |
| dhutch | egilmore | | | |
| egilmore | datastax | mzcassie | | |

# Dynamic Table CQL3 Example

```
CREATE TABLE clicks (
  userid uuid,
  url text,
  timestamp date
  PRIMARY KEY (userid, url)
) WITH COMPACT STORAGE;

SELECT url, timestamp
  FROM clicks
  WHERE userid = 148e9150-1dd2-11b2-0000-242d50cf1fff;

SELECT timestamp
  FROM clicks
  WHERE userid = 148e9150-1dd2-11b2-0000-242d50cf1fff
  AND url = 'http://google.com';
```

# Clustering Order (Comparators)

- Sorts columns on disk by default
- Can change the order

```
create table timeseries (
  event_type text,
  insertion_time timestamp,
  event blob,
  PRIMARY KEY (event_type, insertion_time)
)
WITH CLUSTERING ORDER BY (insertion_time DESC);
```

# The Cassandra Schema

Consists of:

- Column
- Column Family
- Keyspace
- Cluster

# Keyspaces

- Are groupings of Column Families
- Replication strategies
- Replication factor

```
CREATE KEYSPACE demodb
    WITH REPLICATION = {‘class’ : ‘SimpleStrategy’, ‘replication_factor’ : 3};

ALTER KEYSPACE dsg
    WITH REPLICATION = {‘class’ : ‘SimpleStrategy’, ‘replication_factor’ : 3};
```

# Replication Strategies

Two replication strategies are available:

- **SimpleStrategy**: Use for a single data center only. If you ever intend more than one data center, use the **NetworkTopologyStrategy**.
- **NetworkTopologyStrategy**: Highly recommended for most deployments because it is much easier to expand to multiple data centers when required by future expansion.

# Snitches

- A snitch determines which data centers and racks are written to and read from.
- Snitches inform Cassandra about the network topology so that requests are routed efficiently and allows Cassandra to distribute replicas by grouping machines into data centers and racks. All nodes must have exactly the same snitch configuration. Cassandra does its best not to have more than one replica on the same rack (which is not necessarily a physical location).

# Snitch Types

SimpleSnitch

- single-data center deployments (or single-zone in public clouds)

RackInferringSnitch

- determines the location of nodes by rack and data center, which are assumed to correspond to the 3rd and 2nd octet of the node's IP address

PropertyFileSnitch

- user-defined description of the network details
- cassandra-topology.properties file

GossipingPropertyFileSnitch

- defines a local node's data center and rack
- uses gossip for propagating this information to other nodes
- conf/cassandra-rackdc.properties

Amazon Snitches

- EC2Snitch
- EC2MultiRegionSnitch

# Complex Queries

## Partitioning and Indexing

# Partitioners

Generates the token for a key, determining which node owns the record

Cassandra offers the following partitioners:

- **Murmur3Partitioner** (default): uniformly distributes data across the cluster based on MurmurHash hash values.
- **RandomPartitioner**: uniformly distributes data across the cluster based on MD5 hash values.
- **ByteOrderedPartitioner**: keeps an ordered distribution of data lexically by key bytes

# Partitioners – paging through all rows

- **SELECT** * **FROM** test **WHERE** token(k) > token(**42**);

CQL 3 forbids such a query unless the partitioner in use is ordered. Even when using the random partitioner or the murmur3 partitioner, it can sometimes be useful to page through all rows. For this purpose, CQL 3 includes the token function:

The ByteOrdered partitioner arranges tokens the same way as key values, but the RandomPartitioner and Murmur3Partitioner distribute tokens in a completely unordered manner. The token function makes it possible to page through unordered partitioner results. Using the token function actually queries results directly using tokens. Underneath, the token function makes token-based comparisons and does not convert keys to tokens (not k > 42).

# Primary Index Overview

- Index for all of your row keys
- Per-node index
- Partitioner + placement manages which node
- Keys are just kept in ordered buckets
- Partitioner determines how K → Token

# Natural Keys

Examples:

- An email address
- A user id

- Easy to make the relationship
- Less de-normalization
- More risk of an ‘UPSERT’
- Changing the key requires a bulk copy operation

# Surrogate Keys

- Example:
  - UUID
- Independently generated
- Allows you to store multiple versions of a user
- Relationship is now indirect
- Changing the key requires the creation of a new row, or column

# Compound (Composite) Primary Keys

```
CREATE TABLE playlists (
  id uuid,
  song_id uuid,
  title text,
  album text,
  artist text,
  PRIMARY KEY (id, song_id)
);
```

```
 SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
   ORDER BY song_id DESC LIMIT 50;
```

The output looks something like this:

```
  id            | song_id       | album                   | artist          | title
----------------+---------------+-------------------------+-----------------+----------------------
  62c36092...   | a3e64f8f...   |            Tres Hombres |          ZZ Top |            La Grange
  62c36092...   | 8a172618...   |          We Must Obey   |      Fu Manchu  |     Moving in Stereo
  62c36092...   | 2b09185b...   |              Roll Away  |  Back Door Slam | Outside Woman Blues
  62c36092...   | 7db1a490...   | No One Rides for Free   |      Fu Manchu  |             Ojo Rojo
```

# Key Structure

```
CREATE TABLE playlists (
        id uuid,
        song_id uuid,
        title text,
        album text,
        artist text,
        PRIMARY KEY (id, song_id)
);
```

Partition (row) Key

Clustering Key

# Sorting

- It's Free!
- ONLY on the second column in compound Primary Key

```
SELECT * FROM playlists WHERE id = 62c36092-82a1-3a00-93d1-46196ee77204
  ORDER BY song_id DESC LIMIT 50;
```

**Output**

| id | song_id | album | artist | title |
|---|---|---|---|---|
| 62c36092... | a3e64f8f... | Tres Hombres | ZZ Top | La Grange |
| 62c36092... | 8a172618... | We Must Obey | Fu Manchu | Moving in Stereo |
| 62c36092... | 2b09185b... | Roll Away | Back Door Slam | Outside Woman Blues |
| 62c36092... | 7db1a490... | No One Rides for Free | Fu Manchu | Ojo Rojo |

# Composite Partition Keys

```
CREATE TABLE pricechange(
	storeid int,
	eventid int,
	sku text,
	oldprice double,
	newprice double,
	PRIMARY KEY ((storeid, eventid), sku)
);
```

storeid and eventid are used to create the token

# Secondary Indexes

- Need for an easy way to do limited ad-hoc queries
- Supports multiple per row
- Single clause can support multiple selectors
- Implemented as a hash map, not B-Tree
- Low cardinality ONLY

# Secondary Indexes

```
CREATE INDEX ON playlists(artist);
```

Now, you can query the playlists for songs by Fu Manchu, for example:

```
SELECT * FROM playlists WHERE artist = 'Fu Manchu';
```

The output looks something like this:

```
 id           | song_id      | album                  | artist     | title
-------------+--------------+------------------------+------------+-------------------
62c36092... | 7db1a490... | No One Rides for Free | Fu Manchu |          Ojo Rojo
62c36092... | 8a172618... |           We Must Obey | Fu Manchu | Moving in Stereo
```

# Conditional Operators

```
CREATE TABLE ruling_stewards (
  steward_name text,
  king text,
  reign_start int,
  event text,
  PRIMARY KEY (steward_name, king, reign_start)
);
```

```
Select * FROM ruling_stewards
  WHERE king = 'Brego'
  AND reign_start >= 2450
  AND reign_start < 2500 ALLOW FILTERING;
```

The output is:

```
 steward_name | king   | reign_start | event
--------------+--------+-------------+--------------------
      Boromir | Brego  |        2477 |   Attacks continue
       Cirion | Brego  |        2489 | Defeat of Balchoth
```

# ALLOW FILTERING

```
CREATE TABLE users (
    username text PRIMARY KEY,
    firstname text,
    lastname text,
    birth_year int,
    country text
)
CREATE INDEX ON users(birth_year);
```

Valid Queries:

```
SELECT * FROM users;
SELECT firstname, lastname FROM users WHERE birth_year = 1981;
```

Invalid query:

```
SELECT firstname, lastname FROM users WHERE birth_year = 1981 AND country = 'FR';
```

Cassandra cannot guarantee that it won't have to scan large amount of data even if the result to those query is small. Typically, it will scan all the index entries for users born in 1981 even if only a handful are actually from France.

**SELECT firstname, lastname FROM users WHERE birth_year = 1981 AND country = 'FR' ALLOW FILTERING;**

# Designing a Data Model

# The Basics of C* Modeling

- Work backwards
  - What does your application do?
  - What are the access patterns?
- Now design your data model

## Procedures

Consider use case requirements

- What data?
- Ordering?
- Filtering?
- Grouping?
- Events in chronological order?
- Does the data expire?

# De-Normalization

- De-Normalize in C*
  - Forget third normal form
  - Data Duplication is OK
- Concerns
  - Resource contention
  - Latency
  - Client-side joins
    - Avoid them in your C* code

# Foreign Keys

- There are no foreign keys
- No server-side joins

# Table Design

- Ideally each query will be one row
  - Compared to other resources, disk space is cheap
- Reduce disk seeks
- Reduce network traffic

# Workload Preference

- High level of de-normalization means you may have to write the same data many times
- Cassandra handles large numbers of writes well
- If given the choice:
  - Read once
  - Write many

# Concurrent Writes

- A row is always referenced by a Key
- Keys are just bytes
- They must be unique within a CF
- Primary keys are unique
  - But Cassandra will not enforce uniqueness
  - If you are not careful you will accidentally [UPSERT] the whole thing

# The 5 C* Commandments for Developers

1. Start with queries. Don’t data model for data modeling sake.
2. It’s ok to duplicate data.
3. C* is designed to read and write sequentially. Great for rotational disk, awesome for SSDs, awful for NAS.  If your disk has an Ethernet port, it’s not good for C*.
4. Use secondary indexes strategically and cautiously.
5. Embrace wide rows and de-normalization

# Shopping Cart Data Model

Online stores require 100% uptime

# Shopping cart use case

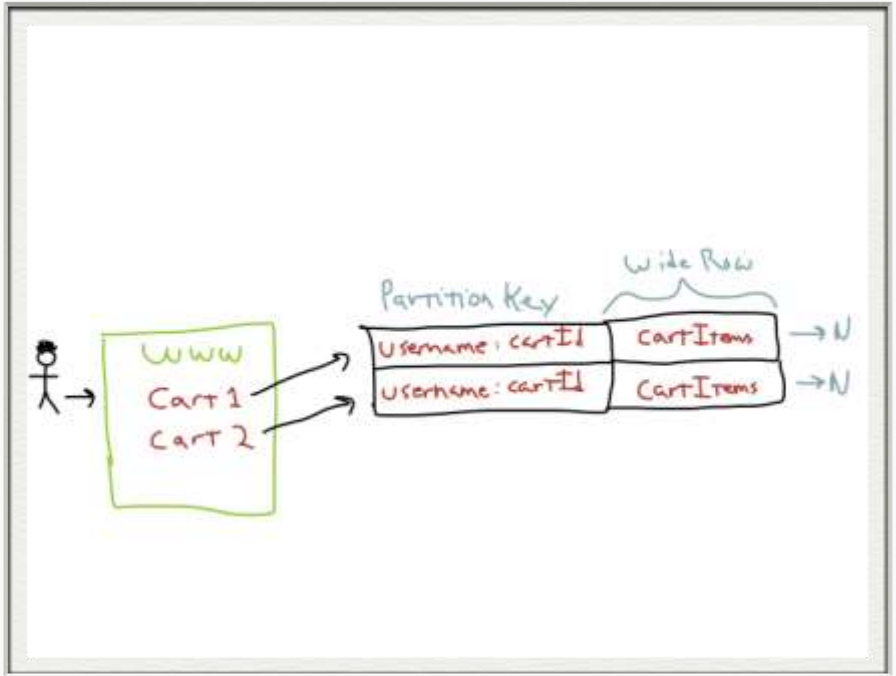
* Store shopping cart data reliably
* Minimize (or eliminate) downtime. Multi-dc
* Scale for the “Cyber Monday” problem

**The bad**

* Every minute off-line is lost $$
* Online shoppers want speed!

# Shopping cart data model

* Each customer can have one or more shopping carts
* De-normalize data for fast access
* Shopping cart == One partition (Row Level Isolation)
* Each item a new column



# Shopping cart data model

```
CREATE TABLE user (
	username varchar,
	firstname varchar,
	lastname varchar,
	shopping_carts set<varchar>,
	PRIMARY KEY (username)
);
```

```
CREATE TABLE shopping_cart (
	username varchar,
	cart_name text
	item_id int,
	item_name varchar,
    description varchar,
	price float,
	item_detail map<varchar,varchar>
	PRIMARY KEY ((username,cart_name),item_id)
);
```

Creates partition row key

One partition (storage row) of data

```
INSERT INTO shopping_cart
(username,cart_name,item_id,item_name,description,price,item_detail)
VALUES ('pmcfadin','Gadgets I want',8675309,'Garmin 910XT','Multisport
training watch',349.99,
{'Related':'Timex sports watch',
'Volume Discount':'10'});
```

Partition row key for one user’s cart

```
INSERT INTO shopping_cart
(username,cart_name,item_id,item_name,description,price,item_detail)
VALUES ('pmcfadin','Gadgets I want',9748575,'Polaris Foot Pod','Bluetooth Smart
foot pod',64.00
{'Related':'Timex foot pod',
'Volume Discount':'25'});
```

Item details. Flexible for any use.

# User Activity Tracking

Watching users, making decisions. Freaky, but cool.

# User activity use case

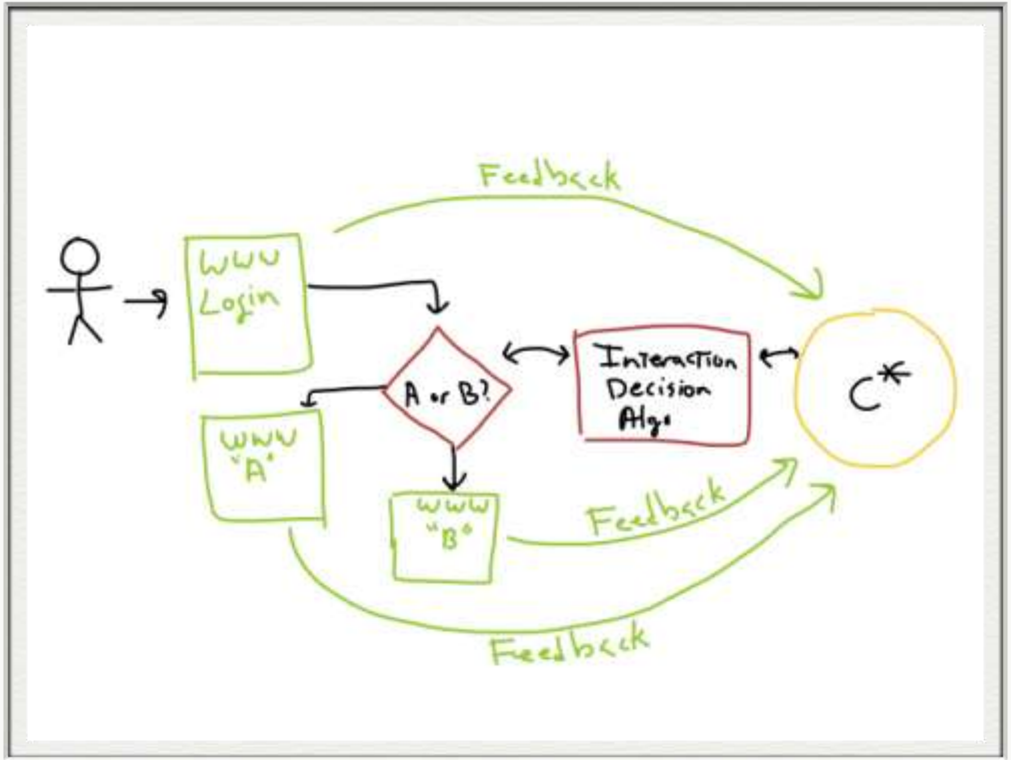
* React to user input in real time
* Support for multiple application pods
* Scale for speed

**The bad**

* Losing interactions is costly
* Waiting for batch(hadoop) is to long

# User activity data model

* Interaction points stored per user in short table
* Long term interaction stored in similar table with date partition
* Process long term later using batch
* Reverse time series to get last N items




DATASTAX

# User activity data model

```
CREATE TABLE user_activity (
    username varchar,
    interaction_time timeuuid,
    activity_code varchar,
    detail varchar,
    PRIMARY KEY (username, interaction_time)
) WITH CLUSTERING ORDER BY (interaction_time
DESC);
```

Reverse order based on timestamp

```
INSERT INTO user_activity
(username,interaction_time,activity_code,detail)
VALUES ('pmcfadin',0D1454E0-D202-11E2-8B8B-0800200C9A66,'100','Normal
login')
USING TTL 2592000;
```

Expire after 30 days

```
CREATE TABLE user_activity_history (
    username varchar,
    interaction_date varchar,
    interaction_time timeuuid,
    activity_code varchar,
    detail varchar,
    PRIMARY KEY ((username,interaction_date),interaction_time)
);
```

```
INSERT INTO user_activity_history
(username,interaction_date,interaction_time,activity_code,detail)
VALUES ('pmcfadin','20130605',0D1454E0-D202-11E2-8B8B-
0800200C9A66,'100','Normal login');
```

# Data model usage

```
select * from user_activity limit 5;
```

```
 username | interaction_time                       | detail                                   | activity_code
----------+--------------------------------------+------------------------------------------+------------------
 pmcfadin | 9ccc9df0-d076-11e2-923e-5d8390e664ec |       Entered shopping area: Jewelry |           301
 pmcfadin | 9c652990-d076-11e2-923e-5d8390e664ec | Created shopping cart: Anniversary gifts |           202
 pmcfadin | 1b5cef90-d076-11e2-923e-5d8390e664ec |     Deleted shopping cart: Gadgets I want |           205
 pmcfadin | 1b0e5a60-d076-11e2-923e-5d8390e664ec |      Opened shopping cart: Gadgets I want |           201
 pmcfadin | 1b0be960-d076-11e2-923e-5d8390e664ec |                             Normal login |           100
```


Maybe put a sale item for flowers too?

# Log collection/aggregation

Machines generate logs at a furious pace. Be ready.

# Log collection use case

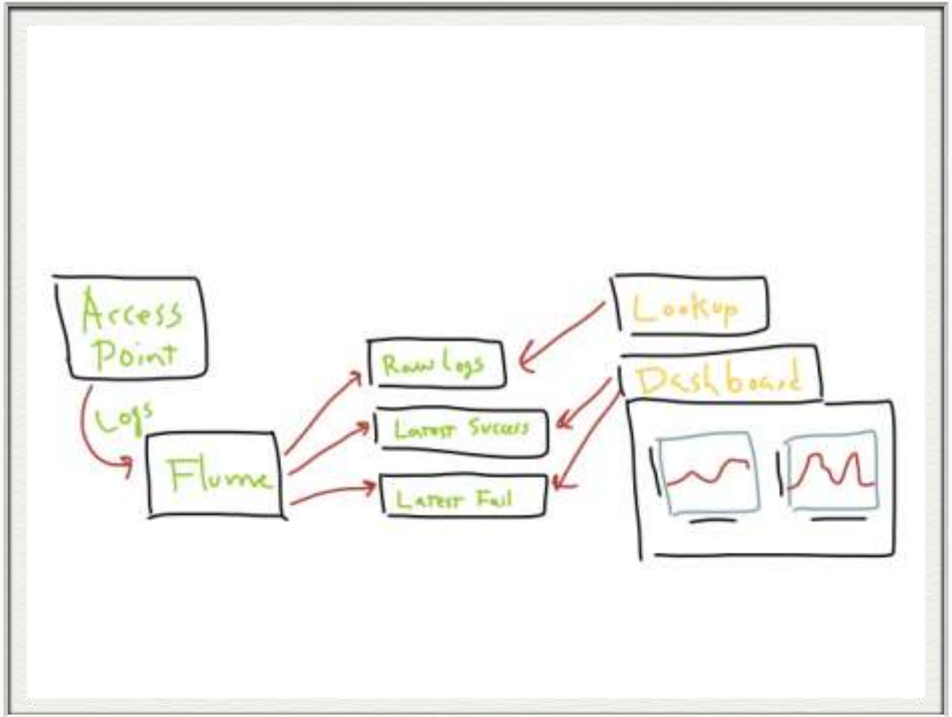
* Collect log data at high speed
* Cassandra near where logs are generated. Multi-datacenter
* Dice data for various uses. Dashboard. Lookup. Etc.

## The bad

* The scale needed for RDBMS is cost prohibitive
* Batch analysis of logs too late for some use cases

# Log collection data model

* Collect and fan out data to various tables
* Tables for lookup based on source and time
* Tables for dashboard with aggregation and summation



# Log collection data model

```
CREATE TABLE log_lookup (
	source varchar,
	date_to_minute varchar,
	timestamp timeuuid,
	raw_log blob,
	PRIMARY KEY ((source,date_to_minute),timestamp)
);
```

Consider storing raw logs as GZIP

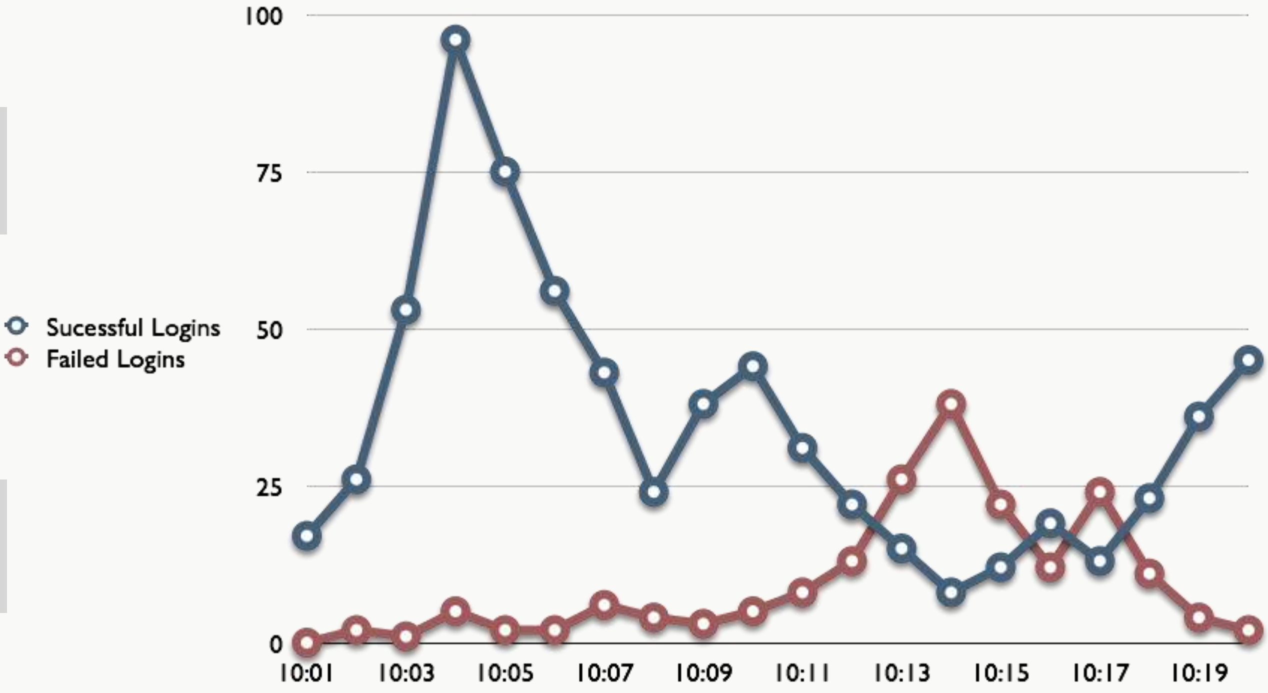
```
CREATE TABLE login_success (
	source varchar,
	date_to_minute varchar,
	successful_logins counter,
	PRIMARY KEY (source,date_to_minute)
) WITH CLUSTERING ORDER BY (date_to_minute DESC);
```

```
CREATE TABLE login_failure (
	source varchar,
	date_to_minute varchar,
	failed_logins counter,
	PRIMARY KEY (source,date_to_minute)
) WITH CLUSTERING ORDER BY (date_to_minute DESC);
```

# Log dashboard

```
SELECT date_to_minute,successful_logins
FROM login_success
LIMIT 20;
```

```
SELECT date_to_minute,failed_logins
FROM login_failure
LIMIT 20;
```



# User Form Versioning

Because mistaks mistakes happen

# Form versioning use case

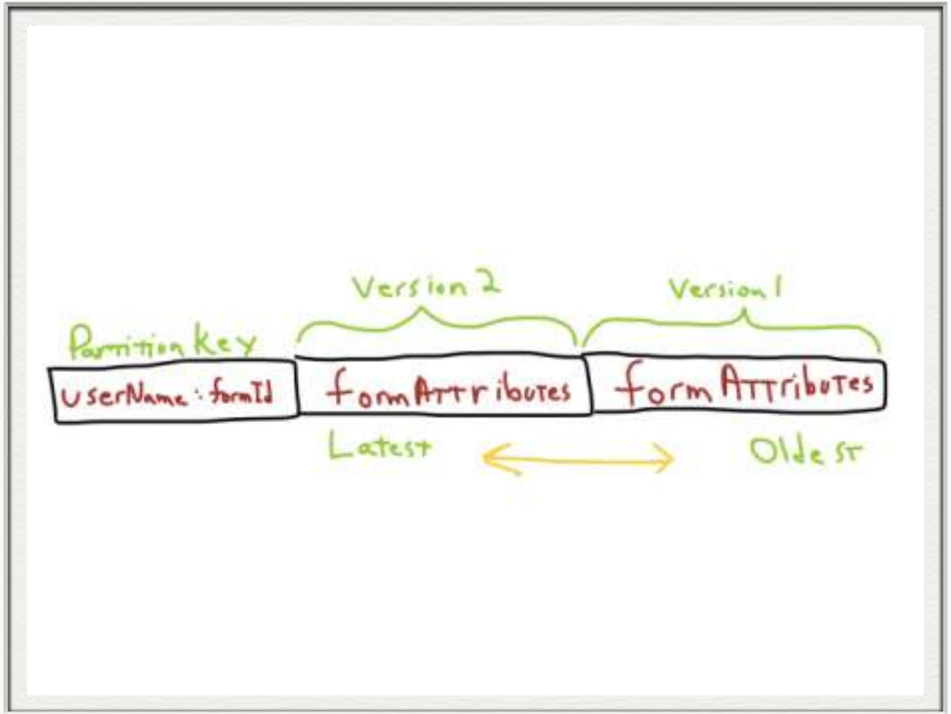
* Store every possible version efficiently
* Scale to any number of users
* Commit/Rollback functionality on a form

## The bad

* In RDBMS, many relations that need complicated join
* Needs to be in cloud and local data center

# Form version data model

* Each user has a form
* Each form needs versioning
* Separate table to store live version
* Exclusive lock on a form



# Form version data model

```
CREATE TABLE working_version (
	username varchar,
	form_id int,
	version_number int,
	locked_by varchar,
	form_attributes map<varchar,varchar>
	PRIMARY KEY ((username, form_id), version_number)
) WITH CLUSTERING ORDER BY (version_number DESC);
```

## 1. Insert first version

```
INSERT INTO working_version
(username, form_id, version_number, locked_by, form_attributes)
VALUES ('pmcfadin',1138,1,'',
{'FirstName<text>':'First Name: ',
'LastName<text>':'Last Name: ',
'EmailAddress<text>':'Email Address: ',
'Newsletter<radio>':'Y,N'});
```

## 2. Lock for one user

```
UPDATE working_version
SET locked_by = 'pmcfadin'
WHERE username = 'pmcfadin'
AND form_id = 1138
AND version_number = 1;
```

## 3. Insert new version. Release lock

```
INSERT INTO working_version
(username, form_id, version_number, locked_by, form_attributes)
VALUES ('pmcfadin',1138,2,null,
{'FirstName<text>':'First Name: ',
'LastName<text>':'Last Name: ',
'EmailAddress<text>':'Email Address: ',
'Newsletter<checkbox>':'Y'});
```

# Light Weight Transactions

# The race is on

## Process 1

**SELECT** firstName, lastName
**FROM** users
**WHERE** username = 'pmcfadin';

(0 rows)

**INSERT INTO** users (username, firstname,
  lastname, email, password, created_date)
**VALUES** ('pmcfadin','Patrick','McFadin',
  ['patrick@datastax.com'],
  'ba27e03fd95e507daf2937c937d499ab',
  '2011-06-20 13:50:00');

This one wins

T0

T1

T2

T3

## Process 2

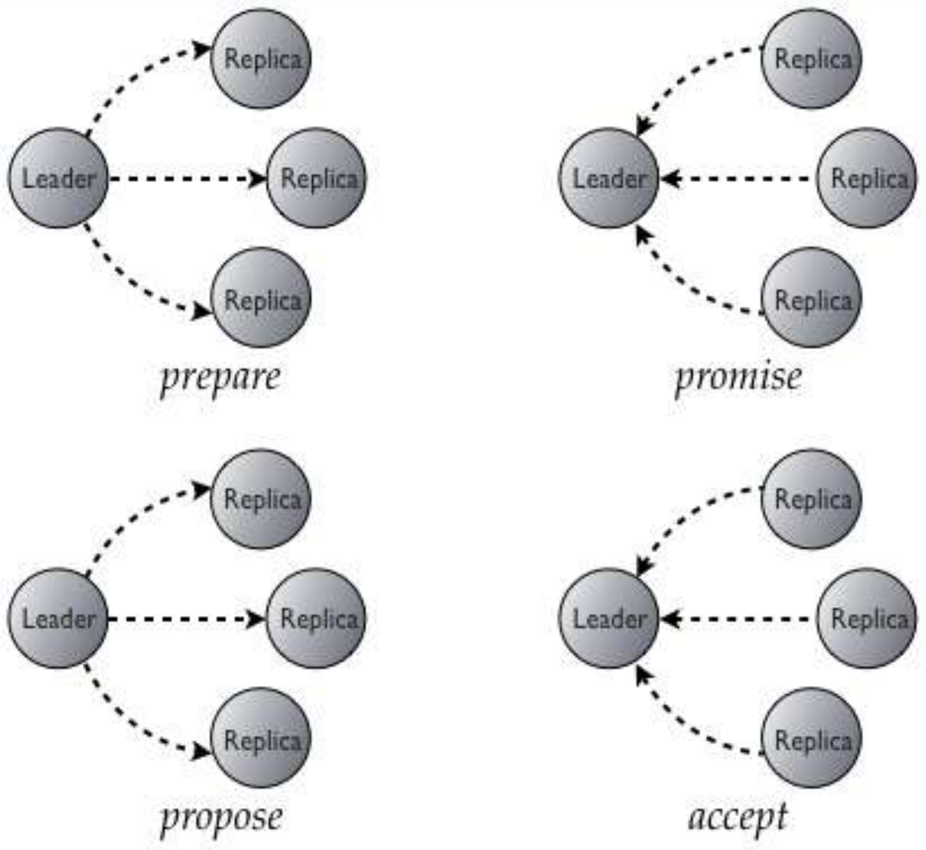
**SELECT** firstName, lastName
**FROM** users
**WHERE** username = 'pmcfadin';

(0 rows)

Got nothing! Good to go!

**INSERT INTO** users (username, firstname,
  lastname, email, password, created_date)
**VALUES** ('pmcfadin','Paul','McFadin',
  ['paul@oracle.com'],
  'ea24e13ad95a209ded8912e937d499de',
  '2011-06-20 13:51:00');

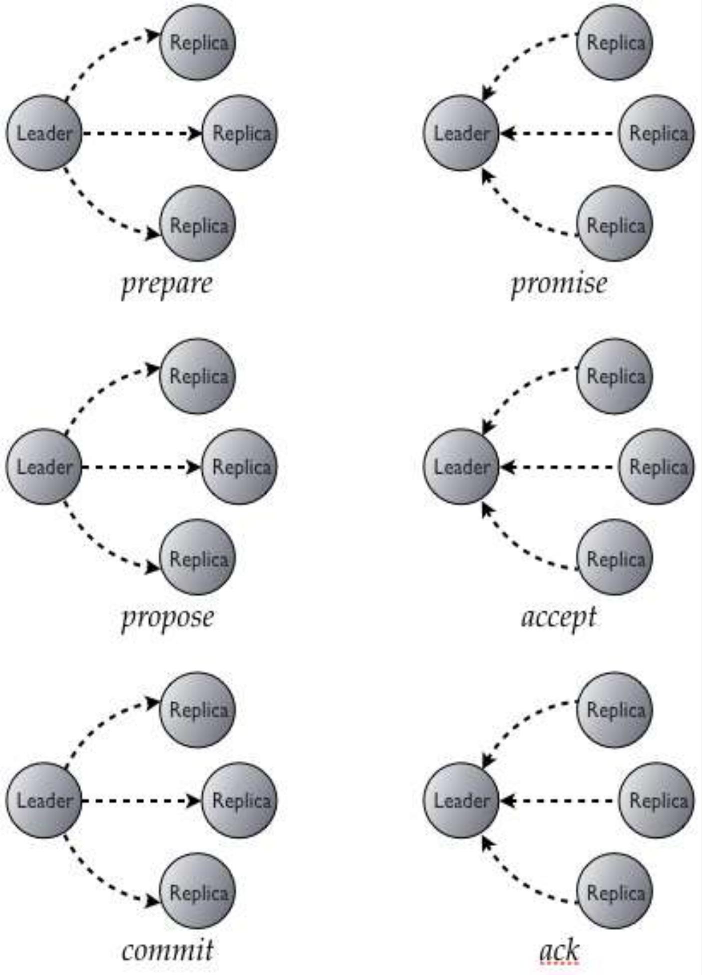
# The Paxos Consensus Protocol




Quorum-based

Algorithm

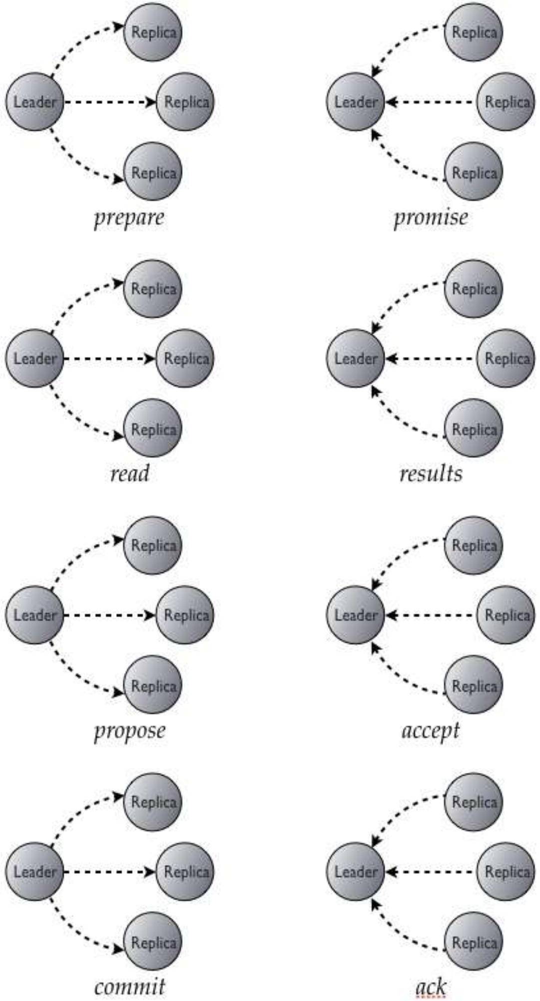
# Extending Paxos




Add third phase to reset state for subsequent proposals

# Compare & Set

Read the current value of the row to see if it matches the expected one



# Solution LWT

Process 1

```
INSERT INTO users (username, firstname,
  lastname, email, password, created_date)
VALUES ('pmcfadin','Patrick','McFadin',
  ['patrick@datastax.com'],
  'ba27e03fd95e507daf2937c937d499ab',
  '2011-06-20 13:50:00')
IF NOT EXISTS;
```

```
[applied]
-----------
   True
```

T0

T1

- Check performed for record
- Paxos ensures exclusive access
- applied = true: Success

# Solution LWT

T2

T3

Process 2

```
INSERT INTO users (username, firstname,
  lastname, email, password, created_date)
VALUES ('pmcfadin','Paul','McFadin',
  ['paul@oracle.com'],
  'ea24e13ad95a209ded8912e937d499de',
  '2011-06-20 13:51:00')
IF NOT EXISTS;
```

```
 [applied] | username | created_date             | firstname | lastname
-----------+----------+--------------------------+-----------+----------
     False | pmcfadin | 2011-06-20 13:50:00-0700 |   Patrick |  McFadin
```

- applied = false: Rejected
- No record stomping!

# Another LWT Example

UPDATE users
SET reset_token = null, password = ‘newpassword’
WHERE login = ‘jbellis’
IF reset_token = ‘some-generated-reset-token’

Prevents a password from being reset twice using the same reset token.

# LWT Fine Print

- Light Weight Transactions solve edge conditions
- They have latency cost.
  - Be aware
  - Load test
  - Consider in your data model

# Indexing with Tables

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

Goals:

1. Create row key for speed
2. Use wide rows for efficiency

# 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

| tag | VideoId1 | .. | VideoIdN |
|---|---|---|---|

Efficie
nt

# 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
```

# Data Modeling Challenge!

DATASTAX

## Products

| Column Name | Condensed... |
|---|---|
| ProductID | int |
| ProductName | nvarchar(40) |
| SupplierID | int |
| CategoryID | int |
| QuantityPerUnit | nvarchar(20) |
| UnitPrice | money |
| UnitsInStock | smallint |
| UnitsOnOrder | smallint |
| ReorderLevel | smallint |
| Discontinued | bit |

## Order Details

| Column ... | Conde... |
|---|---|
| OrderID | int |
| ProductID | int |
| UnitPrice | money |
| Quantity | smallint |
| Discount | real |

## EmployeeTerritories

| Column Name | Condensed T... |
|---|---|
| EmployeeID | int |
| TerritoryID | nvarchar(20) |

## Orders

| Column Name | Condensed... |
|---|---|
| OrderID | int |
| CustomerID | nchar(5) |
| EmployeeID | int |
| OrderDate | datetime |
| RequiredDate | datetime |
| ShippedDate | datetime |
| ShipVia | int |
| Freight | money |
| ShipName | nvarchar(40) |
| ShipAddress | nvarchar(60) |
| ShipCity | nvarchar(15) |
| ShipRegion | nvarchar(15) |
| ShipPostalCode | nvarchar(10) |
| ShipCountry | nvarchar(15) |

## Customers

| Column Name | Condensed... |
|---|---|
| CustomerID | nchar(5) |
| CompanyName | nvarchar(40) |
| ContactName | nvarchar(30) |
| ContactTitle | nvarchar(30) |
| Address | nvarchar(60) |
| City | nvarchar(15) |
| Region | nvarchar(15) |
| PostalCode | nvarchar(10) |
| Country | nvarchar(15) |
| Phone | nvarchar(24) |
| Fax | nvarchar(24) |

## Categories

| Column Name | Condensed... |
|---|---|
| CategoryID | int |
| CategoryName | nvarchar(15) |
| Description | ntext |
| Picture | image |

## Employees

| Column Name | Condensed ... |
|---|---|
| EmployeeID | int |
| LastName | nvarchar(20) |
| FirstName | nvarchar(10) |
| Title | nvarchar(30) |
| TitleOfCourtesy | nvarchar(25) |
| BirthDate | datetime |
| HireDate | datetime |
| Address | nvarchar(60) |
| City | nvarchar(15) |
| Region | nvarchar(15) |
| PostalCode | nvarchar(10) |
| Country | nvarchar(15) |
| HomePhone | nvarchar(24) |
| Extension | nvarchar(4) |
| Photo | image |
| Notes | ntext |
| ReportsTo | int |
| PhotoPath | nvarchar(255) |

## Suppliers

| Column Name | Condensed... |
|---|---|
| SupplierID | int |
| CompanyName | nvarchar(40) |
| ContactName | nvarchar(30) |
| ContactTitle | nvarchar(30) |
| Address | nvarchar(60) |
| City | nvarchar(15) |
| Region | nvarchar(15) |
| PostalCode | nvarchar(10) |
| Country | nvarchar(15) |
| Phone | nvarchar(24) |
| Fax | nvarchar(24) |
| HomePage | ntext |

## Territories

| Column Name | Condensed... |
|---|---|
| TerritoryID | nvarchar(20) |
| TerritoryDescription | nchar(50) |
| RegionID | int |

## Region

| Column Name | Conden... |
|---|---|
| RegionID | int |
| RegionDescription | nchar(50) |

## CustomerCustomerDemo

| Column Name | Condensed T... |
|---|---|
| CustomerID | nchar(5) |
| Customer... | nchar(10) |

## Shippers

| Column Name | Condensed... |
|---|---|
| ShipperID | int |
| CompanyName | nvarchar(40) |
| Phone | nvarchar(24) |

## CustomerDemographics

| Column Name | Conden... |
|---|---|
| CustomerTypeID | nchar(10) |
| CustomerDesc | ntext |

# 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