

DataStax

# Developers

## Apache Cassandra™ Data Modelling

Building efficient applications with Apache Cassandra





## Developer Advocate Lead

[dtsx.io/aleks](https://dtsx.io/aleks)



@aleks-volochnev



@HadesArchitect

- IT Exorcist
- Apache Cassandra™ certified
- Cloud Architect certified



Aleksandr Volochnev



# Developer Advocate



@ArtemChebotko

- Data professional, computer scientist
- Data modeling, data quality, data warehousing, data analytics
- Author of the Cassandra Data Modeling Methodology
- Google Cloud Certified Data Engineer

Astra DB

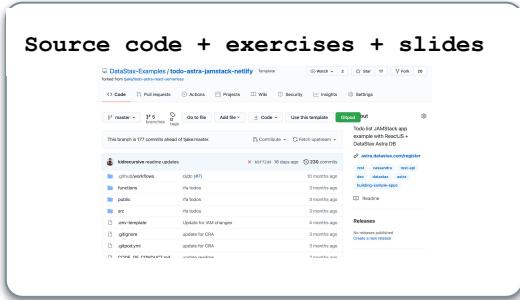
Astra  
STREAMING

PULSAR



Artem Chebotko

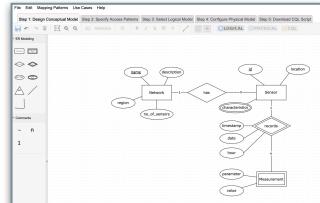
Nothing to install !



## SCENARIOS

K L L R  
C O D A    BETA

## DATA MODELING: KDM



<http://kdm.kashliev.com>

Hands-On Housekeeping



# Data Modelling Methodology Workshop



Awarded to La [REDACTED] @hotmail.com)

Issued on Jul 4, 2022 at 8:00 AM

Offered by  
[DataStax Developers](#)

Upgrade Complete! This badge is to certify successful completion of the DataStax Cassandra Workshop: "Data Modelling Methodology for Apache Cassandra".

## EARNING CRITERIA

Recipients must complete the earning criteria to earn this badge

To earn this badge, individuals must complete the following steps during Data Modelling Methodology Workshop:

- Attend the lecture
- Complete the practical steps by doing all required exercises

[View Badge Public Page](#) ↗



Achievement Unlocked !

93 and counting



# 01



## Data Storage Overview

Data Distribution & Organization

# 02

## Key Definition

Primary, Partition, Clustering

# 03

## CQL Data Types

Brief Review

# 04

## Data Modeling Process

Methodology, Notation, Tool

# 05

## Optimization Techniques

Five important optimizations

# 06

## What's next?

Homework, Next Session



Agenda

# 01



## Data Storage Overview

Data Distribution & Organization

# 02

## Key Definition

Primary, Partition, Clustering

# 03

## CQL Data Types

Brief Review

# 04

## Data Modeling Process

Methodology, Notation, Tool

# 05

## Optimization Techniques

Five important optimizations

# 06

## What's next?

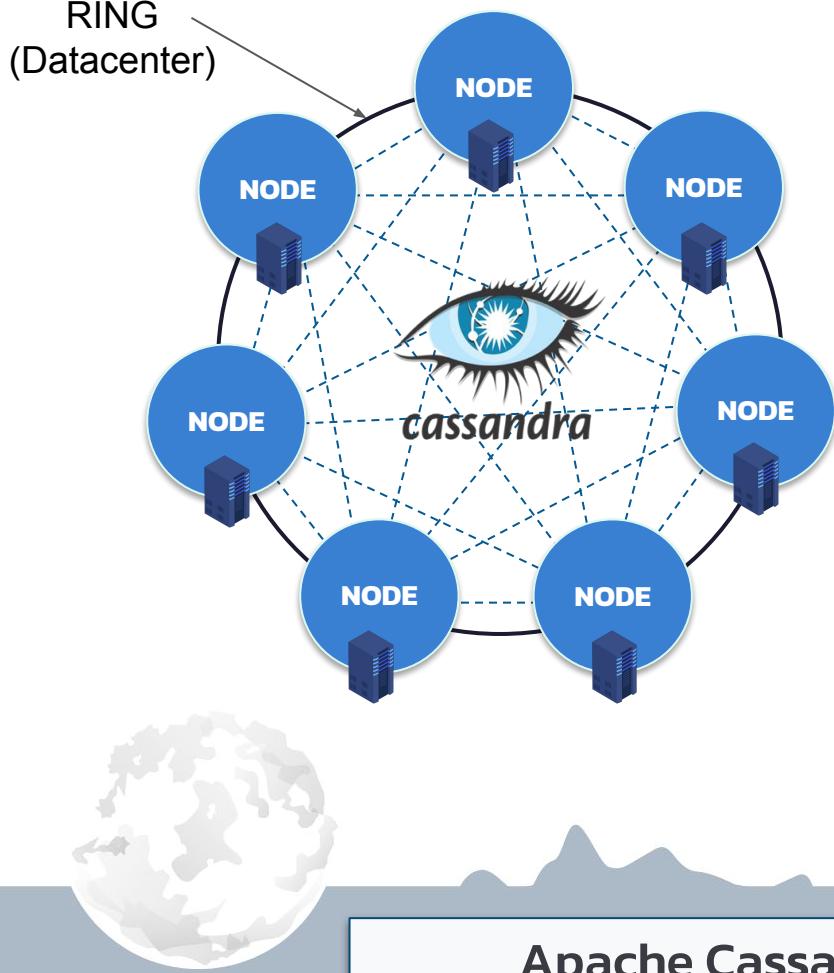
Homework, Next Session



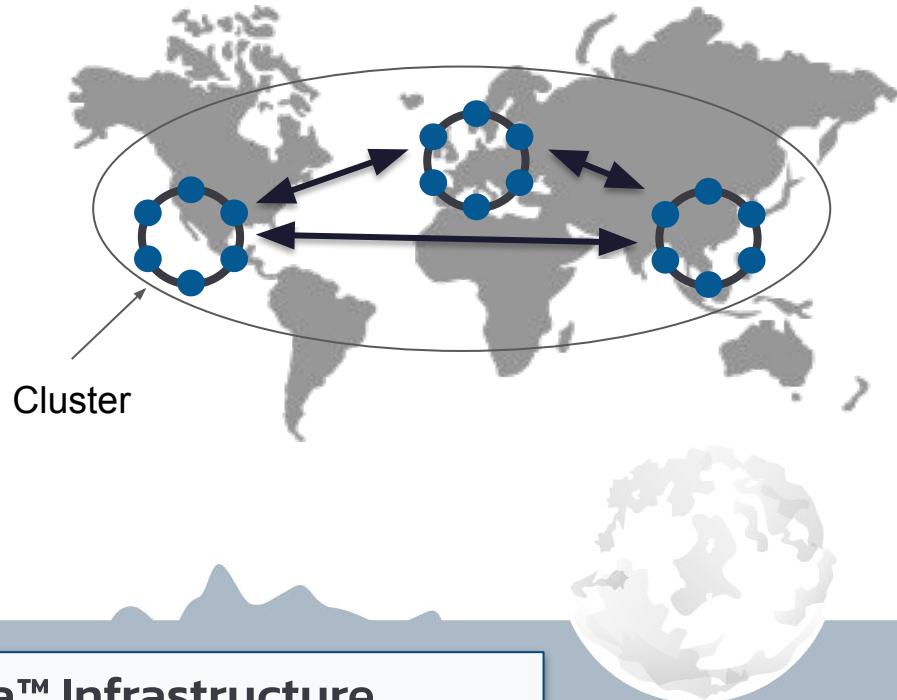
Agenda

# Data Distribution

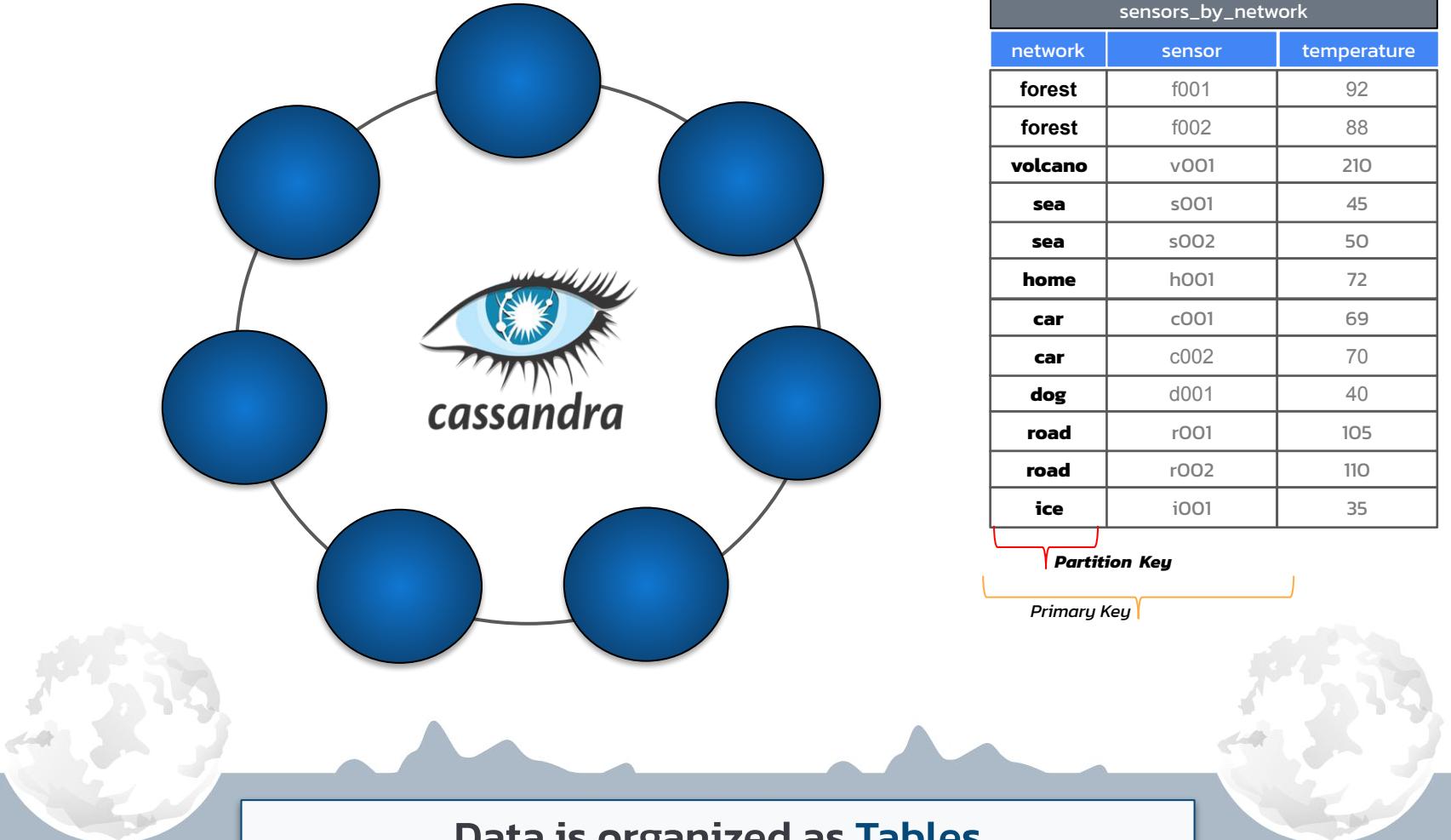




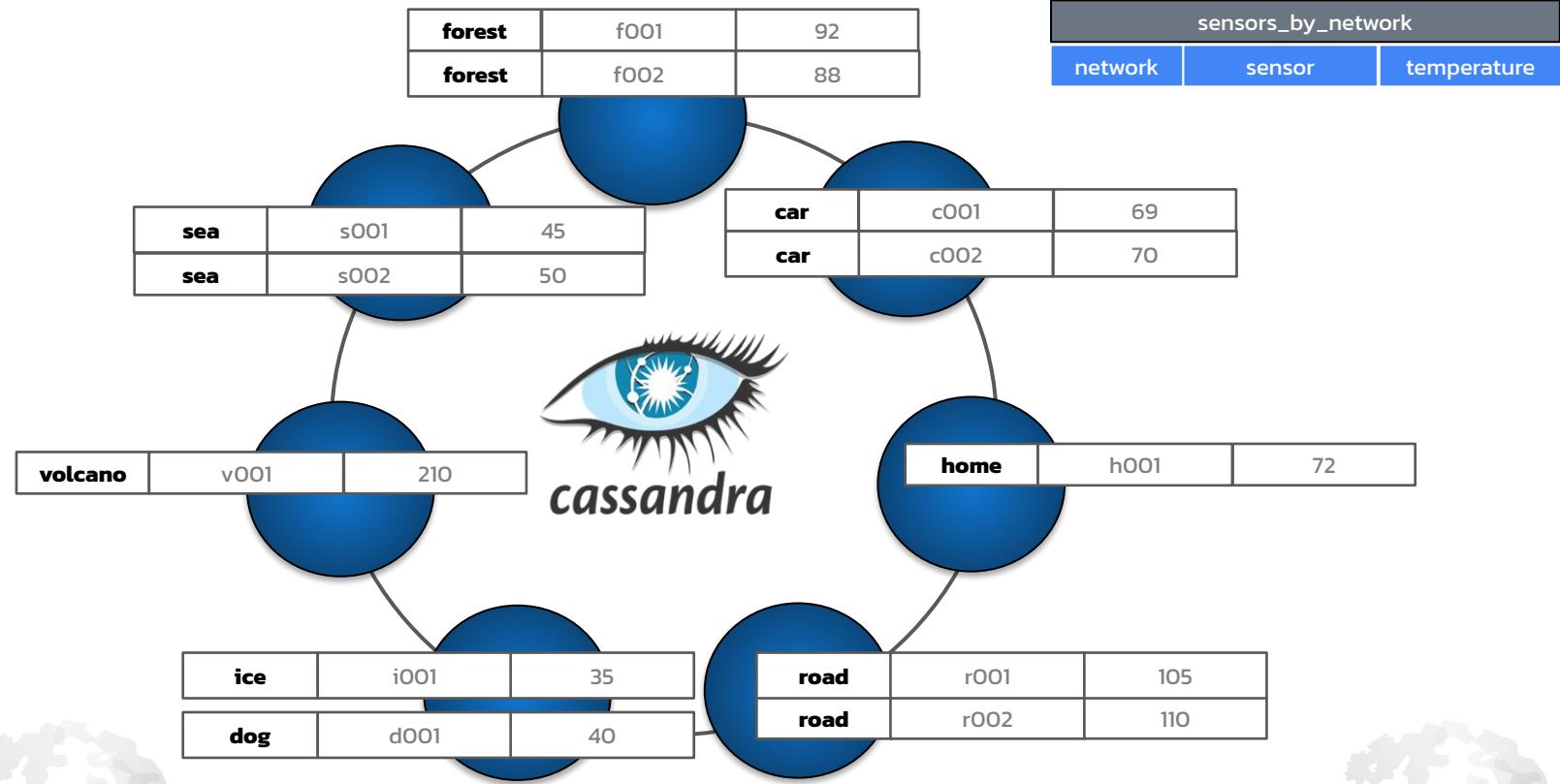
1. **NO** Single Point of Failure (masterless)
2. Scales for writes and reads
3. Application can contact any node  
(in case of failure - just contact next one)



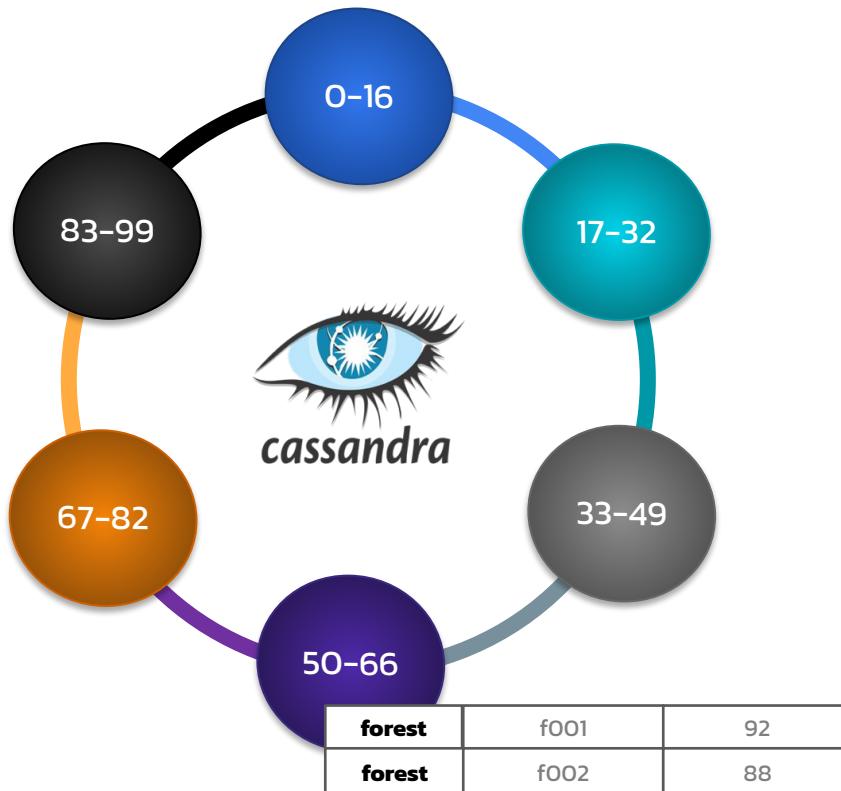
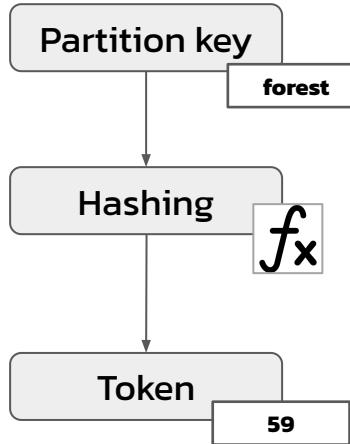
Apache Cassandra™ Infrastructure



Data is organized as Tables



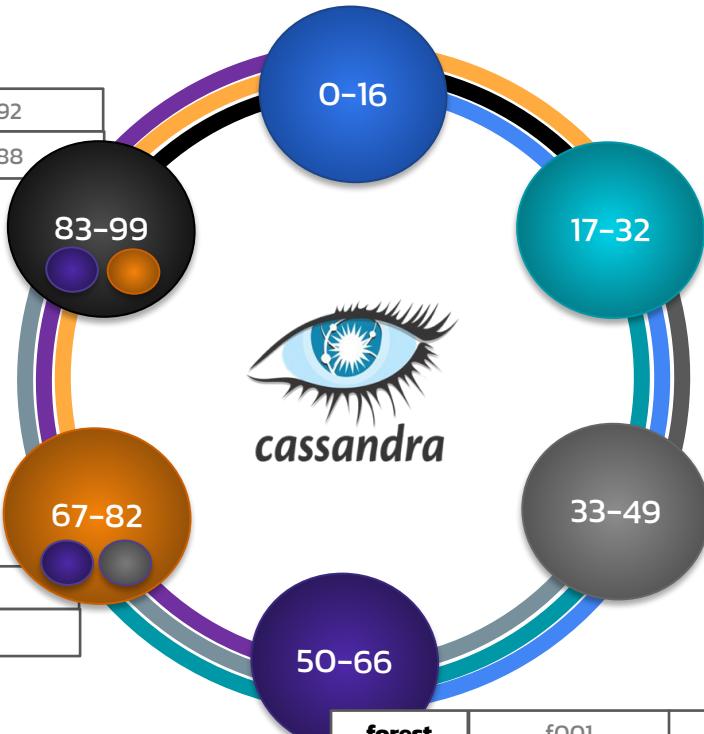
Data is organized as Distributed tables



Per Table, each node owns a range of tokens

RF = 3

<b>forest</b>	f001	92
<b>forest</b>	f002	88



<b>forest</b>	f001	92
<b>forest</b>	f002	88

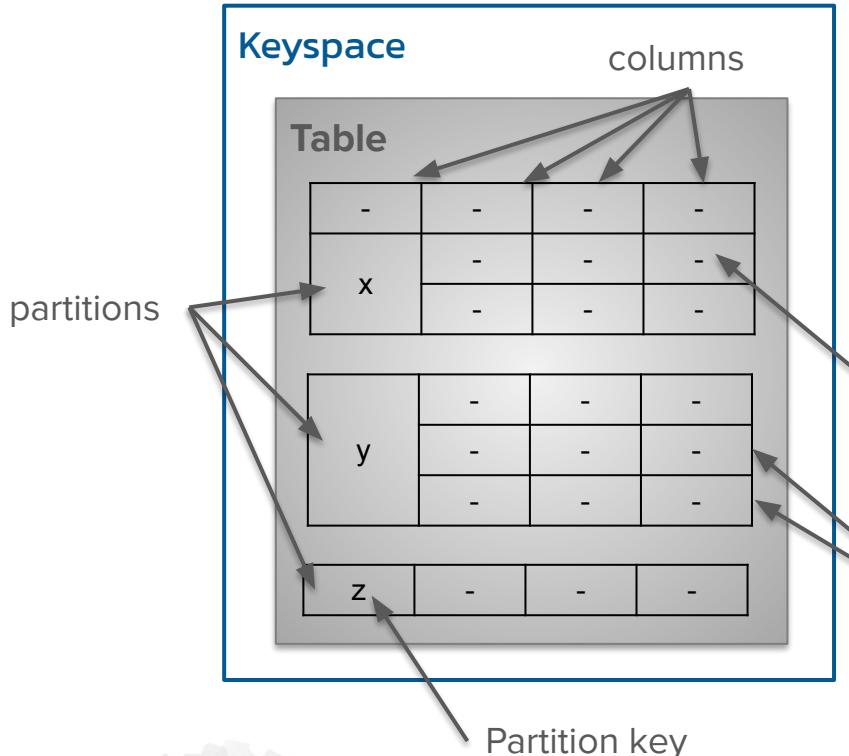
<b>forest</b>	f001	92
<b>forest</b>	f002	88



Data is replicated X times (replication factor)

# Data Organization





- 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

**Cell  $\in$  Row  $\in$  Partition  $\in$  Table  $\in$  Keyspace**

```
CREATE KEYSPACE sensor_data  
WITH REPLICATION = {  
    'class' : 'NetworkTopologyStrategy',  
    'us-west-1' : 3,  
    'eu-central-1' : 5  
};
```

keyspace

replication strategy

Replication factor by data center



Create a Keyspace

```
Keyspace  
↓  
CREATE TABLE sensor_data.sensors_by_network (  
    network      text,  
    sensor       text,  
    temperature integer,  
    PRIMARY KEY ((network), sensor)  
);
```

Table  
↓

Primary key

Partition key

Clustering columns



Create a table



# Lab 1

## Create Your Astra DB Instance

[https://github.com/datastaxdevs/  
workshop-cassandra-data-modeling](https://github.com/datastaxdevs/workshop-cassandra-data-modeling)

# 01



## Data Storage Overview

Data Distribution & Organization

# 02

## Key Definition

Primary, Partition, Clustering

# 03

## CQL Data Types

Brief Review

# 04

## Data Modeling Process

Methodology, Notation, Tool

# 05

## Optimization Techniques

Five important optimizations

# 06

## What's next?

Homework, Next Session



Agenda

# Key Definition



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

## PARTITIONS TABLE.

Defines data distribution strategy

```
CREATE TABLE sensor_data.temperatures_by_sensor (
    sensor      TEXT,
    date        DATE,
    timestamp   TIMESTAMP,
    value       FLOAT,
    PRIMARY KEY ((sensor, date), timestamp)
);
```

Partition key

Examples:

```
PRIMARY KEY ((sensor), timestamp);
```

```
PRIMARY KEY ((sensor), date, timestamp);
```

```
PRIMARY KEY ((sensor, timestamp));
```

Partition Key

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

Defines physical ordering on disk

```
CREATE TABLE sensor_data.temperatures_by_sensor (
    sensor      TEXT,
    date        DATE,
    timestamp   TIMESTAMP,
    value       FLOAT,
    PRIMARY KEY ((sensor, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

Clustering columns

PRIMARY KEY ((**sensor**));

Not Unique

PRIMARY KEY ((**sensor**), timestamp);

Not filter on dates

PRIMARY KEY ((**sensor**), value, timestamp);

Not sorted

PRIMARY KEY ((**sensor**), date, timestamp);



Clustering Columns

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

## UNIQUELY IDENTIFIES A ROW

```
CREATE TABLE sensor_data.temperatures_by_sensor (
    sensor      TEXT,
    date        DATE,
    timestamp   TIMESTAMP,
    value       FLOAT,
    PRIMARY KEY ((sensor, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

Primary key

Examples:

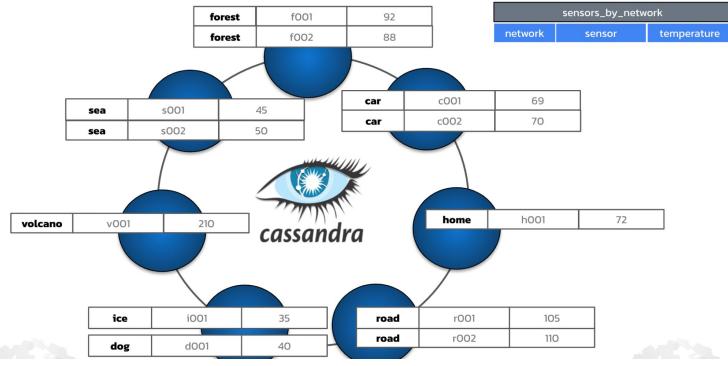
```
PRIMARY KEY ((sensor), timestamp);
```

```
PRIMARY KEY ((sensor, date), timestamp);
```

Primary Key

## Partition Keys:

Defines data distribution over the cluster



## Clustering Columns

Defines how data is physically stored (written on disk)

```
CREATE TABLE sensor_data.temperatures_by_sensor (
    sensor      TEXT,
    date        DATE,
    timestamp   TIMESTAMP,
    value       FLOAT,
    PRIMARY KEY ((sensor, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

```
PRIMARY KEY ((sensor, date), timestamp);
```

```
SELECT * FROM temperatures_by_sensor ...
```

```
WHERE sensor = ?;
```

```
WHERE sensor > ?;
```

```
WHERE sensor = ? AND date > ?:
```

```
WHERE sensor = ? AND date = ? AND timestamp > ?;
```

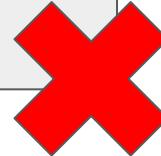
## Important:

Once created, the data model cannot be changed! You will need new tables and migration. Stay lazy, design it right in advance!

```
ALTER TABLE temperatures_by_sensor  
ADD season TEXT;
```



```
ALTER TABLE temperatures_by_sensor  
DROP PRIMARY KEY  
ADD PRIMARY KEY (sensor, timestamp)
```



Consequence II: Schema Immutability

# QUESTION

Why don't we use sequential IDs in Cassandra?



## [My]SQL

```
CREATE TABLE table_name (
    column1 integer NOT NULL AUTO_INCREMENT,
    ...
);
```

```
mysql> INSERT INTO table_name ...
mysql> SELECT LAST_INSERT_ID();
```

*Result: 99 (last value + 1)*

Usually ID generated by a database

## Cassandra

```
CREATE TABLE todoitems (
    item_id UUID,
    ...
);
```

```
cql> INSERT INTO todoitems ( item_id )
      VALUES ( UUID() );
cql> SELECT item_id FROM todoitems ...
```

*Result: 123e4567-e89b-12d3-a456-426614174000*

Usually UUID generated in an application code

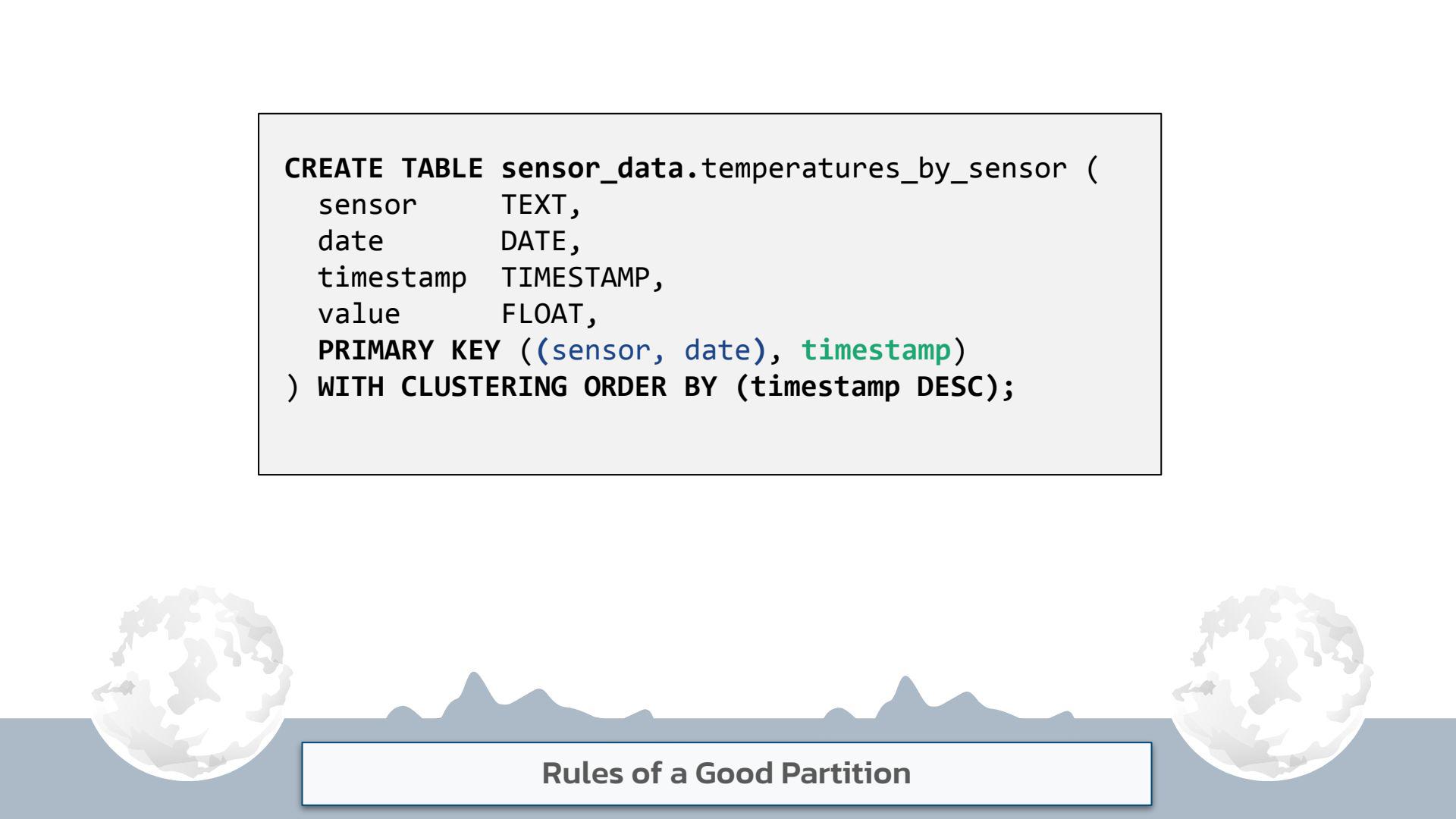


Different Approaches

# Rules for good partitioning



```
CREATE TABLE sensor_data.temperatures_by_sensor (
    sensor      TEXT,
    date        DATE,
    timestamp   TIMESTAMP,
    value       FLOAT,
    PRIMARY KEY ((sensor, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```



Rules of a Good Partition

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

**Q:** Show temperature evolution over time for **sensor X** On july 6th 2022

```
PRIMARY KEY ((sensor, timestamp));
```



```
PRIMARY KEY ((sensor), timestamp);
```



Rules of a Good Partition

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

# BUCKETING

PRIMARY KEY ((sensor), timestamp);



PRIMARY KEY ((sensor, month), timestamp);



- Up to 2 billion cells per partition
- Up to ~100k values in a partition
- Up to ~100MB in a Partition

Rules of a Good Partition

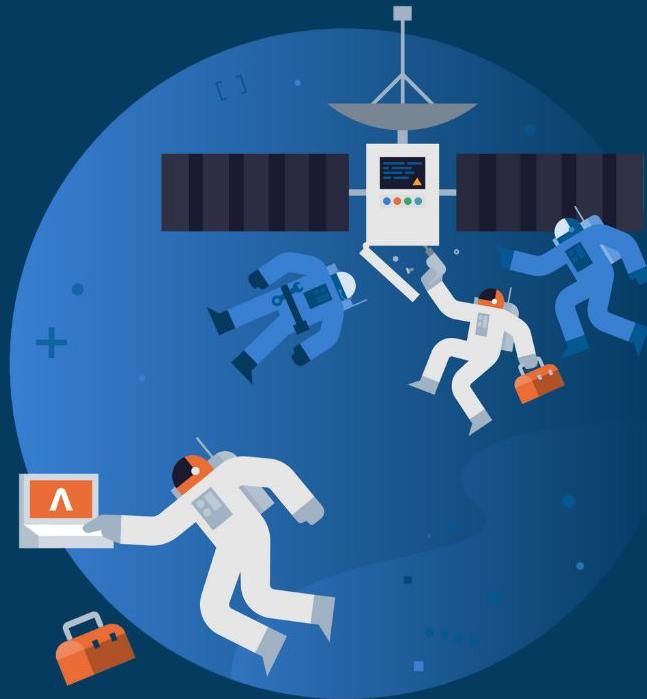
- ❖ Store together what you retrieve together
- ❖ Avoid big partitions
- ❖ **Avoid hot partitions**

```
PRIMARY KEY ((date), sensor, timestamp);
```



```
PRIMARY KEY ((date, sensor), timestamp);
```





# Lab 2

## Dynamic Bucketing

[https://github.com/datastaxdevs/  
workshop-cassandra-data-modeling](https://github.com/datastaxdevs/workshop-cassandra-data-modeling)



# 01



## Data Storage Overview

Data Distribution & Organization

# 02

## Key Definition

Primary, Partition, Clustering

# 03

## CQL Data Types

Brief Review

# 04

## Data Modeling Process

Methodology, Notation, Tool

# 05

## Optimization Techniques

Five important optimizations

# 06

## What's next?

Homework, Next Session



Agenda

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

- Collection types
  - `set<int>`
  - `list<text>`
  - `map<int, text>`
- User-Defined Types (UDTs)

```
CREATE TYPE address (
    street text,
    city text,
    state text,
);
```
- Tuple types
  - `tuple <uuid, text, decimal>`
- Custom types
  - Your own implementation as a Java class (not recommended)



More Types



# Homework

## Working with Data Types

[https://github.com/datastaxdevs/  
workshop-cassandra-data-modeling](https://github.com/datastaxdevs/workshop-cassandra-data-modeling)

```
yyyy-mm-dd HH:mm  
yyyy-mm-dd HH:mm:ss  
yyyy-mm-dd HH:mmZ  
yyyy-mm-dd HH:mm:ssZ  
yyyy-mm-dd 'T' HH:mm  
yyyy-mm-dd 'T' HH:mmZ  
yyyy-mm-dd 'T' HH:mm:ss  
yyyy-mm-dd 'T' HH:mm:ssZ  
yyyy-mm-dd 'T' HH:mm:ss.fffffffZ  
yyyy-mm-dd  
yyyy-mm-ddZ
```

- **Encoded as 64-bit signed integers representing a number of milliseconds since the standard base time known as the epoch**
- **String with ISO 8601 formats**

# Working with Collections

## List, Map, Set



## // Definition

```
CREATE TABLE IF NOT EXISTS table_with_list (
    uid      uuid,
    items    list<text>,
    PRIMARY KEY (uid)
);
```

## // Insert

```
INSERT INTO table_with_list(uid,items)
VALUES (c7133017-6409-4d7a-9479-07a5c1e79306, ['a', 'b', 'c']);
```

## // Replace

```
UPDATE table_with_list SET items = ['d', 'e']
WHERE uid = c7133017-6409-4d7a-9479-07a5c1e79306;
```

```
// Show result
SELECT * FROM table_with_list ;

// Append to list
UPDATE table_with_list SET items = items + ['f']
WHERE uid = c7133017-6409-4d7a-9479-07a5c1e79306;

// Replace an element (not available in Astra because read before write)
UPDATE table_with_list SET items[0] = ['g']
WHERE uid = c7133017-6409-4d7a-9479-07a5c1e79306;
```



## // Definition

```
CREATE TABLE IF NOT EXISTS table_with_set (
    uid      uuid,
    animals  set<text>,
    PRIMARY KEY (uid)
);
```

## // Insert

```
INSERT INTO table_with_set(uid,animals)
VALUES (87fad746-4adf-4107-9858-df8643564186, {'spider', 'cat', 'dog'});
```

## // Replace

```
UPDATE table_with_set SET animals = {'pangolin', 'bat'}
WHERE uid = 87fad746-4adf-4107-9858-df8643564186;
```

```
// Show result
```

```
SELECT * FROM table_with_set;
```

```
// Append to Set
```

```
UPDATE table_with_set SET animals = animals + {'sheep'}  
WHERE uid = 87fad746-4adf-4107-9858-df8643564186;
```



## Collections Set



### // Definition

```
CREATE TABLE IF NOT EXISTS table_with_map (
    uid          text,
    dictionary   map<text, text>,
    PRIMARY KEY (uid)
);
```

### // Insert

```
INSERT INTO table_with_map(uid, dictionary)
VALUES ('fr_en', {'fromage':'cheese', 'vin':'wine', 'pain':'bread'});
```

### // Replace

```
UPDATE table_with_map SET dictionary = {'saucisse': 'sausage'}
WHERE uid = 'fr_en';
```

```
// Show result
```

```
SELECT * FROM table_with_map;
```

```
// Append to Map
```

```
UPDATE table_with_map SET dictionary = dictionary + {'frites':'fries'}  
WHERE uid = 'fr_en';
```



## Collections Map



# Working with “Advanced Data Types” Udt, Counters



```
// Definition  
CREATE TYPE IF NOT EXISTS udt_address (  
    street text,  
    city text,  
    state text,  
);
```

```
// Use the UDT in a table  
CREATE TABLE IF NOT EXISTS table_with_udt (  
    uid      text,  
    address  udt_address,  
    PRIMARY KEY (uid)  
);
```

```
// INSERT (not quote on field names like street)
INSERT INTO table_with_udt(uid, address)
VALUES ('superman', {street:'daily planet',city:'metropolis',state:'CA'});
```

// Replace

```
UPDATE table_with_udt
SET address = {street:'pingouin alley',city:'antarctica',state:'melting'}
WHERE uid = 'superman';
```

// Replace 1 fields

```
UPDATE table_with_udt
SET address.state = 'melt'
WHERE uid = 'superman';
```

## Definition

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

## Constraints

- **Cannot be...**
  - part of primary key
  - mixed with other types in table
  - inserted
  - updated with a value
- **Updates are not idempotent**
- **Should not be used for precise values**



Counters

// Definition

```
CREATE TABLE IF NOT EXISTS table_with_counters (
    handle      text,
    following   counter,
    followers   counter,
    notifications counter,
    PRIMARY KEY (handle)
);
```

// You have a new follower

```
UPDATE table_with_counters SET followers = followers + 1
WHERE handle = 'clunven';
```



Counters

```
// Some counters are... null
SELECT * from table_with_counters;

// Set to 0...but set is not valid
UPDATE table_with_counters
SET following = following + 0, notifications = notifications + 0
WHERE handle = 'clunven';

// Following someone
UPDATE table_with_counters SET following = following + 1
WHERE handle = 'clunven';

// You have a new message
UPDATE table_with_counters SET notifications = notifications + 1
WHERE handle = 'clunven';
```



## Counters



# Hands-on Data Types

Github ?



# 01



## Data Storage Overview

Data Distribution & Organization

# 02

## Key Definition

Primary, Partition, Clustering

# 03

## CQL Data Types

Brief Review

# 04

## Data Modeling Process

Methodology, Notation, Tool

# 05

## Optimization Techniques

Five important optimizations

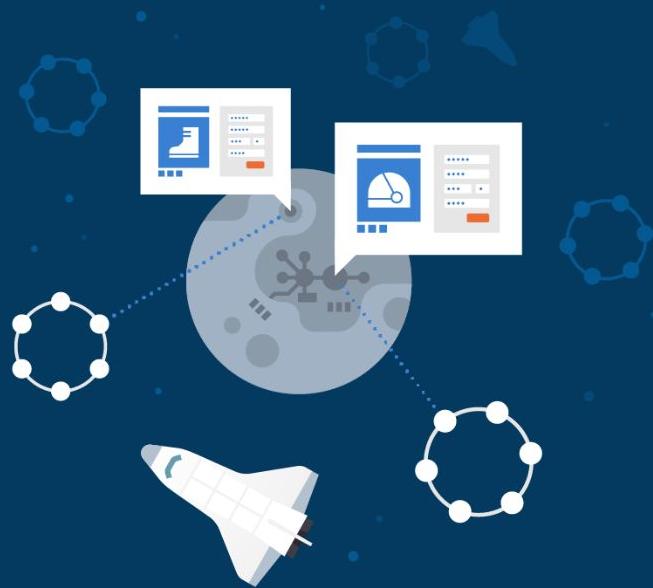
# 06

## What's next?

Homework, Next Session



Agenda



## Normalization vs Denormalization

"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



## Normalization

Employees

userId	deptId	firstName	lastName
1	1	Edgar	Codd
2	1	Raymond	Boyce

Departments

departmentId	department
1	Engineering
2	Math



"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	Engineering
3	Sage	Lahja	Math
4	Juniper	Jones	Botany

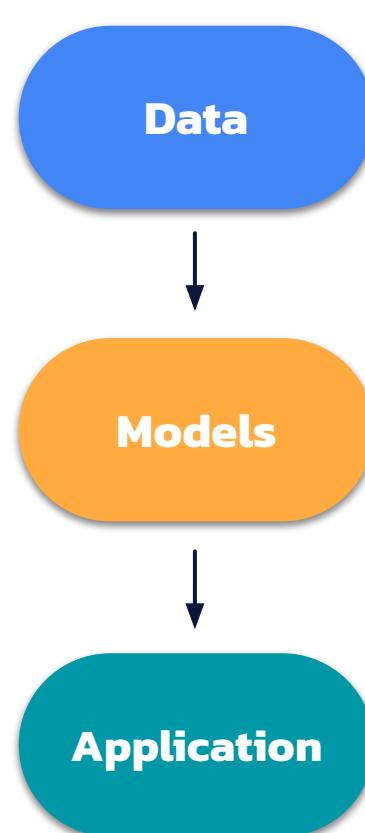
departmentId	department
1	Engineering
2	Math

Denormalization



## Modelling Approach

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 normalized data from multiple tables



A large cyan cylinder is positioned above two tables. A line connects the cylinder to the first table, and another line connects the cylinder to the second table.

Employees		
userId	firstName	lastName
1	Edgar	Codd
2	Raymond	Boyce

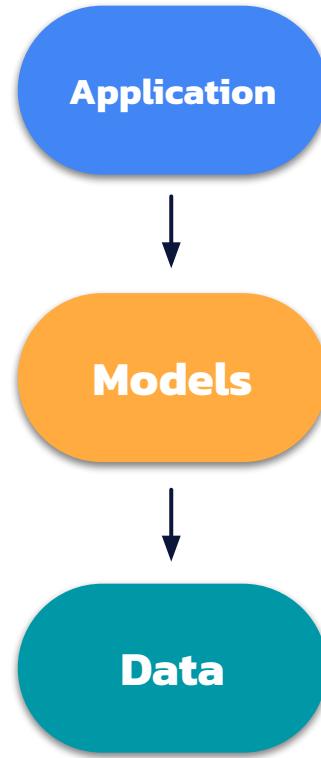
  

Departments	
departmentId	department
1	Engineering
2	Math

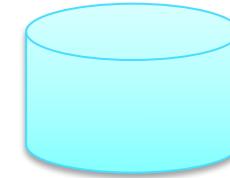


Relational Data Modelling

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



Employees			
userId	firstName	lastName	department
1	Edgar	Codd	Engineering
2	Raymond	Boyce	Math
3	Sage	Lahja	Math
4	Juniper	Jones	Botany



# Data Modelling Methodology

by Dr. Artem Chebotko

Modelling process  
Step by Step



- Collection and analysis of **data requirements**
- Identification of participating **entities and relationships**
- Identification of **data access patterns**
- A particular way of **organizing and structuring data**
- Design and specification of a **database schema**
- Schema **optimization** and data **indexing** techniques



Data Quality: completeness consistency accuracy  
Data Access: queryability efficiency scalability



What is Data Modelling ?

## **Modeling principle 1: "Know your data"**

- Key and cardinality constraints are fundamental to schema design

## **Modeling principle 2: "Know your queries"**

- Queries drive schema design

## **Modeling principle 3: "Nest data"**

- Data nesting is the main data modeling technique

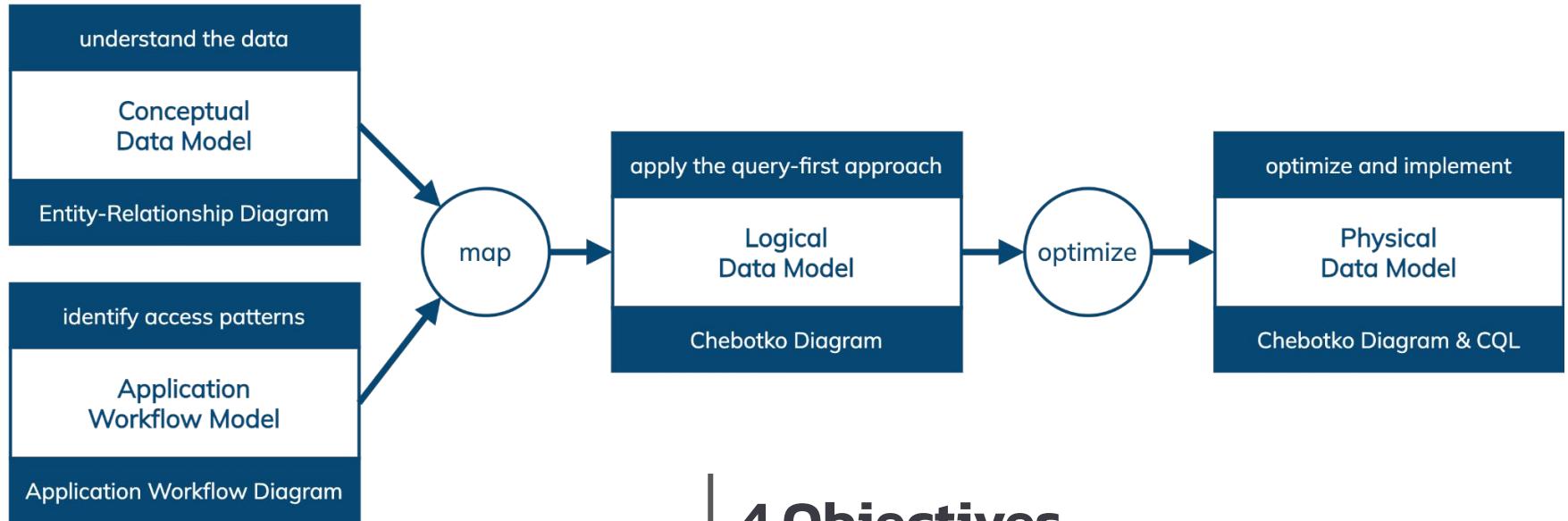
## **Modeling principle 4: "Duplicate data"**

- Better to duplicate than to join



**Cassandra Data Modelling Principles**





**4 Objectives,  
4 Models  
2 Transitions**



**4/4/2 Like at soccer (= *real* football)**

understand the data

### Conceptual Data Model

Entity-Relationship Diagram

Analyze the Domain

identify access patterns

### Application Workflow Model

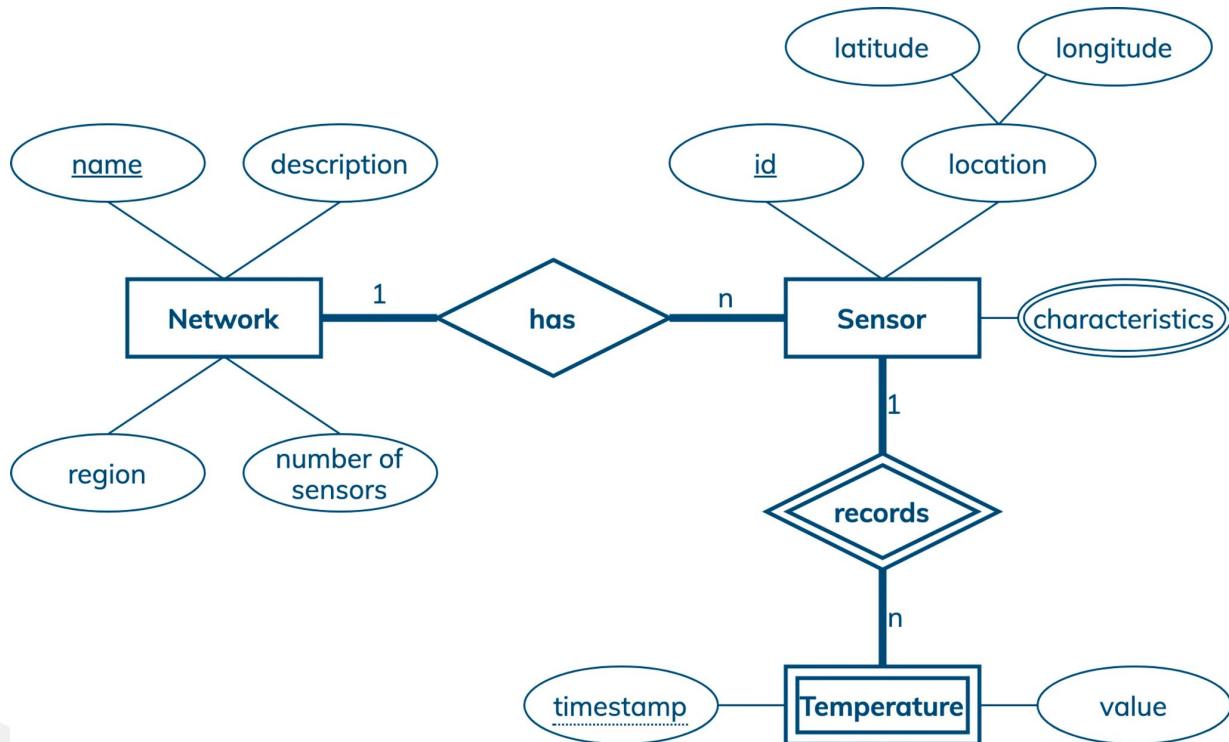
Application Workflow Diagram

Analyze Customer Workflows

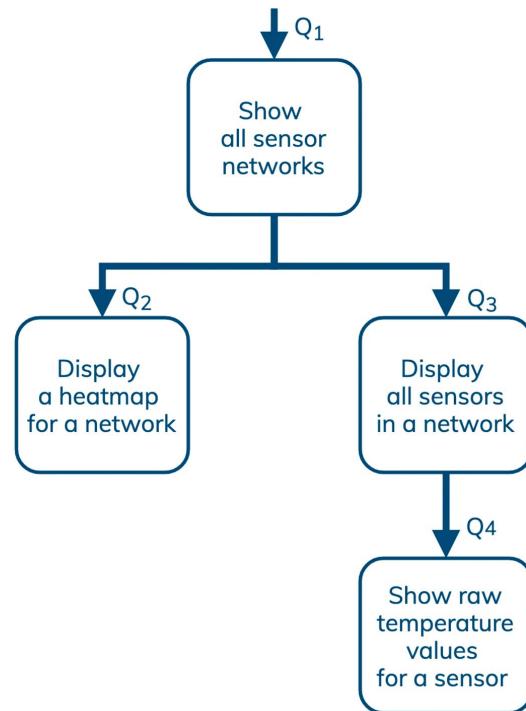


## Data Modelling Methodology: Step I





Entity-Relationship Diagram



#### Data access patterns

Q1: Find information about all networks; order by name (asc)

Q2: Find hourly average temperatures for every sensor in a specified network for a given date range; order by date (desc) and hour (desc)

Q3: Find information about all sensors in a specified network

Q4: Find raw measurements for a particular sensor on a specified date; order by timestamp (desc)

Application Workflow Diagram

apply the query-first approach

Logical  
Data Model

Chebotko Diagram

Design queries and build  
tables based on the queries



Data Modelling Methodology: Step II

#### Mapping rule 1: "Entities and relationships"

- Entity and relationship types map to tables

Based on  
a conceptual  
data model

#### Mapping rule 2: "Equality search attributes"

- Equality search attributes map to the beginning columns of a primary key

Based on  
a query

#### Mapping rule 3: "Inequality search attributes"

- Inequality search attributes map to clustering columns

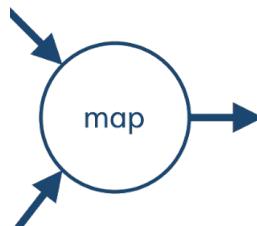
#### Mapping rule 4: "Ordering attributes"

- Ordering attributes map to clustering columns

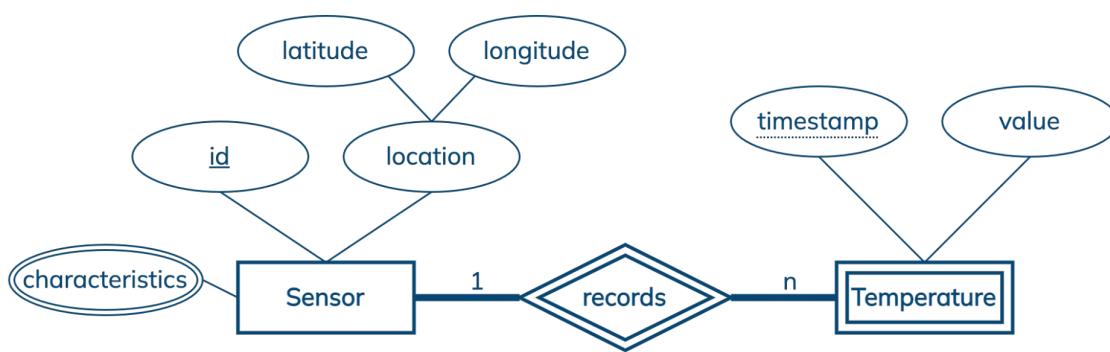
Based on  
a conceptual  
data model

#### Mapping rule 5: "Key attributes"

- Key attributes map to primary key columns



## Mapping Rules

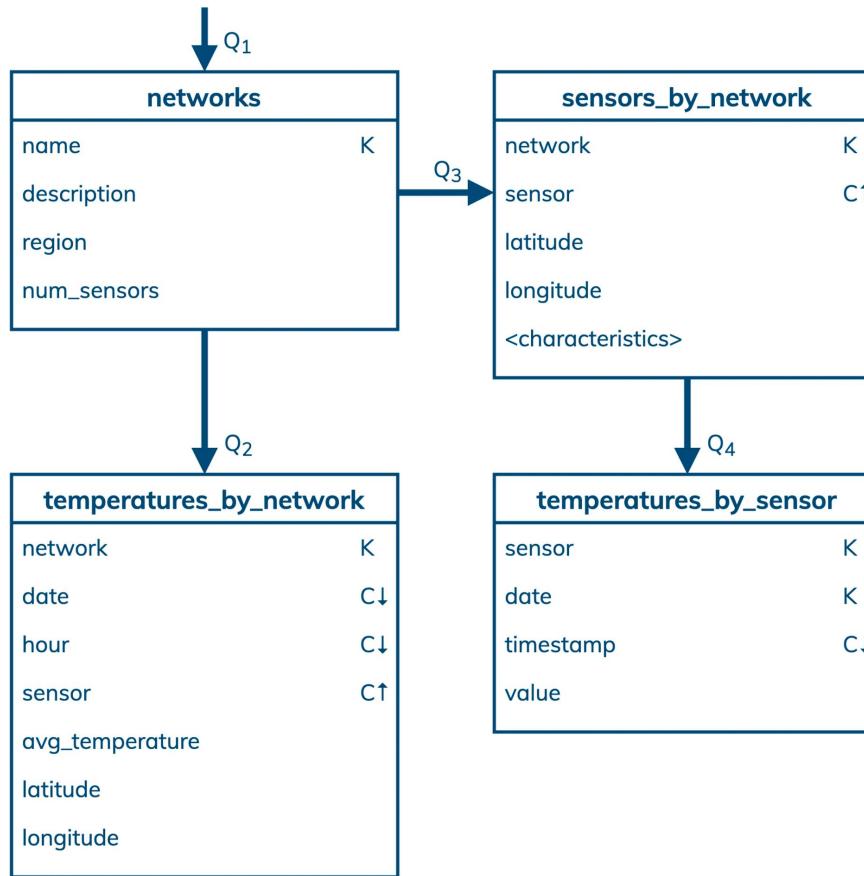


Q5: Find raw measurements for a given location and a date/time range; order by timestamp (desc)

The table shows the results of applying five mapping rules (MR1 to MR5) to the "temps\_by\_sensor" table:

	MR1	MR2	MR3	MR4	MR5
latitude					
longitude					
timestamp				C↑	C↓
value					

## Example: Applying Mapping Rules



### Data access patterns

- Q1: Find information about all networks; order by name (asc)
- Q2: Find hourly average temperatures for every sensor in a specified network for a given date range; order by date (desc) and hour (desc)
- Q3: Find information about all sensors in a specified network
- Q4: Find raw measurements for a particular sensor on a specified date; order by timestamp (desc)



Chebotko Diagram

optimize and implement

## Physical Data Model

Chebotko Diagram & CQL

Optimize and Create



Data Modelling Methodology: Step III

- Partition size limits and splitting large partitions
- Data duplication and batches
- Indexes and materialized views\*
- Concurrent data access and lightweight transactions\*
- Dealing with tombstones\*

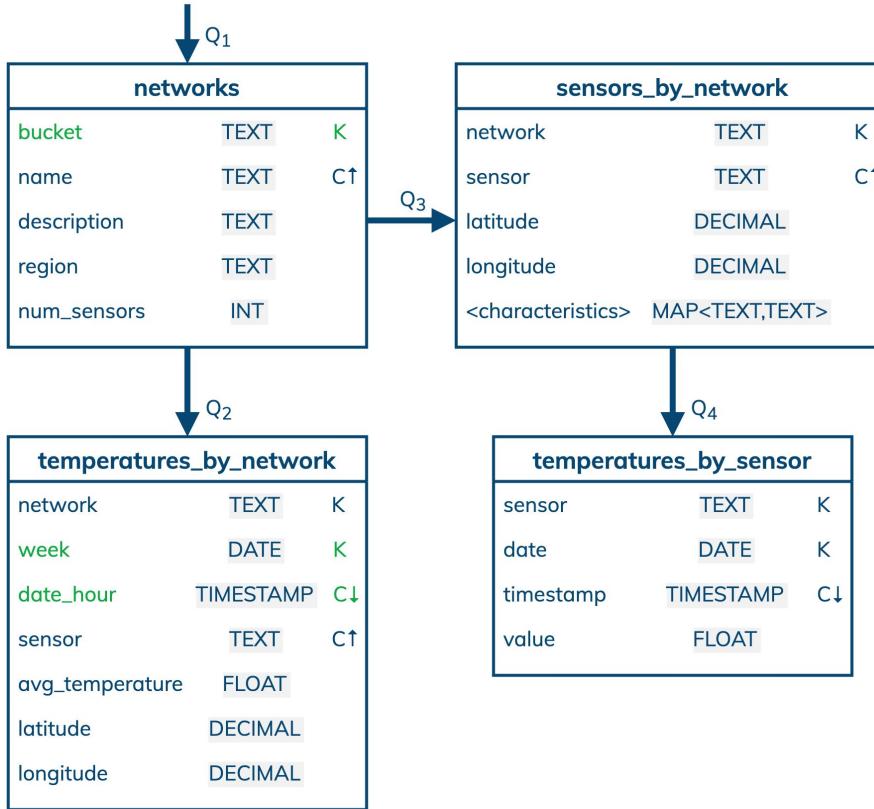


\* Explained in details at our free DS220 course at [academy.datastax.com](http://academy.datastax.com)



## Optimization Techniques





## Physical Data Model: Chebotko Diagram

Q1

```
CREATE TABLE networks (
    bucket TEXT,
    name TEXT,
    description TEXT,
    region TEXT,
    num_sensors INT,
    PRIMARY KEY ((bucket),name)
);
```

Q3

```
CREATE TABLE sensors_by_network (
    network TEXT,
    sensor TEXT,
    latitude DECIMAL,
    longitude DECIMAL,
    characteristics MAP<TEXT,TEXT>,
    PRIMARY KEY ((network),sensor)
);
```

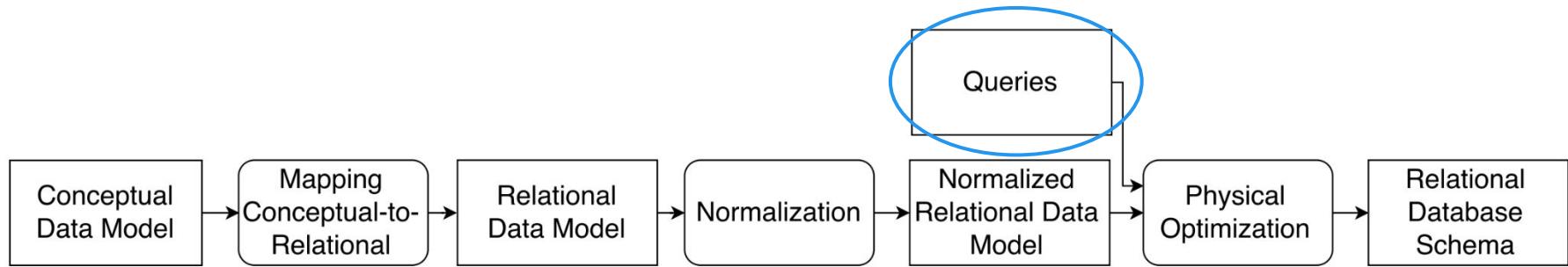
```
CREATE TABLE temperatures_by_network (
    network TEXT,
    week DATE,
    date_hour TIMESTAMP,
    sensor TEXT,
    avg_temperature FLOAT,
    latitude DECIMAL,
    longitude DECIMAL,
    PRIMARY KEY ((network,week),date_hour,sensor)
) WITH CLUSTERING ORDER BY (date_hour DESC, sensor ASC);
```

Q2

Q4

```
CREATE TABLE temperatures_by_sensor (
    sensor TEXT,
    date DATE,
    timestamp TIMESTAMP,
    value FLOAT,
    PRIMARY KEY ((sensor,date),timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

Cassandra Query Language



(a) Relational Data Modeling



(b) Cassandra Data Modeling

Relational vs Cassandra Modeling Process



# Demo

## KDM

[https://github.com/datastaxdevs/  
workshop-cassandra-data-modeling](https://github.com/datastaxdevs/workshop-cassandra-data-modeling)





# Lab 3 (Optional)

## Sensor Data Modeling

[https://github.com/datastaxdevs/  
workshop-cassandra-data-modeling](https://github.com/datastaxdevs/workshop-cassandra-data-modeling)

## Data Modeling Principles

- Know your data
- Know your queries
- Nest data
- Duplicate data

## The Most Efficient Access Pattern

A query is satisfied by accessing one partition

## Primary Key

- Data uniqueness
- Data distribution
- Data queryability

## The Cassandra Data Modeling Methodology

It is not more complex than the relational data modeling methodology



Key Takeaways

# 01



**Data Storage Overview**  
Data Distribution & Organization

# 02

**Key Definition**  
Primary, Partition, Clustering

# 03

**CQL Data Types**  
Brief Review

# 04

**Data Modeling Process**  
Methodology, Notation, Tool

# 05

**Optimization Techniques**  
Five important optimizations

# 06

**What's next?**  
Homework, Next Session



Agenda



# Partition size limits and splitting large partitions



- **Theoretical**
  - 2 billion values
  - Node disk size
- **Practical**

Cassandra 2	Cassandra 3 and 4
100,000 values	10 x 100,000 values
100 MB	10 x 100 MB



## Partition Size Limits



$$N_v = N_{cv} + N_{rv} + N_{sv}$$

$$N_{cv} = N_r \times N_{cc}$$

$$N_{rv} = N_r \times N_{rc}$$

$$N_{sv} = N_{sc}$$

$$S_p = \sum_i \text{sizeOf}(c_{k_i}) + \sum_j \text{sizeOf}(c_{s_j}) + N_r \times \left( \sum_k \text{sizeOf}(c_{r_k}) + \sum_l \text{sizeOf}(c_{c_l}) \right) + (N_{rv} + N_{sv}) \times \text{sizeOf}(t_{avg})$$

$N_v$  – number of **values** in a partition

$N_{cv}$  – number of **clustering column values** in a partition

$N_{rv}$  – number of **regular column values** in a partition

$N_{sv}$  – number of **static column values** in a table definition

$N_r$  – number of **rows** in a partition

$N_{cc}$  – number of **clustering columns** in a table definition

$N_{rc}$  – number of **regular columns** in a table definition

$N_{sc}$  – number of **static columns** in a table definition

$S_p$  – size of a partition in bytes

$\text{sizeOf}$  – size (in bytes) of a CQL data type

$c_k$  – **partition key column** in a table definition

$c_s$  – **static column** in a table definition

$c_r$  – **regular column** in a table definition

$c_c$  – **clustering column** in a table definition

$N_r$  – number of **rows** in a partition

$\text{sizeOf}(t_{avg})$  – average size (in bytes) of a timestamp delta associated with a value

## A user with 1,000 reviews == a partition with 1,000 rows

Reviews_by_user		
user	UUID	K
review	TIMEUUID	C↓
rating	FLOAT	
title	TEXT	
user_name	TEXT	S
user_email	TEXT	S

Values:

+ 1000  
+ 1000  
+ 1000  
+ 1  
+ 1  

---

3002

Bytes:

16  
1000x16  
1000x 4  
1000x60  
12  
20  

---

80048 + 2002x8 = **96064**

Example of Partition size computation

```
$ nodetool flush library reviews_by_user  
$ nodetool tablestats -H library.reviews_by_user
```

Keyspace : library

...

Table: reviews\_by\_user

SSTable count: 1

...

Number of partitions (estimate): 1

...

Compacted partition minimum bytes: 88149

Compacted partition maximum bytes: 105778

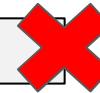
Compacted partition mean bytes: 105778

...

## Getting a Partition Size Empirically

- **Solution: introduce an additional column to a partition key**
  - Use an existing or derived column – convenience
  - Use an artificial “bucket” column – more control
- **Cons: supported access patterns may change**

```
PRIMARY KEY ((sensor), timestamp);
```



```
PRIMARY KEY ((sensor, month), timestamp);
```



## Millions of artifacts with the same keyword across different venues and years

Artifacts_by_keyword		
keyword	TEXT	K
year	INT	C↓
artifact	TEXT	C↑
type	TEXT	
title	TEXT	
authors	LIST<TEXT>	
keywords	SET<TEXT>	
venue	TEXT	



Artifacts_by_keyword		
keyword	TEXT	K
year	INT	K
artifact	TEXT	C↑
type	TEXT	
title	TEXT	
authors	LIST<TEXT>	
keywords	SET<TEXT>	
venue	TEXT	



Artifacts_by_keyword		
keyword	TEXT	K
year	INT	K
bucket	INT	K
artifact	TEXT	C↑
type	TEXT	
title	TEXT	
authors	LIST<TEXT>	
keywords	SET<TEXT>	
venue	TEXT	



Example



# Data duplication and batches



- **Types of data duplication**
  - Duplication across tables
  - Duplication across partitions of the same table
  - Duplication across rows of the same partition
- **Duplication factor: constant ✓ vs non-constant ✗**



Data Duplication



- Each artifact is stored once in Artifacts
- Each artifact is stored once in Artifacts\_by\_venue
- Duplication factor = 2

Artifacts	
artifact	K
type	
title	
authors (list)	
keywords (set)	
venue	
year	

Artifacts_by_venue	
venue	K
year	C↓
artifact	C↑
type	
title	
authors (list)	
keywords (set)	



Duplication Across Tables



- An artifact with 5 authors is stored in 5 different partitions
- Duplication factor = 5

Artifacts_by_author	
author	K
year	C↓
artifact	C↑
type	
title	
authors (list)	
keywords (set)	
venue	



## Duplication Across Partitions



- An artifact with 5 keywords is stored in 5 rows of the same partition
- An artifact with 5 authors is stored in 5 different partitions
- Duplication factor =  $5 \times 5 = 25$

Artifacts_by_author	
author	K
keyword	C↑
year	C↓
artifact	C↑
type	
title	
authors (list)	
keywords (set)	
venue	



Duplication across Rows



- **Users can annotate artifacts with keywords**
- An artifact with  $n$  keywords is stored in  $n$  rows of the same partition
- An artifact with 5 authors is stored in 5 different partitions
- Duplication factor =  $5 \times n$
  
  
  
  
- **Solution: place reasonable limits that can be increased gradually**

Artifacts_by_author	
author	K
keyword	C↑
year	C↓
artifact	C↑
type	
title	
authors (list)	
keywords (set)	
venue	



Non-constant duplication factors



- **Logged, single-partition batches**
  - Efficient and atomic
  - Use whenever possible
- **Logged, multi-partition batches**
  - Somewhat efficient and pseudo-atomic
  - Use selectively
- **Unlogged and counter batches**
  - Do not use

```
BEGIN BATCH  
  INSERT INTO ...;  
  INSERT INTO ...;  
  ...  
  APPLY BATCH;  
BEGIN BATCH  
  UPDATE ...;  
  UPDATE ...;  
  ...  
  APPLY BATCH;
```



- **Experimental feature**
- **Not always applicable**
- **May become inconsistent**
- **10% write penalty to a base table**

```
CREATE MATERIALIZED VIEW  
networks_by_region AS  
    SELECT * FROM networks  
    WHERE region IS NOT NULL  
        AND name IS NOT NULL  
PRIMARY KEY ((region), name);
```



Keeping Duplicated Data Consistent with MVs

# Indexes and materialized views (+ allow filtering)

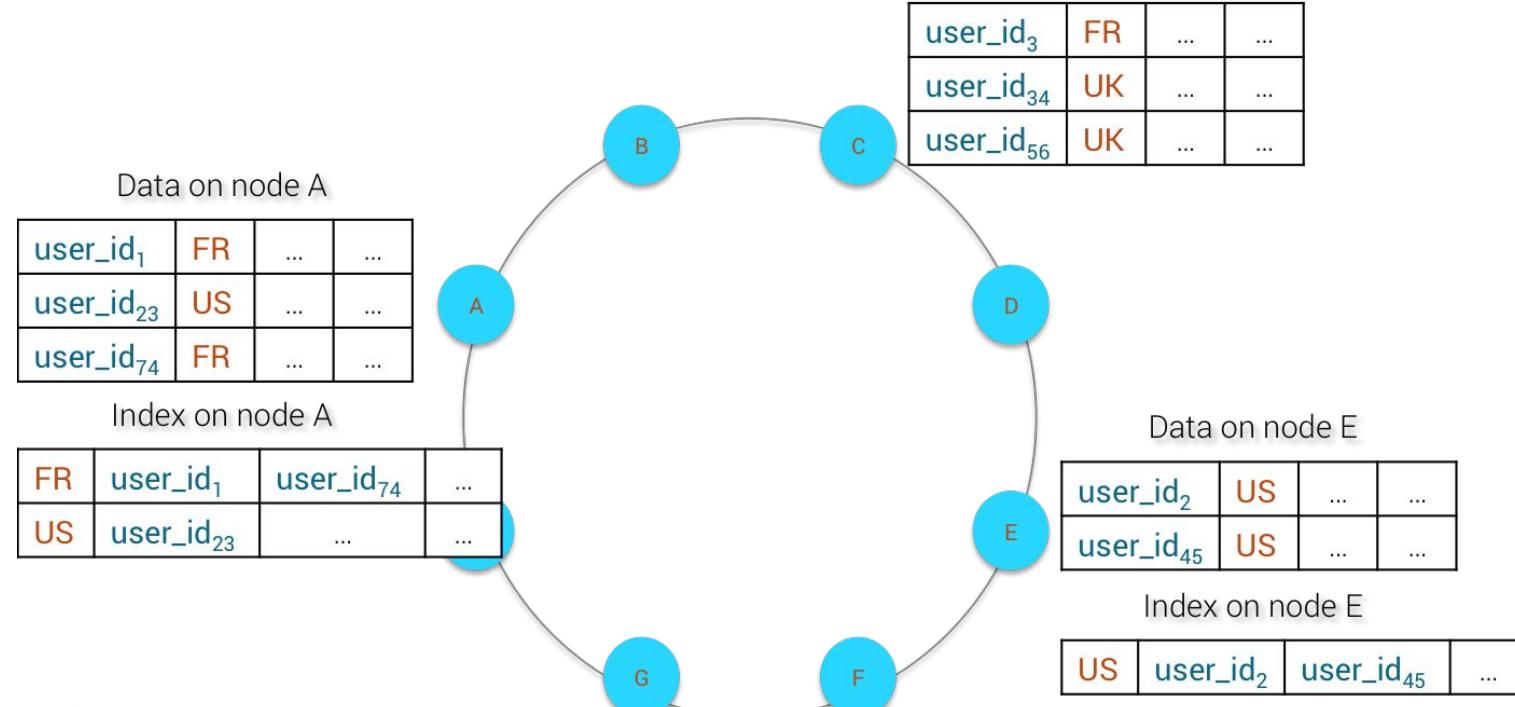


- **Distributed indexes: highly scalable**
  - **Tables ← the best solution available**
  - **Materialized views (experimental)**
- **Local indexes: not highly scalable**
  - **Secondary indexes (2i)**
  - **SSTable-attached secondary indexes (SASI)**
  - **Storage-Attached Indexes (SAI)**



## Indexing Mechanisms





[doanduyhai.com/blog/?p=13191](http://doanduyhai.com/blog/?p=13191)

## How Secondary Indexes Work

- **The best index available**
  - Faster writes
  - Less disk space
  - Multiple indexes can be used in one query
  - Reads can be expensive (similar limitations to 2is and SASIs)
- **Convenience, not performance**

```
CREATE CUSTOM INDEX users_by_name_sai  
ON users (name)  
USING 'org.apache.cassandra.index.sai.StorageAttachedIndex' ;
```



## Storage-Attached Indexes (SAI)



- **Real-time transactional query**: retrieving rows from a large multi-row partition, when both a **partition key** and an indexed column are used in a query.
- **Expensive analytical query**: retrieving a large number of rows from potentially many partitions, when only an indexed **low-cardinality column** is used in a query.



When to use secondary index (1 of 2)

- **Small clusters**: 3–10 nodes with RF=3
- **Infrequent queries**: may not affect the throughput much
- **Queries with LIMIT that can be quickly satisfied**: may not affect the throughput much
- **Convenience over performance**: simpler data modeling while potentially sacrificing throughput and query response time



When to use secondary index (2 of 2)



- **Queries on higher-cardinality columns**
  - Similar advantages as those of regular tables
  - Convenience of automatic view maintenance
  - Reads are as fast as for regular tables
- **Important limitations**
  - Restrictions on how PRIMARY KEY is constructed
  - 10% slower writes to the base table
  - Base-view inconsistencies
  - Use LOCAL\_QUORUM and higher for base table writes
  - Disabled in Astra DB, experimental



When to Use a Materialized View

- Generally, using **ALLOW FILTERING** is not recommended
- But scanning may be cheaper than indexing in some cases
  - Scanning within a small partition
  - Scanning within a large partition based on a low-cardinality column (most rows from the partition will be returned)
  - Scanning a table based on a low-cardinality column (most rows from the table will be returned) for analytical purposes



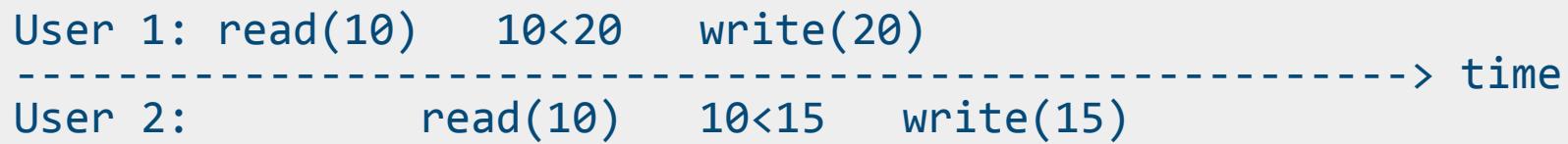
Allow Filtering



# Concurrent data access and lightweight transactions



- Multiple transactions reading and writing the same piece of data concurrently, possibly leaving the data in an inconsistent-with-reality state
- Examples: registering a new username, counting votes, updating account balance, updating an order status, updating the highest bid



Race conditions



- **Linearizable consistency ensures that concurrent transactions produce the same result as if they execute in a sequence, one after another**
- **Lightweight transactions (LWTs) in Cassandra and Astra DB**

User 1: read(10) 10<20 write(20)

-----> time

User 2: read(20) 20>15

User 1: read(15) 15<20 write(20)

-----> time

User 2: read(10) 10<15 write(15)



Linearizable Consistency



- **Guarantee linearizable consistency**
- **Use with race conditions and low data contention**
- **Require 4 coordinator-replica round trips**
- **May become a bottleneck with high data contention**

```
INSERT INTO ... VALUES ...
IF NOT EXISTS;

UPDATE ... SET ... WHERE ...
IF EXISTS | IF predicate [ AND ... ];

DELETE ... FROM ... WHERE ...
IF EXISTS | IF predicate [ AND ... ];
```



Lightweight transactions (LWTs)

- **Change a data model to eliminate race conditions**
  - Isolate computation by isolating data
  - Record each transaction separately and aggregate data later
- **Use a queue to control data contention**
  - Astra Streaming



Dealing with High Data Contention



# Tombstones



- **Written markers for deleted data**
  - **DELETEs**
  - **Data with TTL**
  - **UPDATEs and INSERTs with NULL values**
  - **“Overwritten” collections**



Tombstones



- Tombstone = additional data to store and read
- Query performance degrades, heap memory pressure increases
- **tombstone\_warn\_threshold**
  - Warning when more than 1,000 tombstones are scanned by a query
- **tombstone\_failure\_threshold**
  - Aborted query when more than 100,000 tombstones are scanned



1. Decrease the value of **gc\_grace\_seconds** (default is 864000 or 10 days)
  - Deleted data and tombstones can be purged during compaction after **gc\_grace\_seconds**
2. Run compaction more frequently
  - **nodetool compact keyspace tablename**



Tombstones Issue 1: Large Partitions

1. A replica node is unresponsive and receives no tombstone
2. Other replica nodes receive tombstones
3. The tombstones get purged after `gc_grace_seconds` and compaction
4. The unresponsive replica comes back online and resurrects data that was previously marked as deleted



- Run repairs within the `gc_grace_seconds` and on a regular basis
- `nodetool repair`
- Do not let a node to rejoin the cluster after the `gc_grace_seconds`



Tombstones Issue 2: Zombie or Resurrected Data

- **Design data models that avoid frequent deletes**
- **Truncating/dropping a table is better than deleting each partition in the table**
- **Deleting a partition is better than deleting each row in the partition**
- **Deleting a row is better than deleting each value in the row**



## Other Tombstones Considerations



## Partition Size

100K and 100MB

Bucketing

## Data duplication and batches

Avoid non-constant duplication factors

Single-partition batches are efficient

## Indexing

Materialized views are scalable but have other issues

Secondary indexes are good for searching within large partitions

## LWTs

Linearizable consistency

Race conditions and low data contention

Design data models that eliminate race conditions

## Tombstones

Design data models that avoid frequent deletions

Compact and repair to purge tombstones and avoid resurrected data



## Key Takeaways

# 01



**Data Storage Overview**  
Data Distribution & Organization

# 02

**Key Definition**  
Primary, Partition, Clustering

# 03

**CQL Data Types**  
Brief Review

# 04

**Data Modeling Process**  
Methodology, Notation, Tool

# 05

**Optimization Techniques**  
Five important optimizations

# 06

**What's next?**  
Homework, Next Session



Agenda

[datastax.com/learn/data-modeling-by-example](http://datastax.com/learn/data-modeling-by-example)

[killercoda.com/datastaxdevs/course/cassandra-data-modeling](http://killercoda.com/datastaxdevs/course/cassandra-data-modeling)



The grid contains nine cards, each with a title, a brief description, and a small icon:

- Sensor Data Modeling**: Learn how to create a data model for temperature monitoring sensor networks.
- Messaging Data Modeling**: Learn how to create a data model for an email system.
- Digital Library Data Modeling**: Learn how to create a data model for a digital music library.
- Investment Portfolio Data Modeling**: Learn how to create a data model for investment accounts or portfolios.
- Time Series Data Modeling**: Learn how to create a data model for time series data.
- Shopping Cart Data Modeling**: Learn how to create a data model for online shopping carts.
- Order Management Data Modeling**: Learn how to create a data model for an order management system.



**HomeWork**



- [medium.com/building-the-open-data-stack/data-modeling-in-cassandra-and-datastax-astra-db-3f89b9c133c9](https://medium.com/building-the-open-data-stack/data-modeling-in-cassandra-and-datastax-astra-db-3f89b9c133c9)
- [devm.io/iot/apache-cassandra-iot-174970](https://devm.io/iot/apache-cassandra-iot-174970)



Blog Posts





# Cassandra Data Modelling Homework



Welcome and thank you! Here you can submit your homework for the datastax developers "Data Modelling with Cassandra" workshop. In case of any questions please contact organizers at <https://dtsx.io/aleks> or just send an email to [aleksandr.volochnev@datastax.com](mailto:aleksandr.volochnev@datastax.com)

- Workshop materials: <https://github.com/datastaxdevs/workshop-intro-to-cassandra>
- Discord chat: <https://dtsx.io/discord>

Email \*

Valid email

This form is collecting emails. [Change settings](#)



DataStax **Developers**



# Join our 19k Discord Community

## DataStax Developers



!discord

[dtsx.io/discord](https://dtsx.io/discord)

The screenshot shows the DataStax Developers Discord server interface. The left sidebar lists various channels: Événements, # moderator-only, # WELCOME, start-here, code-of-conduct, # introductions, upcoming-events, useful-resources, # memes, # your-ideas, @ the-stage, # WORKSHOPS, # workshop-chat, # workshop-feedback, # workshop-materials, # upcoming-workshops, # ASTRADB, # getting-started, # astra-apis, # astra-development, # sample-applications, and # APACHE CASSANDRA. The main window is the # workshop-chat channel, which contains a message from a user named RIGGITYREKT asking about mixed version testing for a class and one node that is 5.0.15. Another message from Erick Ramirez follows, stating that mixed versions are not supported. The right sidebar shows a list of presenters and helpers, along with a list of members currently online.

Discord Community

# Thank You!

