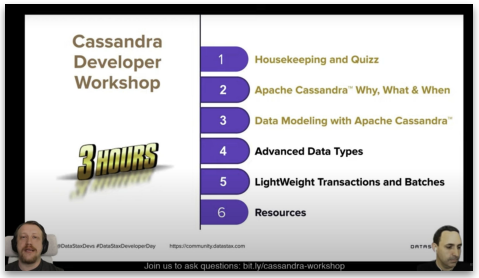
# Welcome

# Intro to Cassandra for Developers

# The Crew

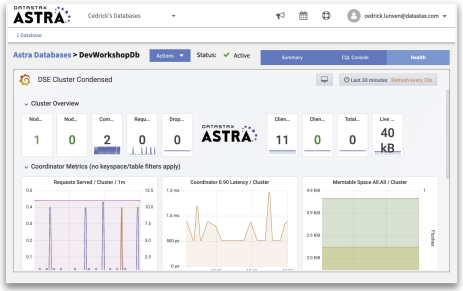
## **Courses**: youtube.com/DataStaxDevs




YouTube

Twitch

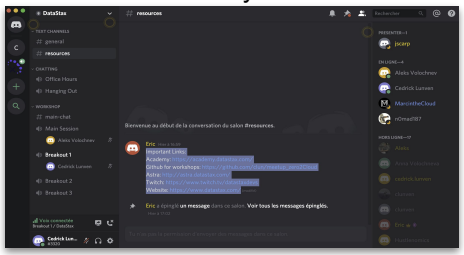
## **Runtime**: dtsx.io/workshop




cassandra

DATASTAX ASTRA

## **Questions**: bit.ly/cassandra-workshop

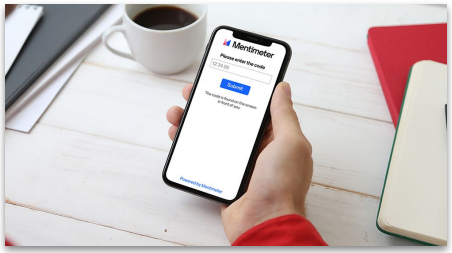



Discord

YouTube

Available on the iPhone App Store

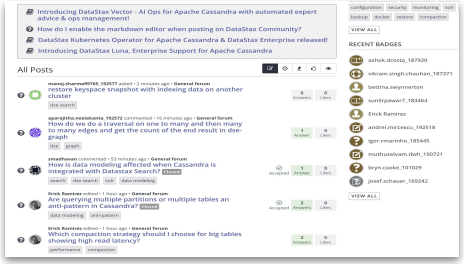
## **Quizz**: menti.com




Mentimeter

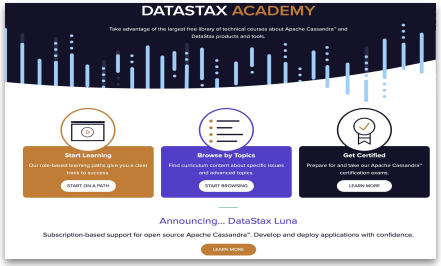
GET IT ON Google play

## **Forum**: community.datastax.com



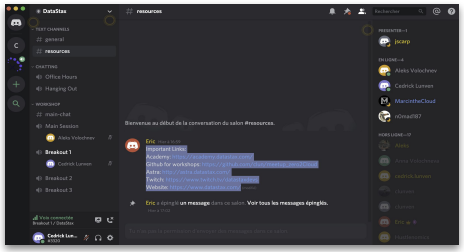

DATASTAX COMMUNITY

## **Training**: academy.datastax.com




DATASTAX ACADEMY

## **Chat**: bit.ly/cassandra-workshop




**Discord**

DATASTAX ASTRA == cassandra

Fully managed Cassandra
Without the ops!

# DataStax Astra

**Global Scale**
Put your data where you need it without compromising performance, availability, or accessibility.

**No Operations**
Eliminate the overhead to install, operate, and scale Cassandra.

**10 Gig Free Tier**
Launch a database in the cloud with a few clicks, no credit card required.


menti.com
Mentimeter
What is the most desirable feature of your ideal workplace?
Flexible time
Social factor
Big paycheck
Available on the iPhone
App Store
GET IT ON
Google play

# Exercise

https://github.com/DataStax-Academy/Intro-to-Cassandra-for-Developers


# Intro to Cassandra for Developers

- ➔ Tables, Partitions
- ➔ The Art of Data Modelling
- ➔ What's NEXT?


- ➔ Tables, Partitions
- ➔ The Art of Data Modelling
- ➔ What's NEXT?

# Data Structure: a Cell


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

# 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.

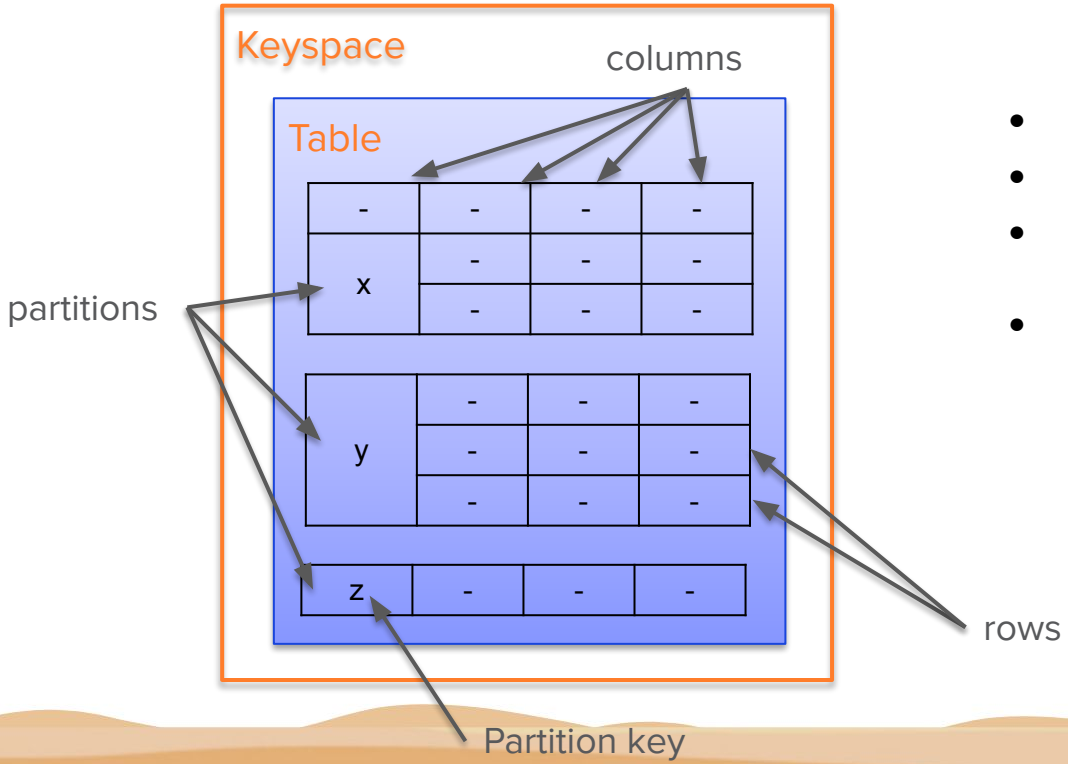
| ID | First Name | Last Name | Department |
|---|---|---|---|
| 1 | John | Doe | Wizardry |
| 399 | Marisha | Chapez | Wizardry |
| 415 | Maximus | Flavius | Wizardry |

# Data Structure: a Table


A group of columns and rows storing partitions.

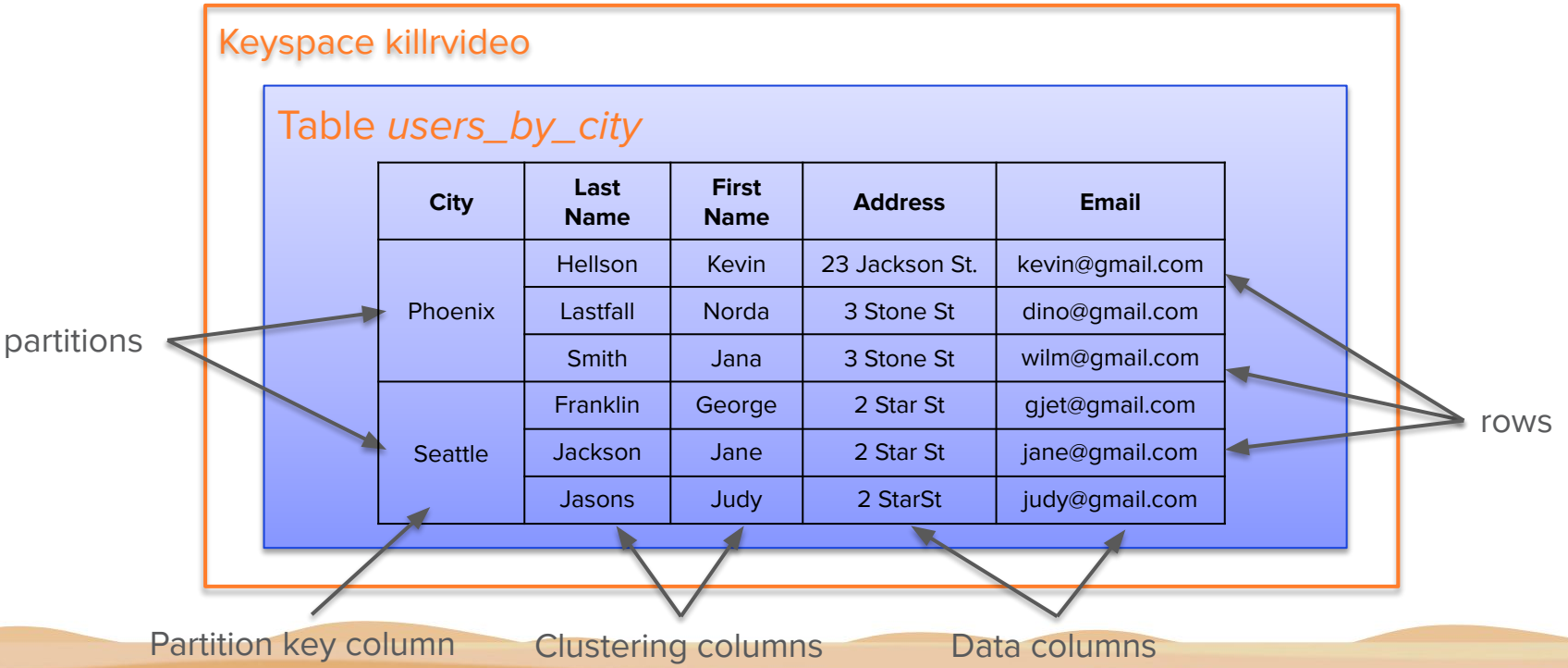
| ID | First Name | Last Name | Department |
|---|---|---|---|
| 1 | John | Doe | Wizardry |
| 2 | Mary | Smith | Dark Magic |
| 3 | Patrick | McFadin | DevRel |

# Data Structure: Overall




- Tabular data model, with one twist
- *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




| 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 |
| Seattle | Franklin | George | 2 Star St | gjet@gmail.com |
| | Jackson | Jane | 2 Star St | jane@gmail.com |
| | Jasons | Judy | 2 StarSt | judy@gmail.com |

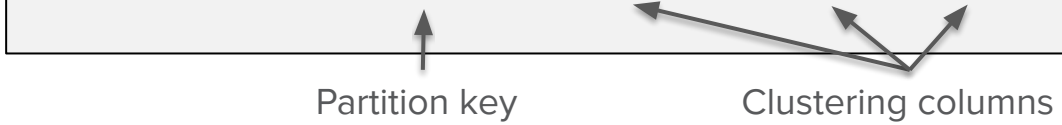
# Creating a Table in CQL

# 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.**

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




Good Examples:

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

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

Bad Example:

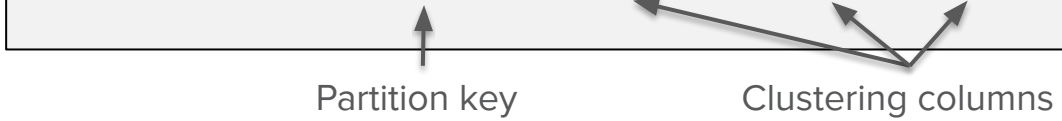
```
PRIMARY KEY ((city), last_name, first_name);
```

# Partition Key

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

**PARTITIONS ROWS.**

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




Good Examples:

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

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

Bad Example:

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

# Clustering Column(s)

**Used to ensure uniqueness and sorting order.** Optional.

```
CREATE TABLE killrvideo.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);
```

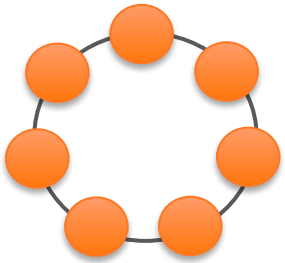
✗ Not Sorted

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

✓

# Partition: The Beginning

```
CREATE TABLE killrvideo.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

# Rules of a Good Partition

The Slide of the Year Award!

- 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

The Slide of the Year Award!

- **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

The Slide of the Year Award!

- 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

The Slide of the Year Award!

- 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.

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

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

# Rules of a Good Partition

The Slide of the Year Award!

- 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.

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

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

# Rules of a Good Partition

The Slide of the Year Award!

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

Example large 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's UUID
Timestamp:
Value: float

BUCKETING

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

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

# Rules of a Good Partition

The Slide of the Year Award!

- 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, month_year), reported_at);
```

## BUCKETING

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

# Rules of a Good Partition

The Slide of the Year Award!

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

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

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

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

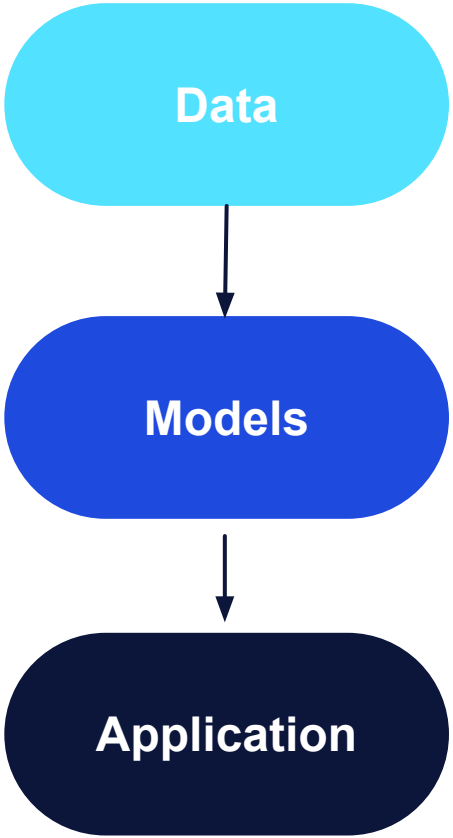
# Exercise

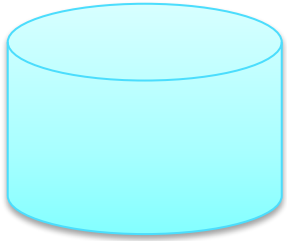
https://github.com/DataStax-Academy/Intro-to-Cassandra-for-Developers#2-create-a-table


- ➔ Tables, Partitions
- ➔ The Art of Data Modelling
- ➔ What's NEXT?

# 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

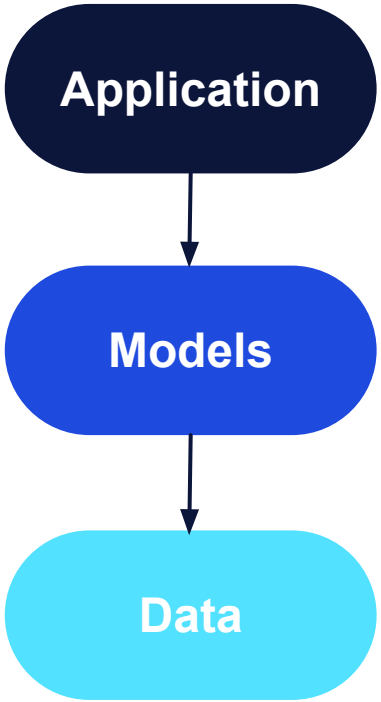
| userId | firstName | lastName |
|---|---|---|
| 1 | Edgar | Codd |
| 2 | Raymond | Boyce |

Department

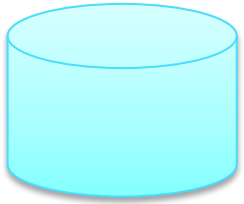
| departmentId | department |
|---|---|
| 1 | Engineering |
| 2 | Math |

# 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



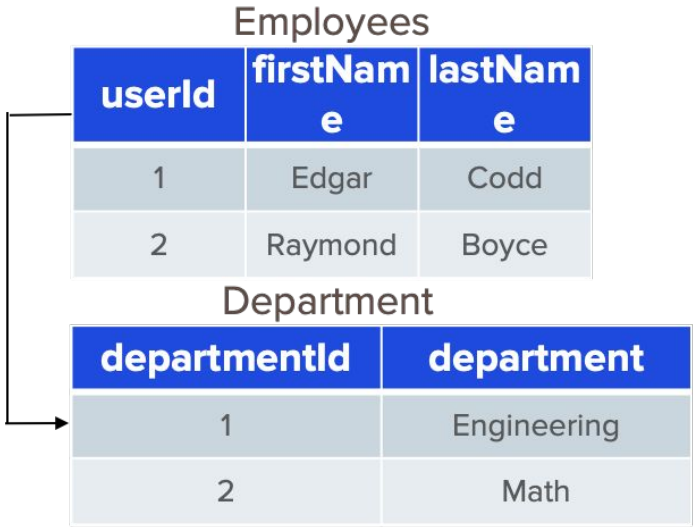

| id | firstName | lastName | department |
|---|---|---|---|
| 1 | Edgar | Codd | Engineering |
| 2 | Raymond | Boyce | Math |



# 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

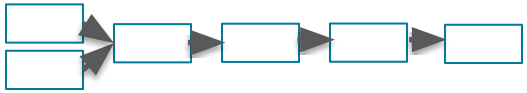
Employees

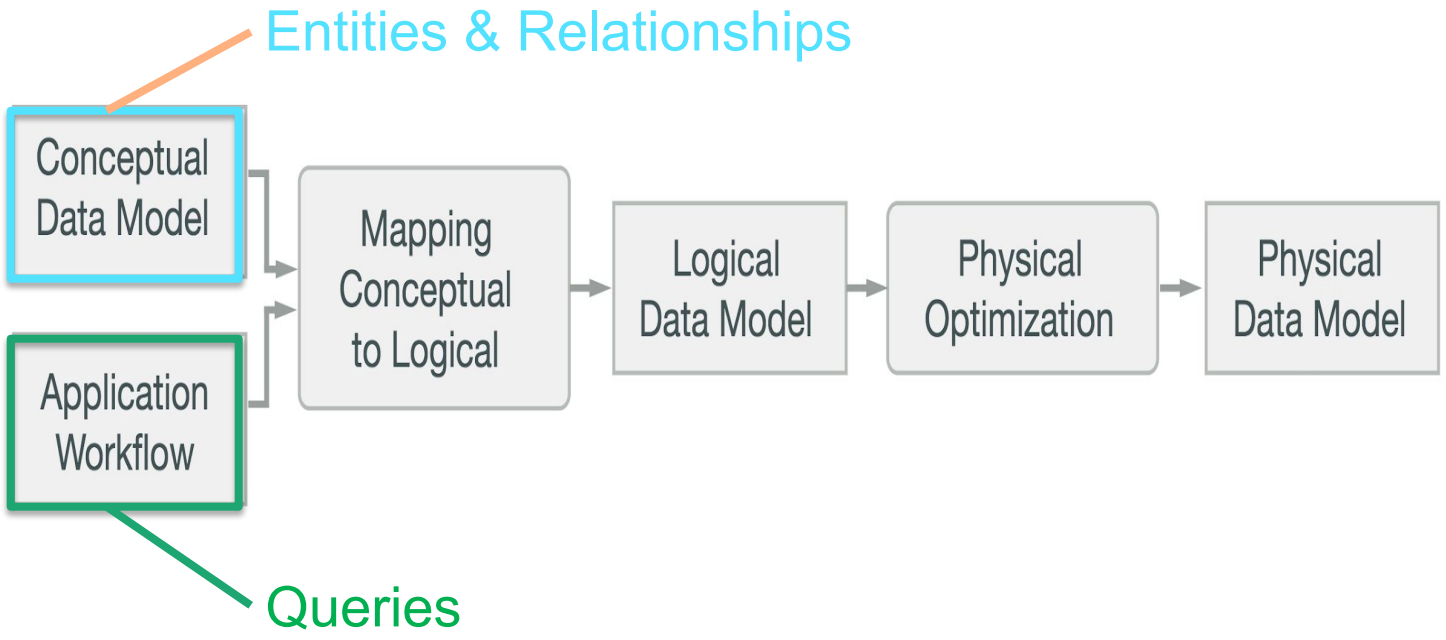
| userId | firstName | lastName | department |
|---|---|---|---|
| 1 | Edgar | Codd | Engineering |
| 2 | Raymond | Boyce | Math |

Department

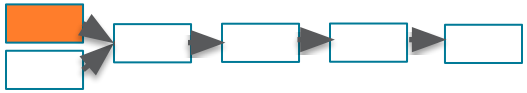
| departmentId | department |
|---|---|
| 1 | Engineering |
| 2 | Math |

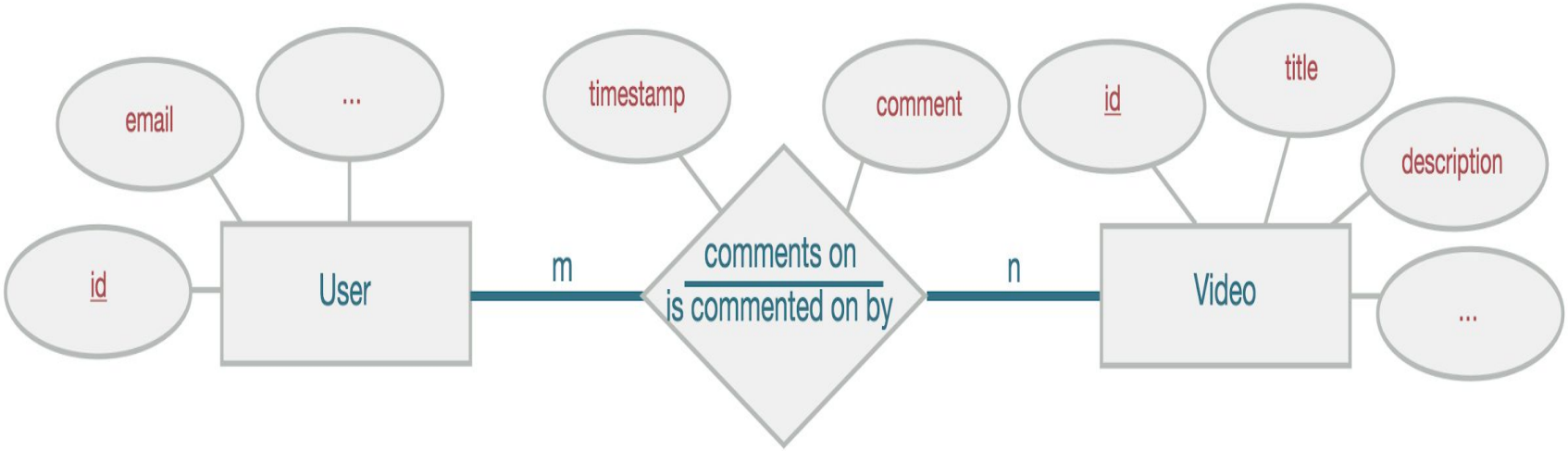
# Designing Process: Step by Step



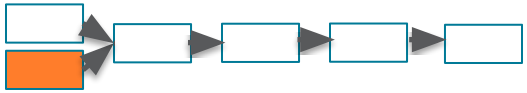


# Designing Process: Conceptual Data Model





# Designing Process: Application Workflow



**Use-Case I:**

- A User opens a Video Page

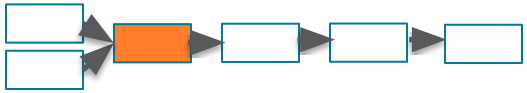
**WF1:** Find comments related to target video using its identifier, most recent first

**Use-Case II:**

- A User opens a Profile

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

# Designing Process: Mapping



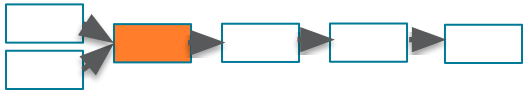
**Query I:** Find comments posted for a user with a known id (show most recent first)

→ comments_by_user

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

→ comments_by_video

# Designing Process: Mapping



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

→ comments_by_user

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

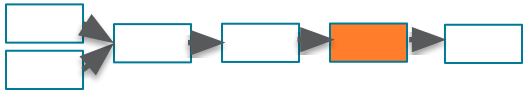
→ comments_by_video

# 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 | |

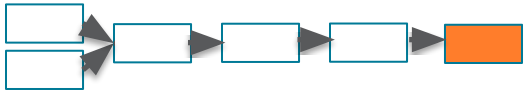
# Designing Process: Physical Data Model



| 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,
    commentid timeuuid,
    userid    uuid,
    comment   text,
    PRIMARY KEY ((videoid), commentid)
) WITH CLUSTERING ORDER BY (commentid DESC);
```

# Exercise

https://github.com/DataStax-Academy/Intro-to-Cassandra-for-Developers#3-execute-crud-operations


menti.com
Mentimeter
What is the most desirable feature of your ideal workplace?
Flexible time
Social factor
Big paycheck
Available on the iPhone
App Store
GET IT ON
Google play


- ➔ Tables, Partitions
- ➔ The Art of Data Modelling
- ➔ What's NEXT?

# MORE LEARNING!!!!

Developer site: datastax.com/dev

- Developer Stories
- New hands-on learning scenarios with Katacoda
  - Try it Out
  - Cassandra Fundamentals
  - New Data Modeling course in progress, sneak preview at https://katacoda.com/datastax/courses/cassandra-data-modeling

Classic courses still available at DataStax Academy


Katacoda

# Developer Resources

| | |
|---|---|
| **LEARN** | New hands-on learning at www.datastax.com/dev<br>Classic courses available at DataStax Academy |
| **ASK/SHARE** | Join community.datastax.com<br>Ask/answer community user questions - share your expertise |
| **CONNECT** | Follow us<br>We are on Youtube - Twitter - Twitch! |
| **MATERIALS** | Slides and code for this course are available at<br>https://github.com/DataStax-Academy/cassandra-workshop-series |

# Walk through of /dev

DATASTAX®

# Thank You