

Data Modelling in Cassandra

Anju Munoth

Infrastructure: a Node



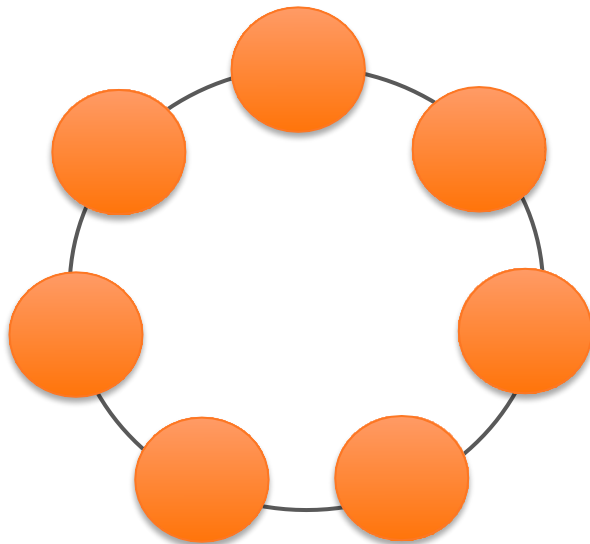
A single bare-metal server, a virtual instance or a docker container.



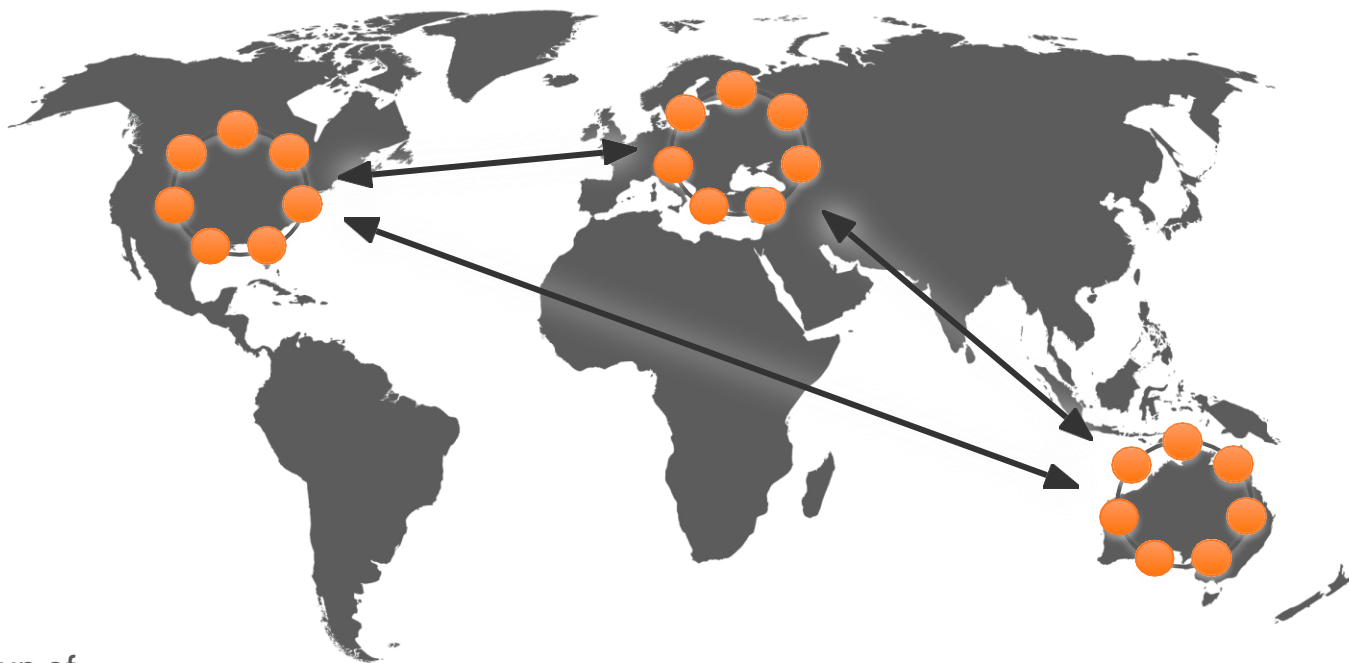
Infrastructure: a Datacenter (Ring)



A group of nodes located in the same physical location, a cloud datacenter or an availability zone.



Infrastructure: a Cluster



A group of datacenters configured to work together.

Data Structure: a Cell



An intersect of a row and a column, stores data.

John

Data Structure: a Row



A single, structured data item in a table.

1	John	Doe	Wizardry
---	------	-----	----------

Data Structure: a Partition



A group of rows having the same partition token, a base unit of access in Cassandra.

IMPORTANT: stored together, all the rows are guaranteed to be neighbours.

ID	First Name	Last Name	Department
1	John	Doe	Wizardry
399	Marisha	Chapez	Wizardry
415	Maximus	Flavius	Wizardry

Data Structure: a Table



A group of columns and rows storing partitions.

ID	First Name	Last Name	Department
1	John	Doe	Wizardry
2	Mary	Smith	Dark Magic
3	Patrick	McFadin	DevRel

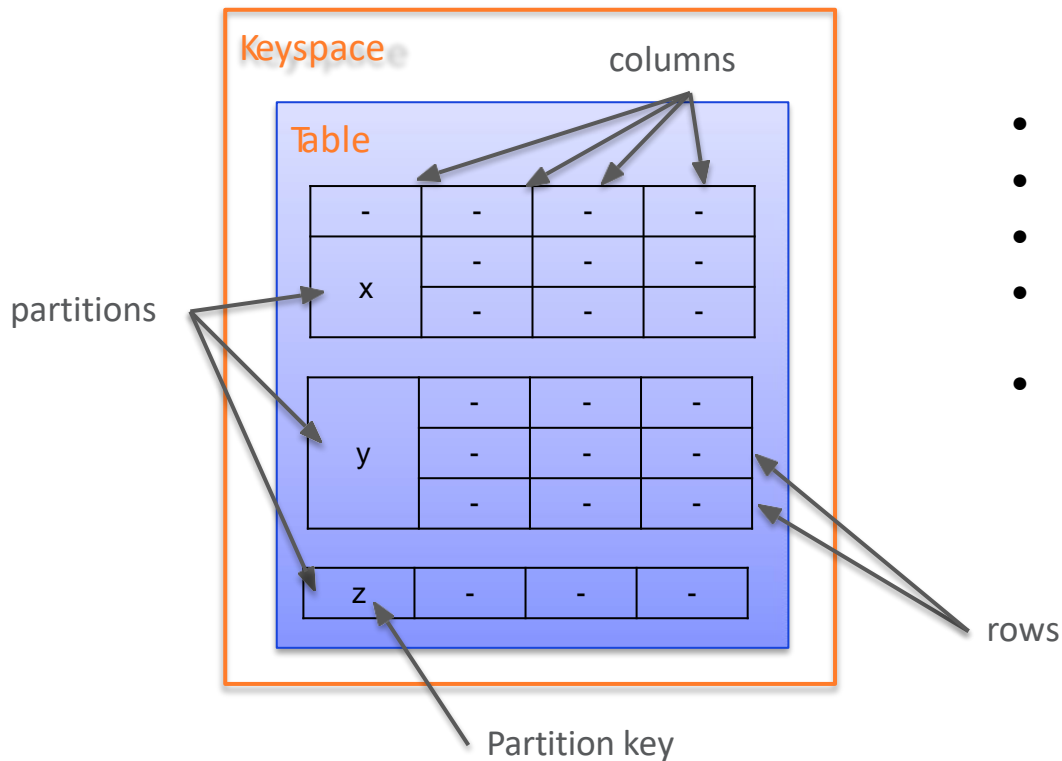
Data Structure: a Keyspace



A group of tables sharing replication strategy, replication factor and other properties

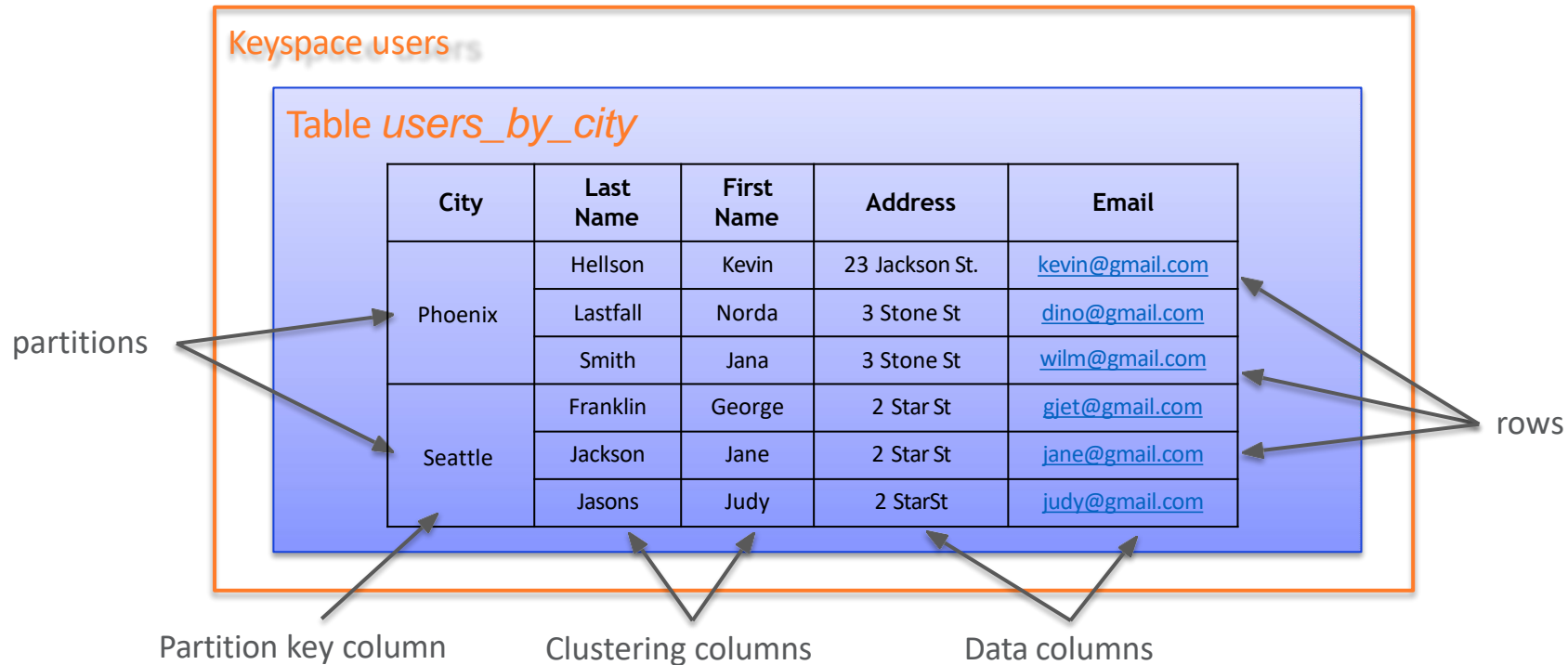
ID	First Name	Last Name	Department
1	John	Doe	Wizardry
2	Mary	Smith	Dark Magic
3	Patrick	McFadin	DevRel

Data Structure: Overall

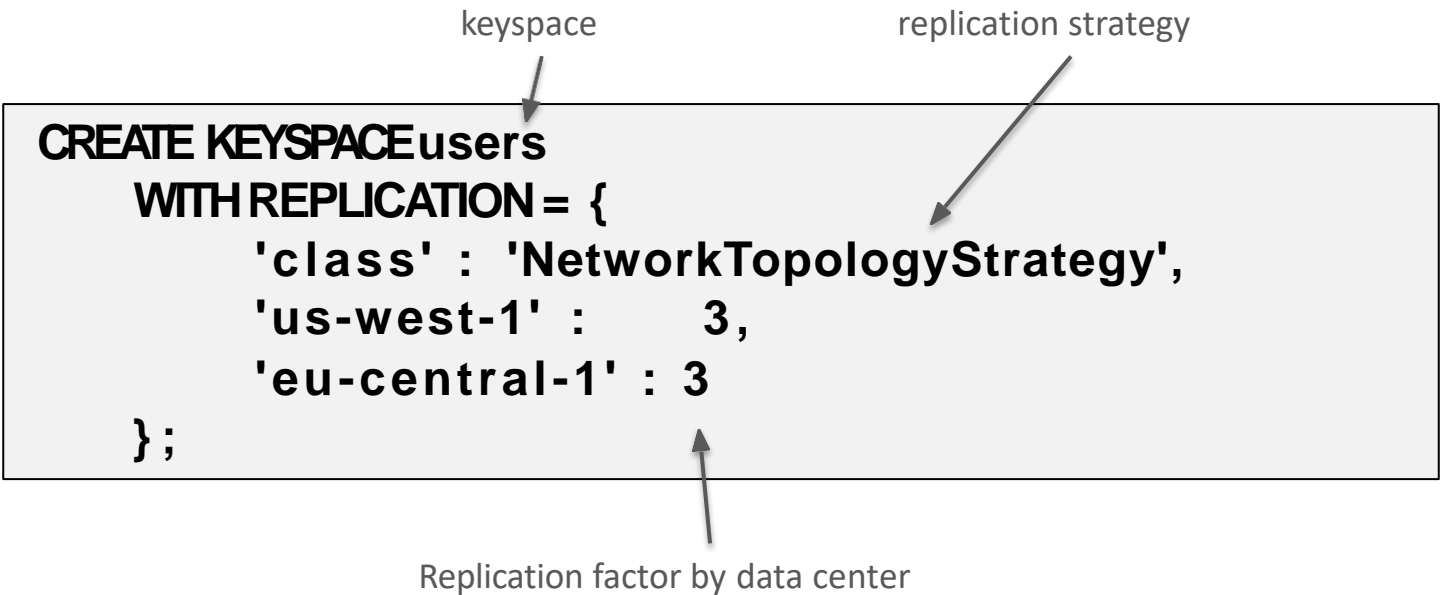


- Tabular data model, with one twist
- *Keyspaces* contain *tables*
- *Tables* are organized in *rows* and *columns*
- Groups of related rows called *partitions* are stored together on the same node (or nodes)
- Each row contains a *partition key*
 - One or more columns that are hashed to determine which node(s) store that data

Example Data: Users organized by city



Creating a Keyspace in CQL



The diagram illustrates the CQL command to create a keyspace named 'users' with a specific replication strategy. The code is enclosed in a light gray box. Three annotations with arrows point to parts of the code: 'keyspace' points to 'users', 'replication strategy' points to 'NetworkTopologyStrategy', and 'Replication factor by data center' points to the value '3' for the 'eu-central-1' data center.

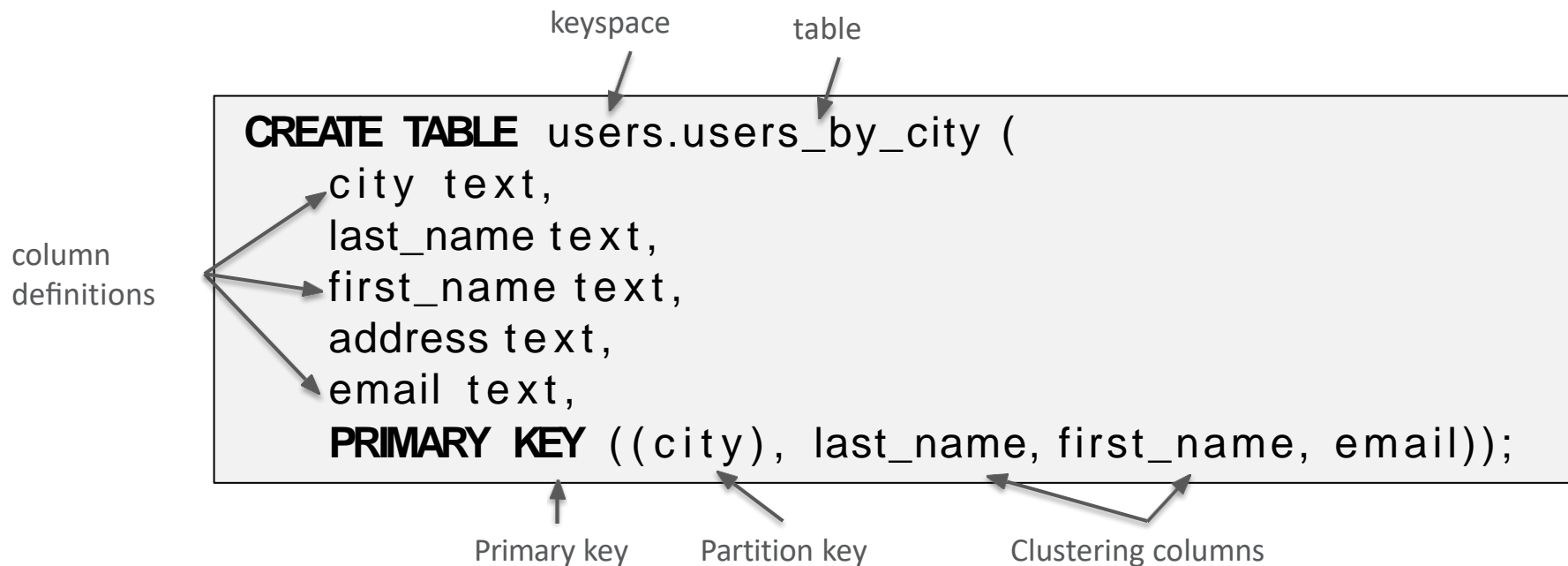
```
CREATE KEYSPACE users
  WITH REPLICATION = {
    'class' : 'NetworkTopologyStrategy',
    'us-west-1' : 3,
    'eu-central-1' : 3
  };
```

keyspace

replication strategy

Replication factor by data center

Creating a Table in CQL



Primary Key

An identifier for a row. Consists of at least one Partition Key and zero or more Clustering Columns.

**MUST ENSURE UNIQUENESS.
MAY DEFINE SORTING.**

Good Examples:

```
PRIMARY KEY((city), last_name, first_name, email);
```

```
PRIMARY KEY(user_id);
```

Bad Example:

```
PRIMARY KEY((city), last_name, first_name);
```

```
CREATE TABLE users.users_by_city (  
  city text,  
  last_name text,  
  first_name text,  
  address text,  
  email text,  
  PRIMARY KEY((city), last_name, first_name, email));
```

Partition key

Clustering columns

Partition Key

An identifier for a partition.
Consists of at least one column,
may have more if needed

PARTITIONS ROWS.

Good Examples:

```
PRIMARY KEY(user_id);
```

```
PRIMARY KEY((video_id), comment_id);
```

Bad Example:

```
PRIMARY KEY((sensor_id), logged_at);
```

```
CREATE TABLE users.users_by_city (  
  city text,  
  last_name text,  
  first_name text,  
  address text,  
  email text,  
  PRIMARY KEY((city), last_name, first_name, email));
```

Partition key

Clustering columns

Clustering Column(s)

Used to ensure uniqueness
and sorting order. Optional.

```
CREATE TABLE users.users_by_city (  
  city text,  
  last_name text,  
  first_name text,  
  address text,  
  email text,  
  PRIMARY KEY((city), last_name, first_name, email));
```

Partition key

Clustering columns

PRIMARY KEY((city), last_name, first_name);



Not Unique

PRIMARY KEY((city), last_name, first_name, email);



PRIMARY KEY((video_id), comment_id);



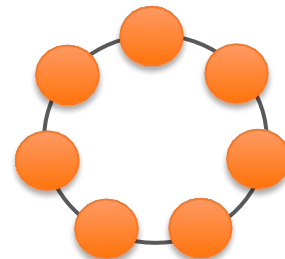
Not Sorted

PRIMARY KEY((video_id), created_at, comment_id);



Partition: The Beginning

```
CREATE TABLE users.users_by_city (  
  city text,  
  last_name text,  
  first_name text,  
  address text,  
  email text,  
  PRIMARY KEY((city), last_name, first_name, email));
```



- Every node is responsible for a range of tokens (0-100500, 100501-201000...)
- INSERT a new row, we get the value of its Partition Key (can't be null!)
- We hash this value using MurMur3 hasher <http://murmurhash.shorelabs.com/>
"Seattle" becomes 2466717130 **Partition Key** = Seattle, **Partition Token** = 2466717130
- This partition belongs to the node[s] responsible for this token
- The INSERT query goes to the nodes storing this partition (Notice Replication Factor)

Rules of a Good Partition

- Store together what you retrieve together
- Avoid big partitions
- Avoid hot partitions

PRIMARY KEY(user_id);



PRIMARY KEY((video_id), comment_id);



PRIMARY KEY((country), user_id);



Rules of a Good Partition

- **Store together what you retrieve together**
- Avoid big partitions
- Avoid hot partitions

Example: open a video? Get the comments in a single query!

```
PRIMARY KEY((video_id), created_at, comment_id);
```



```
PRIMARY KEY((comment_id), created_at);
```



Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big partitions**
- Avoid hot partitions

```
PRIMARY KEY((video_id), created_at, comment_id);
```



```
PRIMARY KEY((country), user_id);
```



- No technical limitations, but...
- Up to ~100k rows in a partition
- Up to ~100MB in a Partition

Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big partitions?**
- Avoid hot partitions

Example: a huge IoT infrastructure, hardware all over the world, different sensors reporting their state every 10 seconds. Every sensor reports its UUID, timestamp of the report, sensor's value.

```
PRIMARY KEY((sensor_id), reported_at);
```



The Rules of the Good Partition

- Sensor ID: UUID
- Timestamp: Timestamp
- Value: float

Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big and constantly growing partitions!**
- Avoid hot partitions

Example: a huge IoT infrastructure, hardware all over the world, different sensors reporting their state every 10 seconds. Every sensor reports its UUID, timestamp of the report, sensor's value.

The Rules of the Good Report

- Sensor ID: UUID
- Timestamp: Timestamp
- Value: float

```
PRIMARY KEY((sensor_id), reported_at);
```



Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big and constantly growing partitions!**
- Avoid hot partitions

Example: IoT infrastructure. Hardware all over the world, different sensors reporting their state every 10 seconds. Every sensor reports its UUID, timestamp of the report, sensor's value.

Sensor ID: UUID
Timestamp: float
Value: float

BUCKETING

```
PRIMARY KEY((sensor_id), reported_at);
```



```
PRIMARY KEY ((sensor_id, _____), reported_at);
```



Rules of a Good Partition

- Store together what you retrieve together
- **Avoid big and constantly growing partitions!**
- Avoid hot partitions

Example: a huge IoT infrastructure, hardware all over the world, different sensors reporting their state every 10 seconds. Every sensor reports its UUID, timestamp of the report, sensor's value.

PRIMARY KEY((sensor_id), reported_at);



PRIMARY KEY ((sensor_id, _____), reported_at);



PRIMARY KEY((sensor_id, month_year), reported_at);



BUCKETING

- Sensor ID: UUID
- **MonthYear**: Integer or String
- Timestamp: Timestamp
- Value: float

Rules of a Good Partition

- Store together what you retrieve together
- Avoid big partitions
- **Avoid hot partitions**

The Rules of the Good Partition

```
PRIMARY KEY(user_id);
```



```
PRIMARY KEY((video_id), created_at, comment_id);
```

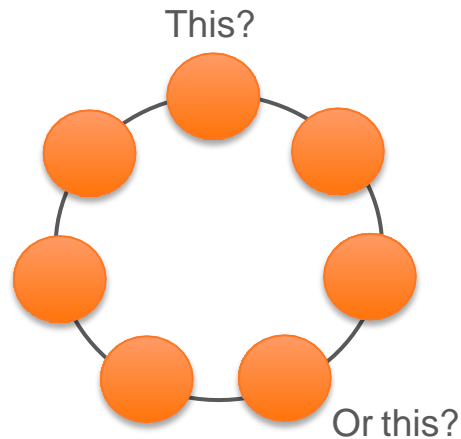


```
PRIMARY KEY((country), user_id);
```



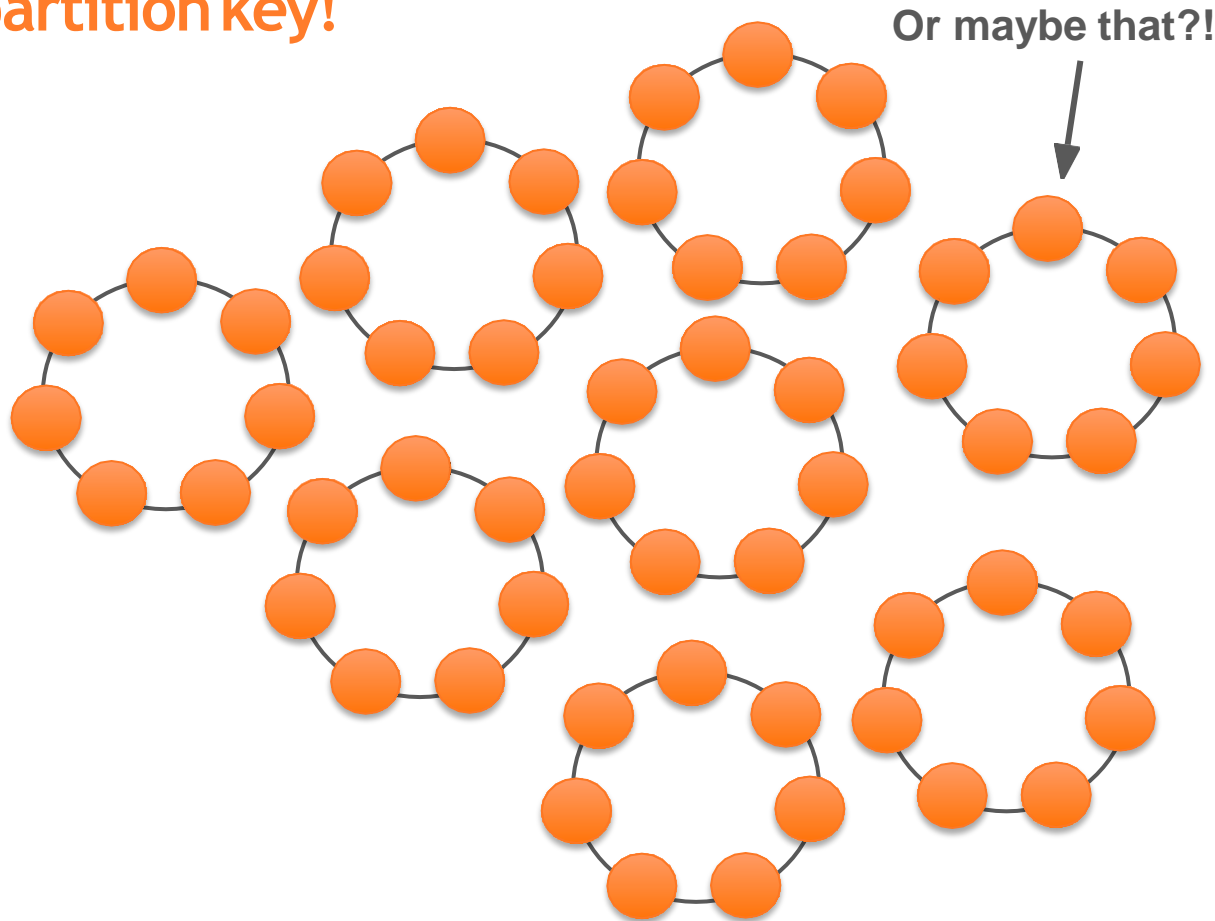
Always specify the partition key!

If there is no partition key in a query, which node you will ask?



Always specify the partition key!

If there is no partition key in a query, which node you will ask?



Always specify the partition key!

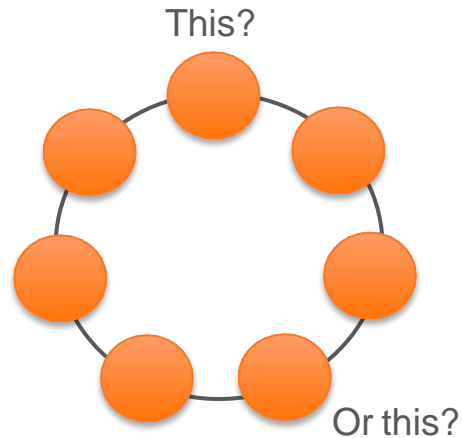
If there is no partition key in a query, which node you will ask?

```
CREATE TABLE users.users_by_city (  
  city text,  
  last_name text,  
  first_name text,  
  address text,  
  email text,  
  PRIMARY KEY((city), last_name, first_name, email));
```

```
SELECT address FROM users_by_city WHERE first_name = "Anna";
```

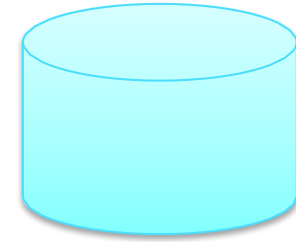
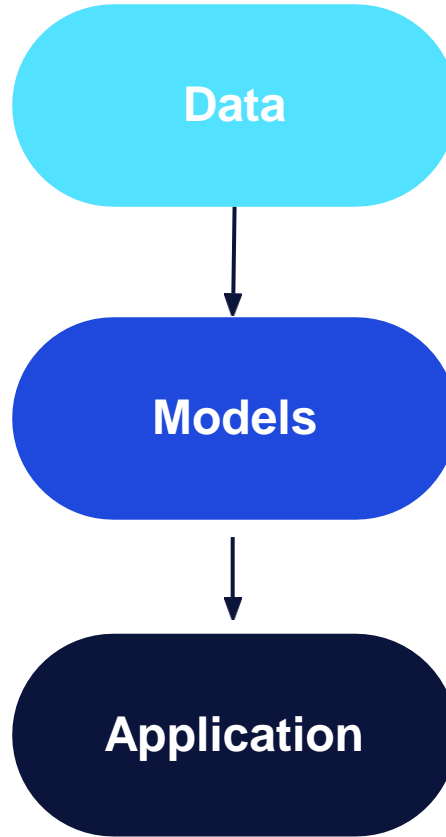


```
SELECT address FROM users_by_city WHERE city = "Otterberg" AND last_name = "Koshkina";
```



Relational Data Modelling

1. Analyze raw data
2. Identify entities, their properties and relations
3. Design tables, using normalization and foreign keys.
4. Use JOIN when doing queries to join denormalized data from multiple tables



Employees

userId	firstName	lastName
1	Edgar	Codd
2	Raymond	Boyce

Department

departmentId	department
1	Engineering
2	Math

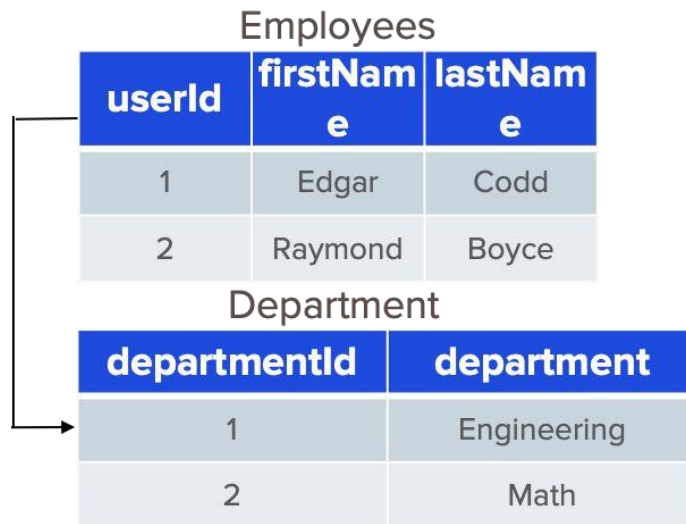


Normalization

“Database normalization is the process of structuring a relational database in accordance with a series of so-called normal forms in order to reduce data redundancy and improve data integrity. It was first proposed by Edgar F. Codd as part of his relational model.”

PROS: Simple write, Data Integrity

CONS: Slow read, Complex Queries



Denormalization

“Denormalization is a strategy used on a database to increase performance. In computing, denormalization is the process of trying to improve the read performance of a database, at the expense of losing some write performance, by adding redundant copies of data”

PROS: Quick Read, Simple Queries

CONS: Multiple Writes, Manual Integrity

Employees

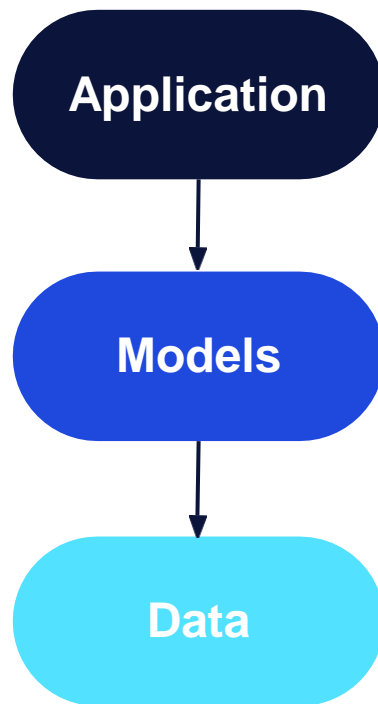
userId	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math

Department

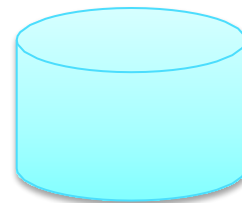
departmentId	department
1	Engineering
2	Math

NoSQL Data Modelling

1. Analyze user behaviour (customer first!)
2. Identify workflows, their dependencies and needs
3. Define Queries to fulfill these workflows
4. Knowing the queries, design tables, using denormalization.
5. Use BATCH when inserting or updating denormalized data of multiple tables



id	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math



Design Differences Between RDBMS and Cassandra

- No joins
- No referential integrity
- Denormalization
- Query-first design
- Designing for optimal storage
- Sorting is a design decision

Designing for optimal storage

- Cassandra tables are each stored in separate files on disk, it's important to keep related columns defined together in the same table.
- A key goal as you begin creating data models in Cassandra is to minimize the number of partitions that must be searched in order to satisfy a given query.
- Because the partition is a unit of storage that does not get divided across nodes, a query that searches a single partition will typically yield the best performance.

Wide Partition Pattern

- Group multiple related rows in a partition in order to support fast access to multiple rows within the partition in a single query.

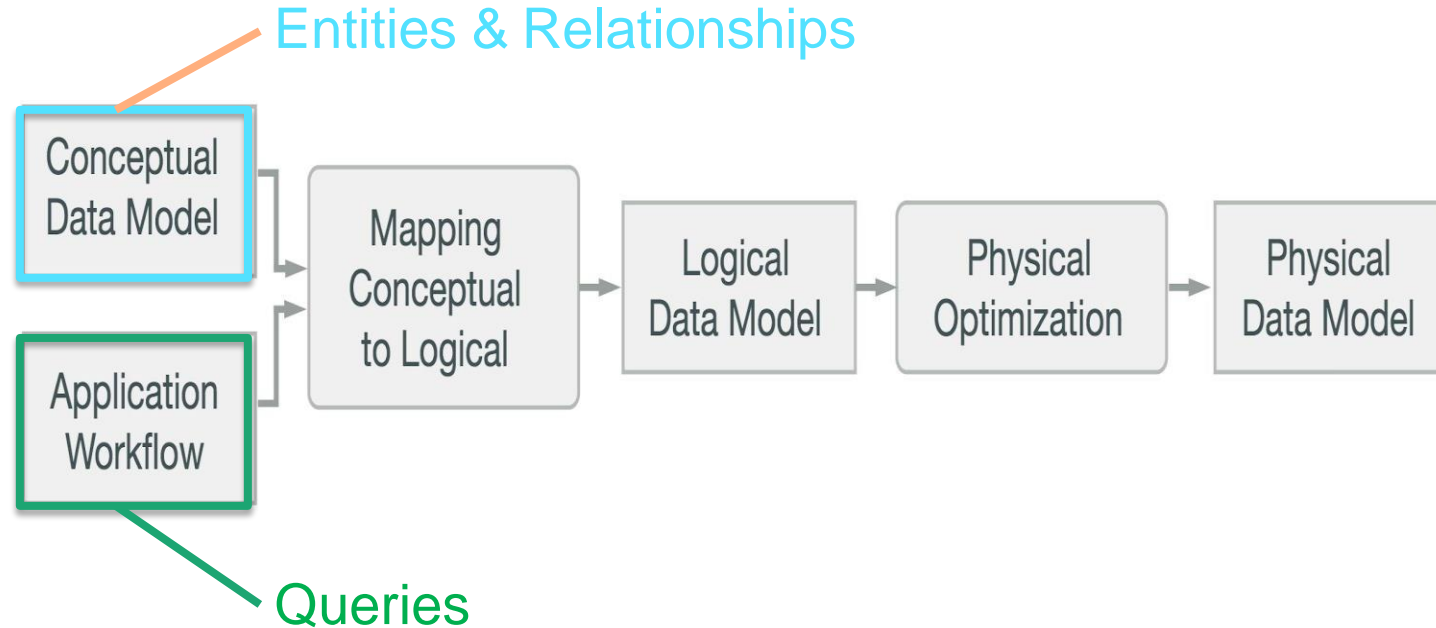
More Patterns and Anti-Patterns

- There are some well-known patterns and antipatterns for data modeling in Cassandra.
- One of the most common patterns -- *wide partition* pattern.
- *time series* pattern is an extension of the wide partition pattern.-series of measurements at specific time intervals are stored in a wide partition, where the measurement time is used as part of the partition key.
 - Pattern is frequently used in domains including business analysis, sensor data management, and scientific experiments.
 - The time series pattern is also useful for data other than measurements.
 - Consider the example of a banking application. You could store each customer's balance in a row, but that might lead to a lot of read and write contention as various customers check their balance or make transactions.
 - Probably be tempted to wrap a transaction around your writes just to protect the balance from being updated in error.
 - In contrast, a time series-style design would store each transaction as a timestamped row and leave the work of calculating the current balance to the application.

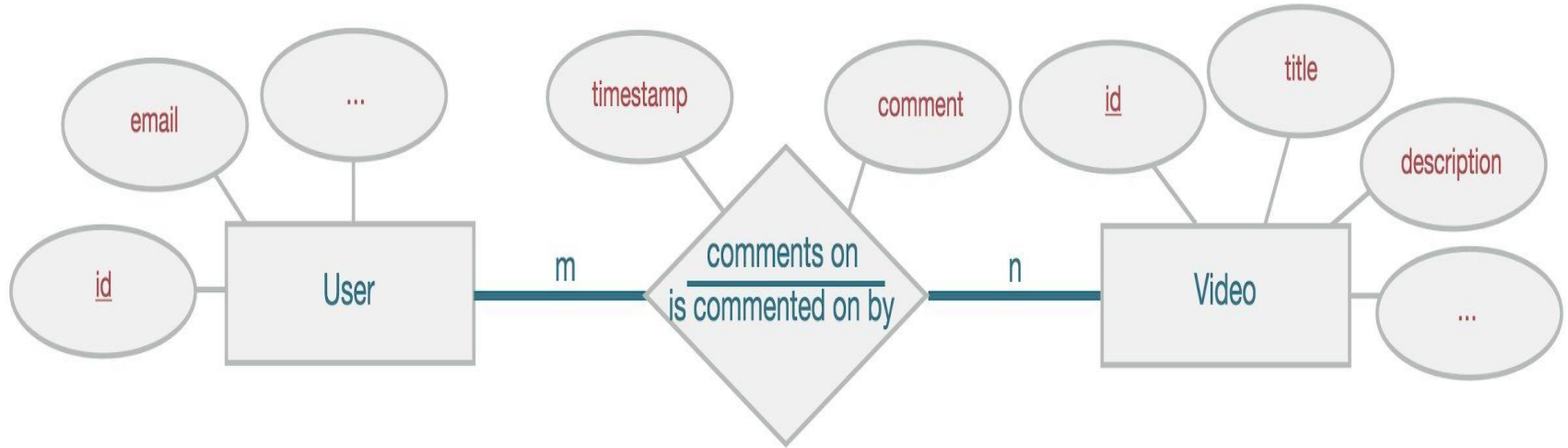
More Patterns and Anti-Patterns

- One design trap that many new users fall into is attempting to use Cassandra as a queue.
- Each item in the queue is stored with a timestamp in a wide partition. Items are appended to the end of the queue and read from the front, being deleted after they are read.
- This is a design that seems attractive, especially given its apparent similarity to the time series pattern.
- The problem with this approach is that the deleted items are now tombstones that Cassandra must scan past in order to read from the front of the queue.
- Over time, a growing number of tombstones begins to degrade read performance.
- The queue anti-pattern serves as a reminder that any design that relies on the deletion of data is potentially a poorly performing design.

Designing Process: Step by Step



Designing Process: Conceptual Data Model



Designing Process: Application Workflow



Use-Case I:

- A User opens a Video Page

WF1: Find **comments** related to target **video** using its identifier, most recent first

Use-Case II + III:

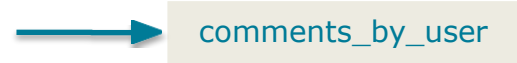
- A User opens a Profile
- A Moderator verifies a User if spammer or not

WF2: Find **comments** related to target **user** using its identifier, get most recent first

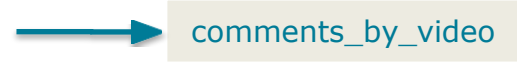
Designing Process: Mapping



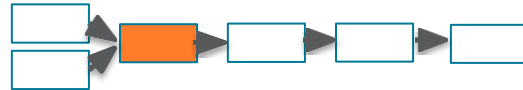
Query I: Find comments posted for a user with a known id (show most recent first)



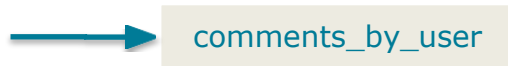
Query II: Find comments for a video with a known id (show most recent first)



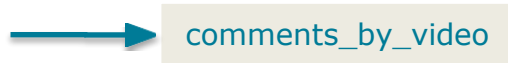
Designing Process: Mapping



```
SELECT * FROM comments_by_user  
WHERE userid = <some UUID>
```



```
SELECT * FROM comments_by_video  
WHERE videoid = <some UUID>
```



Designing Process: Logical Data Model



comments_by_user

userid	K
creationdate	C ↓
commentid	C ↑
videoid	
comment	

comments_by_video

videoid	K
creationdate	C ↓
commentid	C ↑
userid	
comment	

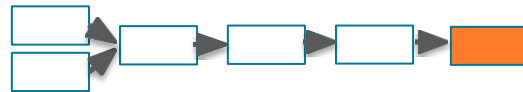
Designing Process: Physical Data Model



comments_by_user			
userid	UUID	K	
commentid	TIMEUUID	C	↓
videoid	UUID		
comment	TEXT		

comments_by_video			
videoid	UUID	K	
commentid	TIMEUUID	C	↓
userid	UUID		
comment	TEXT		

Designing Process: Schema DDL



```
CREATE TABLE IF NOT EXISTS comments_by_user (  
  userid uuid,    commentid timeuuid,  
  videoid uuid,   comment text,  
  PRIMARY KEY ((userid), commentid)  
) WITH CLUSTERING ORDER BY (commentid DESC);
```

```
CREATE TABLE IF NOT EXISTS comments_by_video (  
  videoid      uuid,  
  userid       uuid,  
  comment      text,  
  commentid    timeuuid,  
  
  PRIMARY KEY ((videoid), commentid)  
) WITH CLUSTERING ORDER BY (commentid DESC);
```

Basic Data Types

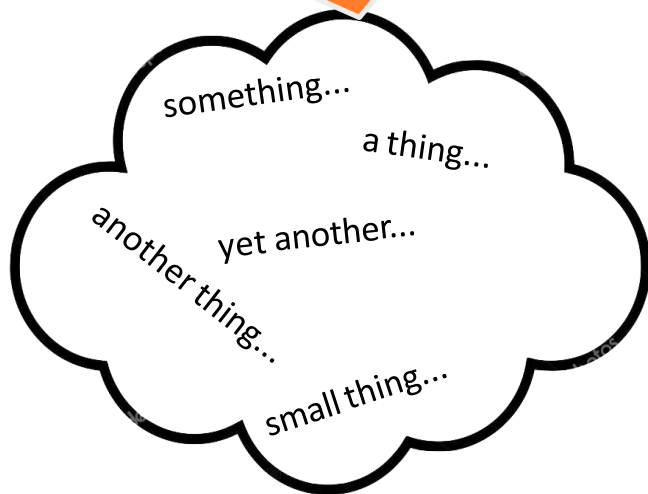
type	constants supported	description
<code>ascii</code>	<code>string</code>	ASCII character string
<code>bigint</code>	<code>integer</code>	64-bit signed long
<code>blob</code>	<code>blob</code>	Arbitrary bytes (no validation)
<code>boolean</code>	<code>boolean</code>	Either <code>true</code> or <code>false</code>
<code>counter</code>	<code>integer</code>	Counter column (64-bit signed value). See Counters for details
<code>date</code>	<code>integer</code> , <code>string</code>	A date (with no corresponding time value). See Working with dates below for details
<code>decimal</code>	<code>integer</code> , <code>float</code>	Variable-precision decimal
<code>double</code>	<code>integer</code> <code>float</code>	64-bit IEEE-754 floating point
<code>duration</code>	<code>duration</code> ,	A duration with nanosecond precision. See Working with durations below for details
<code>float</code>	<code>integer</code> , <code>float</code>	32-bit IEEE-754 floating point
<code>inet</code>	<code>string</code>	An IP address, either IPv4 (4 bytes long) or IPv6 (16 bytes long). Note that there is no <code>inet</code> constant, IP address should be input as strings
<code>int</code>	<code>integer</code>	32-bit signed int
<code>smallint</code>	<code>integer</code>	16-bit signed int
<code>text</code>	<code>string</code>	UTF8 encoded string
<code>time</code>	<code>integer</code> , <code>string</code>	A time (with no corresponding date value) with nanosecond precision. See Working with times below for details
<code>timestamp</code>	<code>integer</code> , <code>string</code>	A timestamp (date and time) with millisecond precision. See Working with timestamps below for details
<code>timeuuid</code>	<code>uuid</code>	Version 1 <code>UUID</code> , generally used as a “conflict-free” timestamp. Also see Timeuuid functions
<code>tinyint</code>	<code>integer</code>	8-bit signed int
<code>uuid</code>	<code>uuid</code>	A <code>UUID</code> (of any version)
<code>varchar</code>	<code>string</code>	UTF8 encoded string
<code>varint</code>	<code>integer</code>	Arbitrary-precision integer

uuid

- A universally unique identifier (UUID) is a 128-bit value in which the bits conform to one of several types, of which the most commonly used are known as Type 1 and Type 4.
- The CQL uuid type is a Type 4 UUID, which is based entirely on random numbers.
- UUIDs are typically represented as dash-separated sequences of hex digits.
- For example: 1a6300ca-0572-4736-a393-c0b7229e193e
- The uuid type is often used as a surrogate key, either by itself or in combination with other values.
- Because UUIDs are of a finite length, they are not absolutely guaranteed to be unique.
- However, most operating systems and programming languages provide utilities to generate IDs that provide adequate uniqueness.
- You can also obtain a Type 4 UUID value via the CQL uuid() function and use this value in an INSERT or UPDATE.

Collections

I'm a SET with a bunch
of unordered things



I'm an ordered LIST

0	1	2	3	4	6
---	---	---	---	---	---

I'm a MAP of
key/value pairs

Key	Value
K1	V1
K2	V2
K3	V3
K4	V4
K5	V5

set

- set data type stores a collection of elements.
- Elements are unordered when stored, but are returned in sorted order.
- For example, text values are returned in alphabetical order.
- Sets can contain the simple types, as well as user-defined types and even other collections.
- One advantage of using set is the ability to insert additional items without having to read the contents first.

Collection: Set

```
CREATE killrvideo.videos (  
  videoid          uuid,  
  userid           uuid,  
  name             text,  
  description       text,  
  location         text,  
  location_type    int,  
  preview_image_location text,  
  tags             set<text>,  
  added_date       timestamp,  
  PRIMARY KEY (videoid)  
);
```

{'Family', 'Disney', 'Princess'}

{'Thriller', 'Short'}

{'Tragicomedy', 'Western'}

Collection: Set

```
INSERT INTO killrvideo.videos (videoid, tags)
VALUES(12345678-1234-1234-1234-123456789012,
{'Side-splitter', 'Short'});
```

Insert

```
UPDATE killrvideo.videos
SET tags = {'Dark', 'Sad'}
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Replace entire set

```
UPDATE killrvideo.videos
SET tags = tags + {'Enthralling'}
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Add to set

list

- list data type contains an ordered list of elements.
- By default, the values are stored in order of insertion

Collection: List

```
CREATE killrvideo.actors_by_video (  
  videoid  uuid,  
  actors   list<text>, // alphabetical list of actors  
  PRIMARY KEY (videoid)  
);
```



Collection: List

```
INSERT INTO killrvideo.actors_by_video (videoid, actors)  
VALUES(12345678-1234-1234-1234-123456789012,  
['Adams', 'Baker', 'Cox']);
```

Insert

```
UPDATE killrvideo.actors_by_video  
SET actors = ['Arthur', 'Beverly']  
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Replace entire list

```
UPDATE killrvideo.actors_by_video  
SET actors=actors + ['Crawford']  
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Append

Collection: List

```
UPDATE killrvideo.actors_by_video
```

```
SET actors[1] = 'Brown'
```

```
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Replace an element

Note: replacing an element requires a read-before-write, which implies performance penalty.

Collection: List

- can delete a specific item directly using its index:

```
cqlsh:my_keyspace> DELETE phone_numbers[0] from user WHERE  
first_name = 'Mary' AND last_name = 'Rodriguez';
```


Expensive List Operations

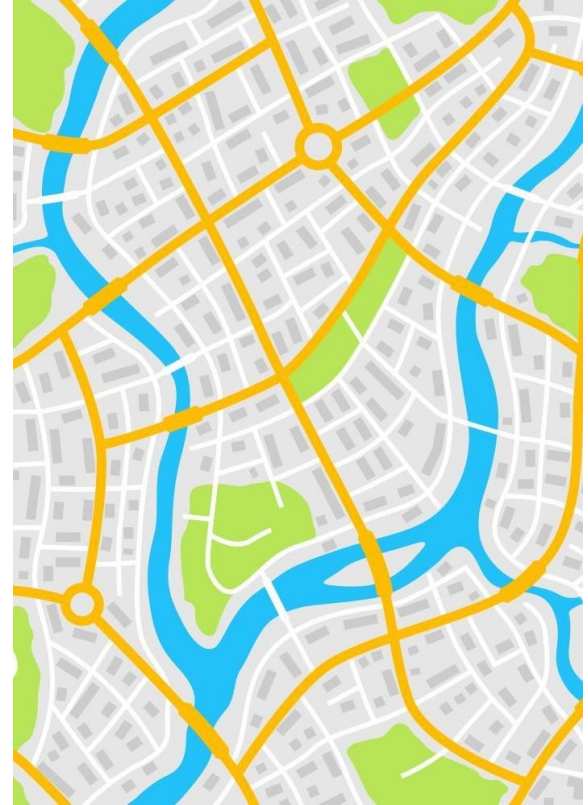
- Because a list stores values according to position, there is the potential that updating or deleting a specific item in a list could require Cassandra to read the entire list, perform the requested operation, and write out the entire list again.
- Could be an expensive operation if you have a large number of values in the list.
- For this reason, many users prefer to use the set or map types, especially in cases where there is the potential to update the contents of the collection.

map

- map data type contains a collection of key-value pairs.
- The keys and the values can be of any type except counter.
- can also reference an individual item in the map by using its key.

Collection: Map

```
CREATE TABLE killrvideo.users(  
  userid      uuid,  
  phone_nos   map<text, text>,  
  PRIMARY KEY (userid)  
);
```



Collection: Map

```
INSERT INTO killrvideo.users (userid, phone_nos)
VALUES(12345678-1234-1234-1234-123456789012,
{'cell':'867-5309', 'home':'555-1212',
'busi':'800-555-1212'});
```

Insert

```
UPDATE killrvideo.users
SET phone_nos = {'cell':'867-5310', 'office':'555-1212'}
WHERE userid = 12345678-1234-1234-1234-123456789012;
```

Replace entire map

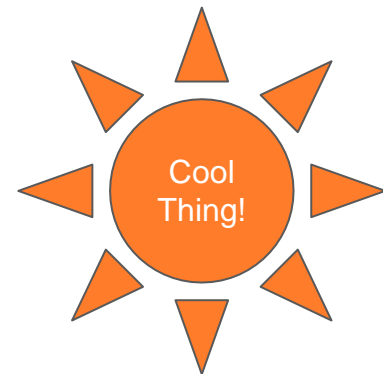
```
UPDATE killrvideo.users
SET phone_nos = phone_nos + {'desk': '270-555-1213'}
WHERE userid = 12345678-1234-1234-1234-123456789012;
```

Add to map

User Defined Types

```
CREATE TYPE killrvideo.address(  
  street text,  
  city   text,  
  state  text,  
);
```

```
CREATE TABLE killrvideo.users(  
  userid      uuid,  
  location    address,  
  PRIMARY KEY (userid)  
);
```



User Defined Types

```
INSERT INTO killrvideo.users (userid, location)  
VALUES(12345678-1234-1234-1234-123456789012,  
{street:'123 Main', city:'Metropolis', state:'CA'});
```

Insert

```
UPDATE killrvideo.users  
SET location = {street:'234 Elm', city:'NYC', state:'NY'}  
WHERE userid = 12345678-1234-1234-1234-123456789012;
```

Replace entire UDT

```
UPDATE killrvideo.users  
SET location.city = 'Albany'  
WHERE userid = 12345678-1234-1234-1234-123456789012;
```

Replace one UDT field

User Defined Types

```
SELECT location.city FROM killrvideo.users  
WHERE userid = 12345678-1234-1234-1234-123456789012;
```

Select field

Freezing Collections

- Cassandra releases prior to 2.2 do not fully support the nesting of collections.
- Specifically, the ability to access individual attributes of a nested collection is not yet supported, because the nested collection is serialized as a single object by the implementation.
- Therefore, the entire nested collection must be read and written in its entirety.
- Freezing is a concept that was introduced as a forward compatibility mechanism.
- For now, you can nest a collection within another collection by marking it as frozen, which means that Cassandra will store that value as a blob of binary data.
- In the future, when nested collections are fully supported, there will be a mechanism to
- “unfreeze” the nested collections, allowing the individual attributes to be accessed.
- You can also use a collection as a primary key if it is frozen.

Freezing Collections

```
cqlsh:my_keyspace> ALTER TABLE user ADD addresses map<text,  
frozen<address>>;
```

- Now let's add a home address for Mary:

```
cqlsh:my_keyspace> UPDATE user SET addresses = addresses +  
{ 'home': { street: '7712 E. Broadway', city: 'Tucson',  
state: 'AZ', zip_code: 85715 } }  
WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
```

```
cqlsh:my_keyspace> SELECT addresses FROM user WHERE first_name = 'Mary' AND  
last_name = 'Rodriguez';
```

Counters

- 64-bit signed integer
- Value cannot be set directly, but only incremented or decremented.
- Cassandra is one of the few databases that provides race-free increments across data centers.
- Counters are frequently used for tracking statistics such as numbers of page views, tweets, log messages, and so on.
- cannot be used as part of a primary key.
- If a counter is used, all of the columns other than primary key columns must be counters.

Counters

- Use-case:
 - Imprecise values such as likes, views, etc.
- Two operations:
 - Increment
 - Decrement
 - First op assumes the value is zero
- Cannot be part of primary key
- Counters not mixed with other types in table
- Value cannot be set
- Rows with counters cannot be inserted
- Updates are not idempotent
 - Counters should *not* be used for precise values

Counters

```
CREATE TABLE killrvideo.video_playback_stats (  
  videoid uuid,  
  views counter,  
  PRIMARY KEY (videoid)  
);
```

Counters

Incrementing a counter:

```
UPDATE killrvideo.videos SET views = views + 1  
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

This format must be
observed

This can be an
integer value

Decrementing a counter:

```
UPDATE killrvideo.videos SET views = views - 1  
WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Just change the
sign

Important note about tables

- In Cassandra, on one hand, a table is a set of rows containing values and, on the other hand, a table is also a set of partitions containing rows.
- Specifically, each row belongs to exactly one partition and each partition contains one or more rows.
- A primary key consists of a mandatory partition key and optional clustering key, where a partition key uniquely identifies a partition in a table and a clustering key uniquely identifies a row in a partition.
- A table with single-row partitions is a table where there is exactly one row per partition. A table with single-row partitions defines a primary key to be equivalent to a partition key.
- A table with multi-row partitions is a table where there can be one or more rows per partition.
- A table with multi-row partitions defines a primary key to be a combination of both partition and clustering keys.
- Rows in the same partition have the same partition key values and are ordered based on their clustering key values using the default ascendant order.

Indexing

- An index provides a means to access data in Cassandra using attributes other than the partition key.
- The benefit is fast, efficient lookup of data matching a given condition.
- The index indexes column values in a separate, hidden table from the one that contains the values being indexed.
- Cassandra has a number of techniques for guarding against the undesirable scenario where data might be incorrectly retrieved during a query involving indexes on the basis of stale values in the index.
- Indexes can be used for collections, collection columns, and any other columns except counter columns and static columns.

When to use an index

- Cassandra's built-in indexes are best on a table having many rows that contain the indexed value.
- The more unique values that exist in a particular column, the more overhead you will have, on average, to query and maintain the index.
- For example, suppose you had a races table with a billion entries for cyclists in hundreds of races and wanted to look up rank by the cyclist.
- Many cyclists' ranks will share the same column value for race year. The race_year column is a good candidate for an index.

When not to use an index

- Do not use an index in these situations:
- On high-cardinality columns for a query of a huge volume of records for a small number of results.
- In tables that use a counter column.
- On a frequently updated or deleted column.
- To look for a row in a large partition unless narrowly queried.

Problems using a high-cardinality column index

- If you create an index on a high-cardinality column, which has many distinct values, a query between the fields will incur many seeks for very few results.
- In the table with a billion songs, looking up songs by writer (a value that is typically unique for each song) instead of by their artist, is likely to be very inefficient.
- It would probably be more efficient to manually maintain the table as a form of an index instead of using the Cassandra built-in index.
- For columns containing unique data, it is sometimes fine performance-wise to use an index for convenience, as long as the query volume to the table having an indexed column is moderate and not under constant load.
- Conversely, creating an index on an extremely low-cardinality column, such as a boolean column, does not make sense.
- Each value in the index becomes a single row in the index, resulting in a huge row for all the false values, for example.
- Indexing a multitude of indexed columns having `foo = true` and `foo = false` is not useful.

Problems using an index on a frequently updated or deleted column

- Cassandra stores tombstones in the index until the tombstone limit reaches 100K cells.
- After exceeding the tombstone limit, the query that uses the indexed value will fail.

Problems using an index to look for a row in a large partition unless narrowly queried

- A query on an indexed column in a large cluster typically requires collating responses from multiple data partitions.
- The query response slows down as more machines are added to the cluster.
- You can avoid a performance hit when looking for a row in a large partition by narrowing the search.

Using a secondary index

- Using CQL, can create an index on a column after defining a table.
- Can also index a collection column.
- Secondary indexes are used to query a table using a column that is not normally queryable.
- Secondary indexes are tricky to use and can impact performance greatly.
- The index table is stored on each node in a cluster, so a query involving a secondary index can rapidly become a performance nightmare if multiple nodes are accessed.
- A general rule of thumb is to index a column with low cardinality of few values.
- Before creating an index, be aware of when and when not to create an index.

Secondary Indexes

- When you create a secondary index, Cassandra creates a new (hidden) table where the secondary becomes a primary key in this table.
- The visibility of this new table is in terms of a node, not a ring (cluster). That's the case of secondary indexes.
- general recommendation is to have at most one secondary index per table. And most tables do not need any.

- Each table only supports a limited set of queries based on its primary key definition.
- Additional queries can be supported by creating new tables with different primary keys, materialized views or secondary indexes.
- A secondary index can be created on a table column to enable querying data based on values stored in this column.
- Internally, a secondary index is represented by additional data structures that are created and automatically maintained on each cluster node

The table `rank_by_year_and_name` can yield the rank of cyclists for races.

```
cqlsh> CREATE TABLE cycling.rank_by_year_and_name (  
race_year int, race_name text, cyclist_name text, rank  
int, PRIMARY KEY ((race_year, race_name), rank) );
```

Both `race_year` and `race_name` must be specified as these columns comprise the partition key.

```
cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE  
race_year=2015 AND race_name='Tour of Japan - Stage 4 -  
Minami > Shinshu';
```


A logical query to try is a listing of the rankings for a particular year. Because the table has a composite partition key, this query will fail if only the first column is used in the conditional operator.

```
cqlsh> SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;
```

```
cqlsh:cycling> SELECT * from rank_by_year_and_name where race_year=2015;  
InvalidRequest: code=2200 [Invalid query] message="Partition key parts: race_name must be restricted as other parts are"
```

An index is created for the race year, and the query will succeed.

An index name is optional and must be unique within a keyspace.

If you do not provide a name, Cassandra will assign a name like race_year_idx.

```
cqlsh> CREATE INDEX ryear ON cycling.rank_by_year_and_name (race_year);  
SELECT * FROM cycling.rank_by_year_and_name WHERE race_year=2015;
```

A clustering column can also be used to create an index. An index is created on rank, and used in a query.

```
cqlsh> CREATE INDEX rrank ON cycling.rank_by_year_and_name (rank);  
SELECT * FROM cycling.rank_by_year_and_name WHERE rank = 1;
```

Types of secondary indexes:

- Regular secondary index (2i): a secondary index that uses hash tables to index data and supports equality (=) predicates.
- SSTable-attached secondary index (SASI): an experimental and more efficient secondary index that uses B+ trees to index data and can support equality (=), inequality (<, <=, >, >=) and even text pattern matching (LIKE).

Custom Indexes

- To create regular secondary indexes (2i) and SSTable-attached secondary indexes (SASI), Cassandra Query Language provides statements CREATE INDEX and CREATE CUSTOM INDEX, respectively, with the following simplified syntax:

```
CREATE INDEX [ IF NOT EXISTS ]
```

```
    index_name
```

```
ON [keyspace_name.] table_name ( column_name );
```

```
CREATE CUSTOM INDEX [ IF NOT EXISTS ]
```

```
    index_name
```

```
ON [keyspace_name.] table_name ( column_name )
```

```
USING 'org.apache.cassandra.index.sasi.SASIIndex'
```

```
[ WITH OPTIONS = { option_map } ];
```