Structuring Your Database

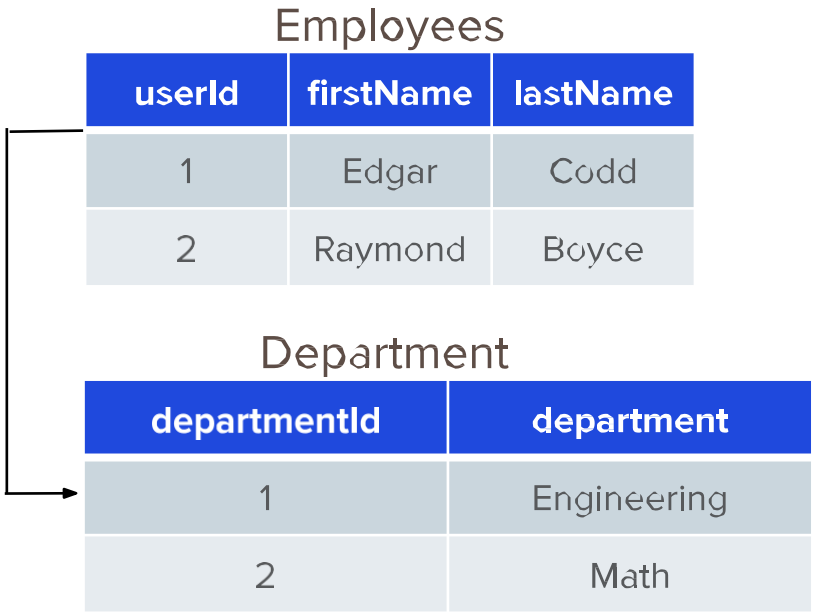
- Normalization: To reduce data redundancy and increase data integrity.
- Denormalization: Must be done in read heavy workloads to increase performance

Normalization

- Structuring a relational database
- Normal forms (3NF max)
- Why?
  - Reduce data redundancy
  - Increase data integrity.

Relational Data Models

- Multiple normal forms
  - most do not go beyond 3NF
- Foreign Keys
- Joins



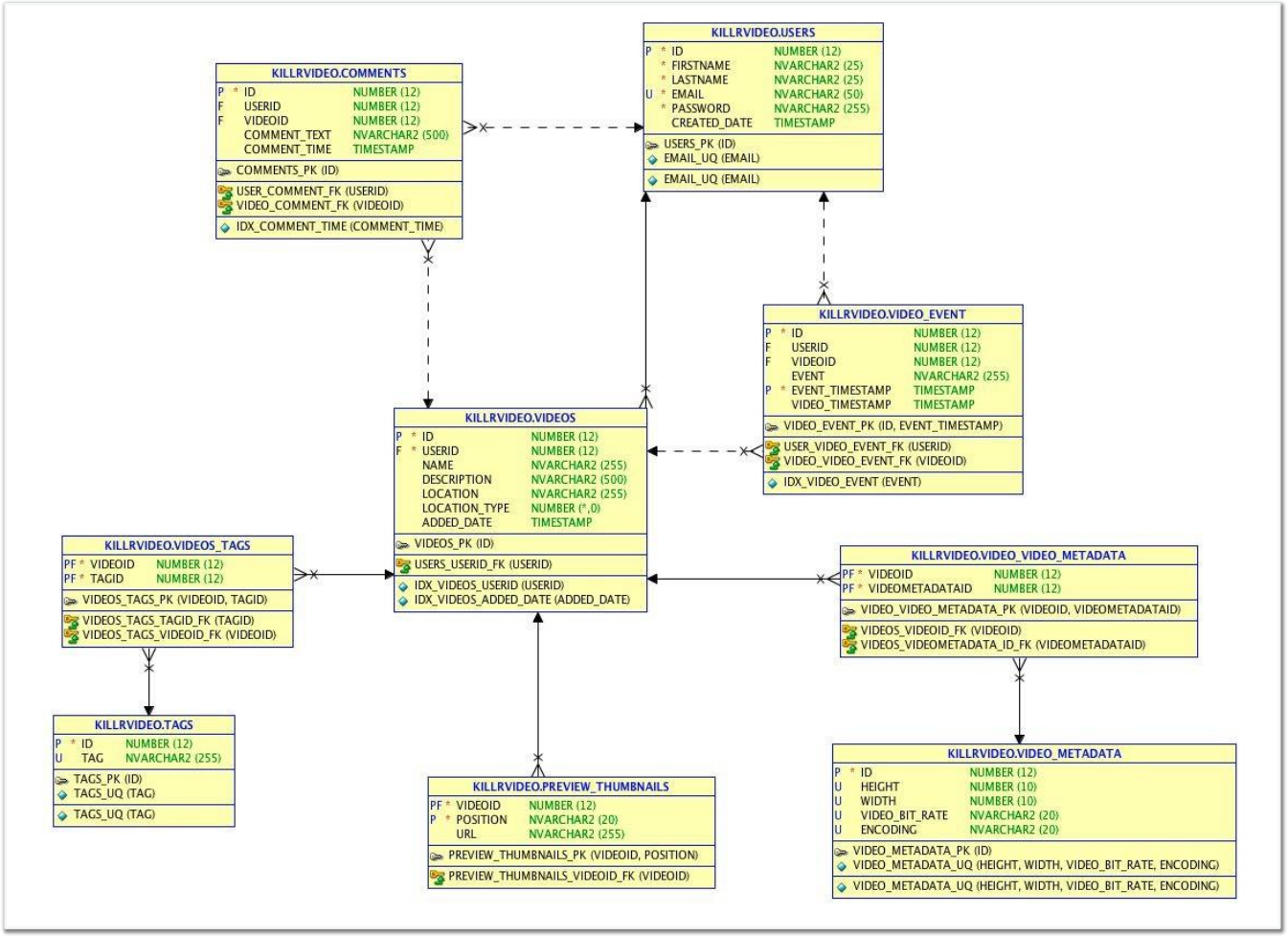
Relational Modeling

- Create entity table
- Add constraints
- Index fields
- Foreign Key relationships

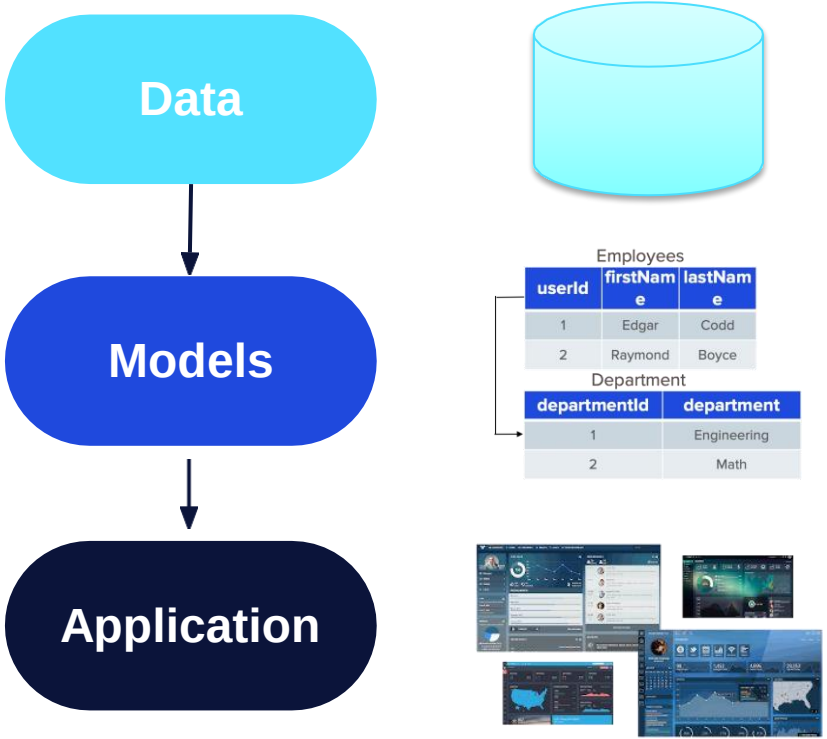
```
CREATE TABLE users (
  id          number(12) NOT NULL ,
  firstname    nvarchar2(25) NOT NULL ,
  lastname     nvarchar2(25) NOT NULL,
  email       nvarchar2(50) NOT NULL,
  password     nvarchar2(255) NOT NULL,
  created_date timestamp(6),
  PRIMARY KEY (id),
  CONSTRAINT email_uq UNIQUE (email)
);

-- Users by email address index
CREATE INDEX idx_users_email ON users (email);
```

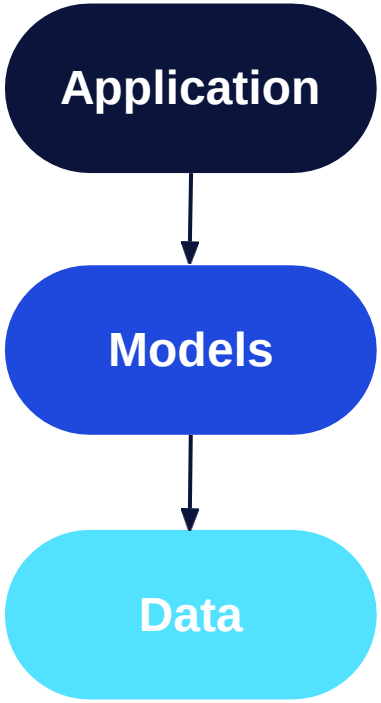
```
CREATE TABLE videos (
  id number(12),
  userid number(12) NOT NULL,
  name nvarchar2(255),
  description nvarchar2(500),
  location nvarchar2(255),
  location_type int,
  added_date timestamp,
  CONSTRAINT users_userid_fk
    FOREIGN KEY (userid)
    REFERENCES users (Id) ON DELETE CASCADE,
  PRIMARY KEY (id)
);
```


KILLRVIDEO.USERS
P * ID NUMBER (12)
* FIRSTNAME NVARCHAR2 (25)
* LASTNAME NVARCHAR2 (25)
U * EMAIL NVARCHAR2 (50)
* PASSWORD NVARCHAR2 (255)
CREATED_DATE TIMESTAMP
USERS_PK (ID)
EMAIL_UQ (EMAIL)
EMAIL_UQ (EMAIL)
KILLRVIDEO.COMMENTS
P * ID NUMBER (12)
F USERID NUMBER (12)
F VIDEOID NUMBER (12)
COMMENT_TEXT NVARCHAR2 (500)
COMMENT_TIME TIMESTAMP
COMMENTS_PK (ID)
USER_COMMENT_FK (USERID)
VIDEO_COMMENT_FK (VIDEOID)
IDX_COMMENT_TIME (COMMENT_TIME)
KILLRVIDEO.VIDEO_EVENT
P * ID NUMBER (12)
F USERID NUMBER (12)
F VIDEOID NUMBER (12)
EVENT NVARCHAR2 (255)
P * EVENT_TIMESTAMP TIMESTAMP
VIDEO_TIMESTAMP TIMESTAMP
VIDEO_EVENT_PK (ID, EVENT_TIMESTAMP)
USER_VIDEO_EVENT_FK (USERID)
VIDEO_VIDEO_EVENT_FK (VIDEOID)
IDX_VIDEO_EVENT (EVENT)
KILLRVIDEO.VIDEOS
P * ID NUMBER (12)
F * USERID NUMBER (12)
NAME NVARCHAR2 (255)
DESCRIPTION NVARCHAR2 (500)
LOCATION NVARCHAR2 (255)
LOCATION_TYPE NUMBER (*,0)
ADDED_DATE TIMESTAMP
VIDEOS_PK (ID)
USERS_USERID_FK (USERID)
IDX_VIDEOS_USERID (USERID)
IDX_VIDEOS_ADDED_DATE (ADDED_DATE)
KILLRVIDEO.VIDEOS_TAGS
PF * VIDEOID NUMBER (12)
PF * TAGID NUMBER (12)
VIDEOS_TAGS_PK (VIDEOID, TAGID)
VIDEOS_TAGS_TAGID_FK (TAGID)
VIDEOS_TAGS_VIDEOID_FK (VIDEOID)
KILLRVIDEO.VIDEO_VIDEO_METADATA
PF * VIDEOID NUMBER (12)
PF * VIDEOMETADATAID NUMBER (12)
VIDEO_VIDEO_METADATA_PK (VIDEOID, VIDEOMETADATAID)
VIDEOS_VIDEOID_FK (VIDEOID)
VIDEOS_VIDEOMETADATA_ID_FK (VIDEOMETADATAID)
KILLRVIDEO.TAGS
P * ID NUMBER (12)
U TAG NVARCHAR2 (255)
TAGS_PK (ID)
TAGS_UQ (TAG)
TAGS_UQ (TAG)
KILLRVIDEO.PREVIEW_THUMBNAILS
PF * VIDEOID NUMBER (12)
P * POSITION NVARCHAR2 (20)
URL NVARCHAR2 (255)
PREVIEW_THUMBNAILS_PK (VIDEOID, POSITION)
PREVIEW_THUMBNAILS_VIDEOID_FK (VIDEOID)
KILLRVIDEO.VIDEO_METADATA
P * ID NUMBER (12)
U HEIGHT NUMBER (10)
U WIDTH NUMBER (10)
U VIDEO_BIT_RATE NVARCHAR2 (20)
U ENCODING NVARCHAR2 (20)
VIDEO_METADATA_PK (ID)
VIDEO_METADATA_UQ (HEIGHT, WIDTH, VIDEO_BIT_RATE, ENCODING)
VIDEO_METADATA_UQ (HEIGHT, WIDTH, VIDEO_BIT_RATE, ENCODING)

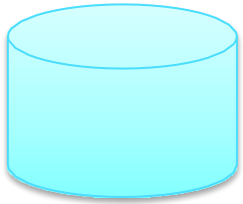
Relational Modeling



Cassandra Modeling



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



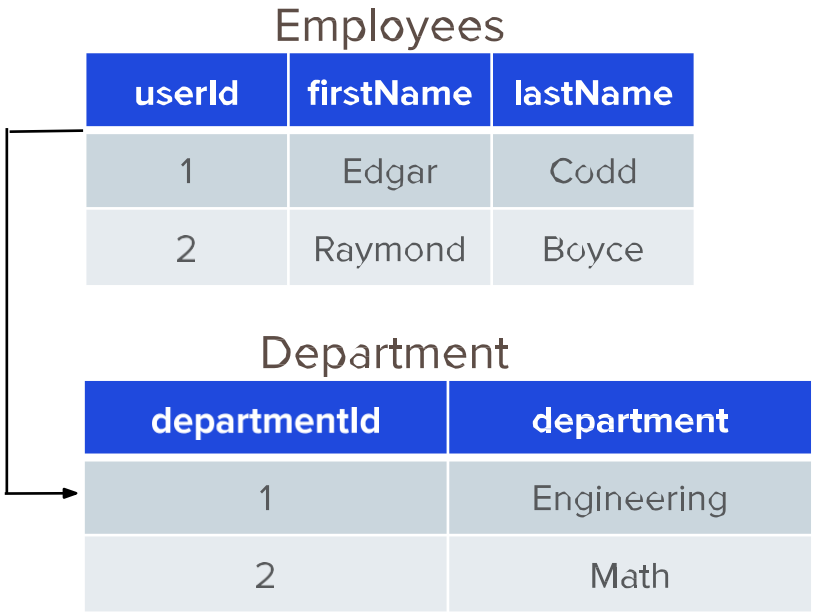
Denormalization - Why?

- Improve read performance of a database
- Reduce write performance
  - Adding redundant copies of data

CQL vs SQL

- No joins
- Limited aggregations

```
SELECT e.First, e.Last, d.Dept
FROM Department d, Employees e
WHERE 'Codd' = e.Last
AND e.deptId = d.id
```



Applying Denormalization

- Combine table columns into a single view
- Eliminate the need for joins
- Queries are concise and easy to understand

Employees

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

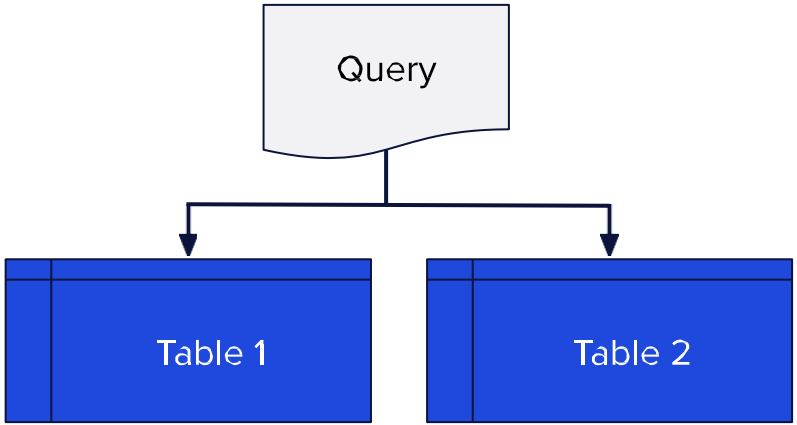
```
SELECT First, Last, Dept
FROM employees
WHERE id = '1'
```

Denormalization in Apache Cassandra

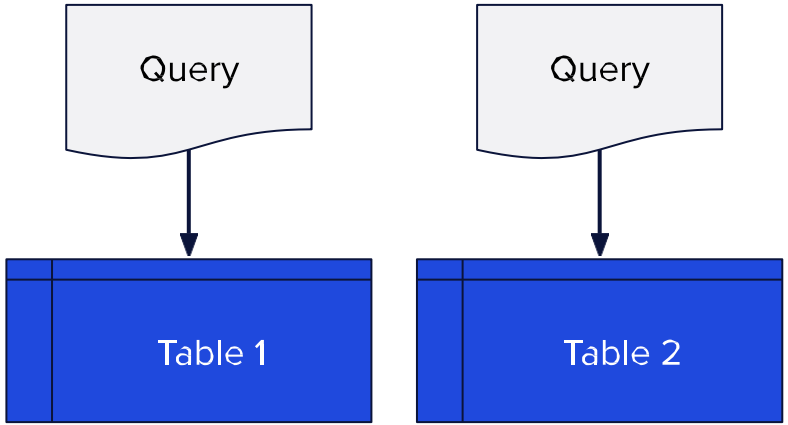
- Denormalization of tables in Apache Cassandra is absolutely critical.
- The biggest take away is to think about your queries first.
- There are no JOINS in Apache Cassandra.

Queries in Relational vs NoSQL Databases

- In a relational database, one query can access and join data from multiple tables



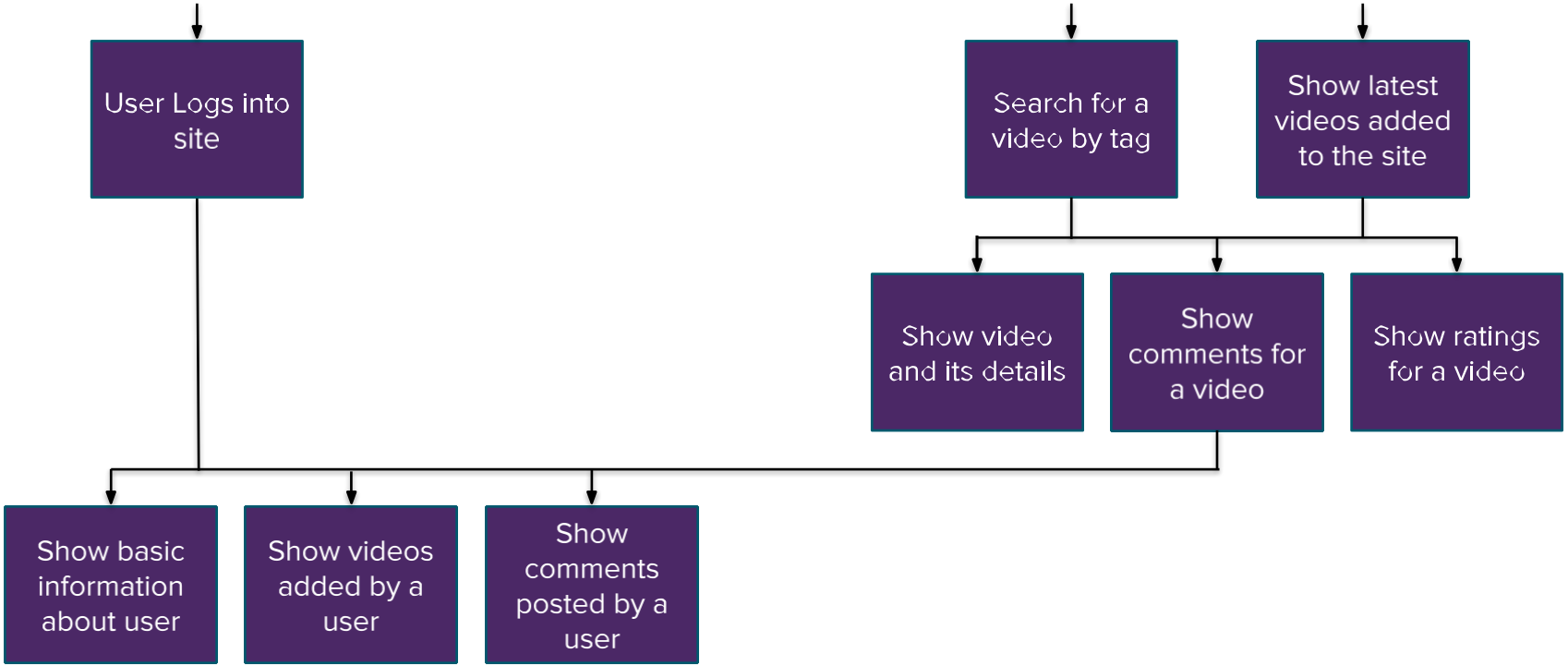
- In Apache Cassandra, you cannot join data, queries can only access data from one table



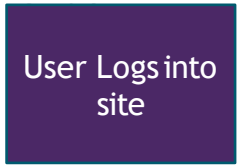
Modeling Queries

- What are your application's workflows?
- Knowing your queries in advance is CRITICAL
- Different from RDBMS because I can't just JOIN or create a new indexes to support new queries
- One table per one query

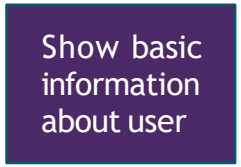
Some Application Workflows in KillrVideo



Us

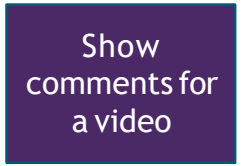


Find user by email address



Find user by id

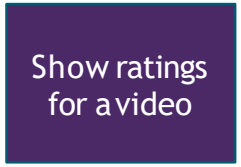
Comments



Find comments by video (latest first)

Find comments by user (latest first)

Ratings

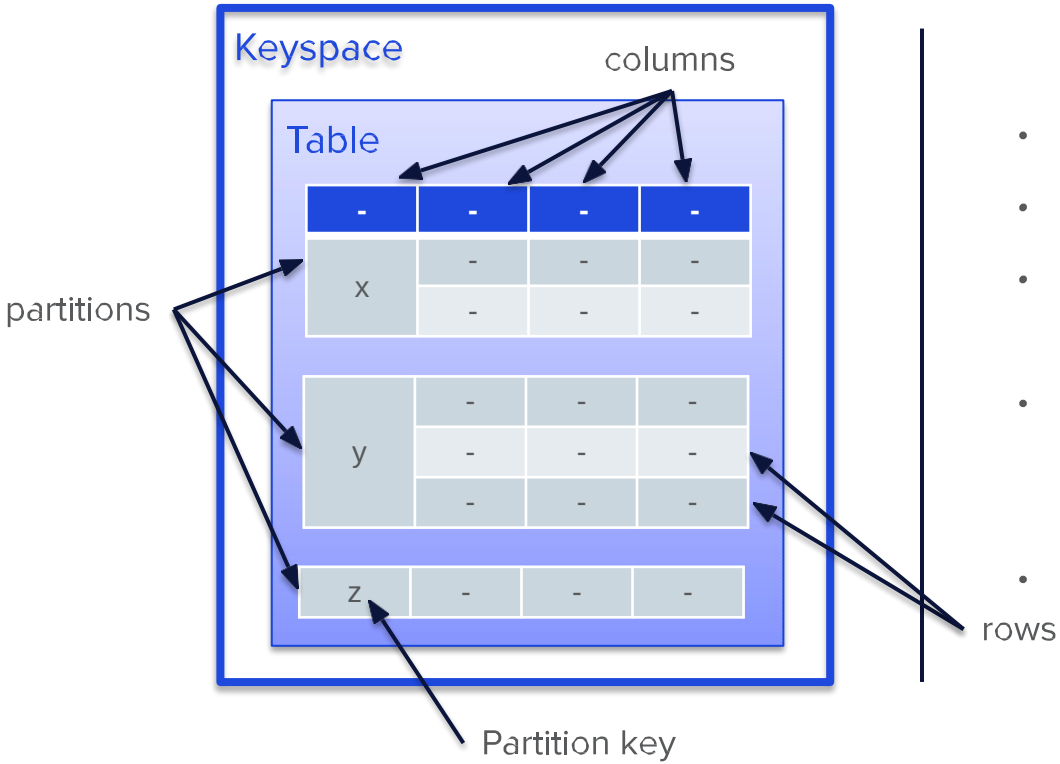


Find ratings by video

Cassandra Data Modeling Principles

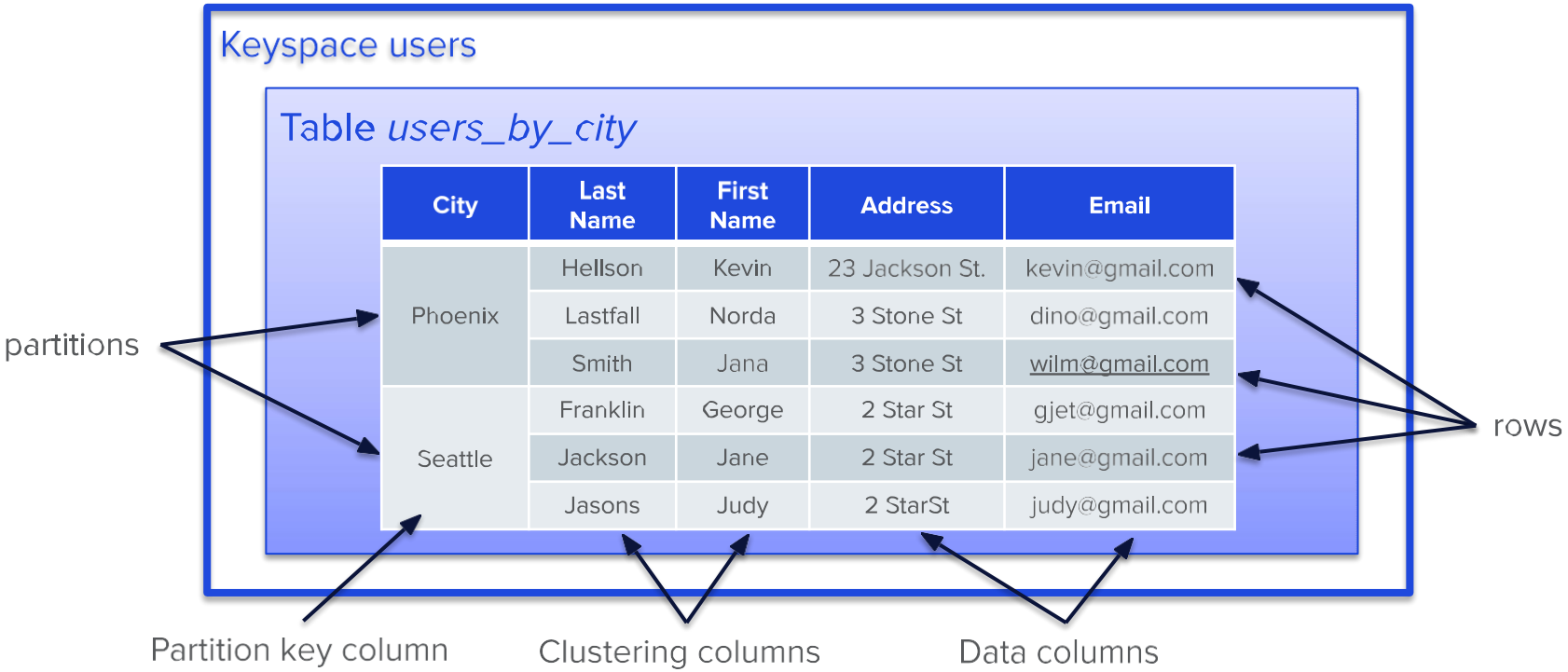
- Design tables around queries
- Use partition key column(s) to group data you would like to be able to get in a single query
- Use clustering columns to guarantee unique rows and control sort order
- Use additional columns to provide the details you need
  - Denormalization - including data that might have been joined from elsewhere in a relational model

Cassandra Structure - Partition



- 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



Tables Hold Many Partitions

City	Last Name	First Name	Address	Email
Phoenix	Hellson	Kevin	23 Jackson St.	kevin@gmail.com
	Lastfall	Norda	3 Stone St	dino@gmail.com
	Smith	Jana	3 Stone St	wilm@gmail.com

Table *users_by_city*

Tables Hold Many Partitions

City	Last Name	First Name	Address	Email
Seattle	Franklin	George	2 Star St	gjet@gmail.com
	Jackson	Jane	2 Star St	jane@gmail.com
	Jasons	Judy	2 StarSt	judy@gmail.com

Table *users_by_city*

City	Last Name	First Name	Address	Email
Phoenix	---	---	---	---
	---	---	---	---
	---	---	---	---

Tables Hold Many Partitions

City	Last Name	First Name	Address	Email
Charlotte	Azrael	Chris	5 Blue St	chris@gmail.com
	Stilson	Brainy	7 Azure In	brain@gmail.com
	Smith	Cristina	4 Teal Cir	clu@gmail.com
	Sage	Grant	9 Royal St	grant@gmail.com
	Seterson	Peter	2 Navy Ct	peter@gmail.com

Table *users_by_city*

City	Last Name	First Name	Address	Email
Phoenix	---	---	---	---
	---	---	---	---
	---	---	---	---
Seattle	---	---	---	---
	---	---	---	---
	---	---	---	---

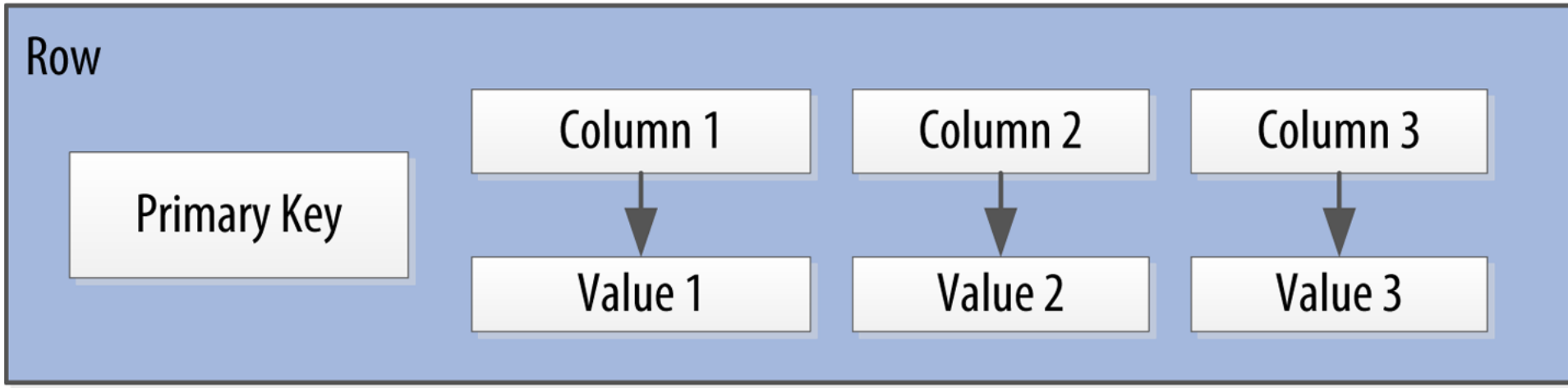
Tables Hold Many Partitions

Table *users_by_city*

City	Last Name	First Name	Address	Email
Phoenix	---	---	---	---
	---	---	---	---
	---	---	---	---
Seattle	---	---	---	---
	---	---	---	---
	---	---	---	---
Charlotte	---	---	---	---
	---	---	---	---
	---	---	---	---

Row

Each separate entity that holds some set of columns -rows.
Unique identifier for each row - row key or primary key.



Tables in Cassandra

- Cassandra defines a *table* to be a logical division that associates similar data.
- Cassandra table is analogous to a table in the relational world.
- Don’t need to store a value for every column every time you store a new entity.
- Maybe you don’t know the values for every column for a given entity.
- For example, some people have a second phone number and some don’t, and in an online form backed by Cassandra, there may be some fields that are optional and some that are required.
- That’s OK. Instead of storing null for those values you don’t know, which would waste space, just don’t store that column at all for that row.
- So now have a sparse, multidimensional array structure .
- This flexible data structure is characteristic of Cassandra and other databases classified as *wide column* stores.

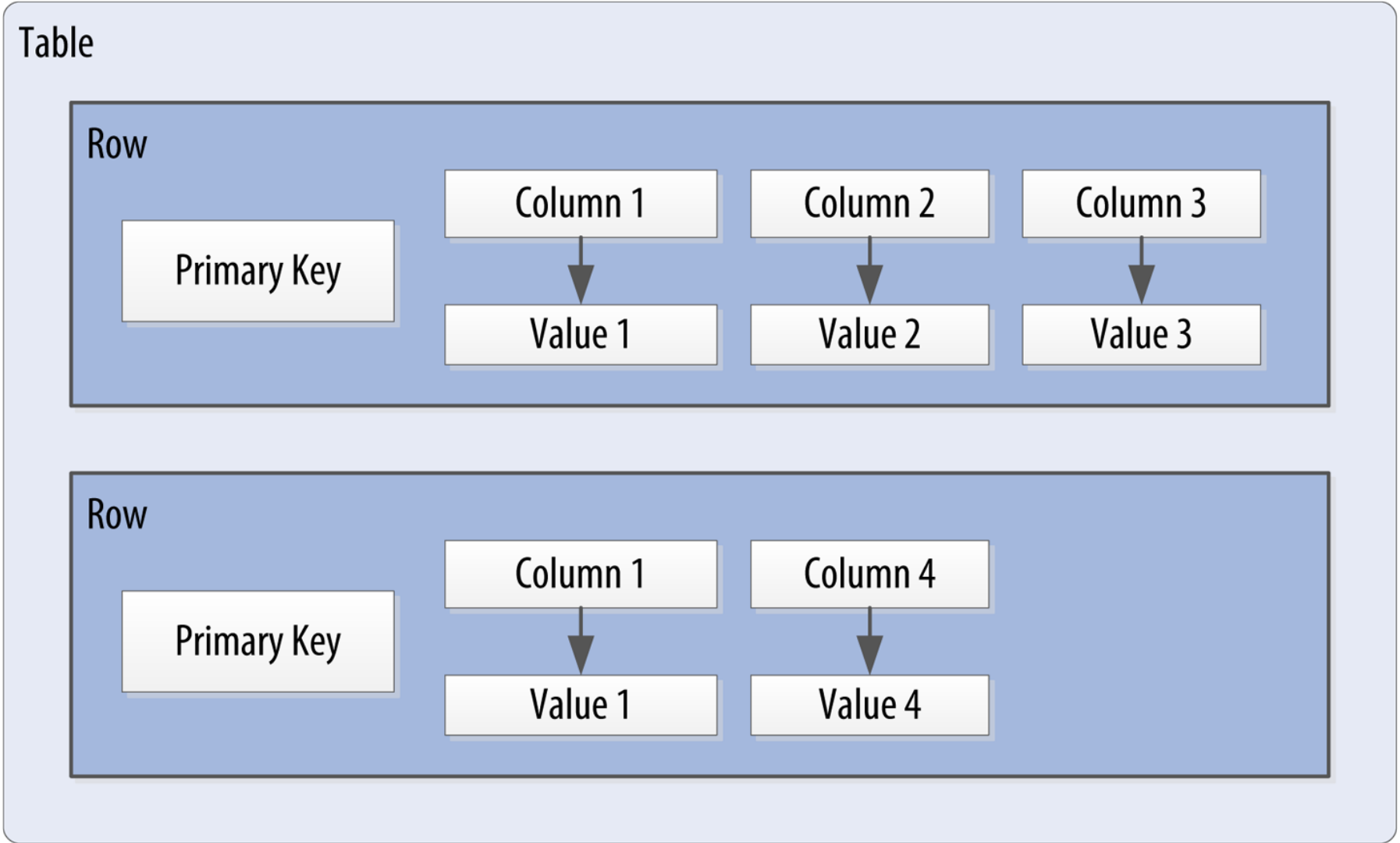
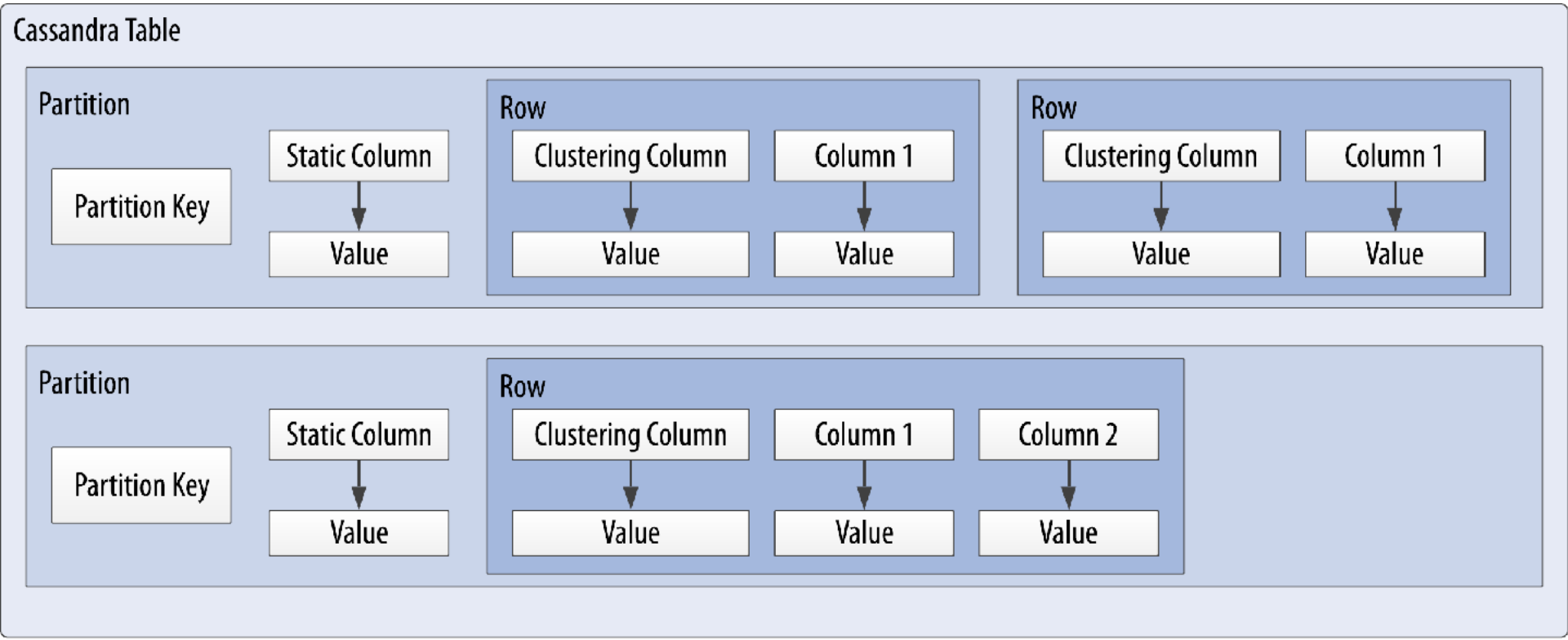


Table
Row
Primary Key
Column 1
Value 1
Column 2
Value 2
Column 3
Value 3
Row
Primary Key
Column 1
Value 1
Column 4
Value 4

Table

- Cassandra uses a special type of primary key called a composite key (or compound key) to represent groups of related rows, also called partitions.
- Composite key consists of a partition key, plus an optional set of clustering columns.
- Partition key is used to determine the nodes on which rows are stored and can itself consist of multiple columns.
- Clustering columns are used to control how data is sorted for storage within a partition.
- Cassandra also supports an additional construct called a static column, which is for storing data that is not part of the primary key but is shared by every row in a partition.


Cassandra Table
Partition
Partition Key
Static Column
Value
Row
Clustering Column
Value
Column 1
Value
Row
Clustering Column
Value
Column 1
Value
Partition
Partition Key
Static Column
Value
Row
Clustering Column
Value
Column 1
Value
Column 2
Value

Clusters

- Cassandra database is specifically designed to be distributed over several machines operating together that appear as a single instance to the end user.
- So the outermost structure in Cassandra is the cluster, sometimes called the ring, because Cassandra assigns data to nodes in the cluster by arranging them in a ring.

Keyspaces

- A cluster is a container for keyspaces. A keyspace is the outermost container for data in Cassandra, corresponding closely to a database in the relational model.
- In the same way that a database is a container for tables in the relational model, a keyspace is a container for tables in the Cassandra data model.
- Like a relational database, a key-space has a name and a set of attributes that define keyspace-wide behavior such as replication.

Timestamps

- Each time you write data into Cassandra, a timestamp, in microseconds, is generated for each column value that is inserted or updated.
- Internally, Cassandra uses these timestamps for resolving any conflicting changes that are made to the same value, in what is frequently referred to as a last write wins approach.
- there is no timestamp for a column that has not been set.
- Not allowed to ask for the timestamp on primary key columns

Timestamps

```
cqlsh:my_keyspace> SELECT first_name, last_name, title, writetime(title)
FROM user;

first_name | last_name | title | writetime(title)
------------+-----------+-------+-----------------
Mary | Rodriguez | null | null
Bill | Nguyen | Mr. | 1567876680189474
Wanda | Nguyen | Mrs. | 1567874109804754
(3 rows)
```

Timestamps

- Cassandra also allows you to specify a timestamp you want to use when performing writes.
- To do this, use the CQL UPDATE command for the first time.
- Use the optional USING TIMESTAMP option to manually set a timestamp (note that the timestamp must be later than the one from your SELECT command, or the UPDATE will be ignored):

➢ **cqlsh:my_keyspace> UPDATE user USING TIMESTAMP 1567886623298243**

SET middle_initial = 'Q' WHERE first_name = 'Mary' AND last_name = 'Rodriguez';

➢ **cqlsh:my_keyspace> SELECT first_name, middle_initial, last_name, WRITETIME(middle_initial) FROM user WHERE first_name = 'Mary' AND last_name = 'Rodriguez';**

first_name | middle_initial | last_name | writetime(middle_initial)

Mary | Q | Rodriguez | 1567886623298243

Time to live (TTL)

- One very powerful feature that Cassandra provides is the ability to expire data that is no longer needed.
- Expiration is very flexible and works at the level of individual column values.
- The time to live (or TTL) is a value that Cassandra stores for each column value to indicate how long to keep the value.
- The TTL value defaults to null, meaning that data that is written will not expire.
- Let's show this by adding the TTL() function to a SELECT command in cqlsh to see the TTL value for Mary's title:

```
cqlsh:my_keyspace> SELECT first_name, last_name, TTL(title)
FROM user WHERE first_name = 'Mary' AND last_name = 'Rodriguez';
first_name | last_name | ttl(title)
------------+-----------+------------
Mary | Rodriguez | null
```

Time to live (TTL)

- To set the TTL on the last name column to an hour (3,600 seconds) by adding the USING TTL option to your UPDATE command:

cqlsh:my_keyspace> UPDATE user USING TTL 3600 SET middle_initial =

'Z' WHERE first_name = 'Mary' AND last_name = 'Rodriguez';

cqlsh:my_keyspace> SELECT first_name, middle_initial,

last_name, TTL(middle_initial)

FROM user WHERE first_name = 'Mary' AND last_name = 'Rodriguez';

- first_name | middle_initial | last_name | ttl(middle_initial)
- ------------+----------------+-----------+--------------------
- Mary | Z | Rodriguez | 3574

Time to live (TTL)

- can also set TTL on INSERTS using the same USING TTL option, in which case the entire row will expire.
- Can try inserting a row using TTL of 60 seconds and check that the row is initially there:

cqlsh:my_keyspace> INSERT INTO user (first_name, last_name)

VALUES ('Jeff', 'Carpenter') USING TTL 60;

cqlsh:my_keyspace> SELECT * FROM user WHERE first_name='Jeff' AND

last_name='Carpenter';

last_name | first_name | middle_initial | title

- ----------+------------+---------------+-------
- Carpenter | Jeff | null | null

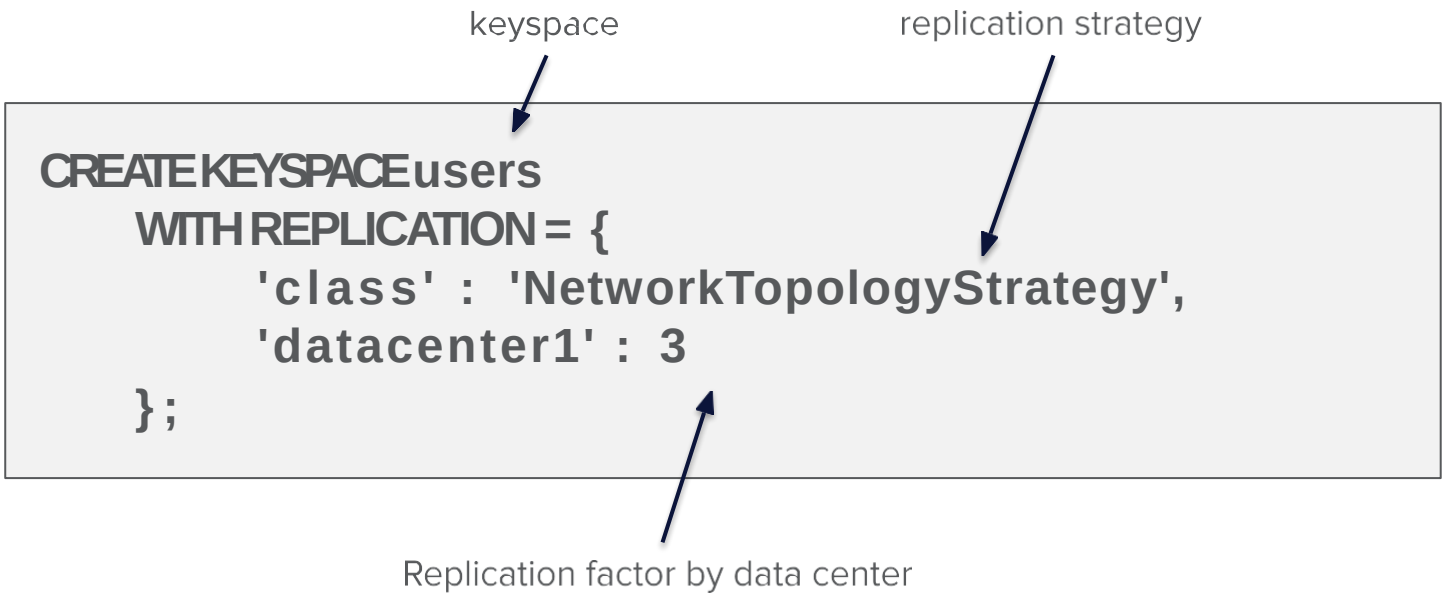
Using TTL

- Remember that TTL is stored on a per-column level for nonprimary key columns.
- Currently no mechanism for setting TTL at a row level directly after the initial insert;
- Would instead need to reinsert the row, taking advantage of Cassandra’s upsert behavior.
- As with the timestamp, there is no way to obtain or set the TTL value of a primary key column, and the TTL can only be set for a column when you provide a value for the column.

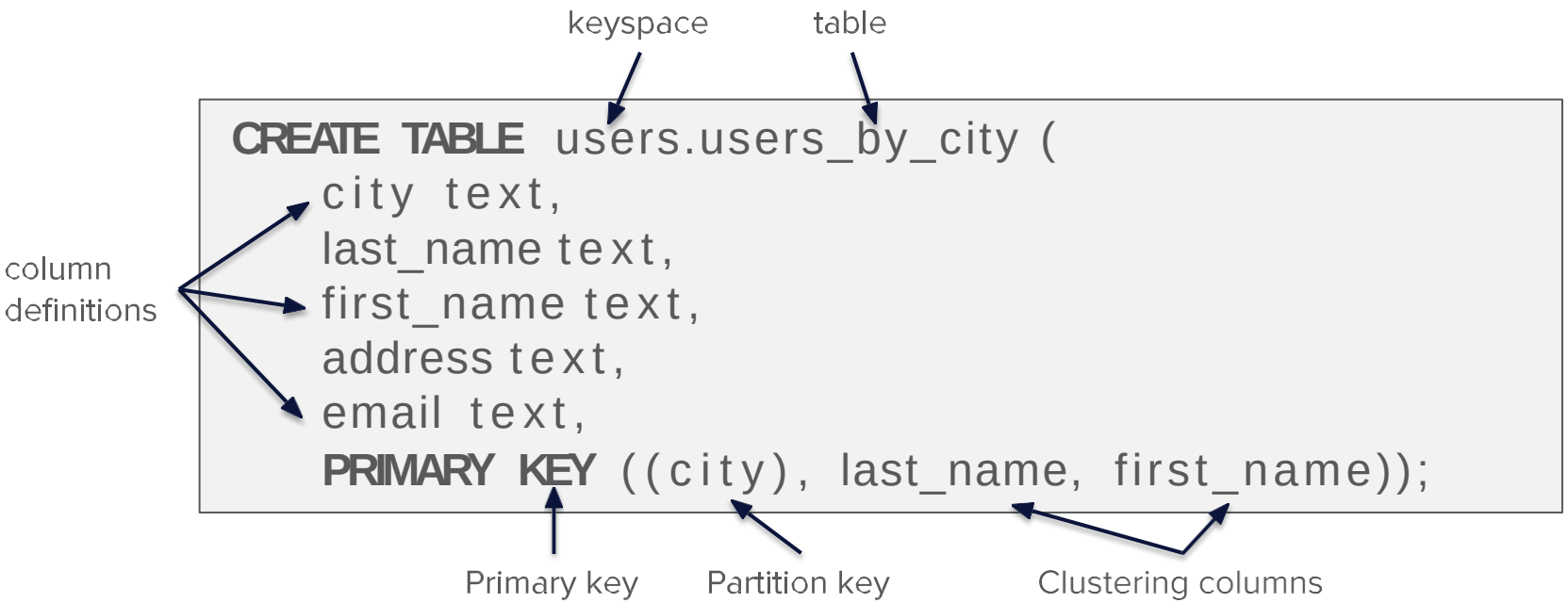
Summary

- Column, which is a name/value pair
- Row, which is a container for columns referenced by a primary key
- Partition, which is a group of related rows that are stored together on the same nodes
- Table, which is a container for rows organized by partitions
- Keyspace, which is a container for tables
- Cluster, which is a container for keyspaces that spans one or more nodes

Creating a Keyspace in CQL



Creating a Table in CQL



Data Modeling – Key Concepts

- Keyspace – contains tables
- Table – contains partitions
- Row – has a primary key and data columns
- Partition – basic unit of storage/retrieval
  - Identified by partition key embedded within primary key
  - Contains one or more rows
- Primary key – intra-table row identifier
  - Consists of partition key and clustering columns
  - Partition key – partition identifier, hashes to partition token
  - Clustering column – intra-partition key for sorting rows within partition

Cassandra-Land Use Cases

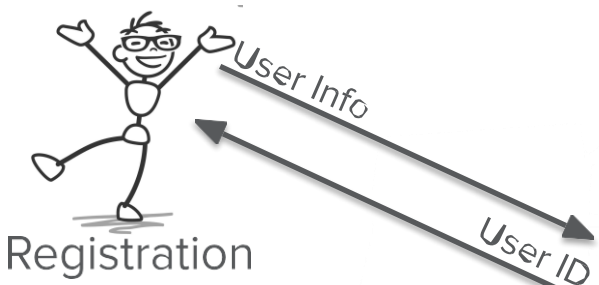
- Creating a Keyspace

```
CREATE KEYSPACE <keyspace name> WITH REPLICATION ={
    'class' : <replication strategy>,
   <datacenter name>: <replication factor>, ... };
```

- For example

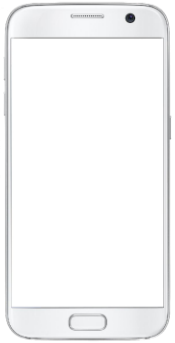
```
CREATE KEYSPACE park WITH REPLICATION ={
    'class' : NetworkTopologyStrategy,
   'USWestDC': 3, 'USEastDC': 3};
```

Cassandra-Land Registration Use Case



User Info

Field	Type
Phone number	text
Password	text



users_by_phone_number

Field	Type
phone_number **K**	text
user_id	UUID
password	text

User ID

Field	Type
User ID	UUID

Cassandra-Land Registration Use Case

- Creating a table

```
CREATE TABLE <keyspace name>.<table name> (
    <field name><field type>,
    // Addadditional field descriptions here
    PRIMARY KEY ( <primary key descriptor> )
);
```

Cassandra-Land Registration Use Case

- Inserting a row into a table

```
INSERT INTO <keyspace name>.<table name>
    ( <column list> )
    VALUES ( <column values> );
```

Cassandra-Land Registration Use Case

- Selecting all rows from a table
  - Typically wouldn't do this in production

```
SELECT * FROM <keyspace name>.<table name>;
```

Cassandra's Upsert Behavior

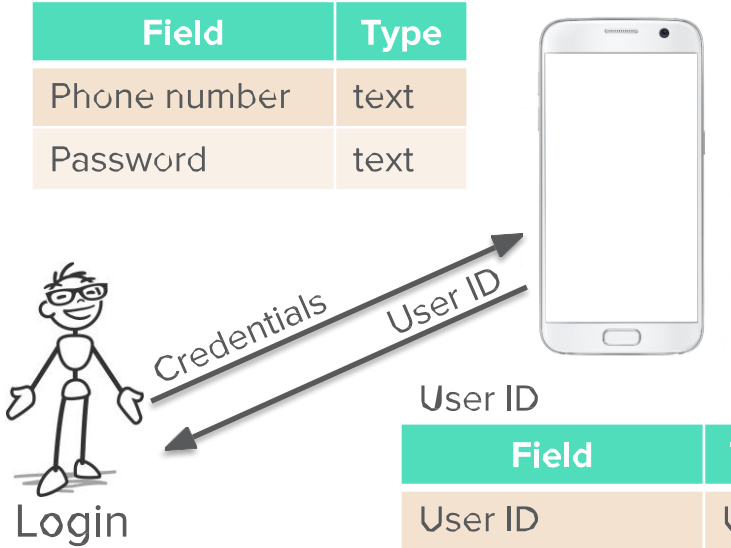
- Cassandra does NOT read before writing
- Inserting a row with the same primary key causes an update called an "upsert"
- Similarly, updates to non-existent rows cause an insert
  - Can use a lightweight transaction to prevent an upsert as it does perform a read before writing

```
INSERT INTO keyspace.table IF NOT EXISTS ...
```

Cassandra-Land Login Use Case

Credentials

Field	Type
Phone number	text
Password	text



User ID

Field	Type
User ID	UUID

users_by_phone_number

Field	Type
phone_number **K**	text
user_id	UUID
password	text

Cassandra-Land Login Use-Case

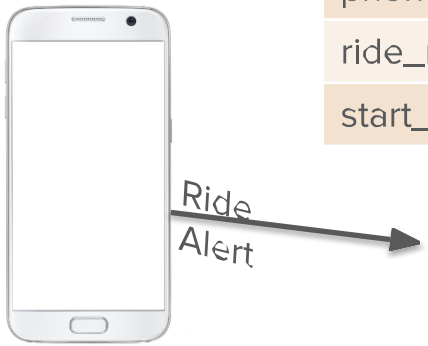
- Writing a SELECT statement
  - Must include full partition key
  - Partition keys do NOT support inequalities
  - Not all clustering columns need be specified, but…
  - Any preceding clustering columns MUST be specified

```
SELECT * FROM <keyspace name>.<table name>
    WHERE <query constraints>;
```

Cassandra-Land Ride Alert Use-Case

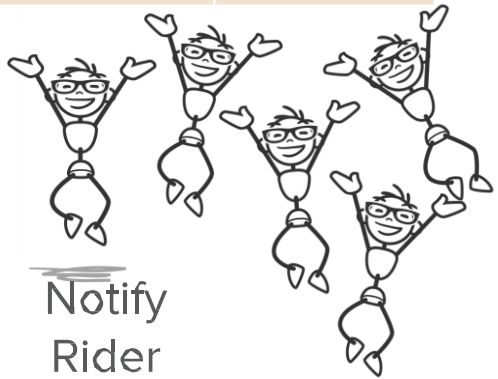
ride_instances_by_start_time

Field	Type
start_time **K**	timestamp
ride_id **C↑**	UUID
user_id **C↑**	UUID
ride_name	text
phone_number	text



Ride Alert

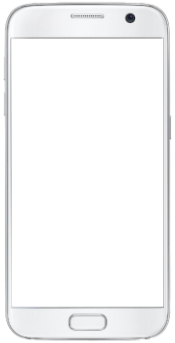
Field	Type
phone_number	text
ride_name	text
start_time	timestamp



Cassandra-Land View Schedule Use-Case

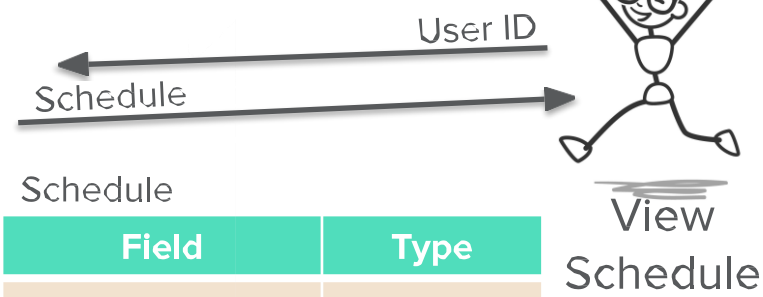
ride_instances_by_user_id

Field	Type
user_id **K**	UUID
start_time **C↑**	timestamp
ride_id	UUID
ride_name	text



User ID

Field	Type
User ID	UUID



Schedule

Field	Type
start_time	timestamp
ride_name	text

Cassandra-Land Schedule Ride Use-Case

ride_list_by_location

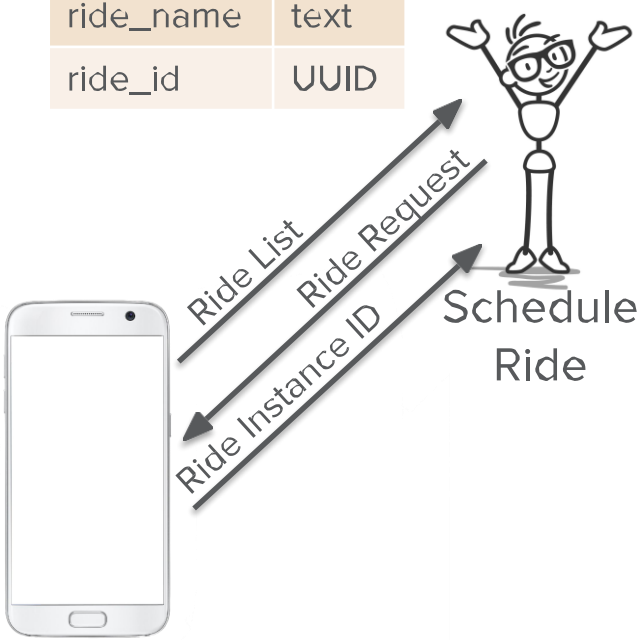
Field	Type
location **K**	text
ride_id **C↑**	UUID
ride_name	text
capacity	int

rider_count_by_time_and_ride

Field	Type
start_time **K**	timestamp
ride_id **K**	UUID
rider_count	int

Ride List

Field	Type
ride_name	text
ride_id	UUID



Ride Request

Field	Type
user_id	UUID
ride_id	UUID
start_time	timestamp

Ride Instance ID

Field	Type
ride_instance_id	UUID

Cassandra-Land Table Summary

users_by_phone_number

Field	Type
phone_number **K**	text
user_id	UUID
password	text

rider_count_by_time_and_ride

Field	Type
start_time **K**	timestamp
ride_id **K**	UUID
rider_count	int

ride_instances_by_start_time

Field	Type
start_time **K**	timestamp
ride_id **C↑**	UUID
user_id **C↑**	UUID
ride_name	text
phone_number	text

ride_list_by_location

Field	Type
location **K**	text
ride_id **C↑**	UUID
ride_name	text
capacity	int

ride_instances_by_user_id

Field	Type
user_id **K**	UUID
start_time **C↑**	timestamp
ride_id	UUID
ride_name	text

Primary Key - What you need to know

- Must have one or more partition key columns
- May have zero or more clustering columns

```
PRIMARY KEY(( <partition key column>,...), <clustering column>,...)
```

Timestamps

- Format:

```
'YYYY-MM-DDTHH:MM:SS[.fff]'
```

- Notice the quotes
- Milliseconds are optional
- Examples:

```
'2020-01-09T11:45:23'
'2020-01-09T11:45:23.898'
```

Update Statement

- Can have multiple <assignment>
- IF is optional – causes a lightweight transaction

```
UPDATE <keyspace name>.<table name>
    SET <assignment>
    WHERE <row specification>
IF <condition>
```

Batch Statement

- What you need to know – BATCH

```
BEGIN BATCH
        INSERT statement
        INSERT statement
        …
APPLY BATCH
```

- Once a statement succeeds, Cassandra will ensure all the others succeed
- Can use for inserting into multiple tables