NoSQL: Cassandra

School of Information Studies
Syracuse University

Introduction to Cassandra

Cassandra

Developed Originally by Facebook

- Developed by Facebook, open sourced in 2008
- Inspired by Google Bigtable and Amazon DynamoDB papers

Eventual Consistency (BASE)

- Can always write, but not guaranteed to read, the same thing
- Scales well horizontally

Data Model

- Distributed wide-column store database
- Key maps to one or more columns
- Has an SQL-like query language CQL
- There are no integrity constraints

Transaction Management

ACID without the "C"

Atomic	Integrated	Durable	*Not consistent*

When you query, you can set the level of consistency you desire from the data.

School of Information Studies
Syracuse University

Cassandra Is...

Good for

Time series data

High-volume writes with subject-specific reads

Any application where you must guarantee writes

Not Good for

Highly normalized data

Ad hoc queries across multiple subjects

Data warehouses

Cassandra Use Cases

Time series data

IoT applications

User activity tracking

Performance monitoring

Social media analytics

E-commerce

Messaging

Databases the Cassandra Way

- Query design influences table design
- Redundant, de-normalized data, including different versions of the same conceptual entity
- One large flat table

School of Information Studies
Syracuse University

Cassandra Ring Architecture

All nodes play the same role

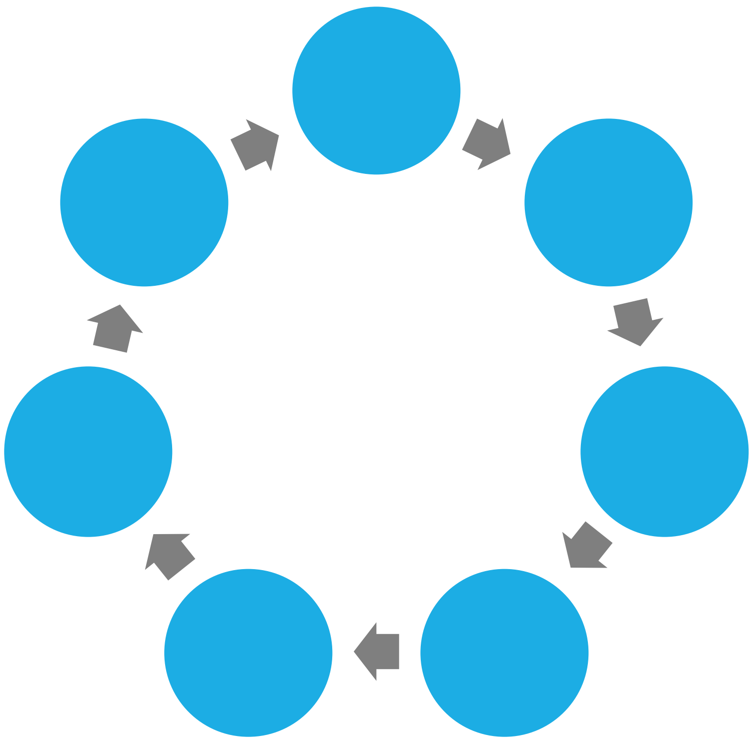
All nodes communicate with each other

There is no “single master” like MongoDB, Redis, and RDBMS solutions

No single point of failure

To scale—add more nodes!

You can configure replication and number of replicas



Cassandra Data Model

Concepts

Cluster: a collection of Cassandra nodes

Keyspace: a container for data tables and indexes; this is like a "database" in a RDBMS; defines the replication strategy

Table: similar to a RDBMS table, but all columns are optional "wide column store" implementation

Primary key: used to uniquely identify a row in the table and to distribute the rows across the cluster

Index: similar function as an RDBMS index, but implemented differently

Keyspaces

More than just a logical container for tables and indexes

You must define a replication strategy for the keyspace

Strategies

- Simple—all nodes in the same data center, values distributed evenly over each node
- NetworkTopology—for use in multiple data centers; rack aware

```
CREATE KEYSPACE keyspace_name WITH
 replication = {
       'class' :'SimpleStrategy',
       'replication_factor' : number
};
USE keyspace_name;
```

Demo: Keyspaces

DESCRIBE KEYSPACES; command to list all keyspaces

CREATE KEYSPACE to make our sysmon keyspace

Set the working keyspace with USE

School of Information Studies
Syracuse University

TABLES AND KEYS

Tables

Tables similar to RDBMS tables, except since there are no "joins" in Cassandra; the tables should be highly de-normalized

Cassandra tables are best suited for the capture of events such as orders, sensor readings, and so on; these data are usually time series

Tables are in wide-column store format; this means that all columns are optional, so there are no integrity constraints

Use the CREATE TABLE statement to make tables

Common Column Data Types

Basic Types

INT/BIGINT

VARCHAR (or TEXT)

DECIMAL/DOUBLE

TIMESTAMP (date + time)

- Milliseconds since epoch
- yyyy-mm-dd hh:MM:ss

UUID

TIMEUUID—conflict-free time stamp

Collection Types

LIST—list of the same items, order matters

MAP—key/value pairs; like a Python dictionary

SET—collection of items with no duplicates, order does not matter; more efficient than lists

http://cassandra.apache.org/doc/latest/cql/types.html

What Is a Sparse/Wide Table?

Key	Columns
mafudge	**Name:** *Mike* **Gender:** *M* **Age:** *47*
kmfudge	**Name:** *Kim* **Gender:** *F*
jafudge	**Name:** *Jackson* **Age:** *12*
dmfudge	**Name:** *Dominick* **Gender:** *M*

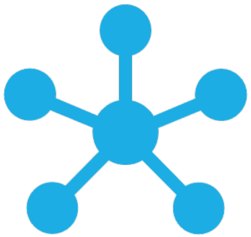
Primary Keys Are Important

Must uniquely identify a row, but will also limit how we can retrieve rows.

The first attribute in the primary key list is the partition key. This value is hashed and the value determines to which node of the cluster data are written.

Any additional parts of the primary key are the clustering key, which determines the order by which the data are written to that node (similar to an RDBMS primary key).

Keys

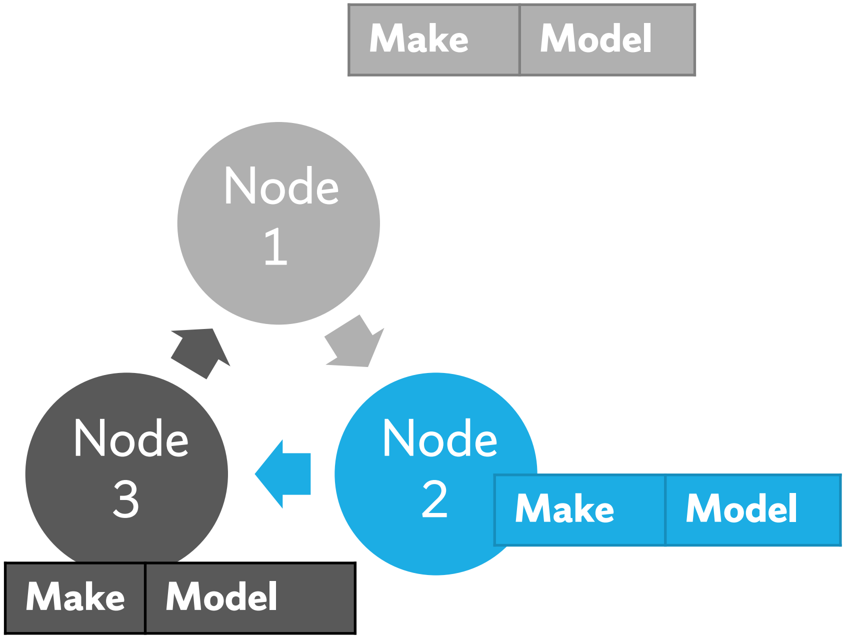


Partition key—distributes data across nodes

Clustering key—sorts data within a node

CASSANDRA CLUSTER VISUALIZED

Cassandra Cluster Visualized



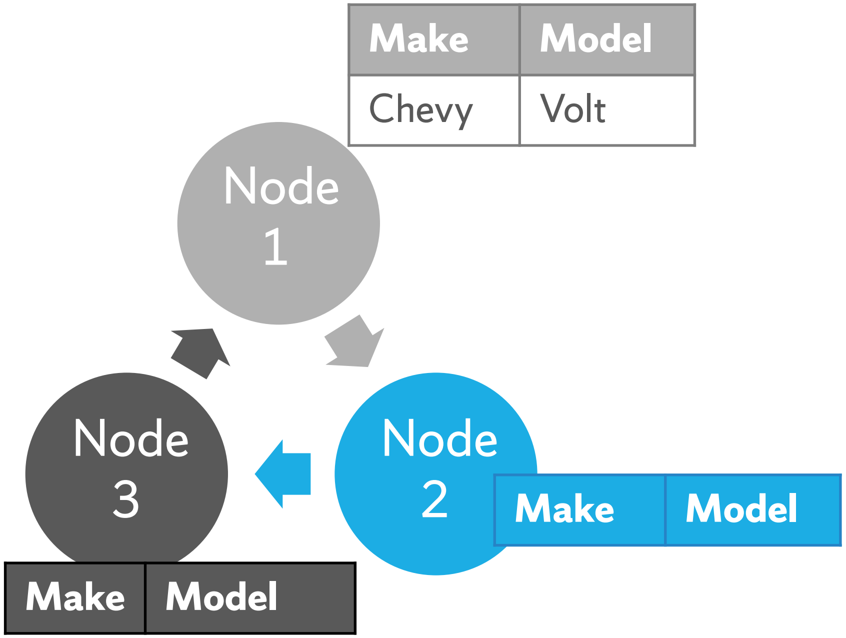
Let's assume that we have a cars table with:

- Partition key = make
- Cluster key = model

Writing the following data:

School of Information Studies
Syracuse University

Cassandra Cluster Visualized



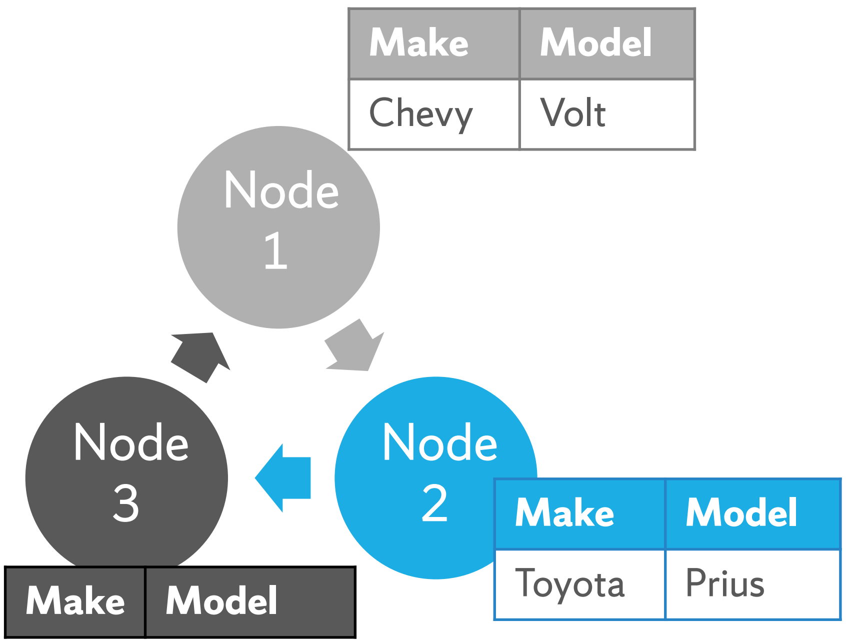
Let's assume that we have a cars table with:

- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt

Cassandra Cluster Visualized



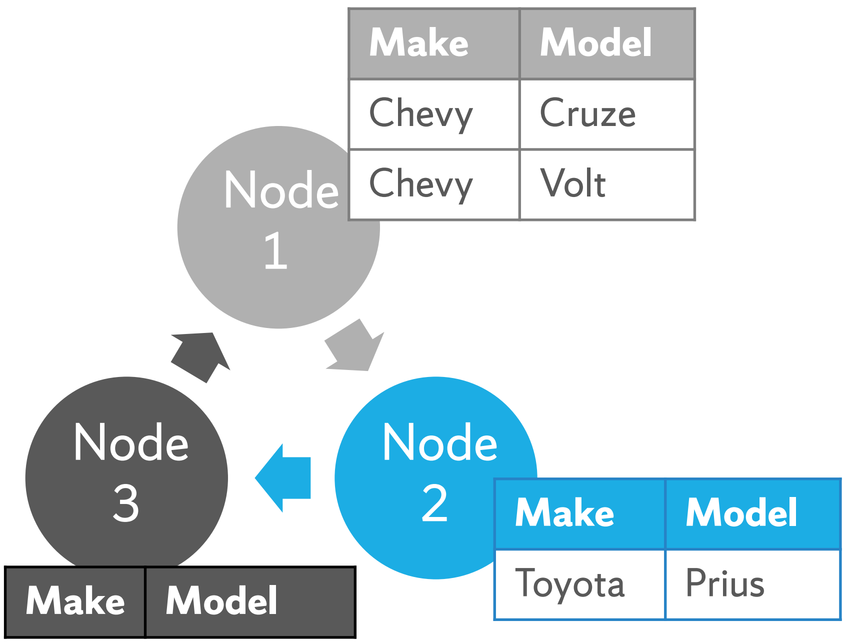
Let's assume that we have a cars table with:

- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius

Cassandra Cluster Visualized



Let's assume that we have a cars table with:

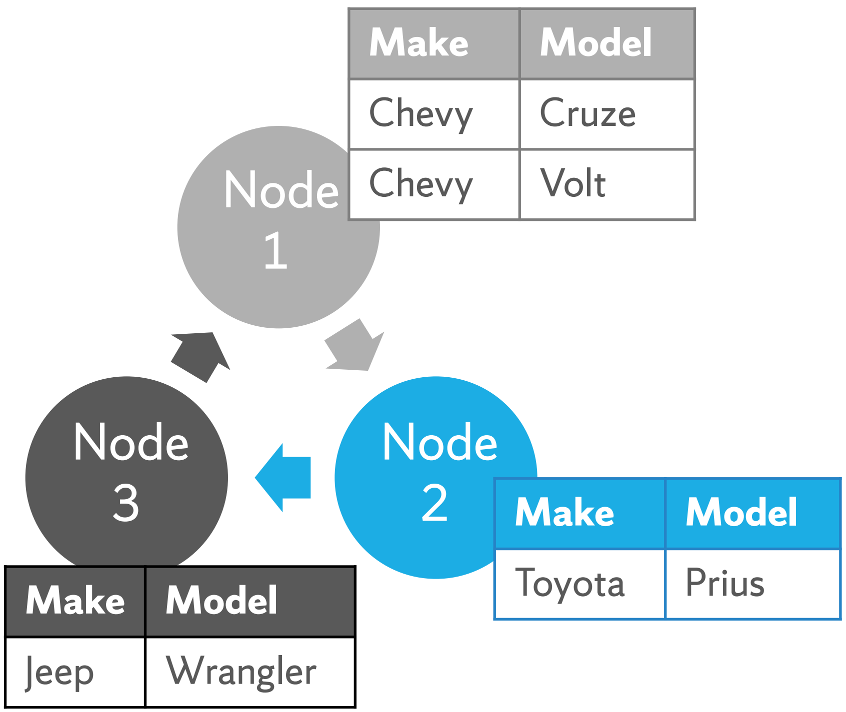
- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius
3. Chevy Cruze

School of Information Studies
Syracuse University

Cassandra Cluster Visualized



Make	Model
Chevy	Cruze
Chevy	Volt

Make	Model
Toyota	Prius

Make	Model
Jeep	Wrangler

Let's assume that we have a cars table with:

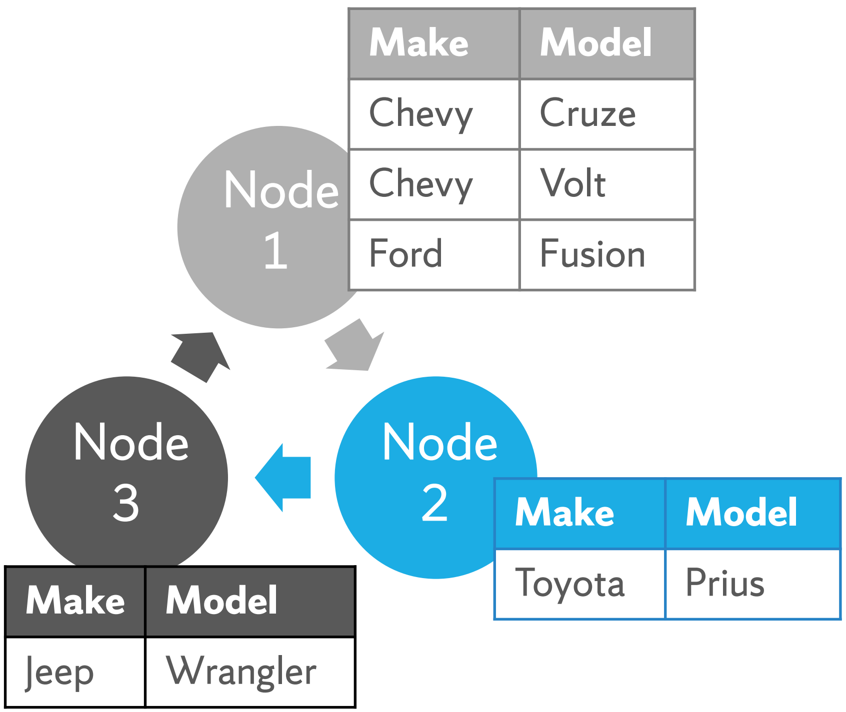
- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius
3. Chevy Cruze
4. Jeep Wrangler

School of Information Studies
Syracuse University

Cassandra Cluster Visualized



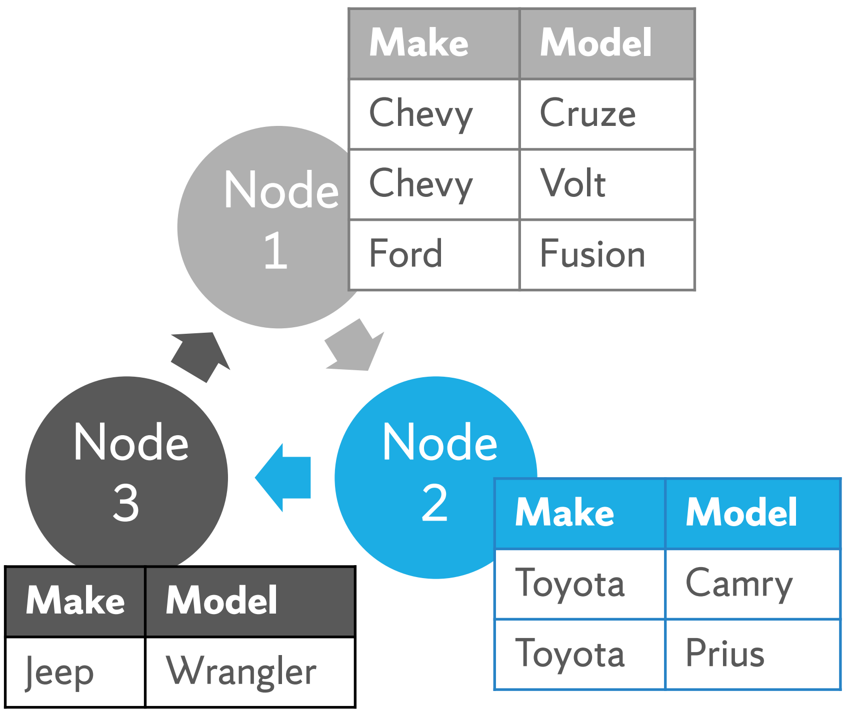
Let's assume that we have a cars table with:

- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius
3. Chevy Cruze
4. Jeep Wrangler
5. Ford Fusion

Cassandra Cluster Visualized



Let's assume that we have a cars table with:

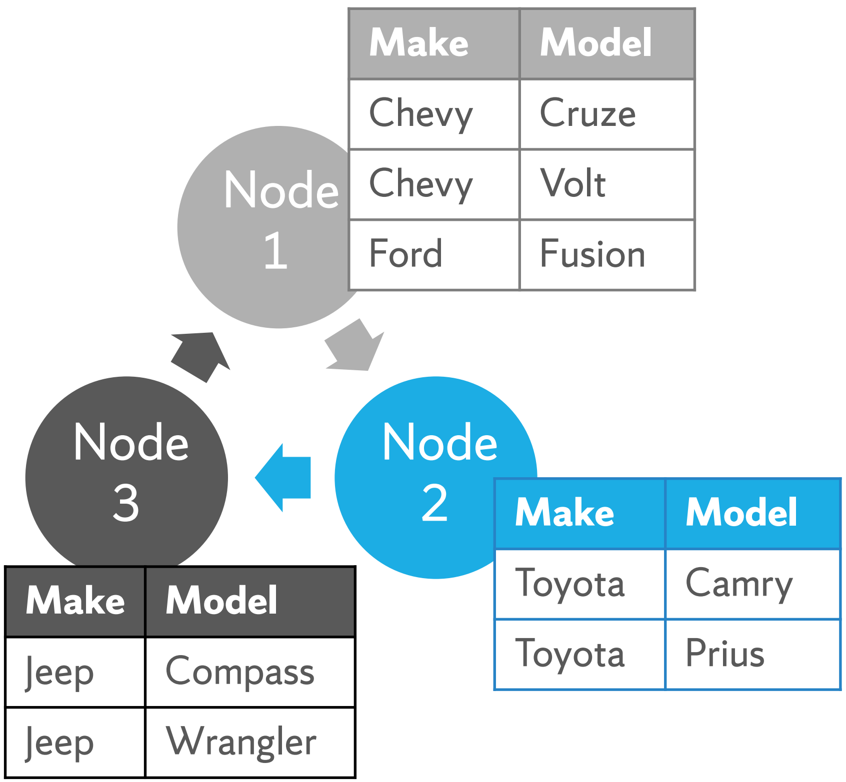
- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius
3. Chevy Cruze
4. Jeep Wrangler
5. Ford Fusion
6. Toyota Camry

School of Information Studies
Syracuse University

Cassandra Cluster Visualized



Make	Model
Chevy	Cruze
Chevy	Volt
Ford	Fusion

Make	Model
Toyota	Camry
Toyota	Prius

Make	Model
Jeep	Compass
Jeep	Wrangler

Let's assume that we have a cars table with:

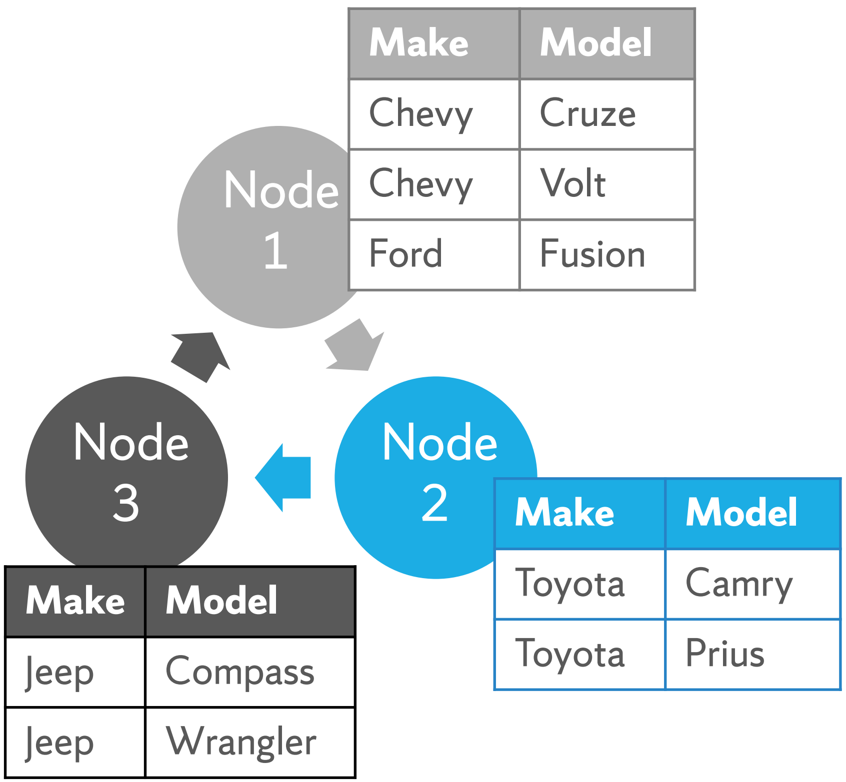
- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius
3. Chevy Cruze
4. Jeep Wrangler
5. Ford Fusion
6. Toyota Camry
7. Jeep Compass

Cassandra Cluster Visualized

8. Audi A8?



Make	Model
Chevy	Cruze
Chevy	Volt
Ford	Fusion

Make	Model
Toyota	Camry
Toyota	Prius

Make	Model
Jeep	Compass
Jeep	Wrangler

Let's assume that we have a cars table with:

- Partition key = make
- Cluster key = model

Writing the following data:

1. Chevy Volt
2. Toyota Prius
3. Chevy Cruze
4. Jeep Wrangler
5. Ford Fusion
6. Toyota Camry
7. Jeep Compass

Demo: Table Basics

Create a users table

Describe the table

Drop a table

Insert the same row multiple times—no integrity constraints

Insert the same key with different values == update

Demo: Understanding Partitioning

Let's create a system_utilization table

Insert several rows

Query with the 'where' clause

Can't filter unless you're using the partition key

ALLOW FILTERING to the rescue?

ALLOW FILTERING FRIEND OF FOE?

“Cannot Execute This Query”

```
SELECT * FROM system_utilization WHERE cpu_pct = 5;
```

hostname	measured_on	cpu_pct
saturn	2018-07-19 09:00	90
saturn	2018-07-19 10:00	5
saturn	2018-07-19 11:00	10
venus	2018-07-19 09:00	5
venus	2018-07-19 10:00	0
venus	2018-07-19 11:00	15
mars	2018-07-19 09:00	5
mars	2018-07-19 10:00	50
mars	2018-07-19 11:00	75

Node 1

Node 2

Node 3

We can add “ALLOW FILTERING” to execute this, but don’t!

School of Information Studies
Syracuse University

Design Your Table With the Query Use Cases in Mind!

```
CREATE TABLE system_utilization2 (
      hostname TEXT,
      year INT,
      os TEXT,
      measured_on TIMESTAMP,
      cpu_pct TINYINT,
      PRIMARY KEY ((year, hostname), measured_on)
);
```

Redundant data, but necessary, as it allows us to partition on year and hostname, thereby limiting the amount of data returned by each node in the query, improving performance

```
SELECT * FROM System utilization where cpu_pct=5 and year=2018
ALLOW FILTERING;
```


School of Information Studies
Syracuse University

SECONDARY INDEXES

Secondary Indexes

Cassandra tables are designed to answer specific queries, but what if we want to answer additional queries? Secondary indexes!

They work similarly to RDMBS non-clustered indexes—indexes are distributed on each node.

They are useful on columns with low cardinality and whose values seldom change.

Covering a column with an index allows us to specify columns in the where clause without using ALLOW FILTERING.

```
CREATE INDEX index_name ON table(column);
```

Demo: Secondary Indexes

Query using the os column in the Where clause—requires ALLOW FILTERING

Create an index for the os column, so we may include it in queries

No longer requires ALLOW FILTERING

Show the index with DESCRIBE

MATERIALIZED VIEWS

Materialized Views

A materialized view is a Cassandra-managed table that is dependent on a base table.

Use it instead of an index when you want a different partition key.

Cassandra is responsible for making sure the data in the materialized view are in sync with their base table, eliminating the need for you to manage two separate tables.

Requirements are as follows:

- The primary key of the base table must be in the PK of the materialized view.
- Only one additional column may be added to the materialized view PK. Typically, this represents the new partition key.

Syntax of Materialized Views

```
CREATE MATERIALIZD VIEW view_name AS
      SELECT columns
      FROM base_table
      WHERE all_key_columns IS NOT NULL
      PRIMARY KEY
        (new_partition_key, original_key_cols_cluster_keys);
```

Demo: Materialized Views

Create MV system utilization by os.

Describe the table to see it.

You can't filter the table by the os column.

But you can filter the materialized view!
The new partition key is the os column.

Insert data to show that it works… one table!

SUMMARY: INDEXING OR MATERIALIZED VIEWS

Index or Materialized Views?

Use a **secondary index** when your query needs a different **cluster key**. Indexes are distributed.

Use a **materialized view** when your query needs a different **partition key**. Materialized views are shadow tables.

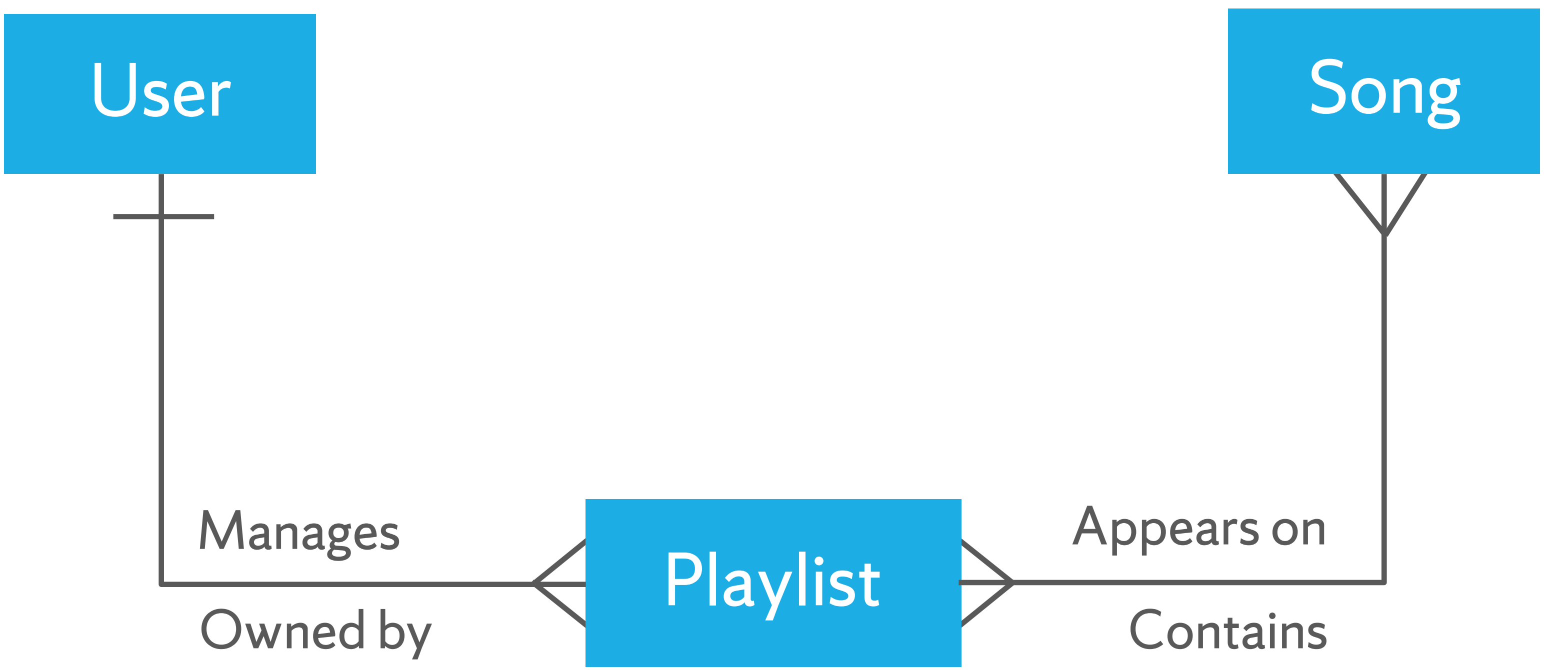
Rule of thumb: Each materialized view/index will add 10% more time to insert or update data.

DATA MODELING FOR CASSANDRA

Example: Music Playlist

Let's consider a Spotify-type application where users of the application can create playlists of songs.

Traditional ERD: Relational



Why Is Cassandra a Good Use Case for This?

Besides the fact that Spotify uses Cassandra for this exact purpose...

- ✓ We must guarantee writes—allow millions of users to manage their playlist.
- ✓ We must be able to read back the user's playlist immediately.
- ✓ Eventually, we will need information regarding who has which songs on their playlists across all users, but this is not an immediate need! Guaranteeing writes is far more important!

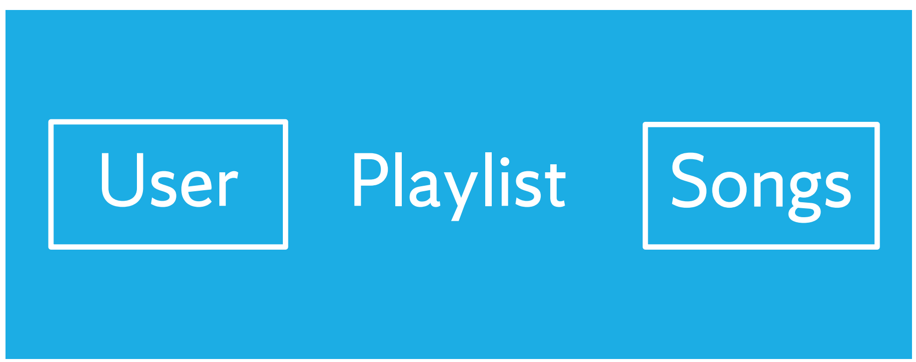
Queries Drive Design in Cassandra

We don't model entities or relationships here.

We model by processes and build the table as it would be queried!

We don't use many tables, but one "big table."

You cannot join or sort, so... de-normalize!



How Do You Manage a Playlist?

Partitioning

- It's just **you**
- You manage one playlist at a time

Clustered index

- Playlist songs are in any order we want

What do we need/need to show?

- User information
- Song information
- Playlist information
- All IDs to retrieve

The Cassandra Design

Columns

```
User ID (Partition Key)
UserInfo: {Email, Name}
Playlist ID (Partition Key)
Playlist Name
Song Order (Cluster Key)
Song ID
SongInfo: {Title, Artist}
Last_update
```

De-normalized design; data stored redundantly, but better I/O due to a flat table design

Partitioned by user and playlist (playlists are personal to a user)

Cluster key is Song Id (so the same song is not added twice)the playlist can be retrieved in the proper song order)

```
PRIMARY KEY ((user_id, playlist_id), song_id)
```

UPDATES AND DELETES

Updates in Cassandra

Cassandra has an UPDATE statement to manipulate a subset of existing columns.

No batch updates—you must include the primary key!

```
UPDATE table
     SET col = value
     WHERE key_col = key_value;
```

Deletes

You can delete an entire row, like in RDBMS:

`DELETE FROM` *`table`* `WHERE` *`key_col`* `=` *`value`*`;`

You can also delete a column from a row:

`DELETE` *`col`* `FROM` *`table`* `WHERE` *`key_col`* `=` *`value`*`;`

This is similar functionality to update set null in RDBMS.

Deleted rows are not actually deleted, but marked as deleted.

Tables are compacted at a later time.

Demo: Updates and Deletes

Alter table and add column

Update rows to include data

Query a multi-valued column with CONTAINS

Insert a row

Delete a column

Delete a row

?

DEMO: CONSISTENCY LEVELS

Consistency Levels

How many replicas must be in sync before the I/O operation (read/write) is complete?

Level	Replicas	Consistency	Availability
ALL	All	Highest	Lowest
ANY	Closest available	Lowest (write)	Highest (write)
ONE	A single available	Lowest (read)	Highest (read)
TWO	Two available		
QUORUM	Simple majority of all nodes		

https://docs.datastax.com/en/cql/3.3/cql/cql_reference/cqlshConsistency.html

Demo: Consistency Levels

Consistency command

Change a level

Run a command

Change it back