Cassandra Day
2022

Hands-on Workshop

Cassandra Data Modeling & Application Development

Sponsored by DataStax

You can't build a successful app without data modeling

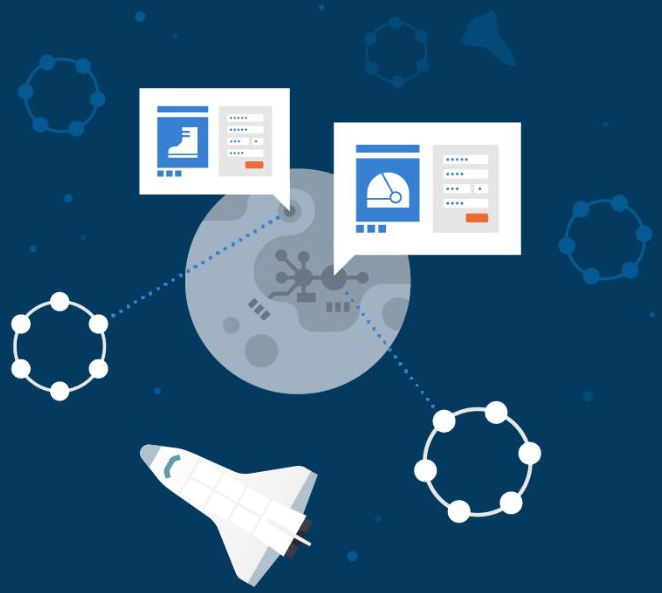
Some words about data modeling

* Not a relational database!
* Denormalization
* Keys and Partitions
* Querying tables
* Data Modeling methodology
* **HANDS-ON**: explore a full data model

Build a successful app

* Apps & Drivers (Python, Java)
* **HANDS-ON**: run your first Cassandra-backed API

Resources, homework, badges



Not a relational database!

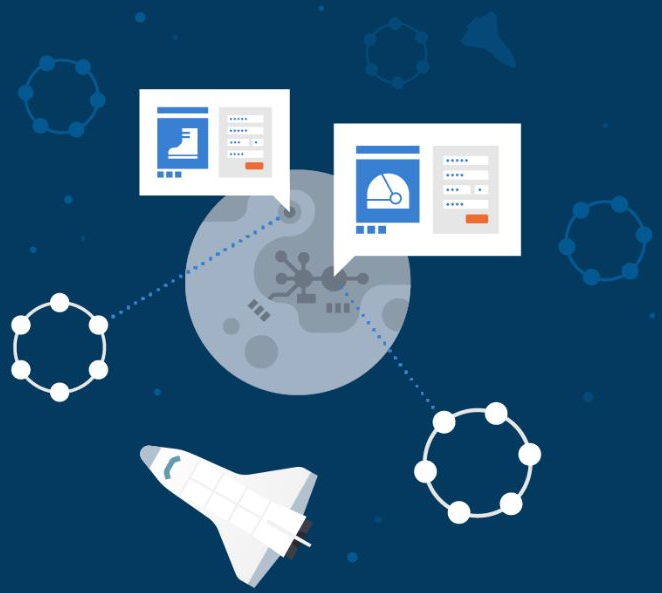
Data modeling =

the act of laying out the structure of your database to fit your application's purposes

Begins with an abstract idea of your application's needs

Ends with the commands that create the physical tables and the read/write operations that they support

Data -> Model -> Application	Application -> Model -> Data
Joins, indexes	Denormalization
Referential integrity	No (would not scale well anyway)
Primary key = uniqueness	Primary key = uniqueness + partitioning + access
Entity-driven	Query-driven



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"

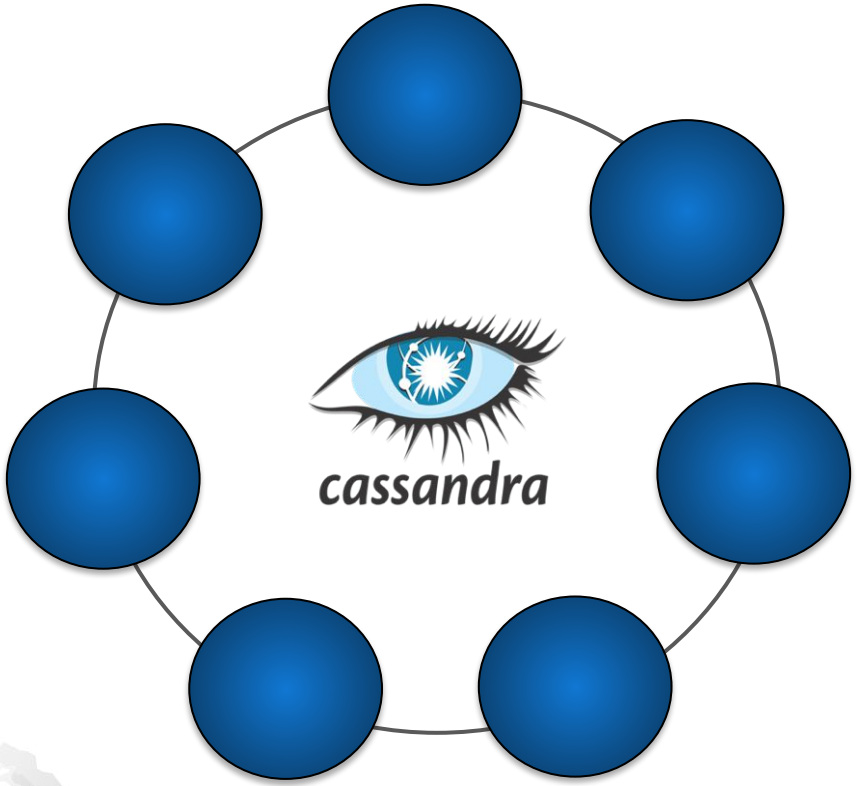
PROS: Quick Read, Simple Queries
CONS: Multiple Writes, Manual Integrity

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

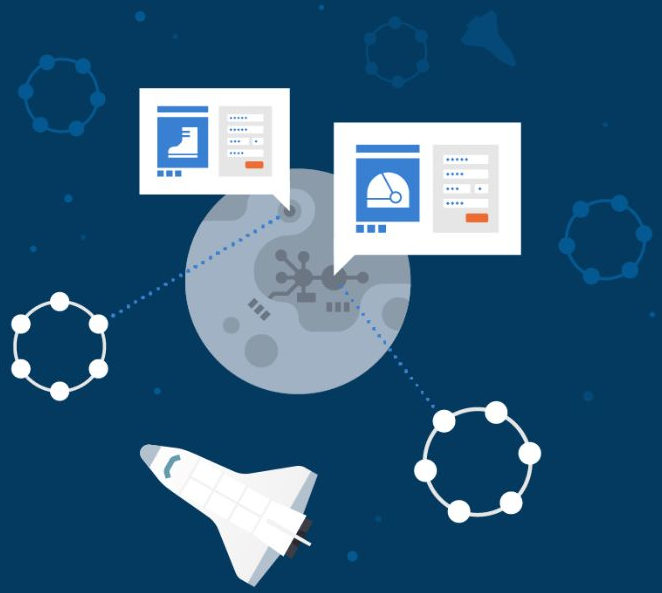


sensors_by_network		
network	sensor	temperature
forest	f001	92
forest	f002	88
volcano	v001	210
sea	s001	45
sea	s002	50
home	h001	72
car	c001	69
car	c002	70
dog	d001	40
road	r001	105
road	r002	110
ice	i001	35

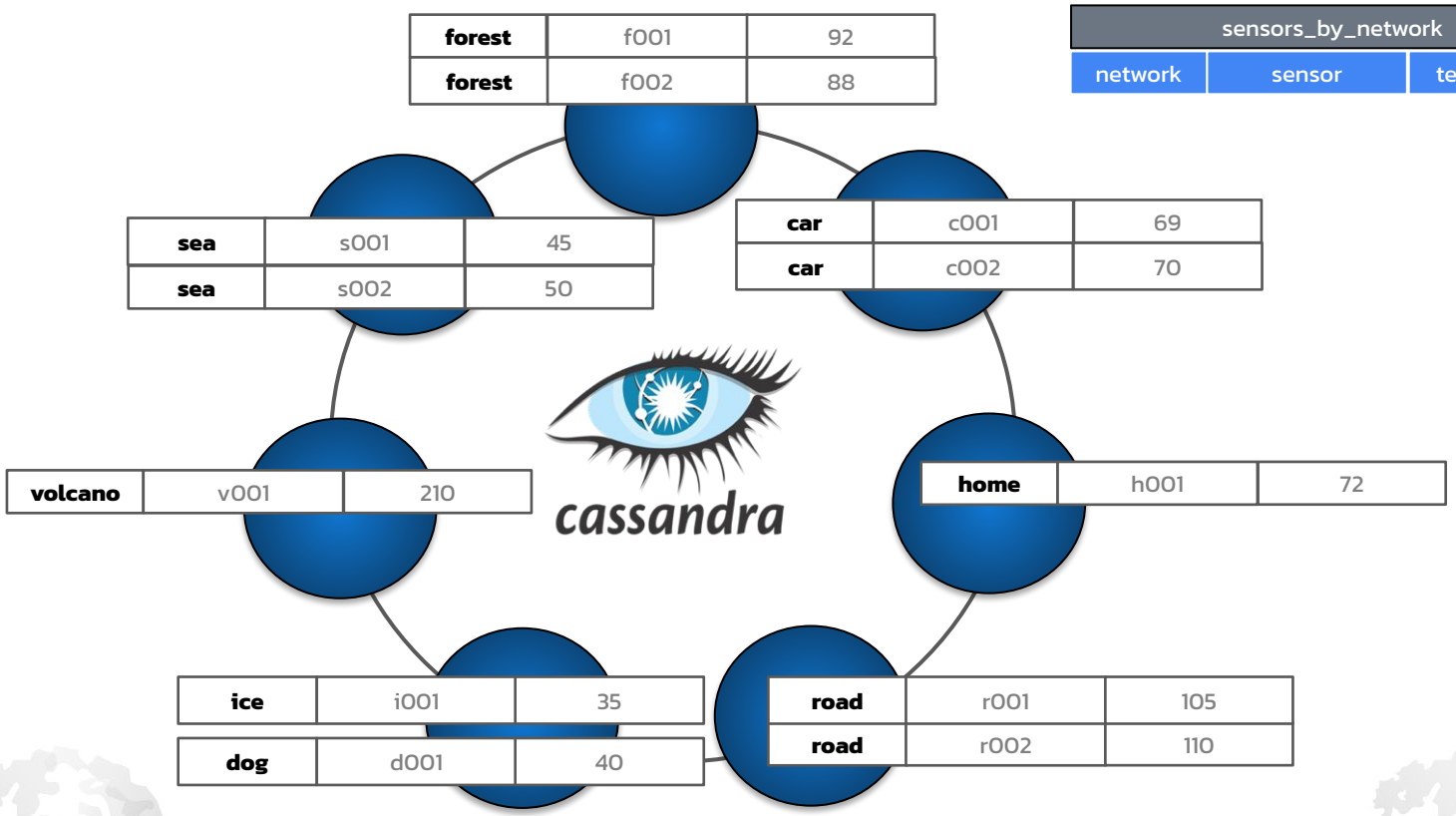
Partition Key

Primary Key

Data is organized as [Tables]



Keys and Partitions



Data is organized as Distributed tables

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:

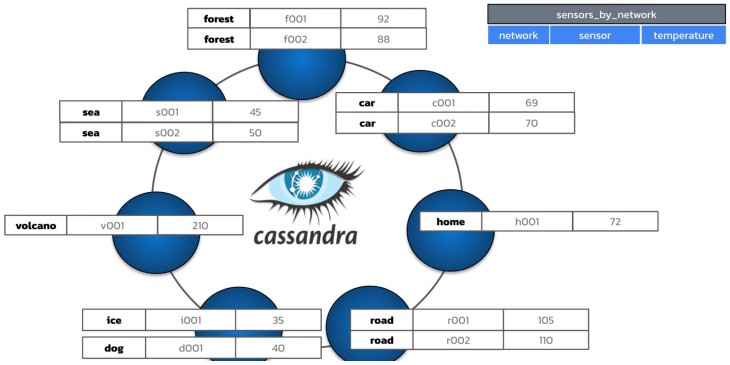
```
PRIMARY KEY ((sensor), timestamp);
```

```
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);
```

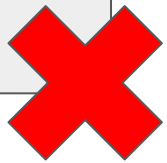
Primary Key = Ensure Uniqueness = PK + CC

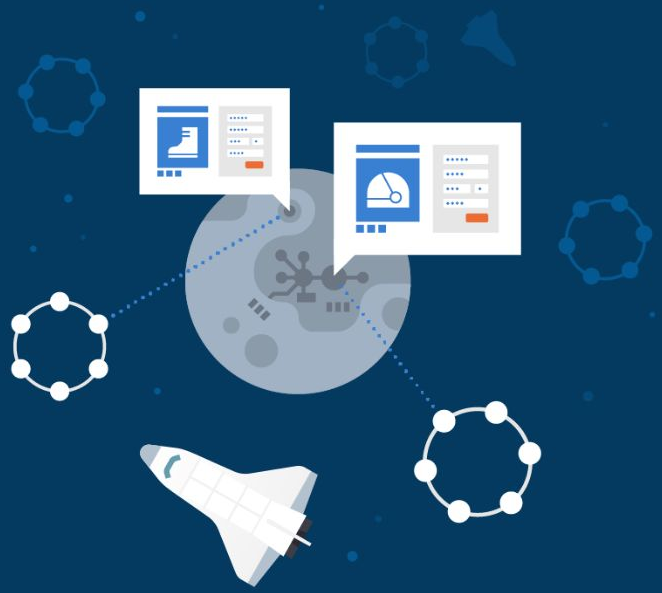
Important:

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;
```

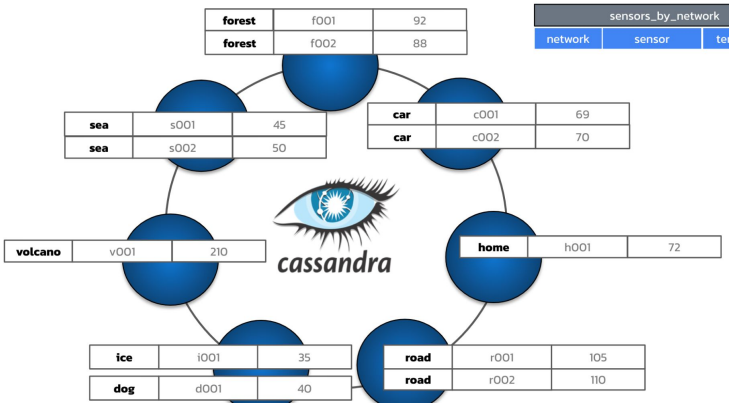
```
ALTER TABLE temperatures_by_sensor
DROP PRIMARY KEY
ADD PRIMARY KEY (sensor, timestamp)
```





Keys and Partitions (recap)

```
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);
```



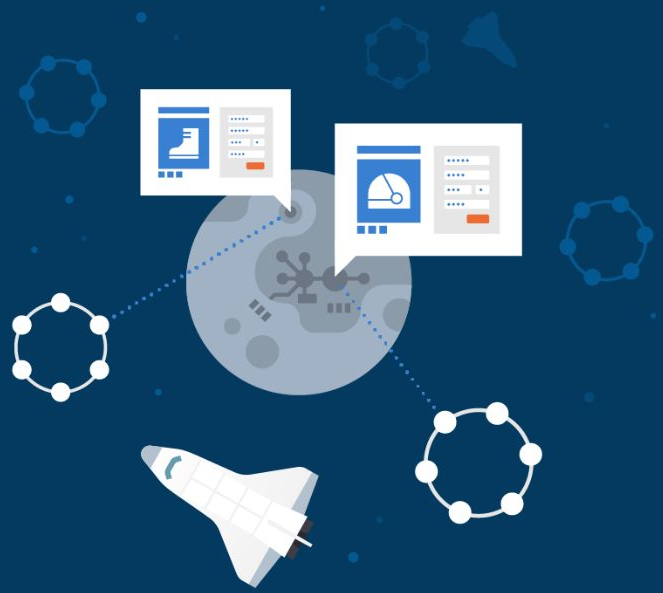
sensors_by_network		
network	sensor	temperature

Partition Key
DEFINES DATA DISTRIB.

Clustering column(s)
ENSURE UNIQUENESS

Primary key
UNIQUELY IDENTIFIES A ROW

Primary Key, Partition Key, Clustering Columns, Oh My!



Querying tables

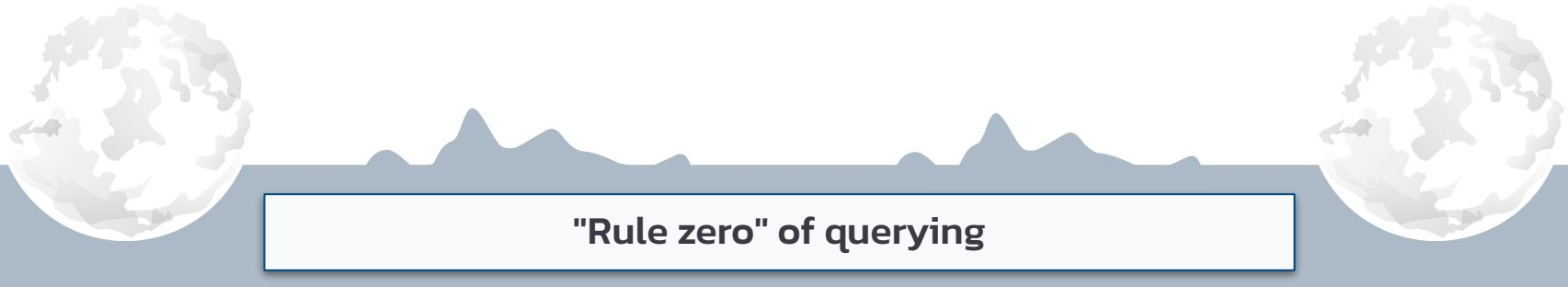
"I want to read ALL rows of my table"

Sorry, **not today**.
If you really need to,

⇒ OLAP workload: use Apache Spark & co.

but in a typical OLTP app,
you actually **don't need** to run such a query!

Reading all rows (*full-table scan*) is a performance capital sin, since it involves a lot of nodes in the cluster!



"Rule zero" of querying

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

```
SELECT * FROM temperatures_by_sensor ...

    WHERE sensor = ?;

    WHERE sensor > ?;

    WHERE sensor = ? AND date > ?;

    WHERE sensor = ? AND date = ?;

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

Table structure ⇒ valid queries

```
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 Sept 20th 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

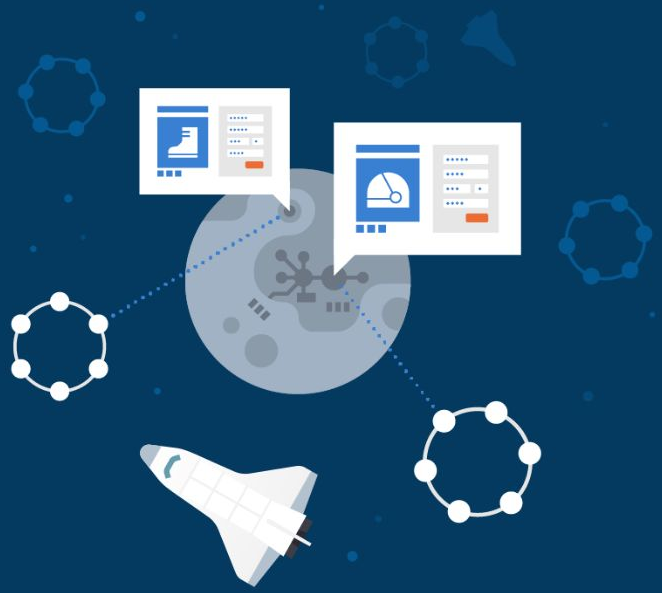
Rules of a Good Partition

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

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

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

Rules of a Good Partition



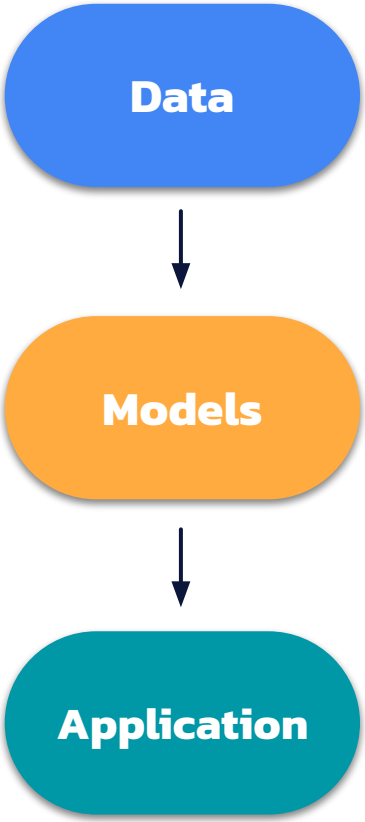
Data modeling methodology

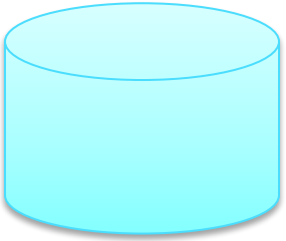
1. Analyze raw data

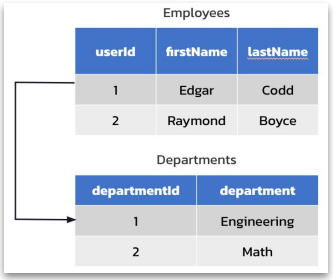
2. Identify entities, their properties and relations

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

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

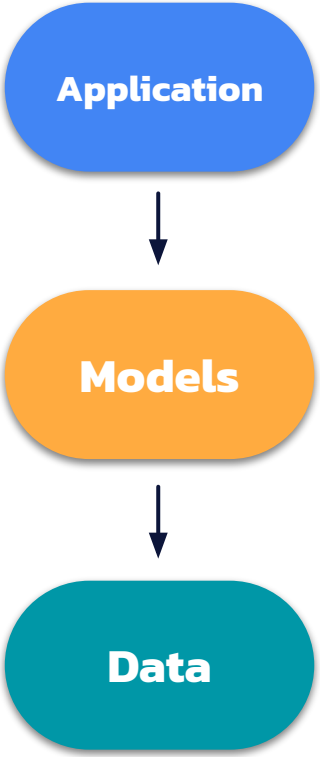






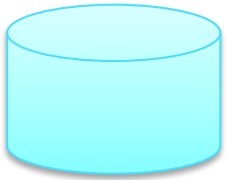
Relational Data Modelling

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



- 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"

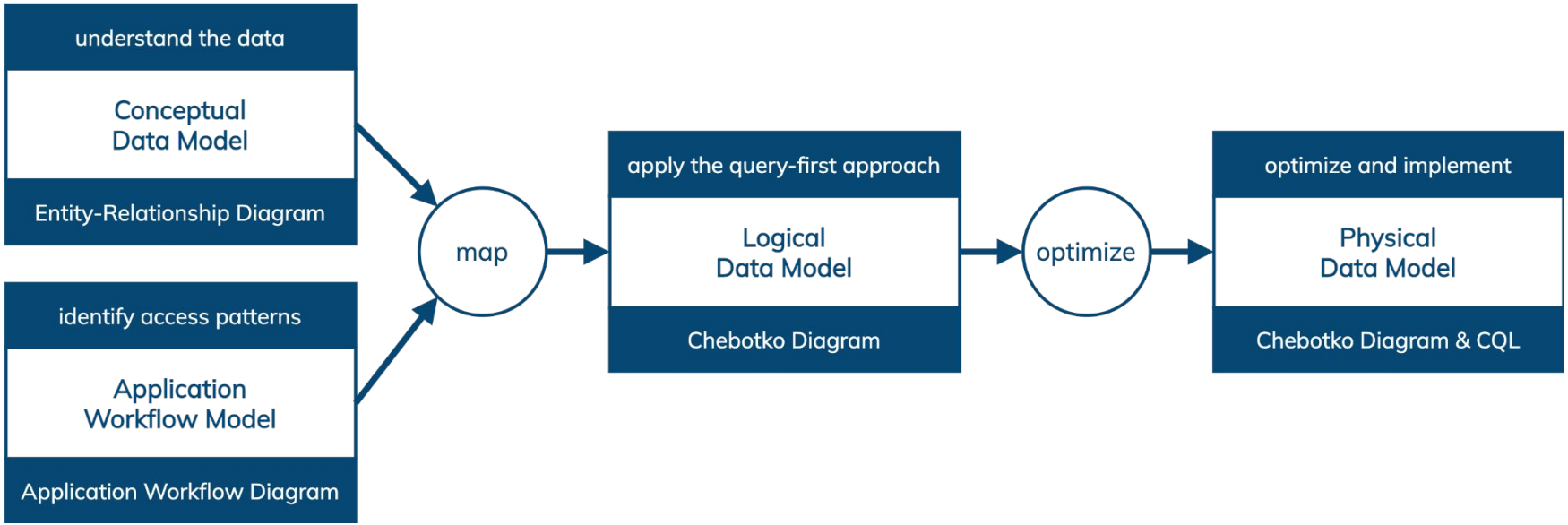
- Queries drive schema design

Modeling principle 3: "Nest data"

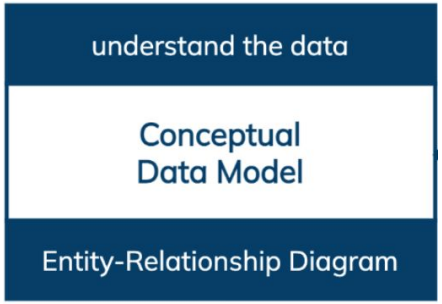
- Data nesting is the main data modeling technique

Modeling principle 4: "Duplicate data"

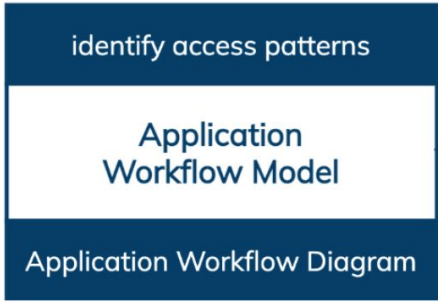
- Better to duplicate than to join



Modeling workflow

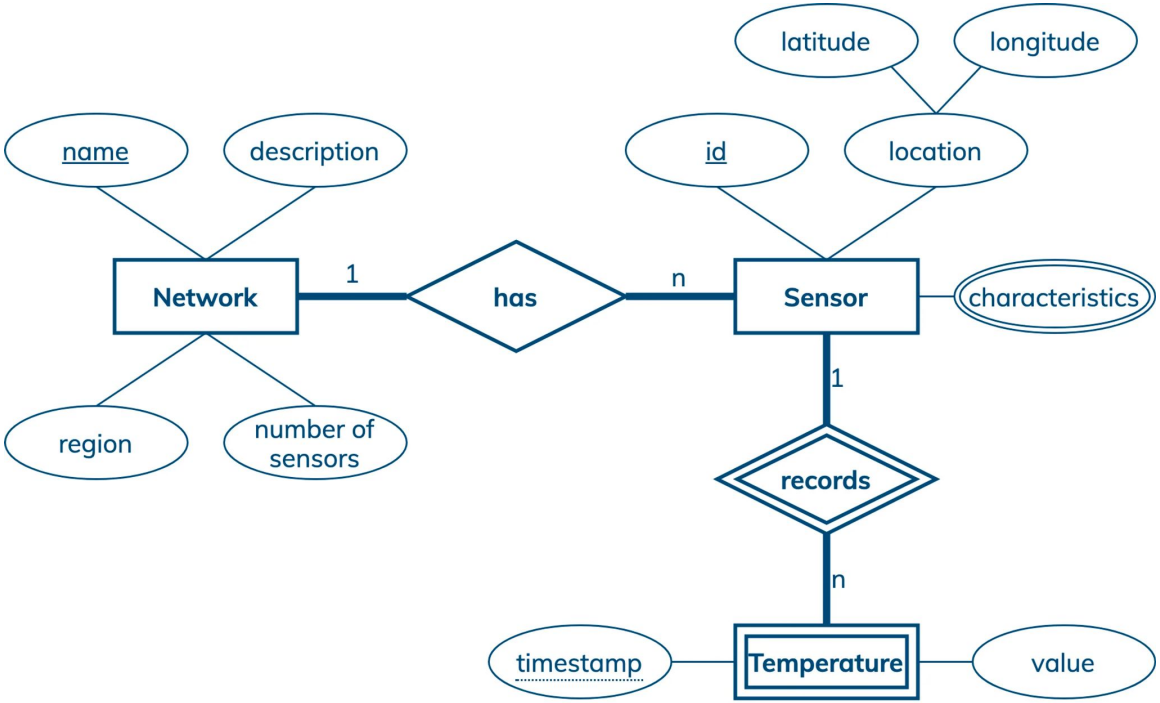


Analyze the Domain

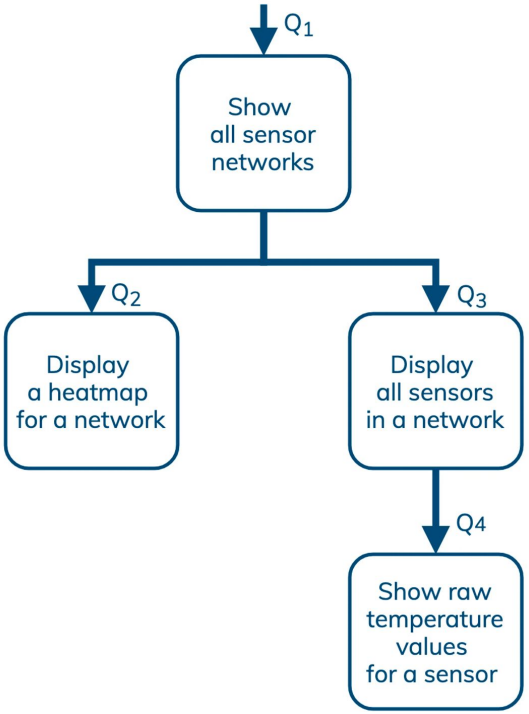


Analyze Customer Workflows

Data Modelling Methodology: Step I



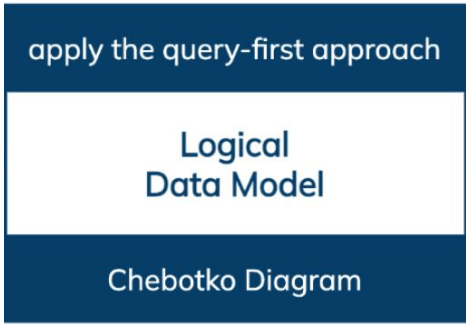
Entity-Relationship Diagram



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

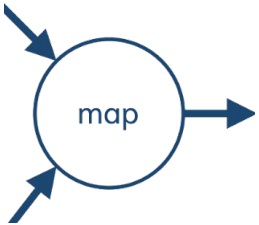
Mapping rule 4: "Ordering attributes"
- Ordering attributes map to clustering columns

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

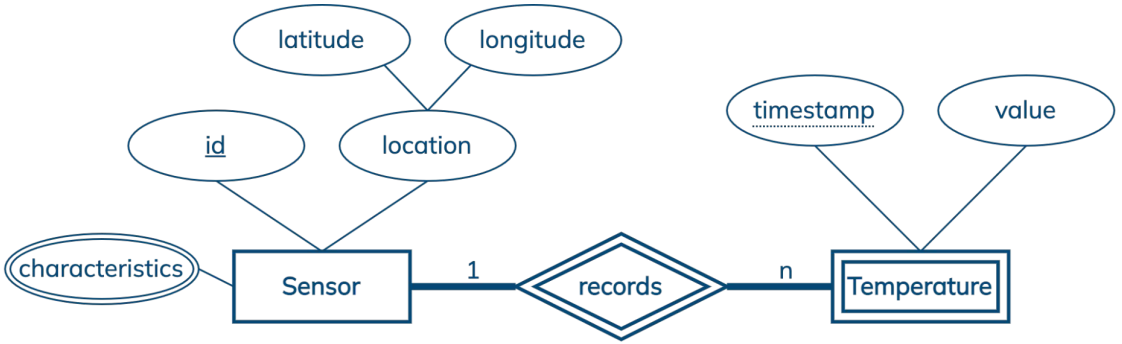
Based on a conceptual data model

Based on a query

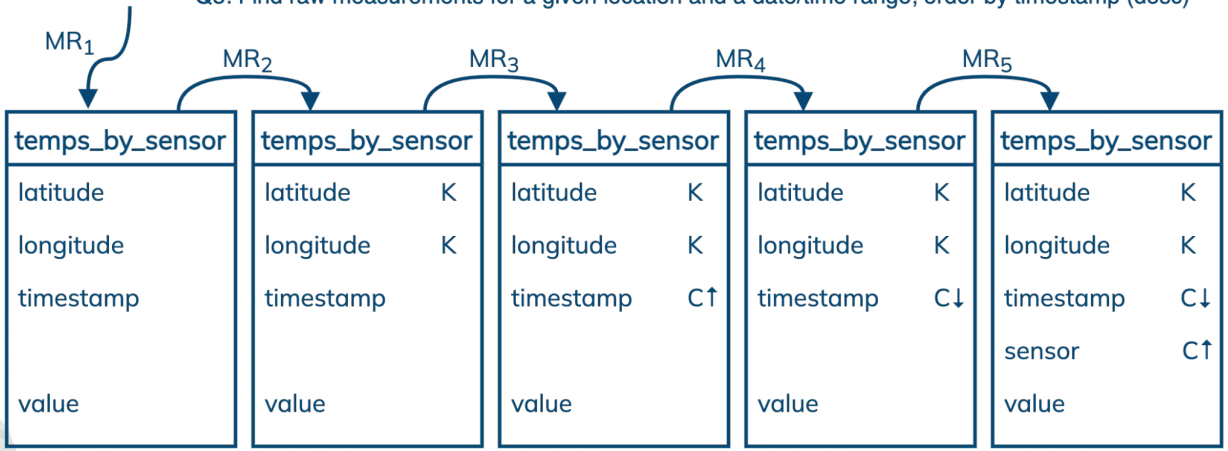
Based on a conceptual data model



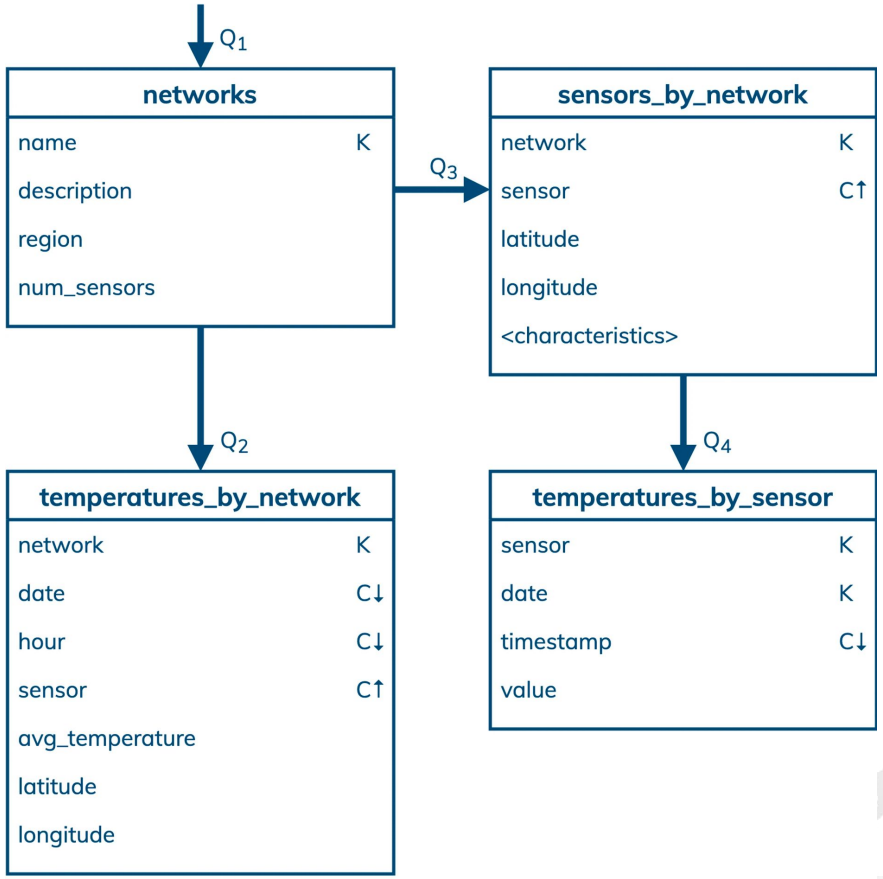
Mapping Rules



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



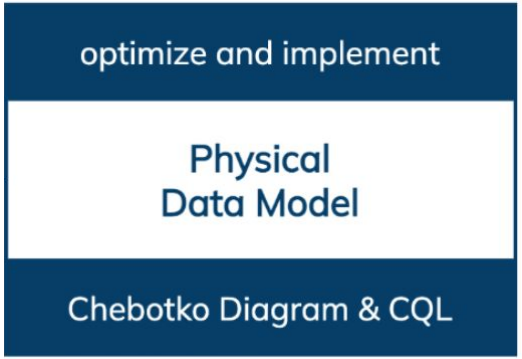
Example: Applying Mapping Rules



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)

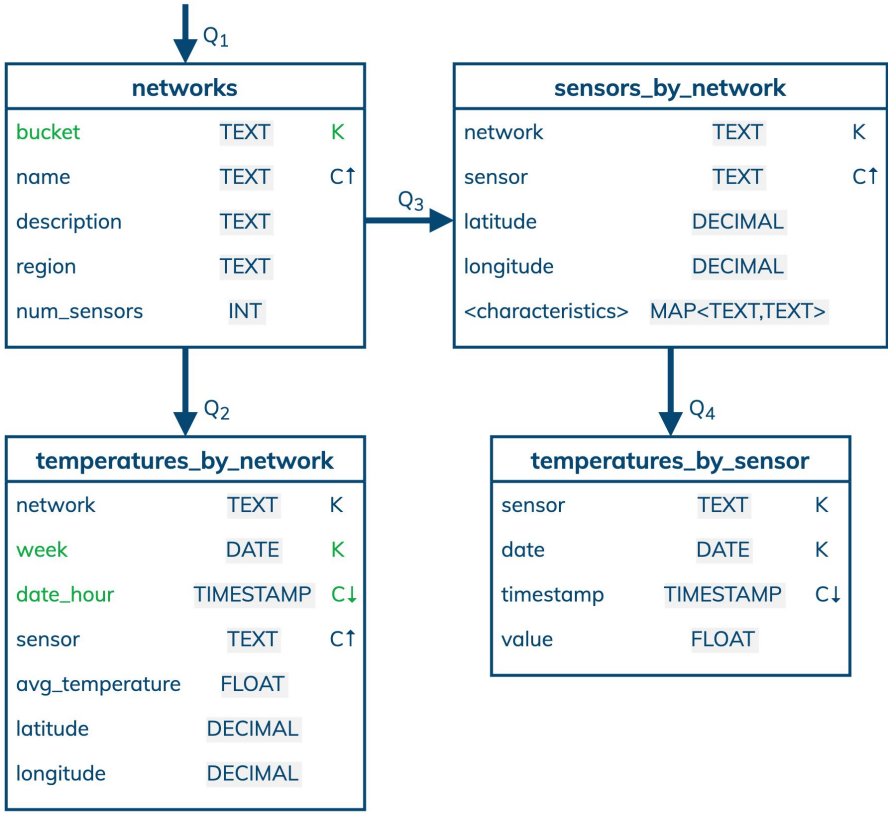
Chebotko Diagram



Optimize and Create

- 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



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);
```

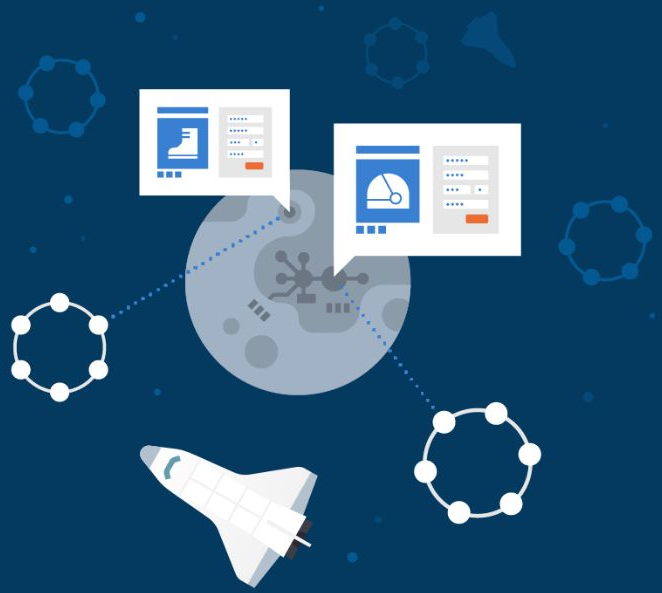
Q4

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

Lab 1

Explore a data model

Start here: **dtsx.io/cday-ws2**



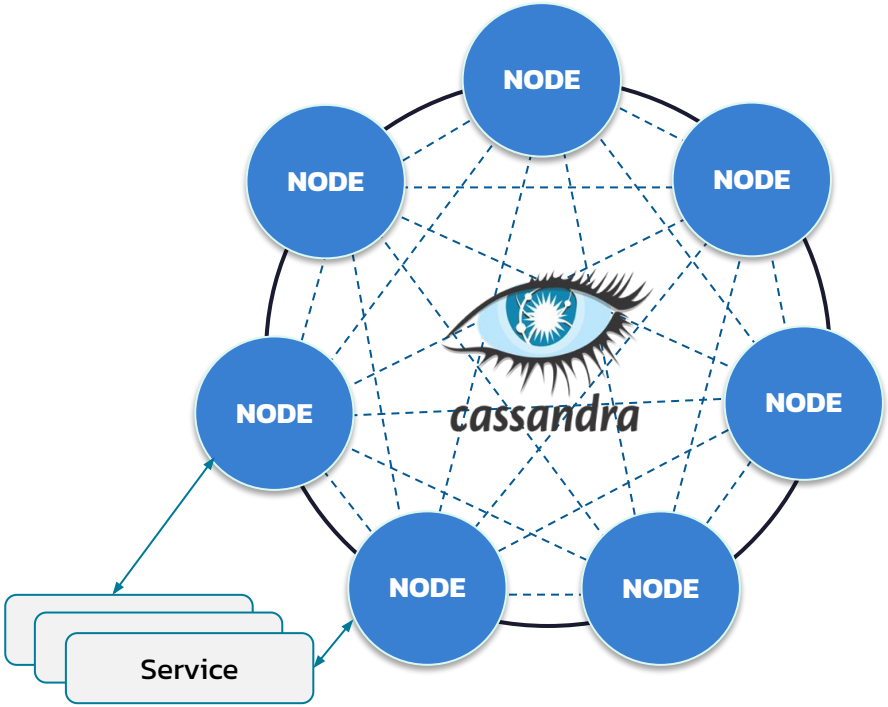
Apps & Drivers

Loose Coupling: Data resiliency

- Data Replicated on multiple Nodes
- Load Balancing at driver side
- Health Check at driver side
- Hinted HandOff

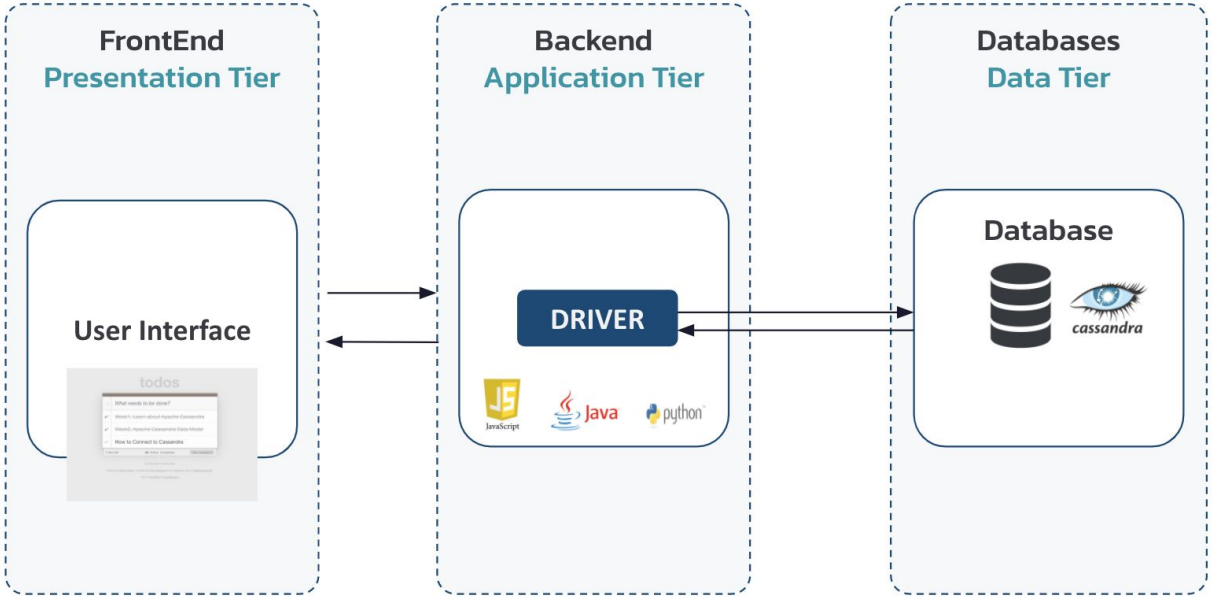
Shared nothing: Data isolation

- Per Keyspace (with replications)
- Per Tables (1 query = 1 table)
- Per profile (RBAC)



Cassandra ❤ Microservices

Application Development with Apache Cassandra



High-level Application Development w/ Cassandra

today: +Spring Boot

today: +FastAPI

Choose your language for today





Connectivity

- ★ Token & Datacenter Aware
- ★ Load Balancing Policies
- ★ Retry Policies
- ★ Reconnection Policies
- ★ Connection Pooling
- ★ Health Checks
- ★ Authentication | Authorization
- ★ SSL

Query

- ★ CQL Support
- ★ Schema Management
- ★ Sync/Async/Reactive API
- ★ Query Builder
- ★ Compression
- ★ Paging

Parsing Results

- ★ Lazy Load
- ★ Object Mapper
- ★ Spring Support
- ★ Paging

Drivers

Maven

```
<dependency>
    <groupId>com.datastax.oss</groupId>
    <artifactId>java-driver-core</artifactId>
    <version>4.13.1</version>
</dependency>
```

Java

4.6.3 npm

```
npm install cassandra-driver
```

```
{
  "dependencies": {
     "cassandra-driver": "^4.6.3"
  }
}
```

JavaScript

3.25.0 python Package Index

```
pip install cassandra-driver
```

python

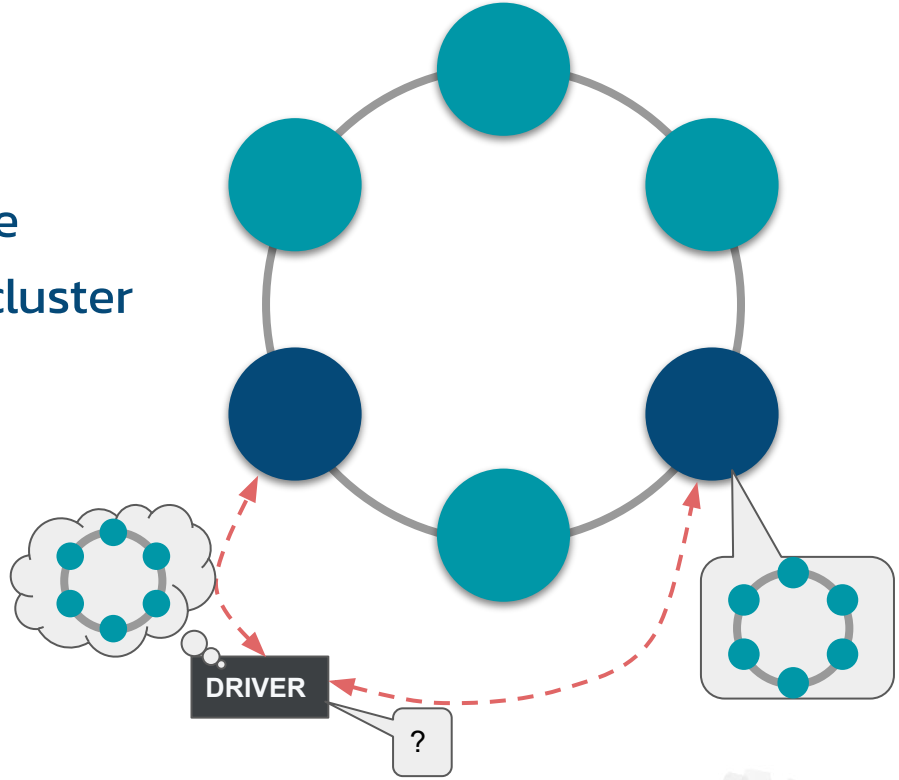
nuget v3.15.0

```
Install-Package CassandraCSharpDriver -Version 3.15.0
```

C#

Installing the drivers

- Only one necessary
- Unless that node is down
- Better ~3 nodes per DC for resilience
- From there, drivers discover whole cluster
- Local Datacenter



Apache Cassandra™ Contact Point

```java
CqlSession cqlSession = CqlSession.builder()
 .addContactPoint(new InetSocketAddress("127.0.0.1", 9042))
 .withKeyspace("killrvideo")
 .withLocalDatacenter("dc1")
 .withAuthCredentials("U","P")
 .build();
```


```node
const client = new cassandra.Client({
  contactPoints: ['127.0.0.1'],
  localDataCenter: 'dc1',
  keyspace: 'killrvideo',
  credentials: { username: 'U', password: 'P' }
});
```


```python
auth_provider = PlainTextAuthProvider(
   username='U', password='P')
cluster = Cluster(['127.0.0.1'],
  auth_provider=auth_provider, protocol_version=2)
session = cluster.connect('todos')
```


```dotnet
Cluster cluster = Cluster.Builder()
.AddContactPoint("127.0.0.1")
.WithCredentials("U", "P")
.Build();
session = cluster.Connect("todos");
```

Create Session/Client with Contact Points

```java
CqlSession cqlSession = CqlSession.builder()
 .withCloudSecureConnectBundle(Paths.get("secure.zip"))
 .withAuthCredentials("U","P")
 .withKeyspace("todos")
 .build();
```


```node
const client = new cassandra.Client({
  cloud: { secureConnectBundle: 'secure.zip' },
  credentials: { username: 'u', password: 'p' }
});
```


```python
auth_provider = PlainTextAuthProvider(
   username='U', password='P')
cluster = Cluster(
   cloud ={'secure_connect_bundle': 'secure.zip'},
   auth_provider=auth_provider)
session= cluster.connect('todos')
```


```c#
var cluster = Cluster.Builder()
 .WithCloudSecureConnectionBundle("secure.zip")
 .WithCredentials("u", "p")
 .Build();
var session = cluster.Connect("todos");
```

Create Session/Client with ASTRA

- Stateful object handling communications with each node
- Should be unique in the Application (*Singleton*)
- Should be closed at application shutdown (*shutdown hook*) in order to free opened TCP sockets (*stateful*)

```
Java:       cqlSession.close();
Python:     session.shutdown();
Node:       client.shutdown();
CSharp:     IDisposable
```


IMPORTANT about Client/Session

Executing CQL Queries


```
session.execute(
    "SELECT * FROM sensors_by_network WHERE network = %s;",
    (network,)
)
```


```
cqlSession.execute(
    "SELECT * FROM sensors_by_network WHERE network = '" + network + "'"
);
```

Running queries from code

Prepared Statements

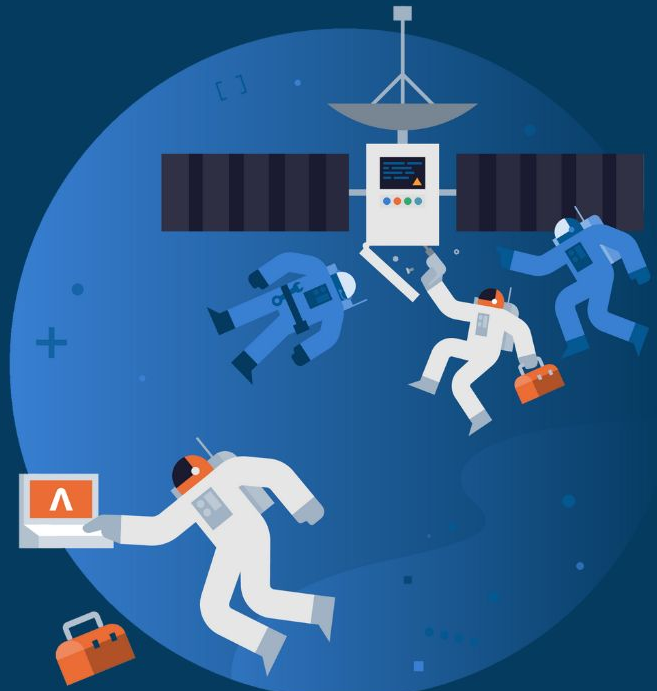

```
q3_statement = session.prepare(
    "SELECT * FROM sensors_by_network WHERE network = ?;"
)
rows = session.execute(q3_statement, (network,) )
```

Prepared Statements


```
PreparedStatement q3Prepared = session.prepare(
    "SELECT * FROM sensors_by_network WHERE network = ?");
BoundStatement q3Bound = q3Prepared.bind(network);
ResultSet rs = session.execute(q3Bound);
```

Advantages of CQL Prepared Statements

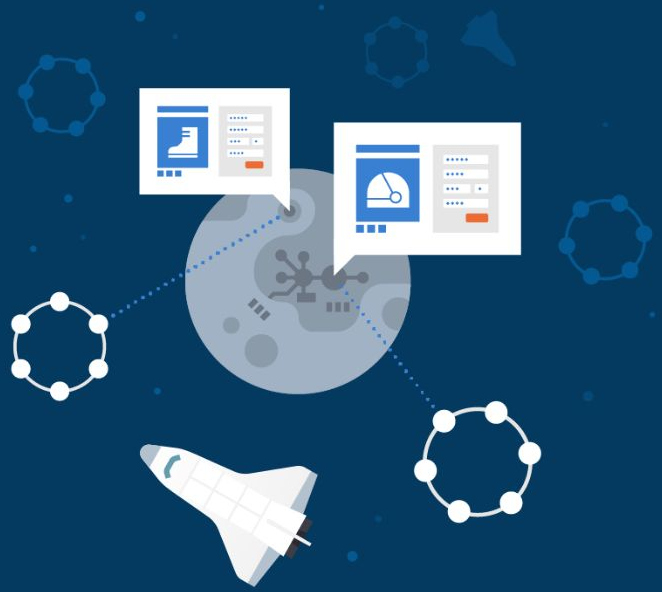
- Parse once, run many times.
- Saves network trips for result set metadata.
- Client-side type validation.
- Statements binding on partition keys compute their own cluster routing.



Lab 2

Create DB and run API !

Start here: **dtsx.io/cday-ws2**



Conclusion & next steps

Homework (see App Dev repo!)

A couple of "theory" questions, plus ...

Coding exercise: *Enrich the API with a* ***new***

GET endpoint *for Q1 ("get all networks")*

Get help about Cassandra

(*):

stackoverflow.com/questions/tagged/cassandra

DBA Stack Exchange(*):

dba.stackexchange.com/questions/tagged/cassandra

Discord:

dtsx.io/discord

(*) for best results, follow the "cassandra" tag

This is just the beginning of your journey

Enjoy the rest of today
Do the homework ⇒ get a badge
We have more workshops & hands-on labs!
Get Cassandra Certified (Academy & Exam voucher)

Apache CASSANDRA

Thank you!

Sponsored by DataStax