DataStax

Developers

Apache Cassandra™ Data Modelling

Building efficient applications with Apache Cassandra

@aleks-volochnev

@HadesArchitect

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

Developer Advocate Lead

dtsx.io/aleks

cassandra

Aleksandr Volochnev

Developer Advocate

@ArtemChebotko

Astra DB

Astra STREAMING

PULSAR

cassandra

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

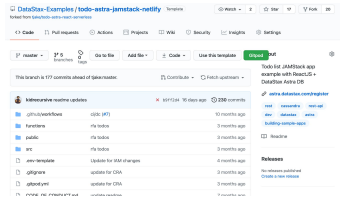
Artem Chebotko

Nothing to install !

SCENARIOS

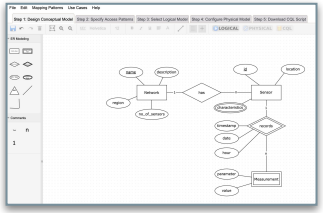
KLLR CODA BETA

Source code + exercises + slides



GitHub

DATA MODELING: KDM



http://kdm.kashliev.com

Hands-On Housekeeping

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

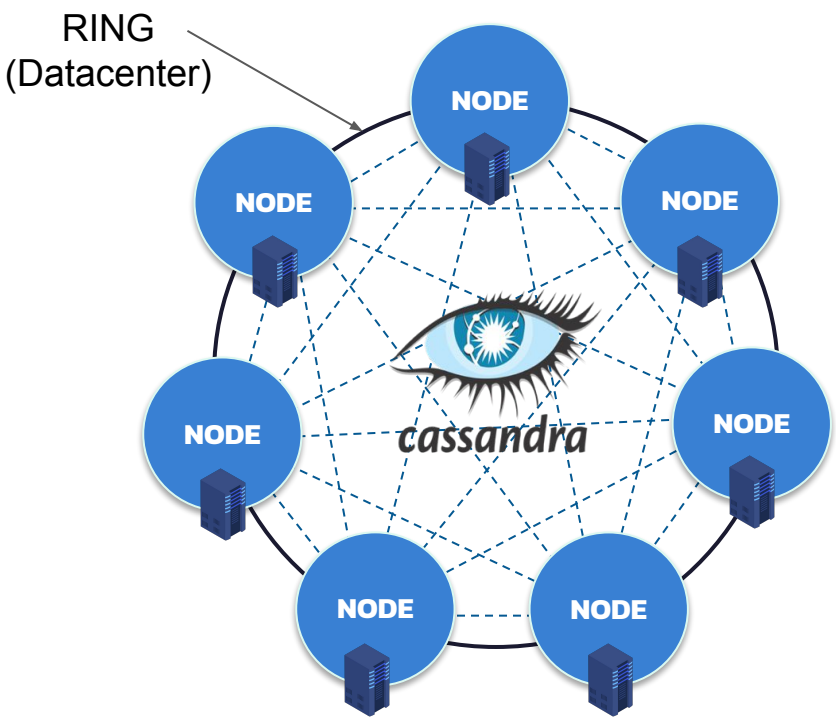
06

What's next?

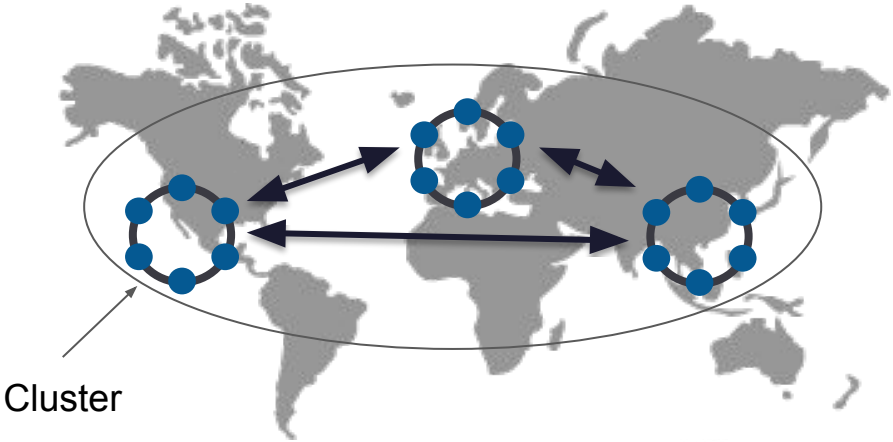
Homework, Next Session

Agenda

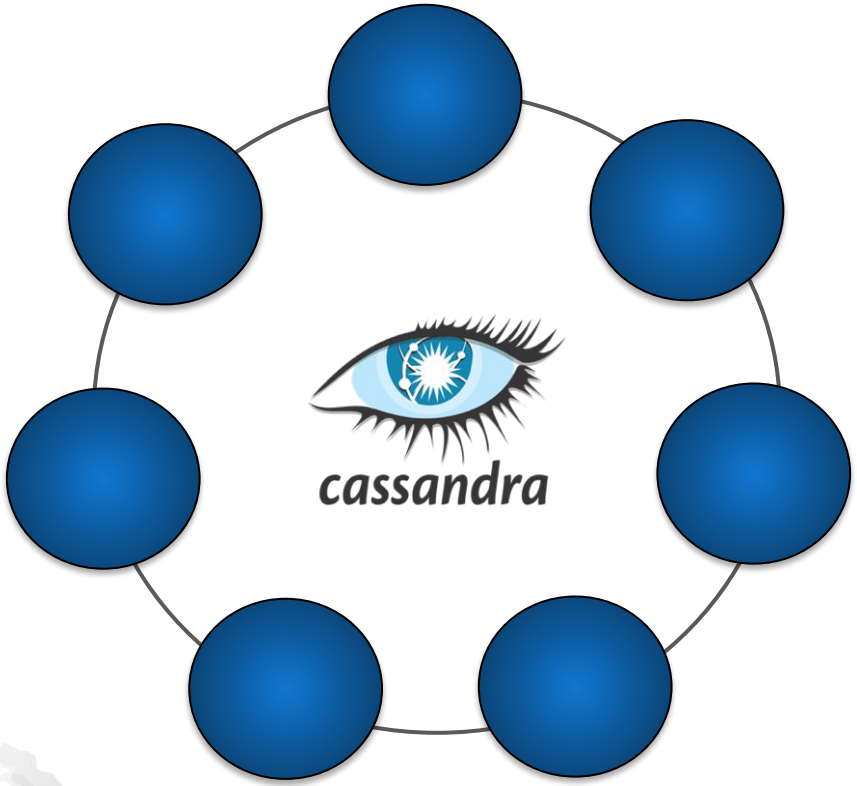
Data Distribution



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

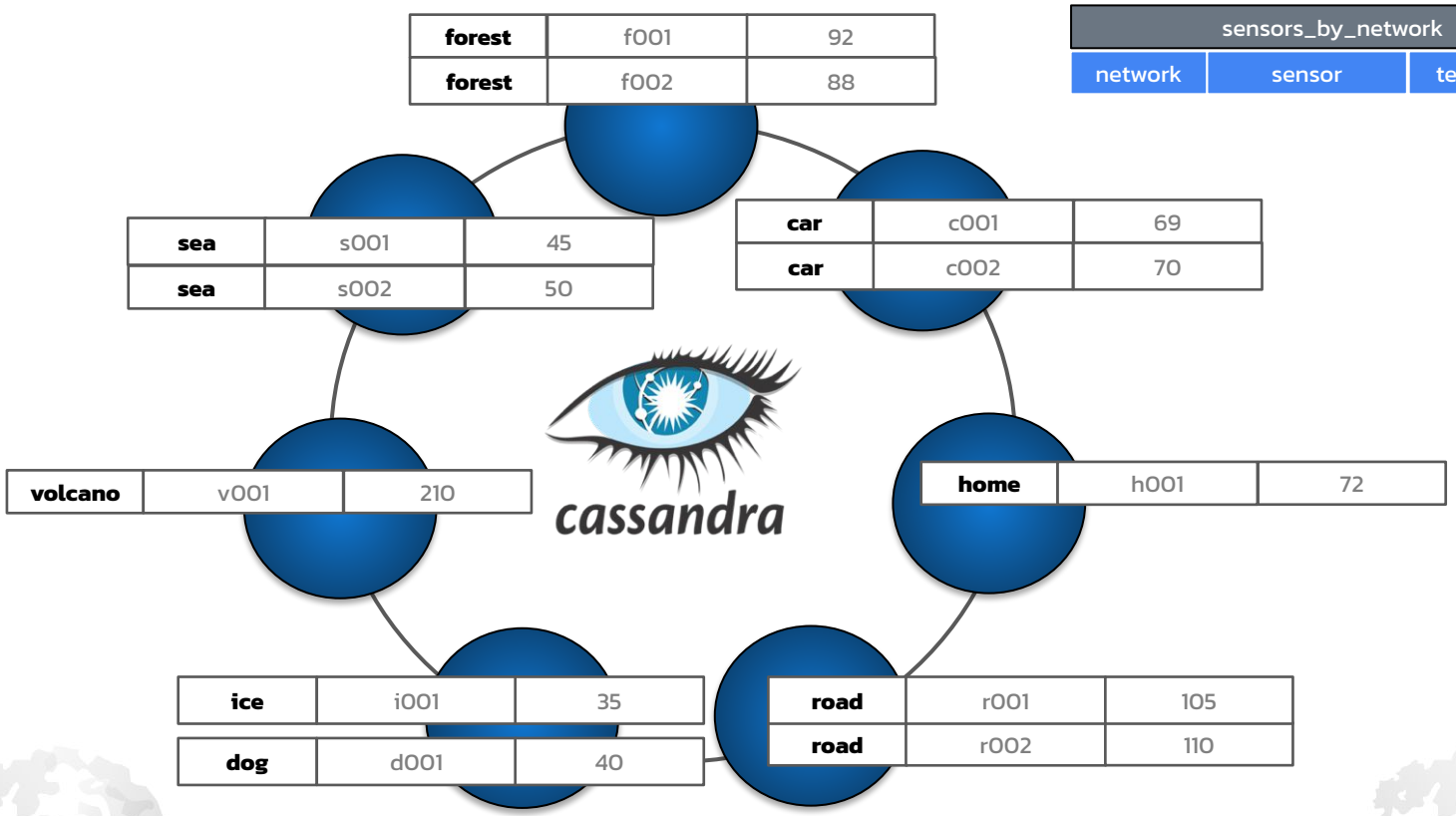


sensors_by_network		
network	sensor	temperature
forest	f001	92
forest	f002	88
volcano	v001	210
sea	s001	45
sea	s002	50
home	h001	72
car	c001	69
car	c002	70
dog	d001	40
road	r001	105
road	r002	110
ice	i001	35

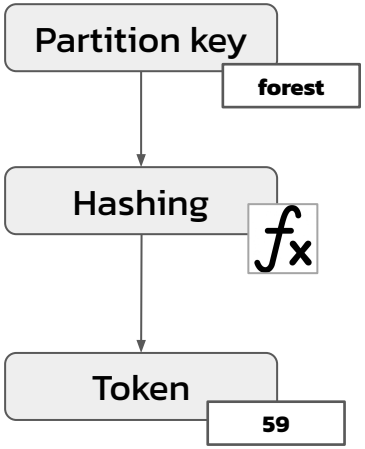
Partition Key

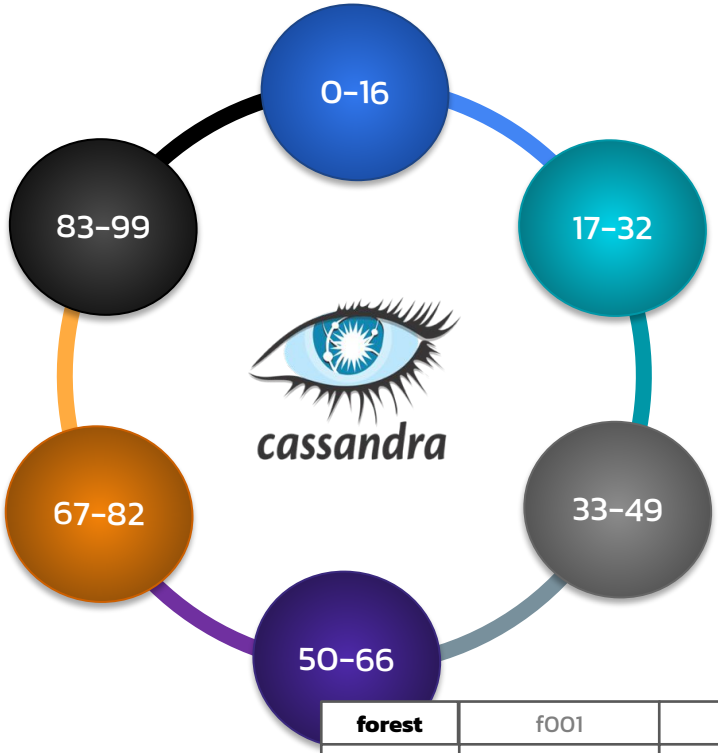
Primary Key

Data is organized as [Tables]



Data is organized as Distributed tables

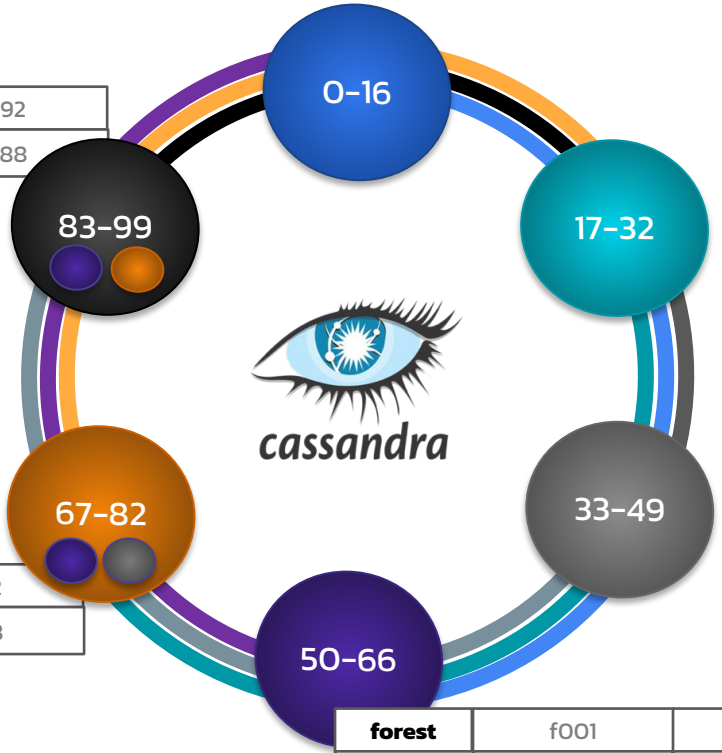




forest	f001	92
forest	f002	88

Per Table, each node owns a range of tokens

RF = 3



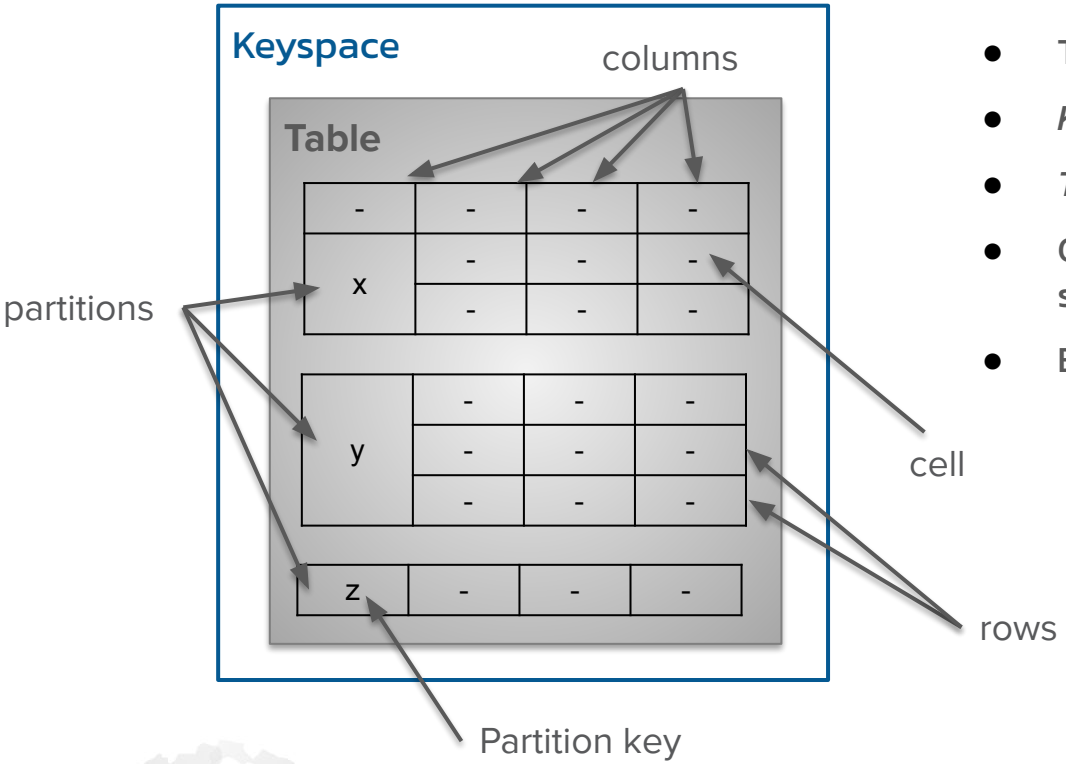
forest	f001	92
forest	f002	88

forest	f001	92
forest	f002	88

forest	f001	92
forest	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 ∈ Row ∈ Partition ∈ Table ∈ Keyspace

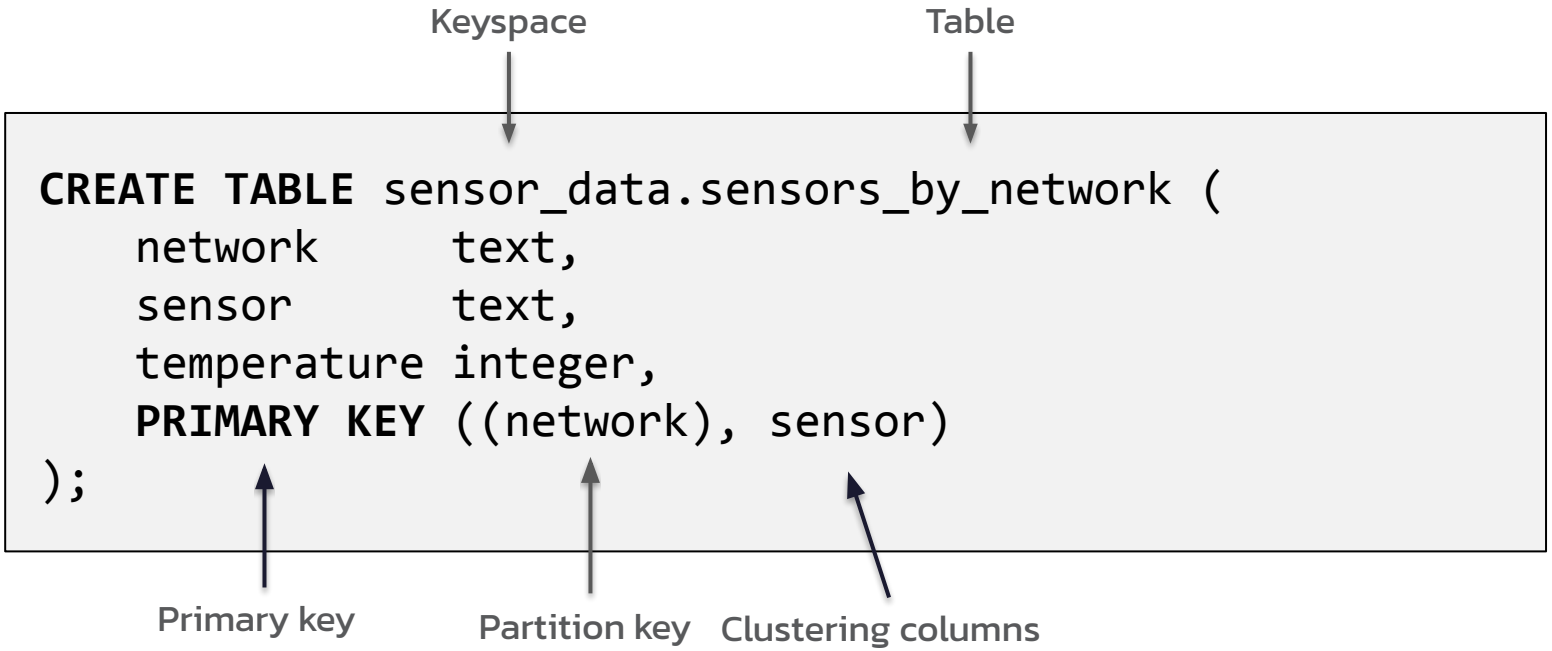
keyspace

replication strategy

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

Replication factor by data center

Create a Keyspace



Create a table

Lab 1

Create Your Astra DB Instance

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:

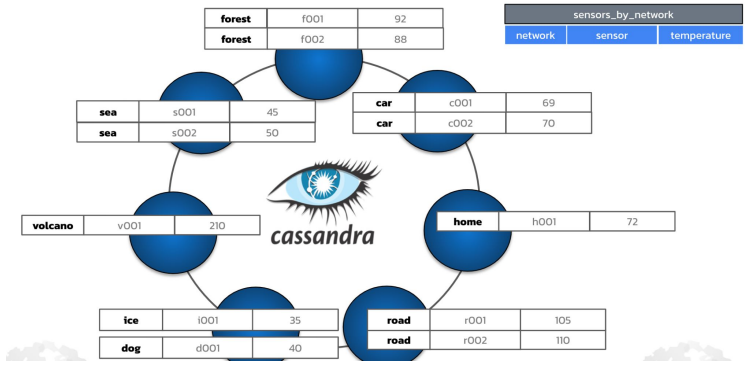
```
PRIMARY KEY ((sensor), timestamp);
```

```
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 = Ensure Uniqueness = PK+ CC

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

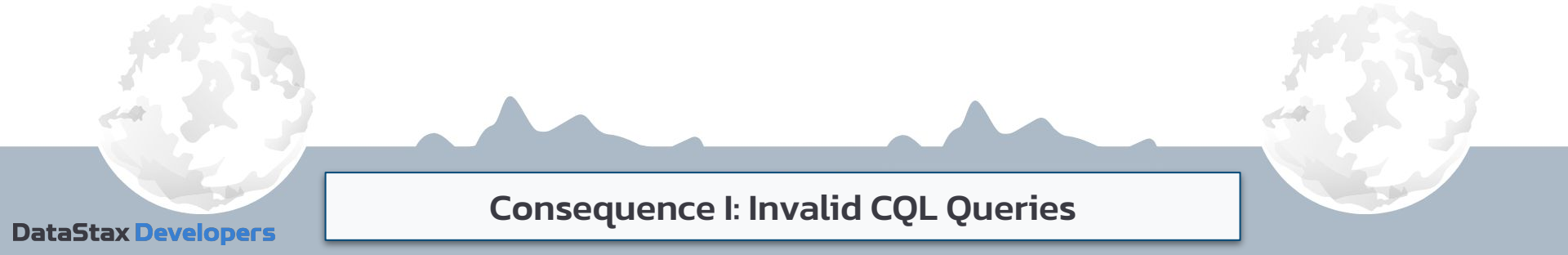
```
SELECT * FROM temperatures_by_sensor ...

    WHERE sensor = ?;

    WHERE sensor > ?;

    WHERE sensor = ? AND date > ?:

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



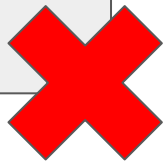
Consequence I: Invalid CQL Queries

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)
```



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

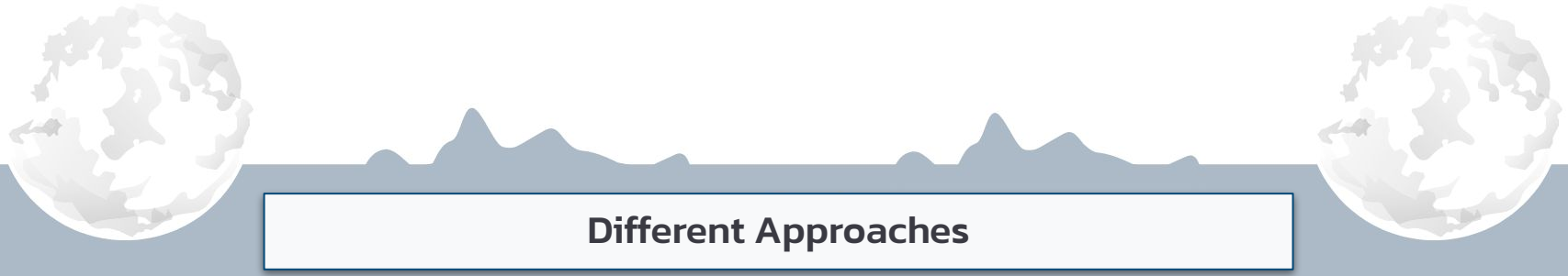
Cassandra

```
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);
```

- 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);
```

Rules of a Good Partition

Lab 2

Dynamic Bucketing

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

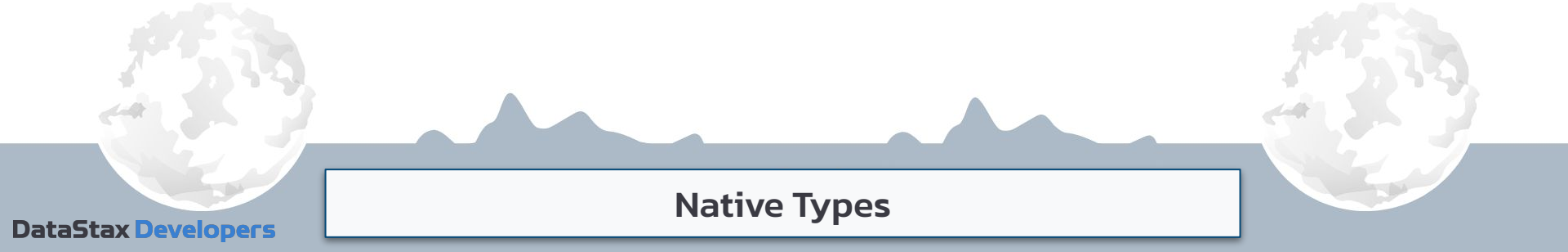
06

What's next?

Homework, Next Session

Agenda

type	constants supported	description
ascii	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

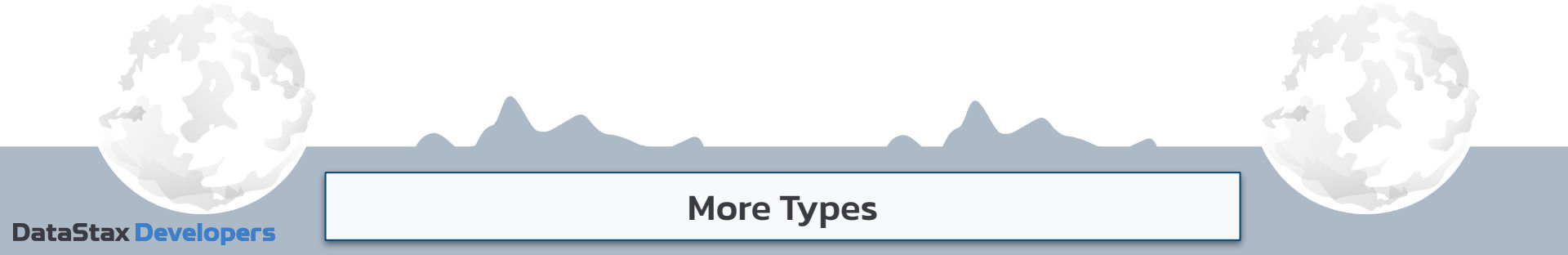


Native Types

- **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

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

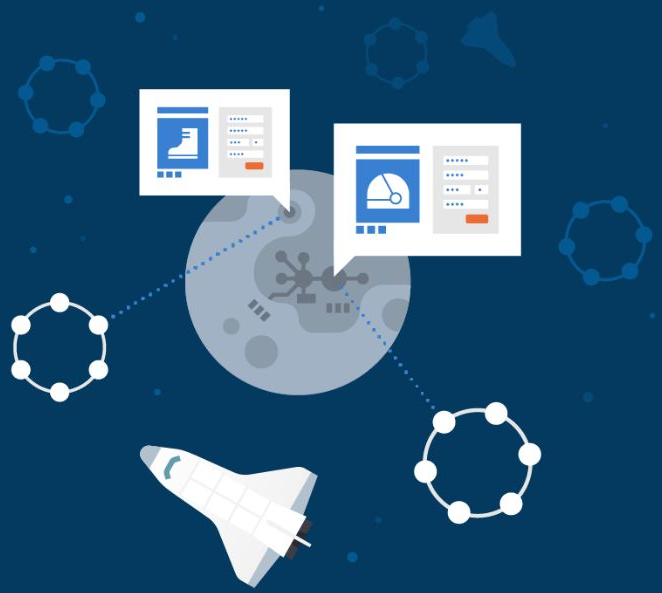
Five important optimizations

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

Employees

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

Departments

departmentId	department
1	Engineering
2	Math

Normalization

"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"

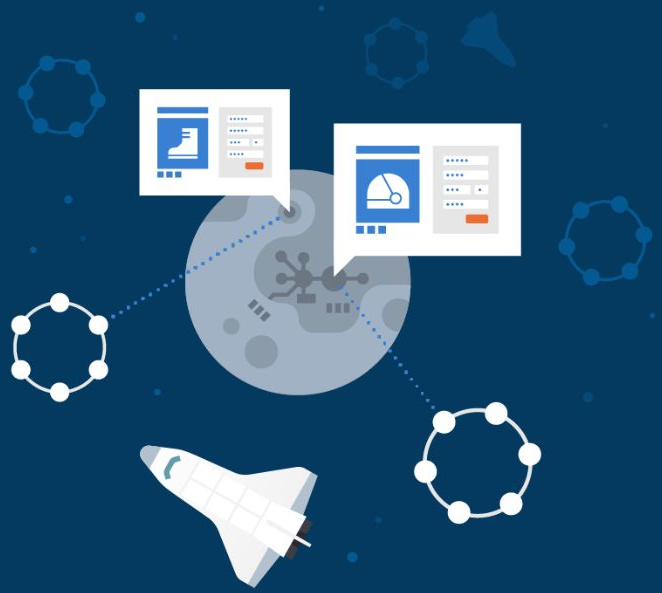
PROS: Quick Read, Simple Queries
CONS: Multiple Writes, Manual Integrity

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



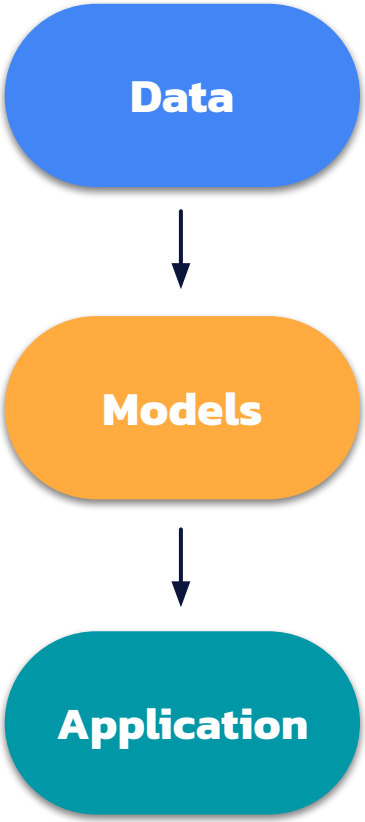
Modelling Approach

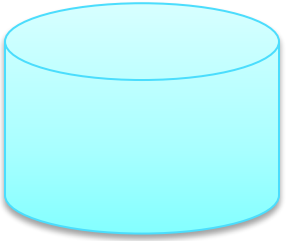
1. Analyze raw data

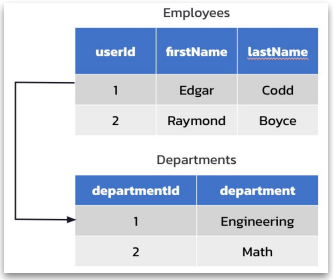
2. Identify entities, their properties and relations

3. Design tables, using **normalization** and foreign keys.

4. Use JOIN when doing queries to join normalized data from multiple tables

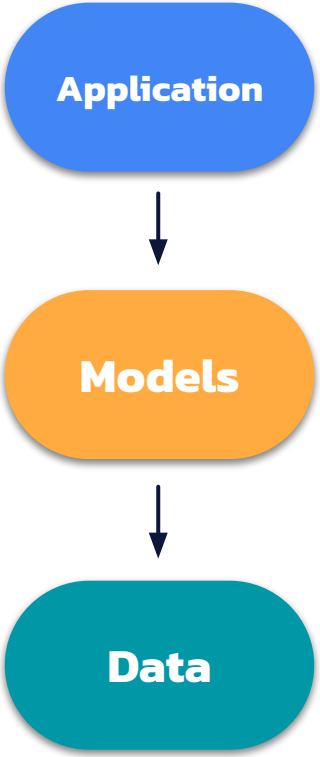






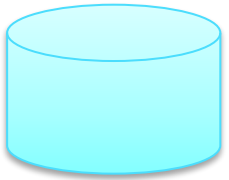
Relational Data Modelling

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"

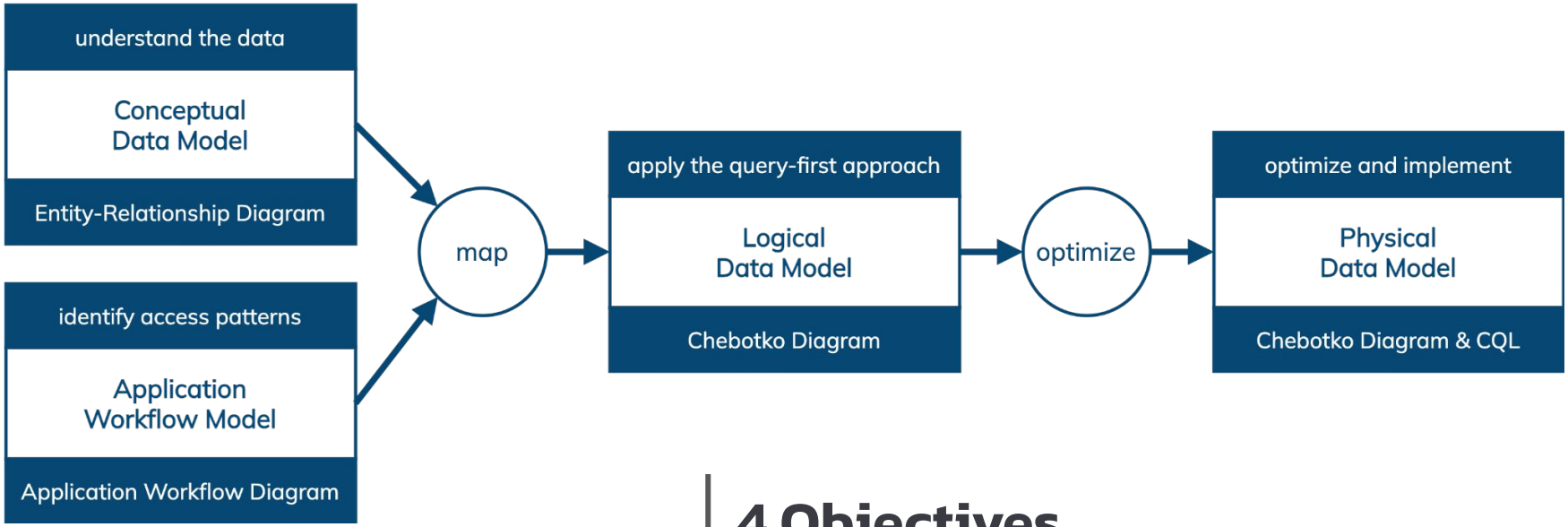
- Queries drive schema design

Modeling principle 3: "Nest data"

- Data nesting is the main data modeling technique

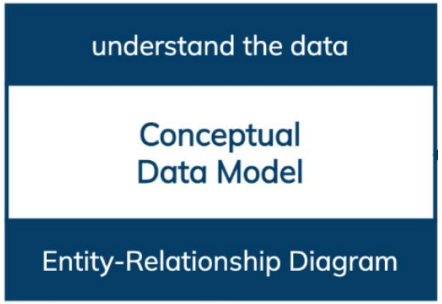
Modeling principle 4: "Duplicate data"

- Better to duplicate than to join

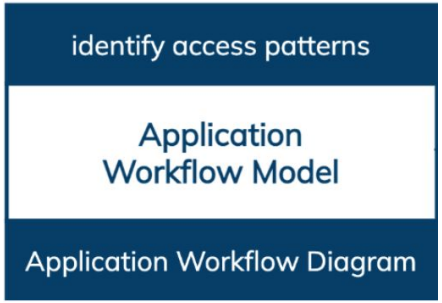


4 Objectives,
4 Models
2 Transitions

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

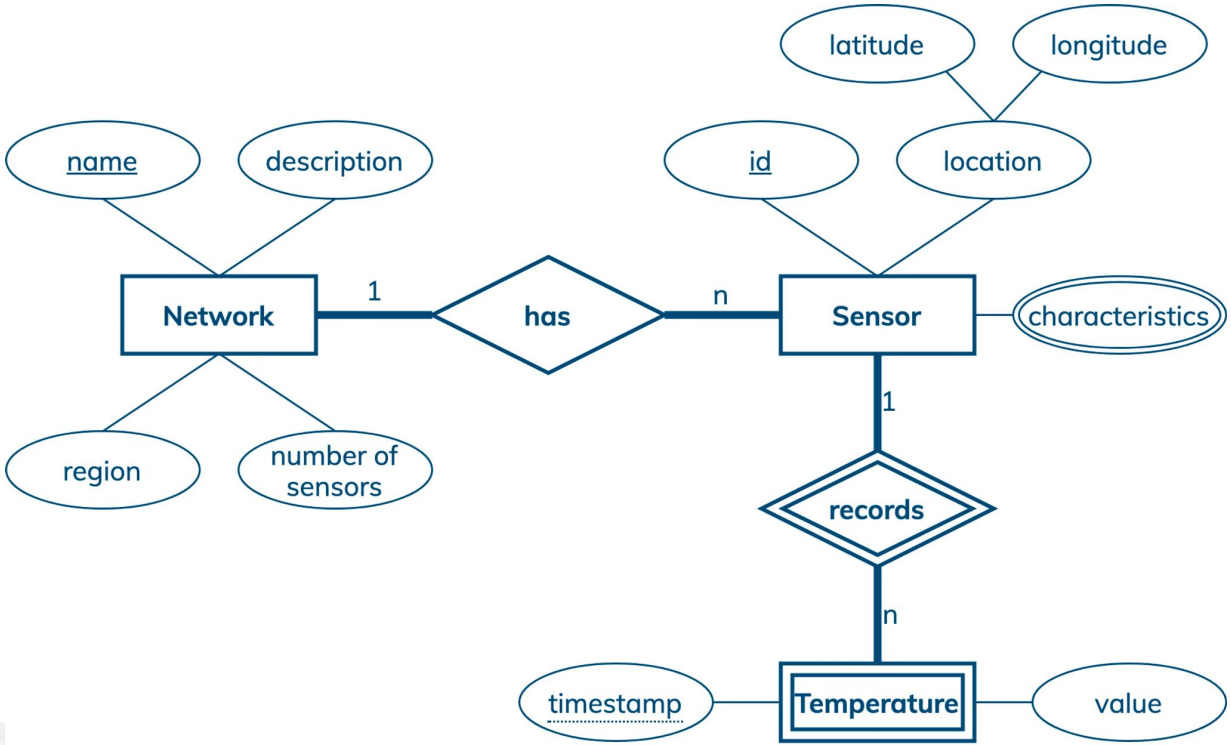


Analyze the Domain

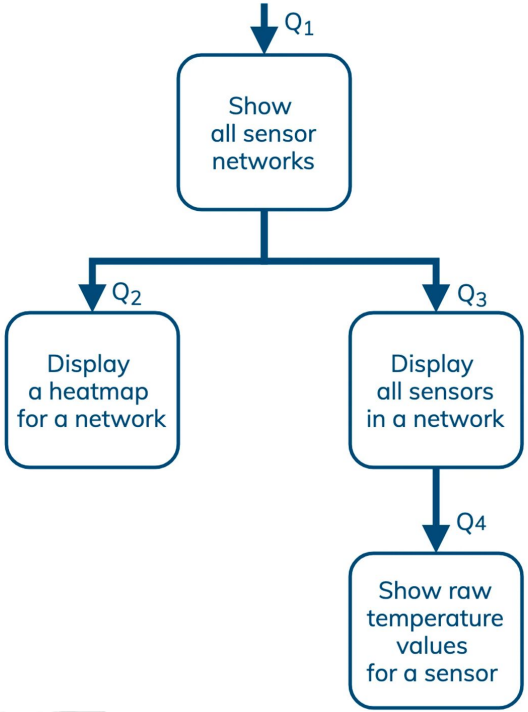


Analyze Customer Workflows

Data Modelling Methodology: Step I



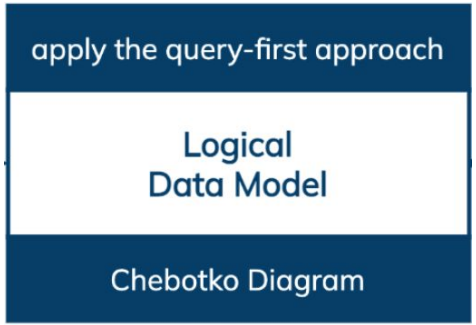
Entity-Relationship Diagram



Data access patterns

Q_1: Find information about all networks; order by name (asc)
Q_2: Find hourly average temperatures for every sensor in a specified network for a given date range; order by date (desc) and hour (desc)
Q_3: Find information about all sensors in a specified network
Q_4: Find raw measurements for a particular sensor on a specified date; order by timestamp (desc)

Application Workflow 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

Mapping rule 2: "Equality search attributes"
- Equality search attributes map to the beginning columns of a primary key

Mapping rule 3: "Inequality search attributes"
- Inequality search attributes map to clustering columns

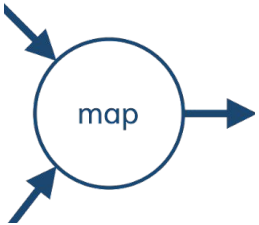
Mapping rule 4: "Ordering attributes"
- Ordering attributes map to clustering columns

Mapping rule 5: "Key attributes"
- Key attributes map to primary key columns

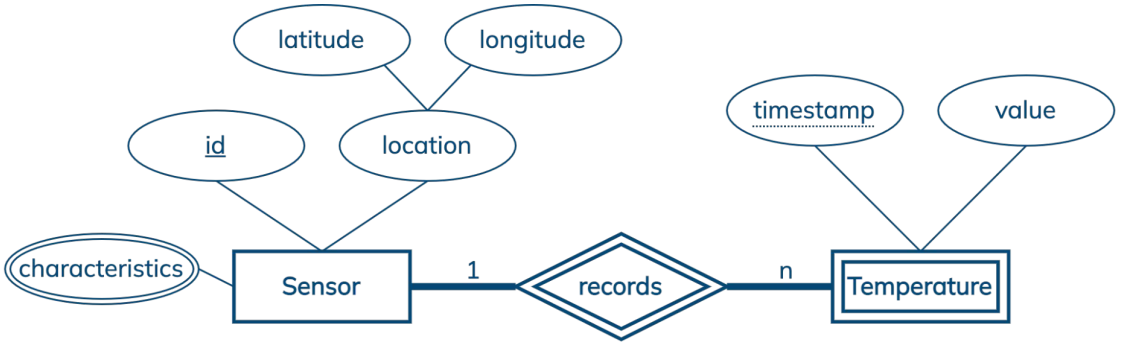
Based on a conceptual data model

Based on a query

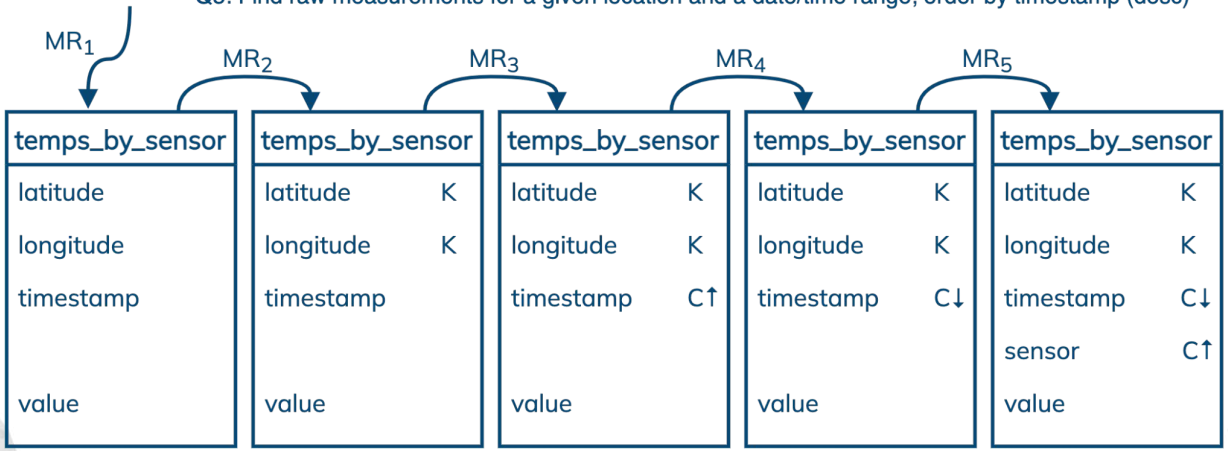
Based on a conceptual data model



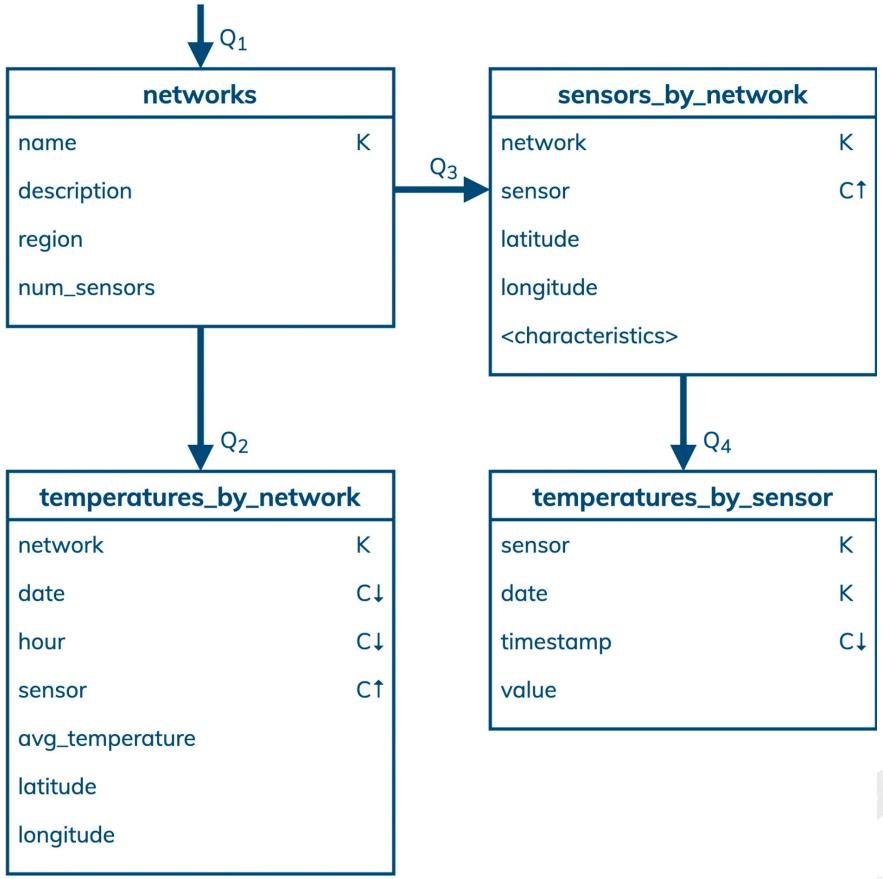
Mapping Rules



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



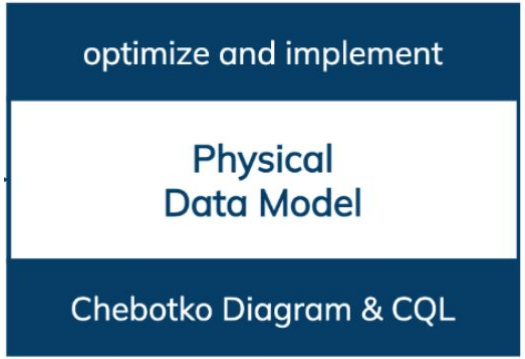
Example: Applying Mapping Rules



Data access patterns

Q_1: Find information about all networks; order by name (asc)
Q_2: Find hourly average temperatures for every sensor in a specified network for a given date range; order by date (desc) and hour (desc)
Q_3: Find information about all sensors in a specified network
Q_4: Find raw measurements for a particular sensor on a specified date; order by timestamp (desc)

Chebotko Diagram

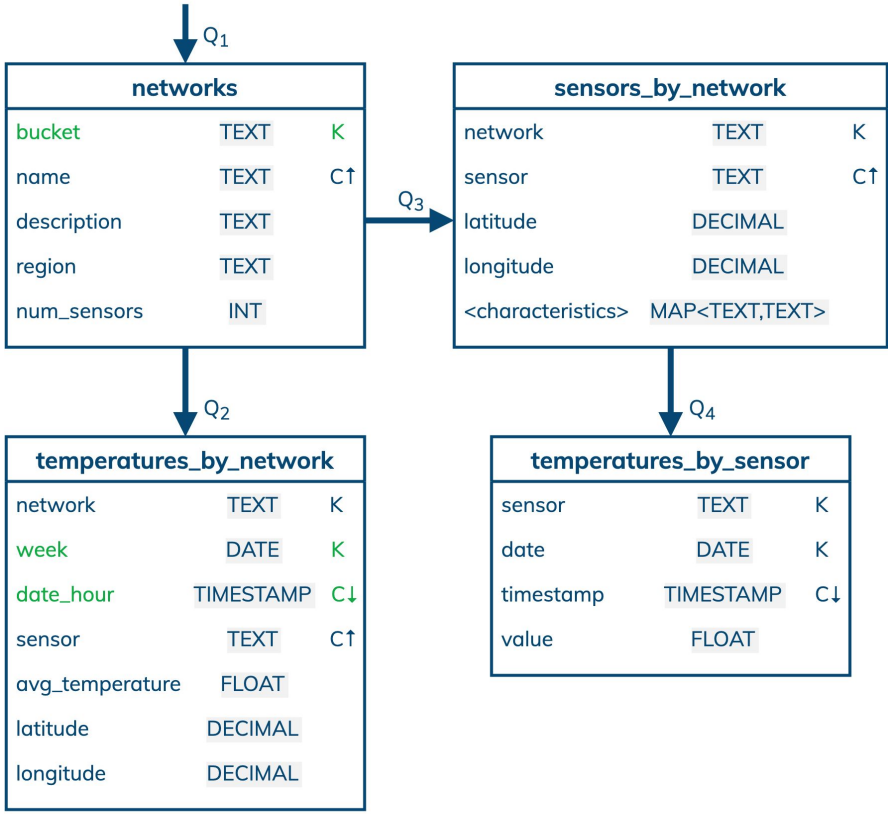


Optimize and Create

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 out free DS220 course at academy.datastax.com



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

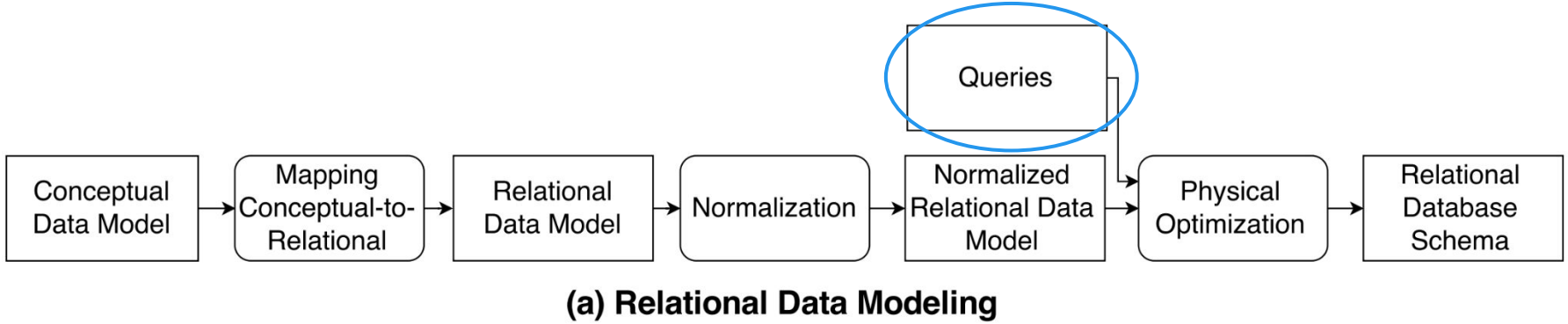
```
CREATE TABLE sensors_by_network (
  network TEXT,
  sensor TEXT,
  latitude DECIMAL,
  longitude DECIMAL,
  characteristics MAP<TEXT,TEXT>,
  PRIMARY KEY ((network),sensor)
);
```

Q2

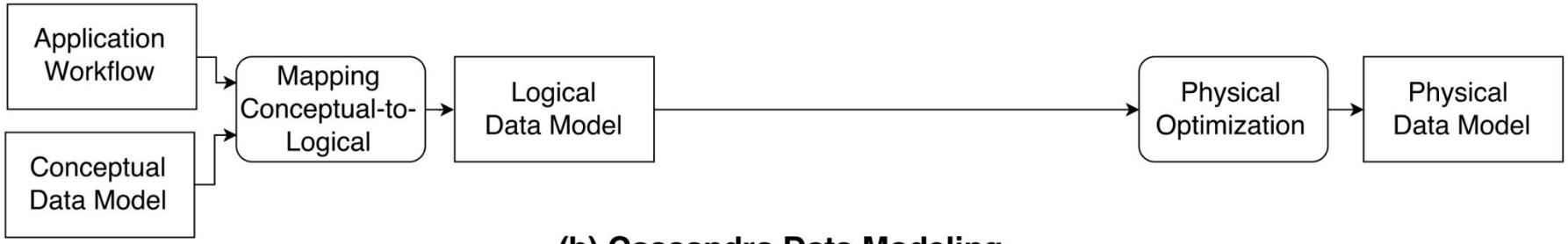
```
CREATE TABLE temperatures_by_network (
  network TEXT,
  week DATE,
  date_hour TIMESTAMP,
  sensor TEXT,
  avg_temperature FLOAT,
  latitude DECIMAL,
  longitude DECIMAL,
  PRIMARY KEY ((network,week),date_hour,sensor)
) WITH CLUSTERING ORDER BY (date_hour DESC, sensor ASC);
```

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);
```



(a) Relational Data Modeling



(b) Cassandra Data Modeling

Relational vs Cassandra Modeling Process

Demo

KDM

https://github.com/datastaxdevs/workshop-cassandra-data-modeling

Lab 3 (Optional)

Sensor 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

- **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);
```

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 ✖

- **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;
```

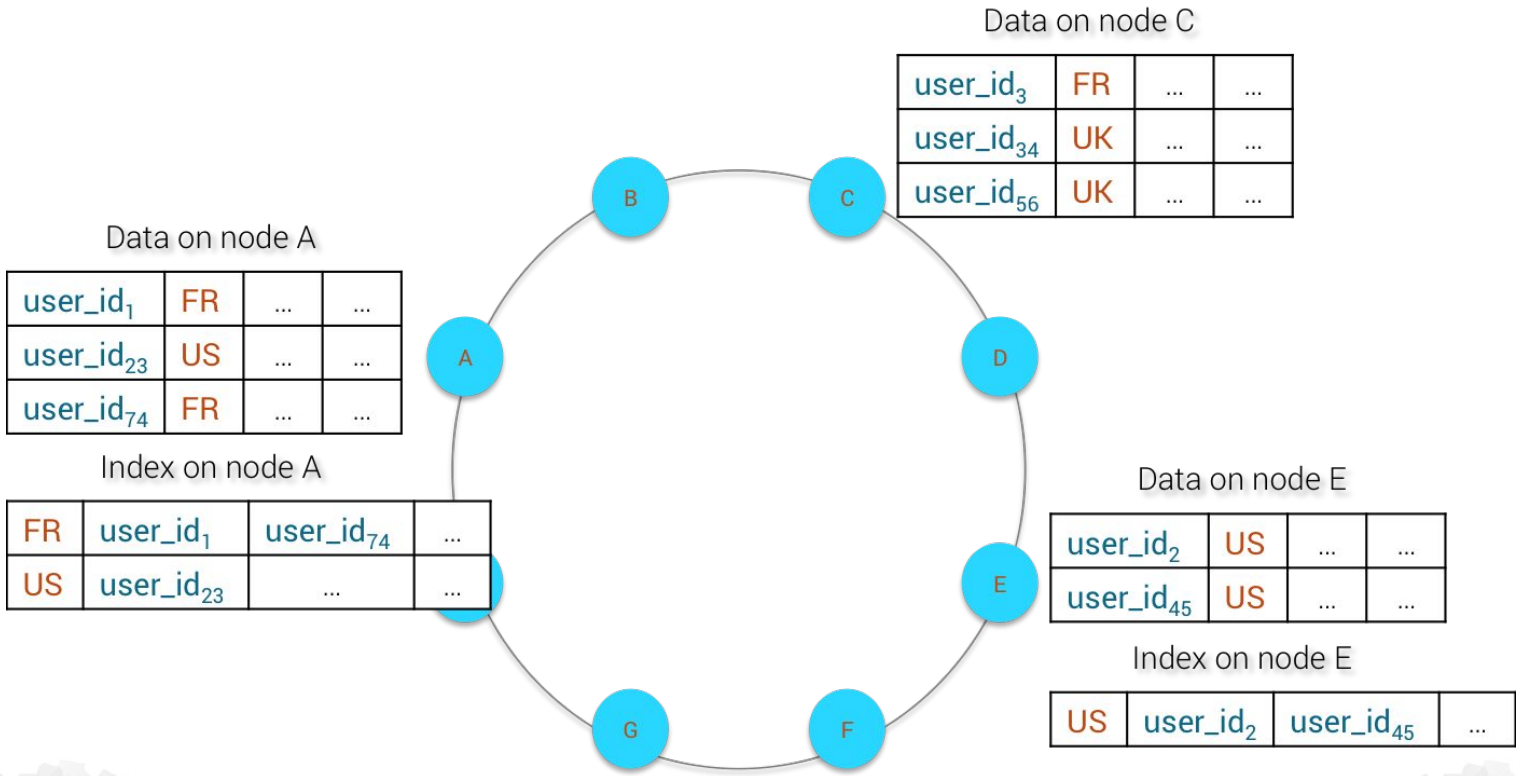
- Experimental feature
- Not always applicable
- May become inconsistent
- 10% write penalty to a base table

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

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)



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';
```

- 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

- 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

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

```
User 1: read(10)   10<20   write(20)
-------------------------------------------------> time
User 2:          read(10)   10<15   write(15)
```


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)
```

- 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 ... ];
```

- 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

Tombstones

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

- 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

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 bathes 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

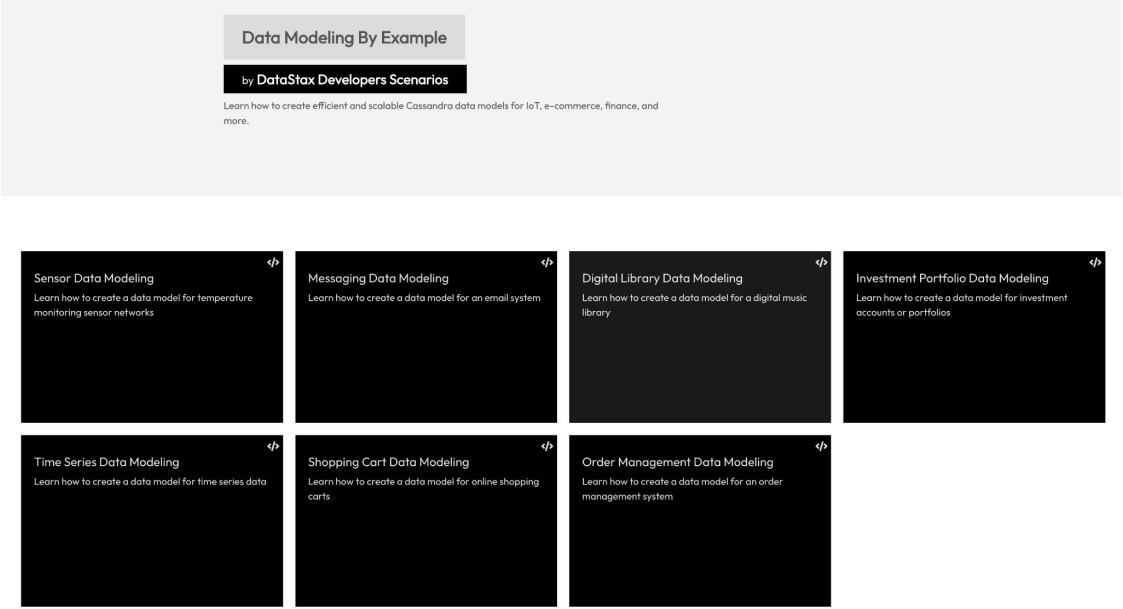
What's next?

Homework, Next Session

Agenda

datastax.com/learn/data-modeling-by-example

killercoda.com/datastaxdevs/course/cassandra-data-modeling



HomeWork

- medium.com/building-the-open-data-stack/data-modeling-in-cassandra-and-datastax-astra-db-3f89b9c133c9

- devm.io/iot/apache-cassandra-iot-174970

Section 1 of 3

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

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

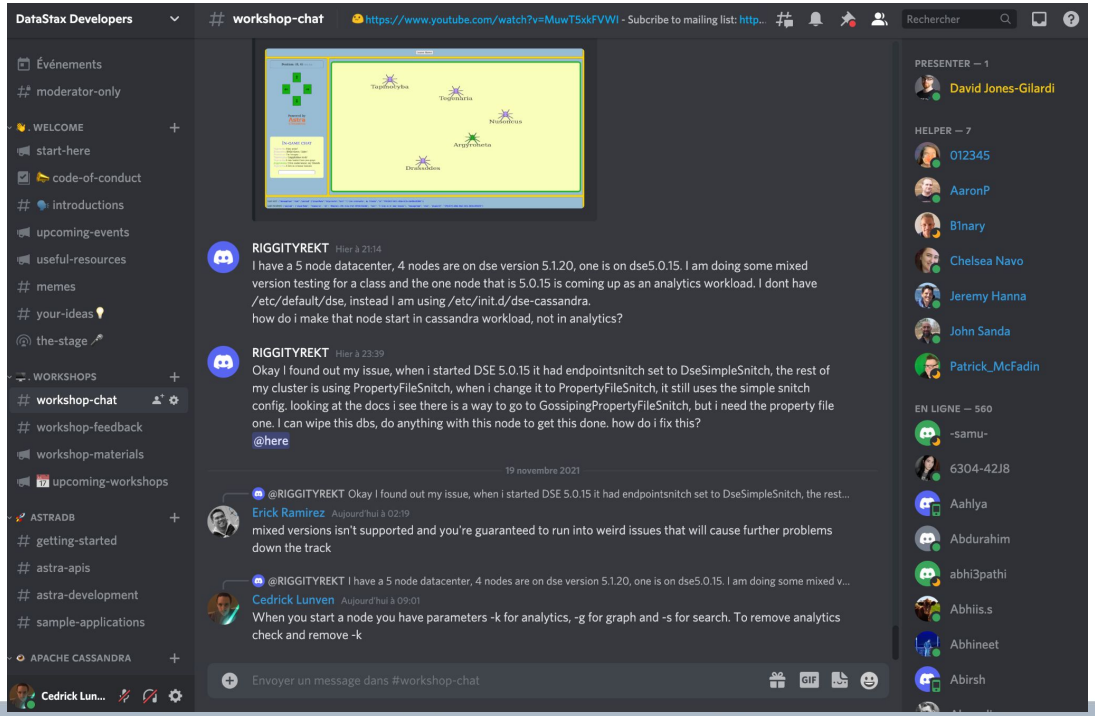
Email *

Valid email

This form is collecting emails. Change settings

Join our 19k Discord Community

DataStax Developers



!discord

dtsx.io/discord

Discord Community

Thank You!