Developer Workshop Series Week II

What we will cover:

- ➔ Keyspaces, Tables, Partitions
- ➔ The Art of Data Modelling
- ➔ Data Types
- ➔ What’s NEXT?

Cassandra Cloud-Native Workshop Series

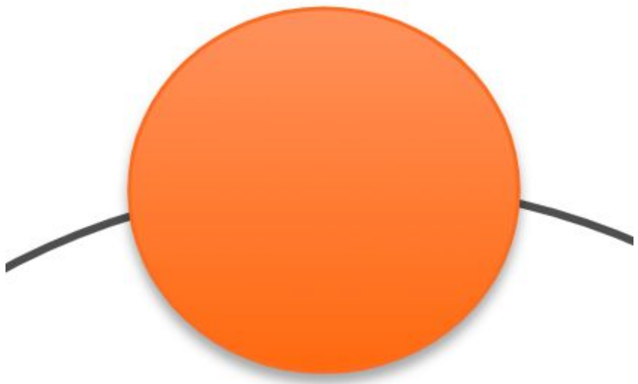
What we will cover:

Developer Workshop Series Week II

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

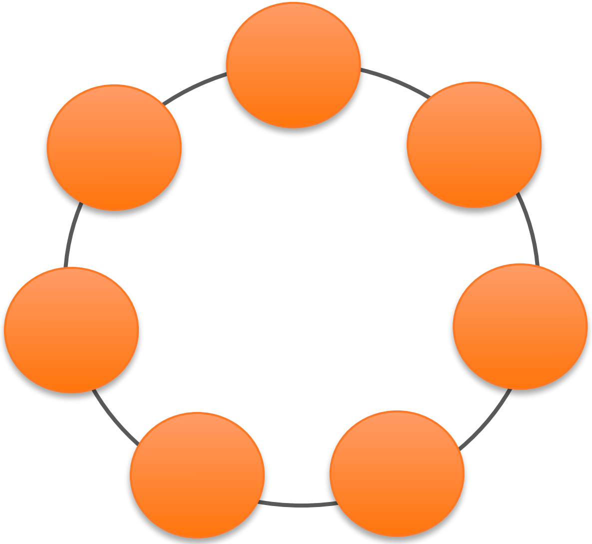
Infrastructure: a Node

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

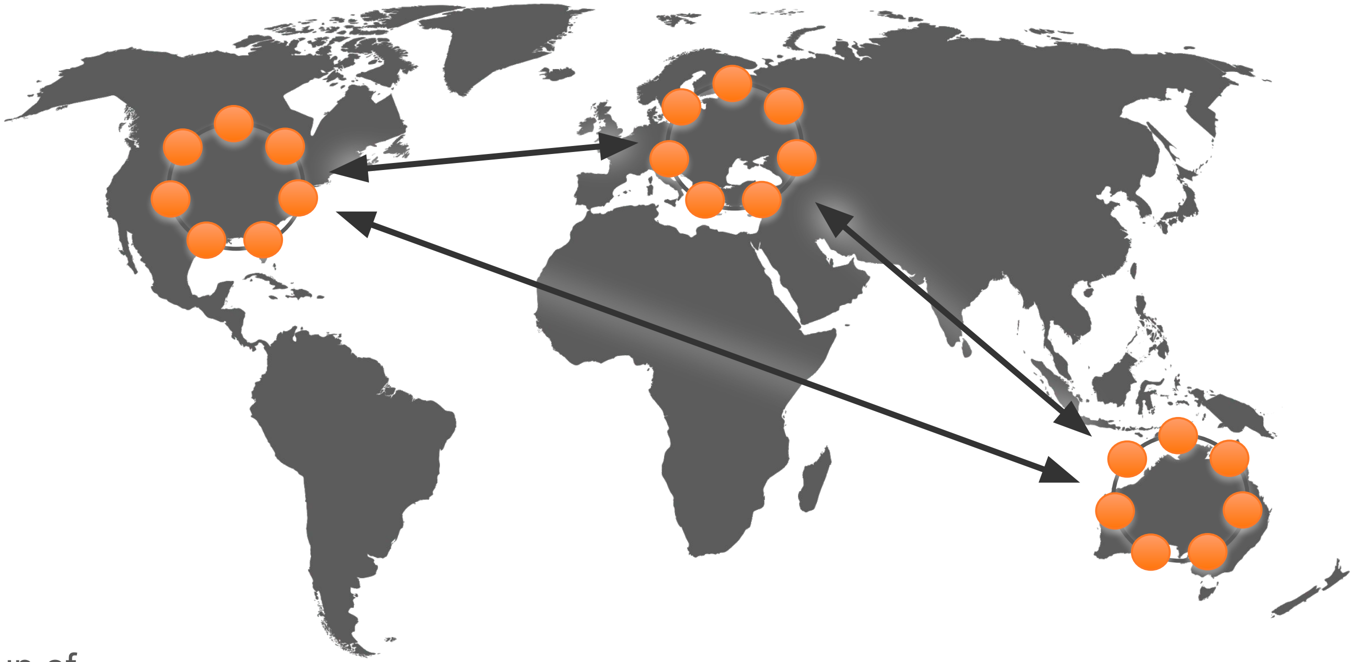


Infrastructure: a Datacenter (Ring)

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.

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.

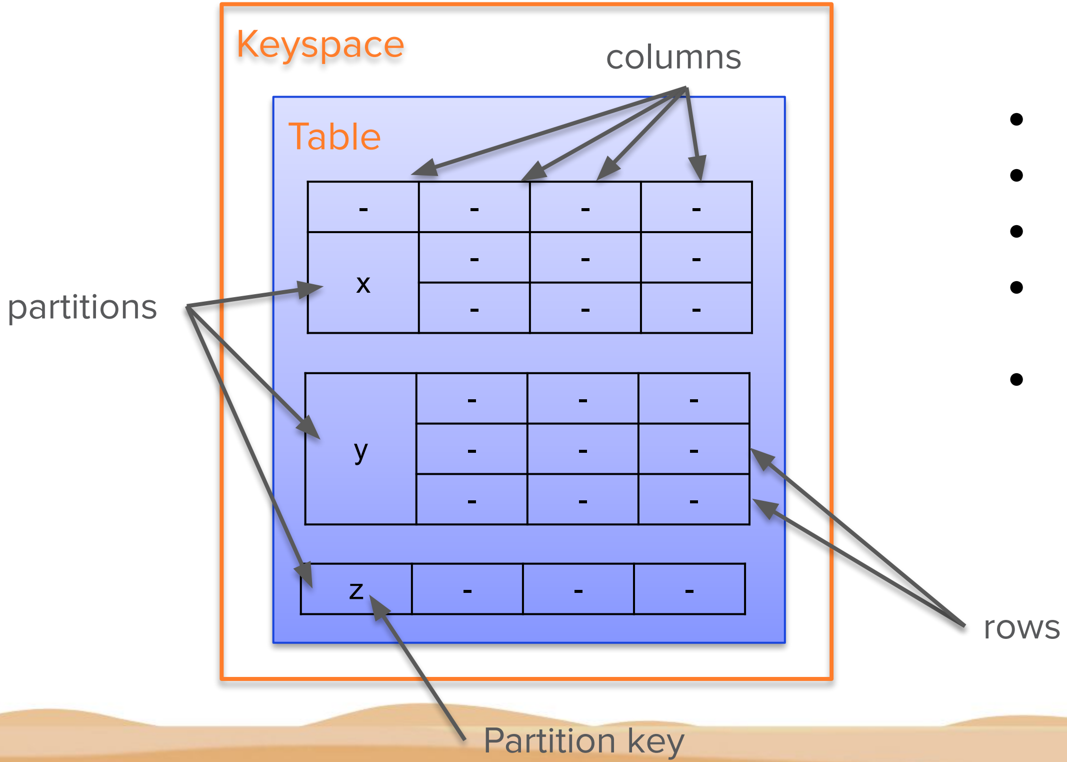
ID	First Name	Last Name	Department
1	John	Doe	Wizardry
2	Mary	Smith	Dark Magic
3	Patrick	McFadin	DevRel

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

Keyspace users

Table *users_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

partitions

rows

Partition key column

Clustering columns

Data columns

Creating a Keyspace in CQL

keyspace

replication strategy

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

Replication factor by data center

Creating a Table in CQL

keyspace

table

column definitions

```
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));
```

Primary key

Partition key

Clustering columns

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 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

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 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

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 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);
```

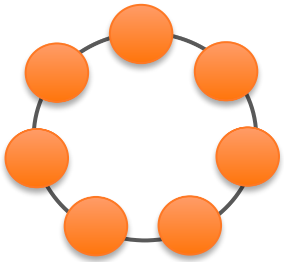
```
PRIMARY KEY ((video_id), comment_id);
```

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

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.

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

?

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

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);
```

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

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 ge IoT infras uctur ardw e all ver t
world, di rent ens s rep rti their stat ev y 10
seconds en r rep rts s UUID, ti mp o
report, se sor's alu

S nsor ID
T es mp:
ue oat

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, _____), 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**

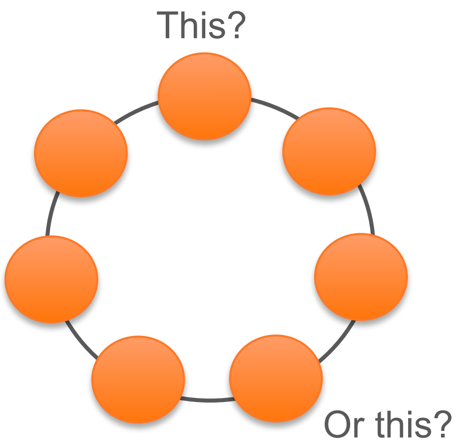
```
PRIMARY KEY (user_id);
```

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

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

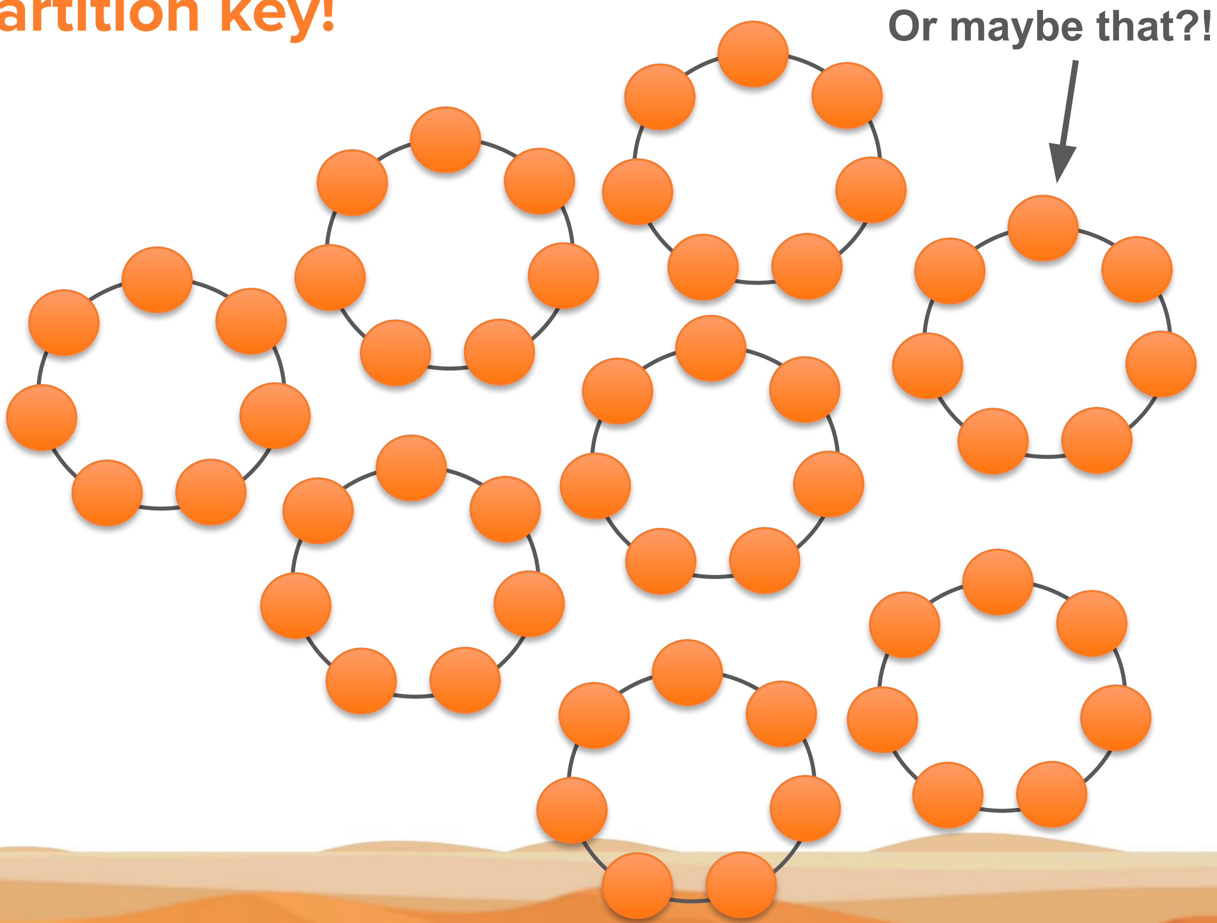
Always specify the partition key!

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



Always specify the partition key!

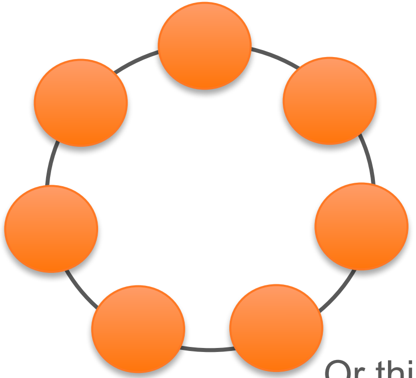
If there is no partition key in a query, which node you will ask?



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";
```

Developer Workshop Series Week II

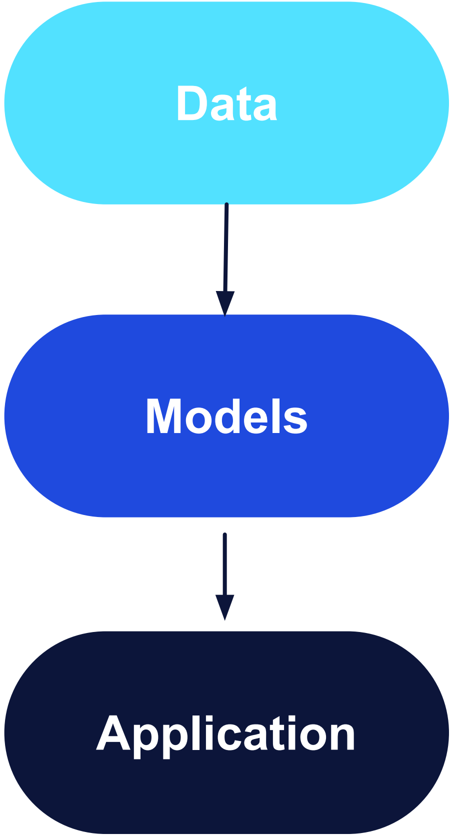
What we will cover:

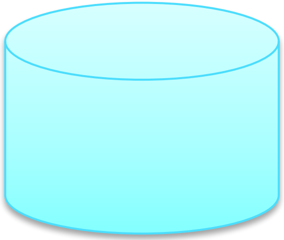
- ➔ Keyspaces, Tables, Partitions
- ➔ The Art of Data Modelling
- ➔ Data Types
- ➔ What's NEXT?

Cassandra Cloud-Native Workshop Series

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

Employees

userId	firstName	lastName
1	Edgar	Codd
2	Raymond	Boyce

Department

departmentId	department
1	Engineering
2	Math

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

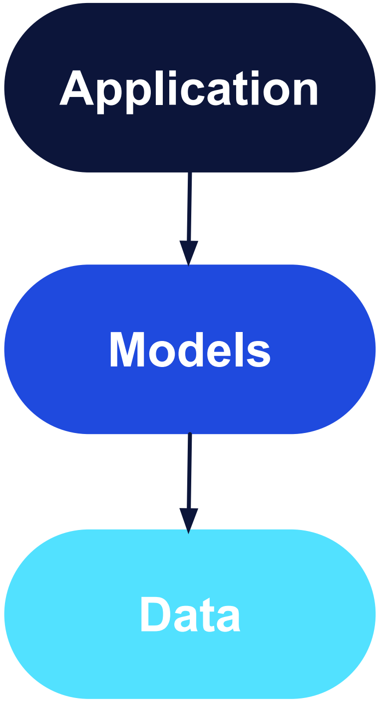
userId	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math

Department

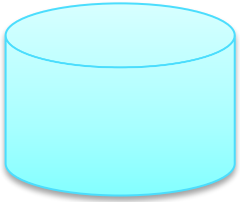
departmentId	department
1	Engineering
2	Math

NoSQL Data Modelling

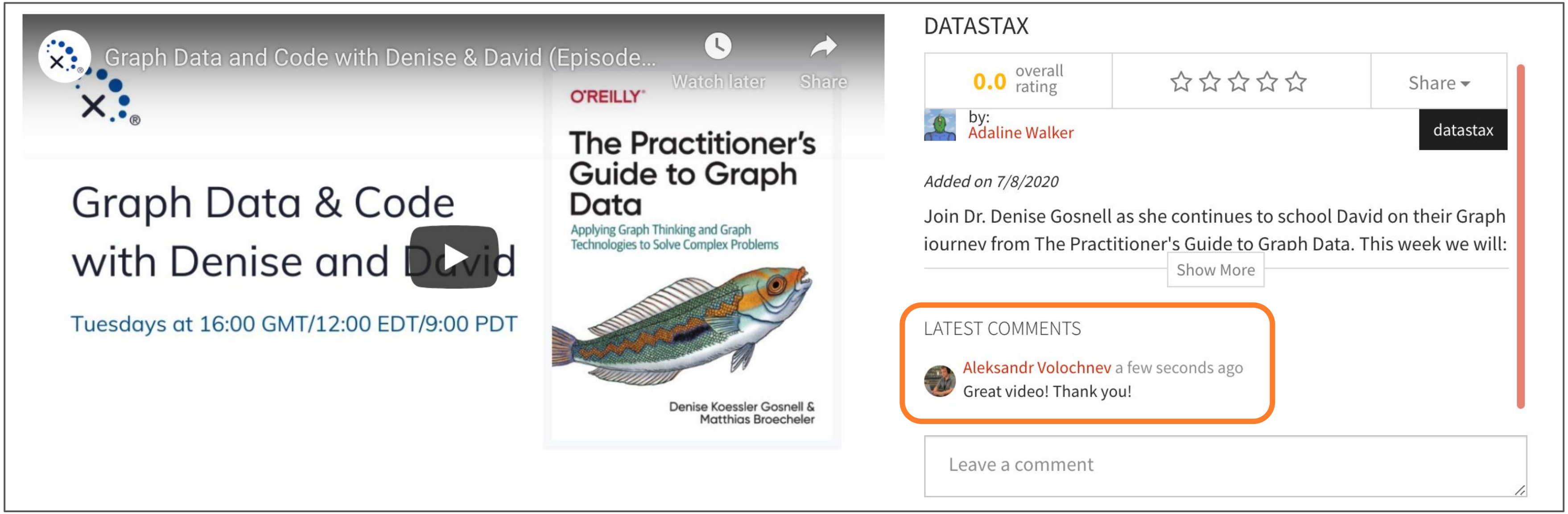
1. Analyze user behaviour (customer first!)
2. Identify workflows, their dependencies and needs
3. Define Queries to fulfill these workflows
4. Knowing the queries, design tables, using denormalization.
5. Use BATCH when inserting or updating denormalized data of multiple tables



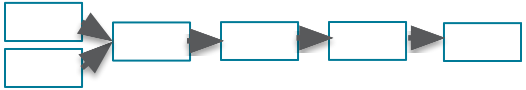
id	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math

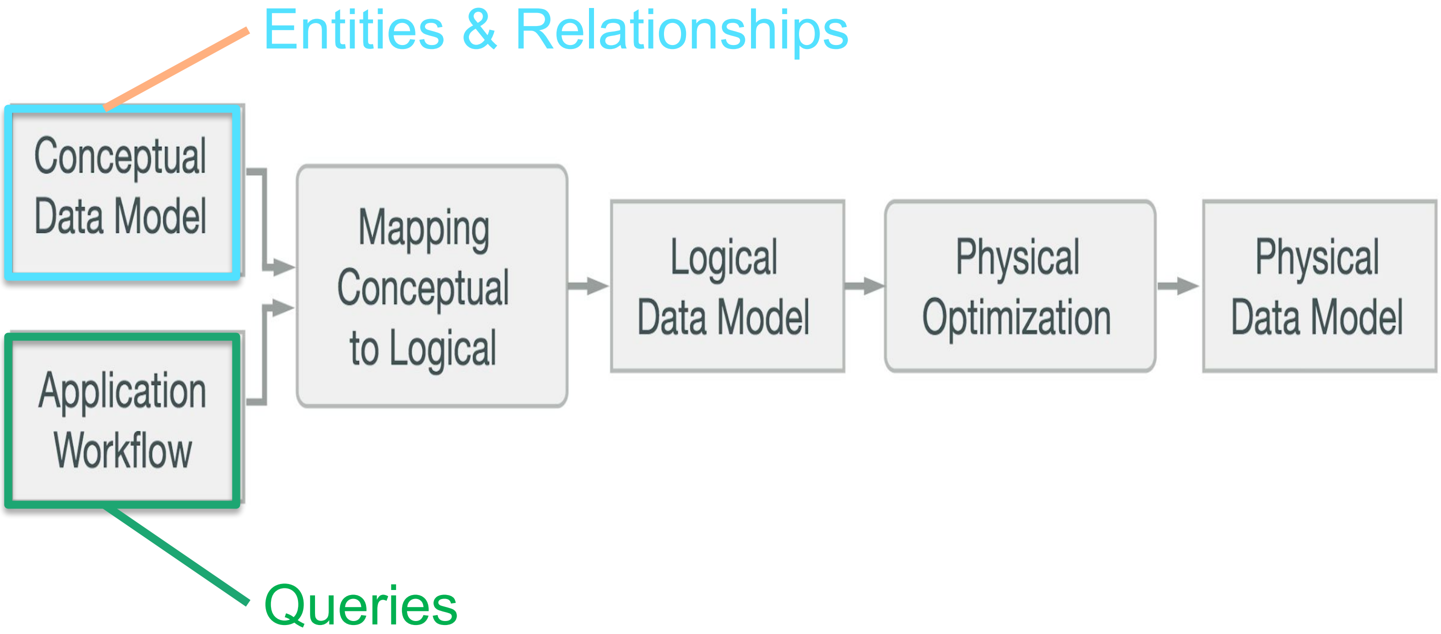


Let's go practical!

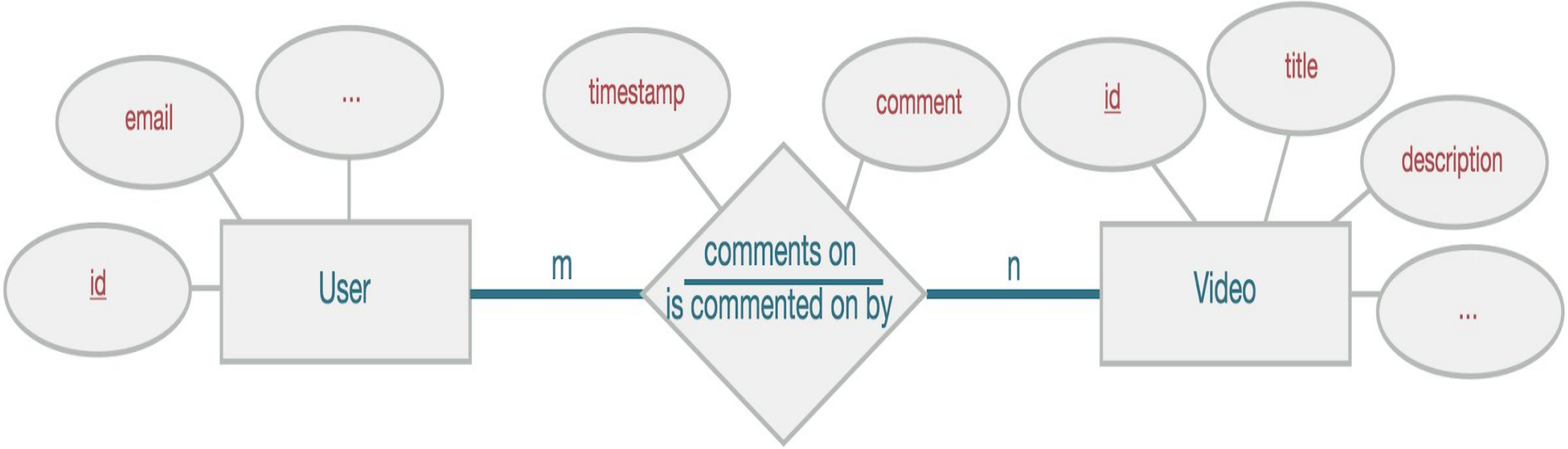


Designing Process: Step by Step

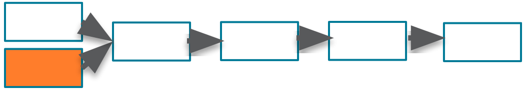




Designing Process: Conceptual Data Model



Designing Process: Application Workflow



Use-Case I:

- A User opens a Video Page

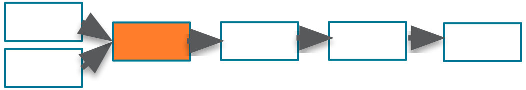
WF1: Find comments related to target video using its identifier, most recent first

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

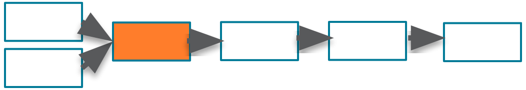
Designing Process: Mapping



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

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

Designing Process: Mapping



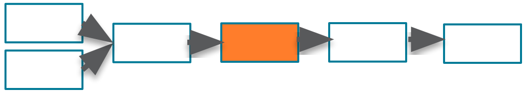
```
SELECT * FROM comments_by_user
WHERE userid = <some UUID>
```

→ comments_by_user

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

→ comments_by_video

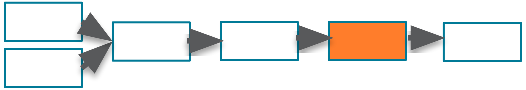
Designing Process: Logical Data Model



comments_by_user	
userid	K
creationdate	C ↓
commentid	C ↑
videoid	
comment	

comments_by_video	
videoid	K
creationdate	C ↓
commentid	C ↑
userid	
comment	

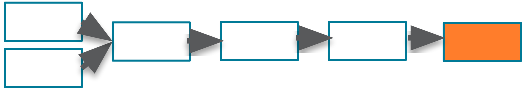
Designing Process: Physical Data Model



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);
```

menti.com

89 49 47

Available on the iPhone App Store

GET IT ON Google play

Developer Workshop Series Week II

What we will cover:

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

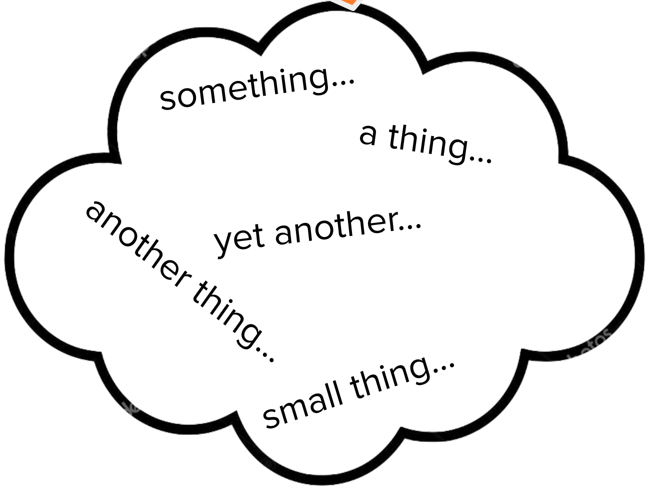
Cassandra Cloud-Native Workshop Series

Basic Data Types

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

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

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

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

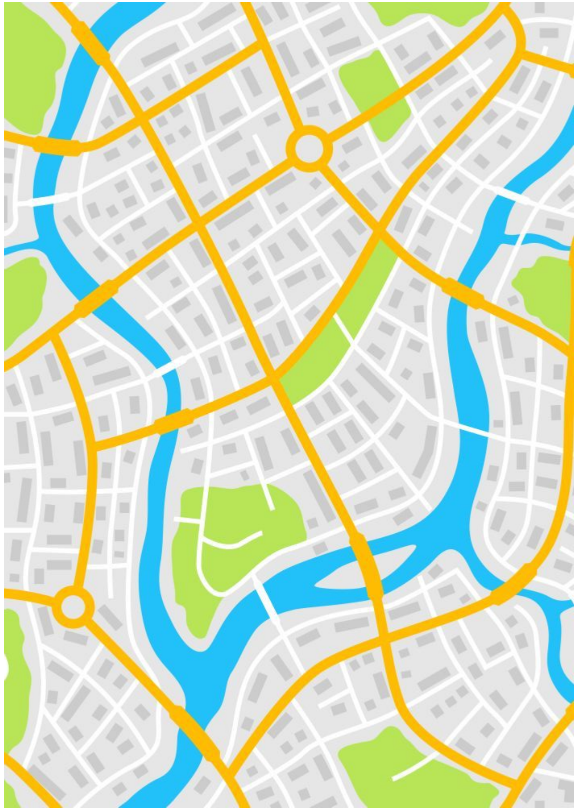
```
UPDATE killrvideo.actors_by_video
  SET actors[1] = 'Brown'
  WHERE videoid = 12345678-1234-1234-1234-123456789012;
```

Replace an element

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

Collection: 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)
);
```

Cool Thing!

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

Counters

- 64-bit signed integer
- Use-case:
  - Imprecise values such as likes, views, etc.
- Two operations:
  - Increment
  - Decrement
  - First op assumes the value is zero

Counters

- 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

This format must be observed

Incrementing a counter:

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

This can be an integer value

Just change the sign

Decrementing a counter:

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