CS 422 - DBMS

Lesson 13

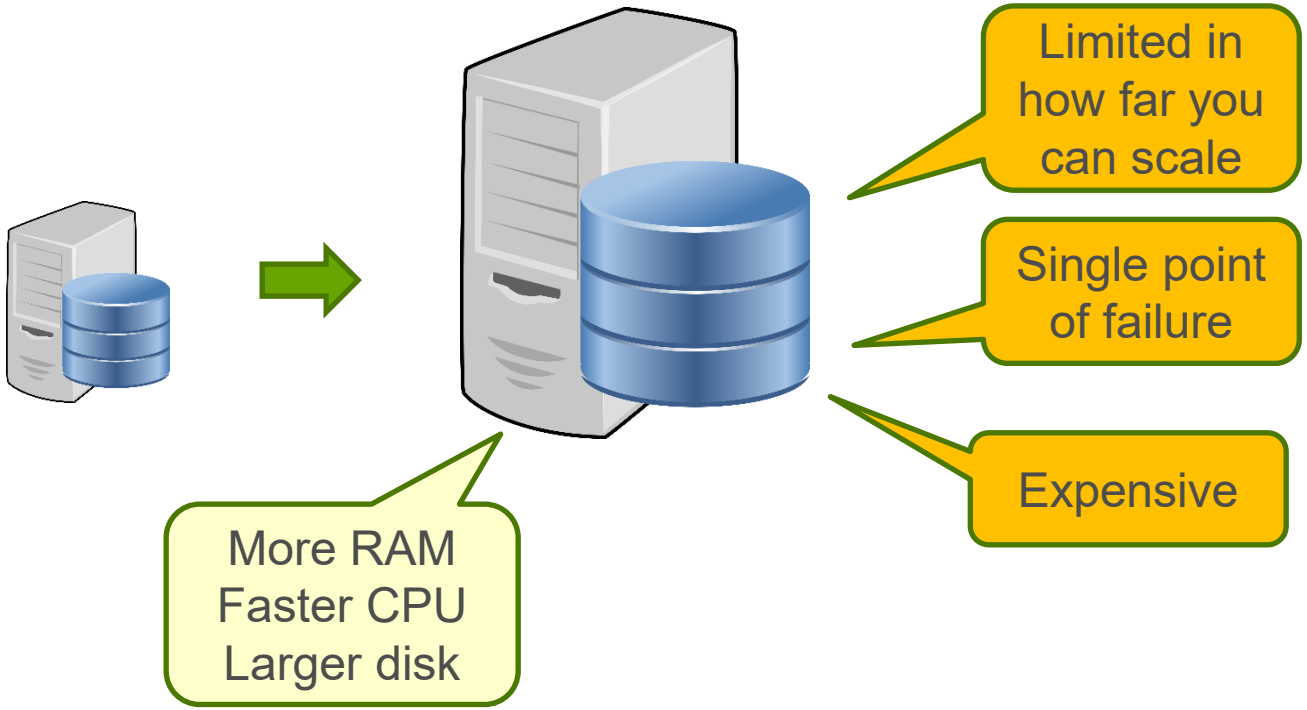
NoSQL Databases: Cassandra

Database Problems

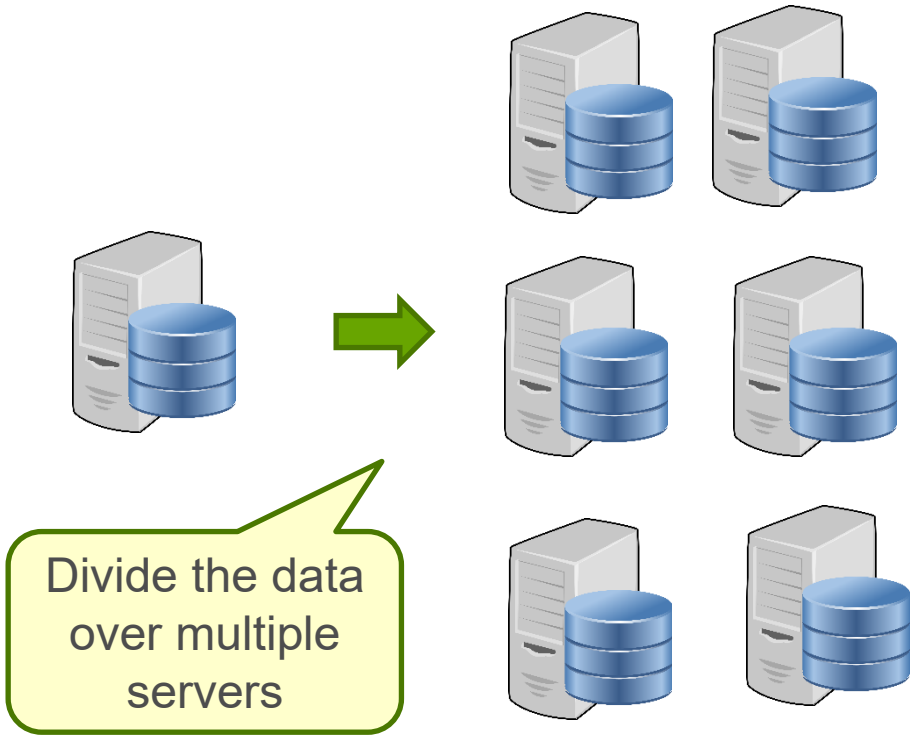
- Too much data
  - Too much load on DB
    - DB becomes the bottleneck

Database Scaling

- Vertical scaling

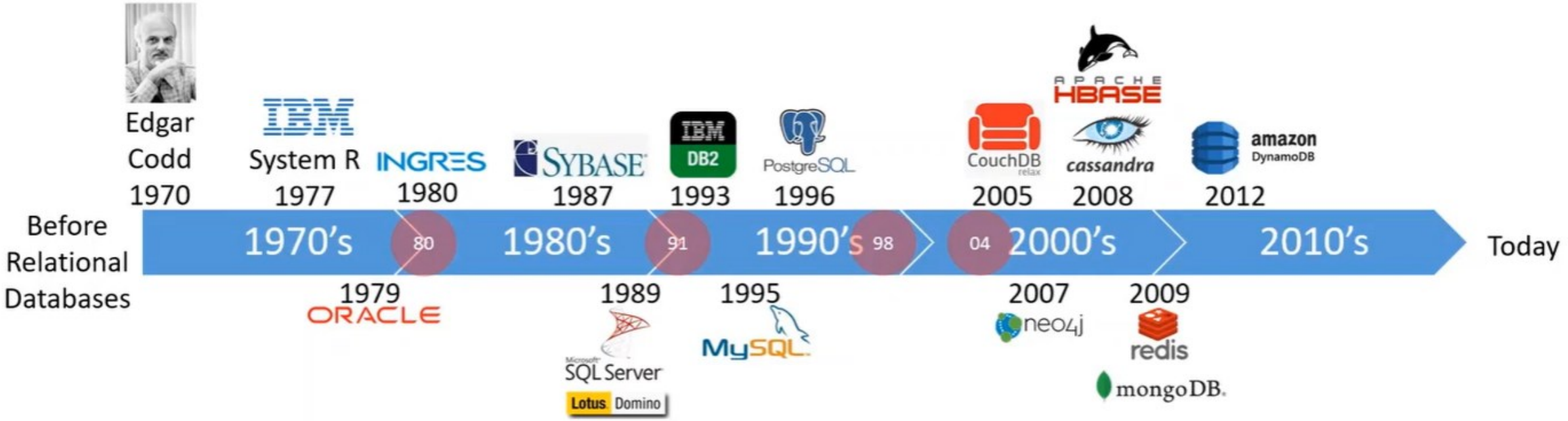


- Horizontal scaling



Today's Requirements on Databases

- Big data (large datasets)
- Varied data formats
- Unstructured/ semi structured data



Sharding in Distributed DBs

Original Table

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

HP1

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
1	TAEKO	OHNUKI	BLUE
2	O.V.	WRIGHT	GREEN

HP2

CUSTOMER ID	FIRST NAME	LAST NAME	FAVORITE COLOR
3	SELDA	BAĞCAN	PURPLE
4	JIM	PEPPER	AUBERGINE

Key based Sharding

Shard Key

COLUMN 1	COLUMN 2	COLUMN 3
A		
B		
C		
D		

Hash values are created from the Shard Key

HASH FUNCTION

COLUMN 1	HASH VALUES
A	1
B	2
C	1
D	2

Shard 1

COLUMN 1	COLUMN 2	COLUMN 3
A		
C		

Shard 2

COLUMN 1	COLUMN 2	COLUMN 3
B		
D		

Range-based Sharding

PRODUCT	PRICE
WIDGET	$118
GIZMO	$88
TRINKET	$37
THINGAMAJIG	$18
DOODAD	$60
TCHOTCHKE	$999

($0-$49.99)

PRODUCT	PRICE
TRINKET	$37
THINGAMAJIG	$18

($50-$99.99)

PRODUCT	PRICE
GIZMO	$88
DOODAD	$60

($100+)

PRODUCT	PRICE
WIDGET	$118
TCHOTCHKE	$999

Problems with Relational Databases

- Scaling writes is very difficult and limited
  - Vertical scaling is limited and is expensive
  - Horizontal scaling is limited and is complex
    - Queries work only within shards
    - Strict consistency and partition tolerance leads to availability problems

A relational database is hard to scale

Problems with Relational Databases contd..

- The schema in a database is fixed
- Schema evolution
  - Adding attributes to an object => have to add columns to table
  - You need to do a migration project
  - Application downtime ...

A relational database is hard to change

Problems with Relational Databases contd..

- Relational schema doesn't easily handle unstructured and semi-structured data
  - Emails, Tweets, Pictures, Audio, Movies, Text

Unstructured data

The university has 5600 students.
John's ID is number 1, he is 18 years old and already holds a B.Sc. degree.
David's ID is number 2, he is 31 years old and holds a Ph.D. degree. Robert's ID is number 3, he is 51 years old and also holds the same degree as David, a Ph.D. degree.

Semi-structured data

```
<University>
 <Student ID="1">
  <Name>John</Name>
  <Age>18</Age>
  <Degree>B.Sc.</Degree>
 </Student>
 <Student ID="2">
  <Name>David</Name>
  <Age>31</Age>
  <Degree>Ph.D. </Degree>
</Student>
....
</University>
```

Structured data

ID	Name	Age	Degree
1	John	18	B.Sc.
2	David	31	Ph.D.
3	Robert	51	Ph.D.
4	Rick	26	M.Sc.
5	Michael	19	B.Sc.

A relational database does not handle unstructured and semi structured data very well

NoSQL Databases

- NoSQL databases, also known as "Not Only SQL" databases, have emerged in response to the limitations of traditional relational databases in handling modern, large-scale, and diverse types of data.
- The need for NoSQL databases arises from various challenges that traditional SQL databases face in certain scenarios.

Advantages of NoSQL (Not Only SQL) Databases

- Capable to handle Big Data
- Suitable for Unstructured and Semi-Structured Data
- Horizontal Scalability
- Provide Flexible Schemas
- Optimized read and write Performance
- Provide Real-time Data processing
- High Availability and Fault Tolerance
- Cost-Effectiveness

Choice between NoSQL and RDBMS

- It's important to note that while NoSQL databases offer many advantages, they are not one-size-fits-all solutions.
- The choice between NoSQL and traditional relational databases should be made based on the specific needs and characteristics of the application.
- RDBMS is a good choice for applications that require strong data consistency, integrity, and transactional support, and where data is well-structured and follows a defined schema.

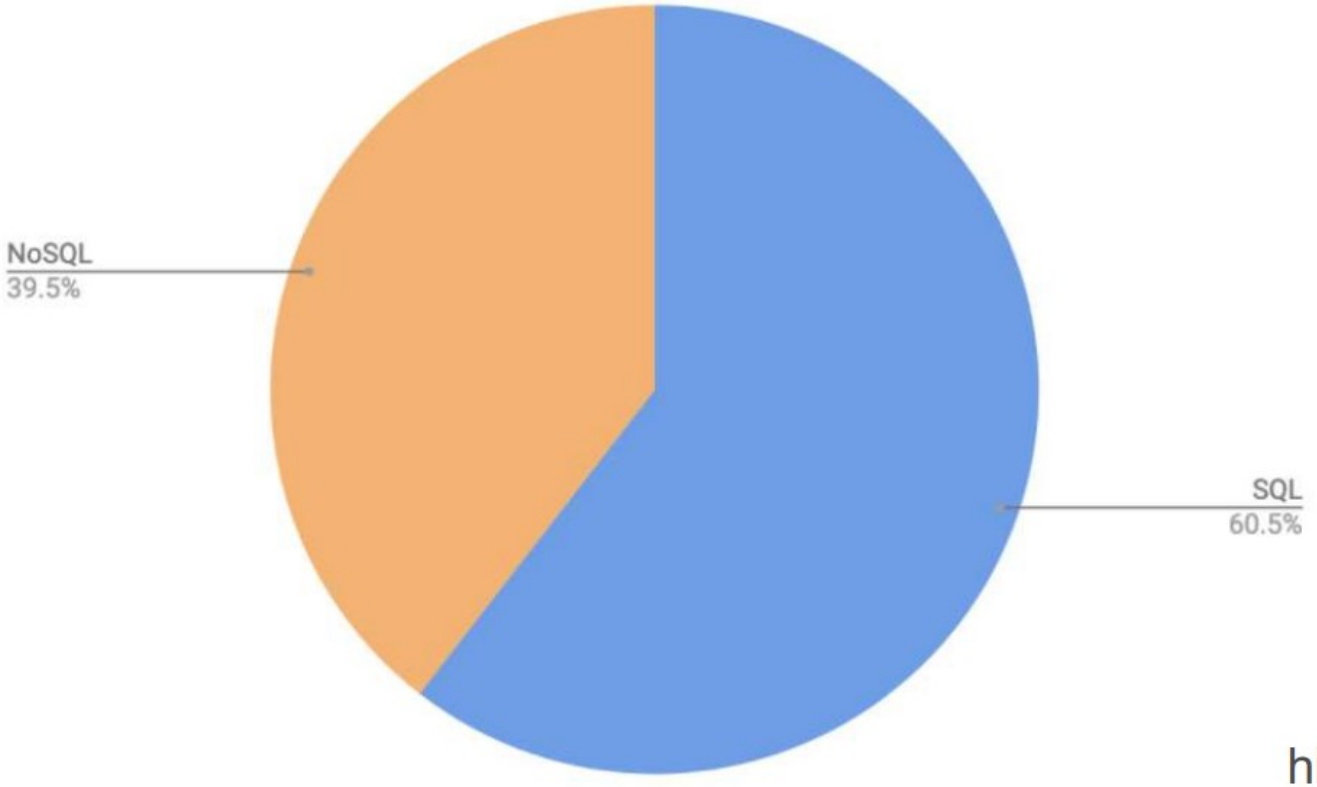
Choice between NoSQL and RDBMS contd..

- NoSQL is a better choice for applications that need to handle large volumes of unstructured or semi-structured data, require extreme scalability, and can tolerate some level of eventual consistency.
- Many modern applications use a combination of both relational and NoSQL databases to meet their diverse data management requirements.

Relational vs. No-SQL usage

SQL Database Use: 60.48%

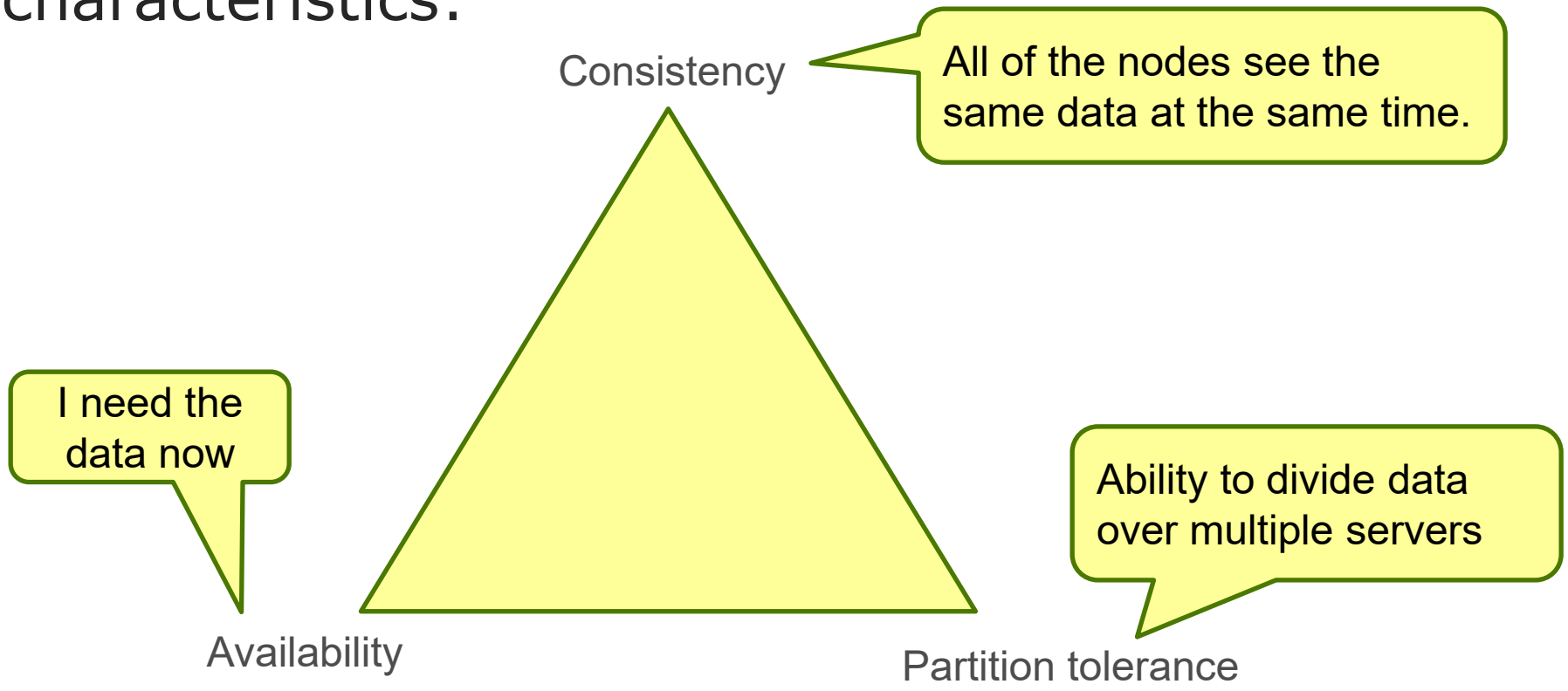
NoSQL Database Use: 39.52%



highscalability.com - 2019

Brewer's CAP Theorem

- A distributed system can support only two of the following three characteristics:



Consistency

- Strict consistency
  - The data that I read is always correct
  - You never loose data
- Eventual consistency
  - The data might not be correct
    - But will eventually become correct

- NoSQL databases prioritize flexibility and scalability over strict consistency.
- They are designed to handle diverse data types and large volumes.

ACID - BASE

- **Relational**
  - ACID
    - **Atomicity**: All parts of transaction complete or none complete
    - **Consistency**: Only valid data written to database
    - **Isolation**: Parallel transactions do not impact each other's execution
    - **Durability**: Once transaction committed, it remains

- **NoSQL**
  - BASE
    - Basically available
    - Soft state
      - No schema
    - Eventual consistency

Types of NoSQL Databases

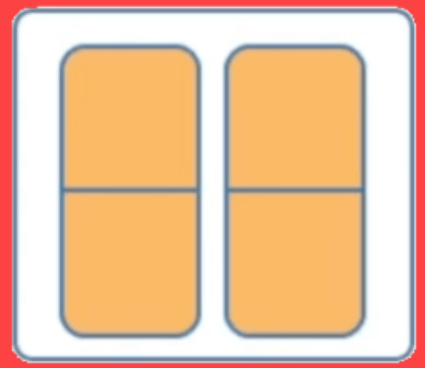
Key Value

Example:
Riak, Tokyo Cabinet, Redis server, Memcached, Scalaris

Document-Based

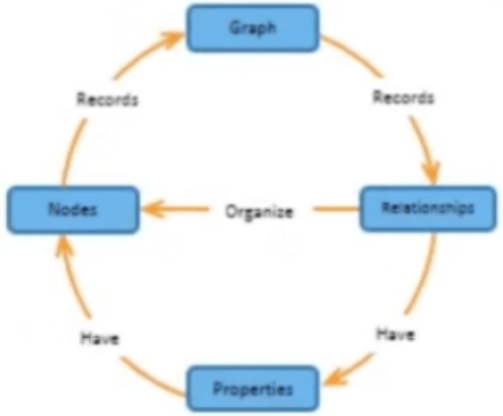
Example:
MongoDB, CouchDB, OrientDB, RavenDB

Column-Based



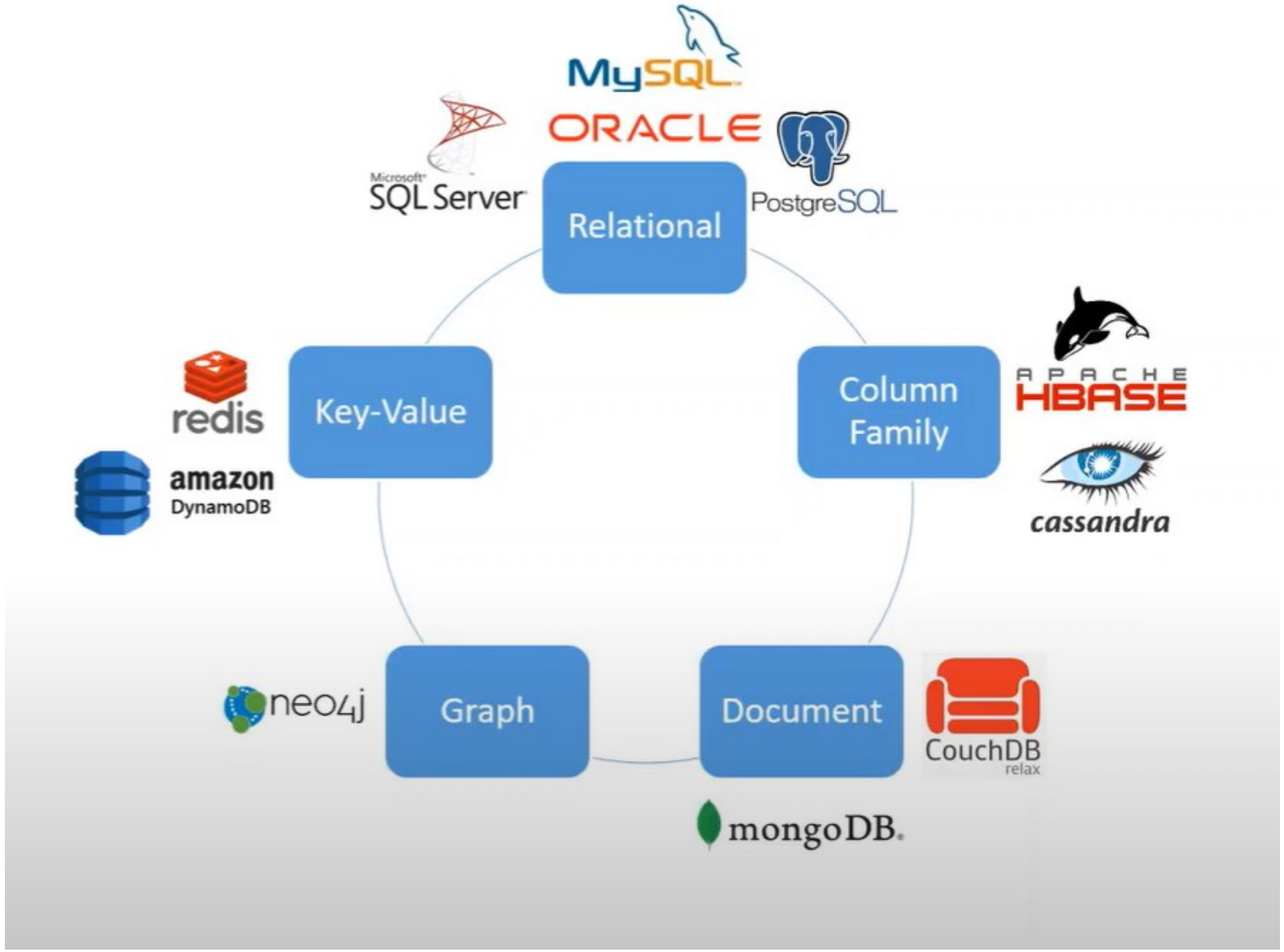
Example:
BigTable, Cassandra, Hbase, Hypertable

Graph-Based



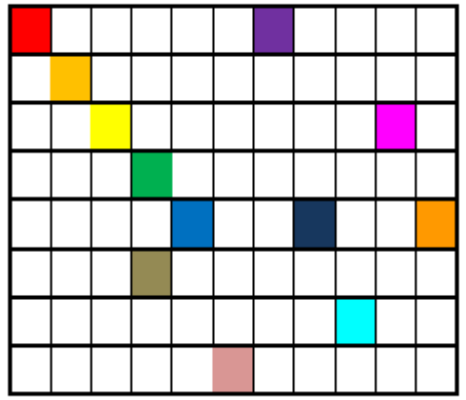
Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

Five Different Types of Databases



Column family

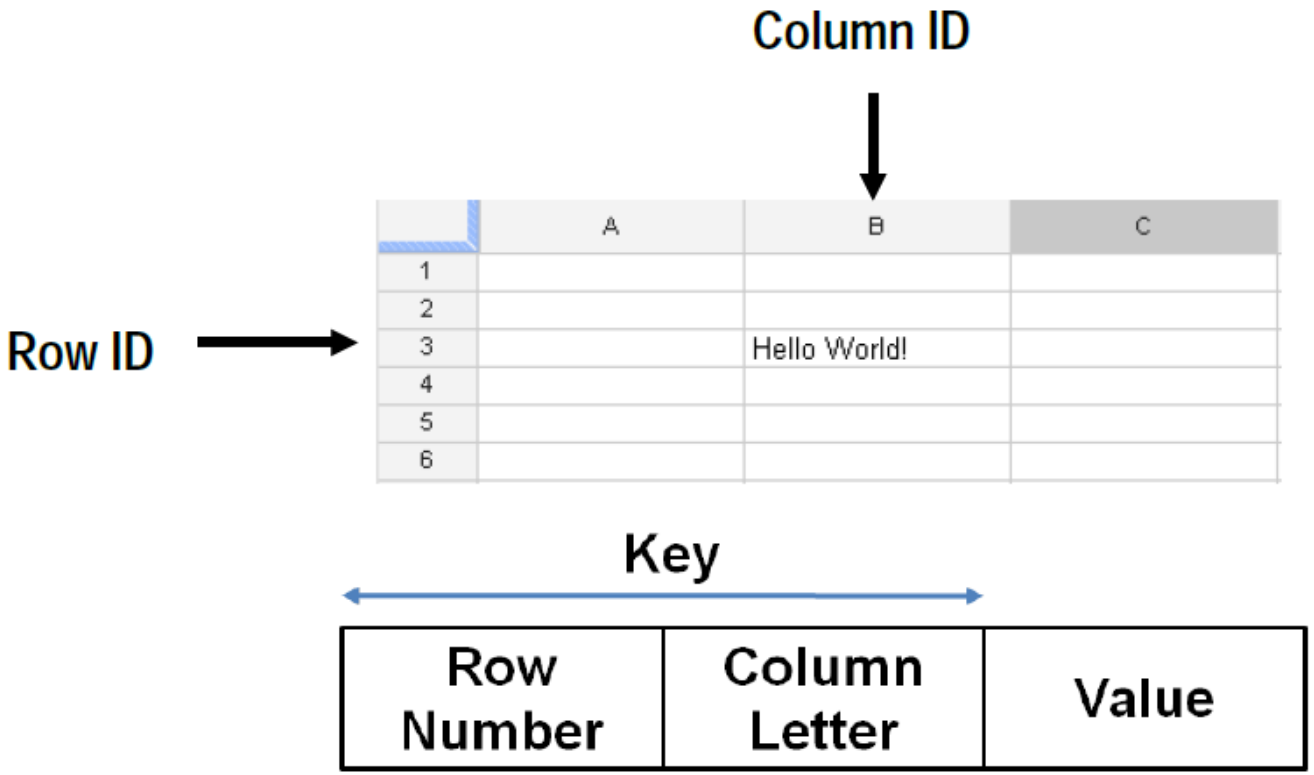
- Place data in a certain column
- Ideal for high-variability data sets
- Colum families allow to query all columns that have a specific property or properties
- Allow new columns to be inserted without doing an "alter table"



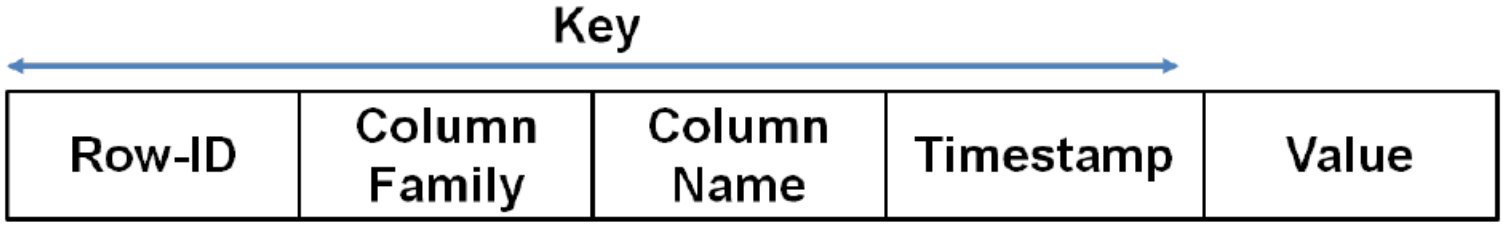
- Players
  - Cassandra
  - HBase
  - Google BigTable

Excel

- The key is the row number and the column letter



Column family



- Keys can have many attributes
  - Column family (group columns in column families)
    - Group column families in super-families
  - Timestamp
- Queries can be done on rows, column families and column names, timestamp

Relational

User

CustID	Name
1	Adam
2	Bob
3	Chris
4	Dave

Watches

CustId	MovieID
1	10
1	11
1	12
2	11
2	13
2	15
2	16
3	10
3	15
4	14
4	15
4	16

Movie

MovieID	Movie Name
10	Oceans 11
11	Pulp Fiction
12	Jurassic Park
13	Sonic the Hedgehog
14	Titanic
15	Deadpool
16	Star Wars

What movies has Adam watched? ⇐ Need to join 3 tables (slow)

Who watched Deadpool? ⇐ Need to join 3 tables (slow)

Column family

User	Movies			
	Oceans 11	Pulp Fiction	Jurassic Park	
Adam	TRUE	TRUE	TRUE	
	Pulp Fiction	Sonic	Deadpool	Star Wars
Bob	TRUE	TRUE	TRUE	TRUE
	Oceans 11	Deadpool		
Chris	TRUE	TRUE		
	Titanic	Deadpool	Star Wars	
Dave	TRUE	TRUE	TRUE	

What movies has Adam watched? ⇐ Very easy and fast

Who watched Deadpool? ⇐ Difficult

Column Family NoSQL Database: Cassandra

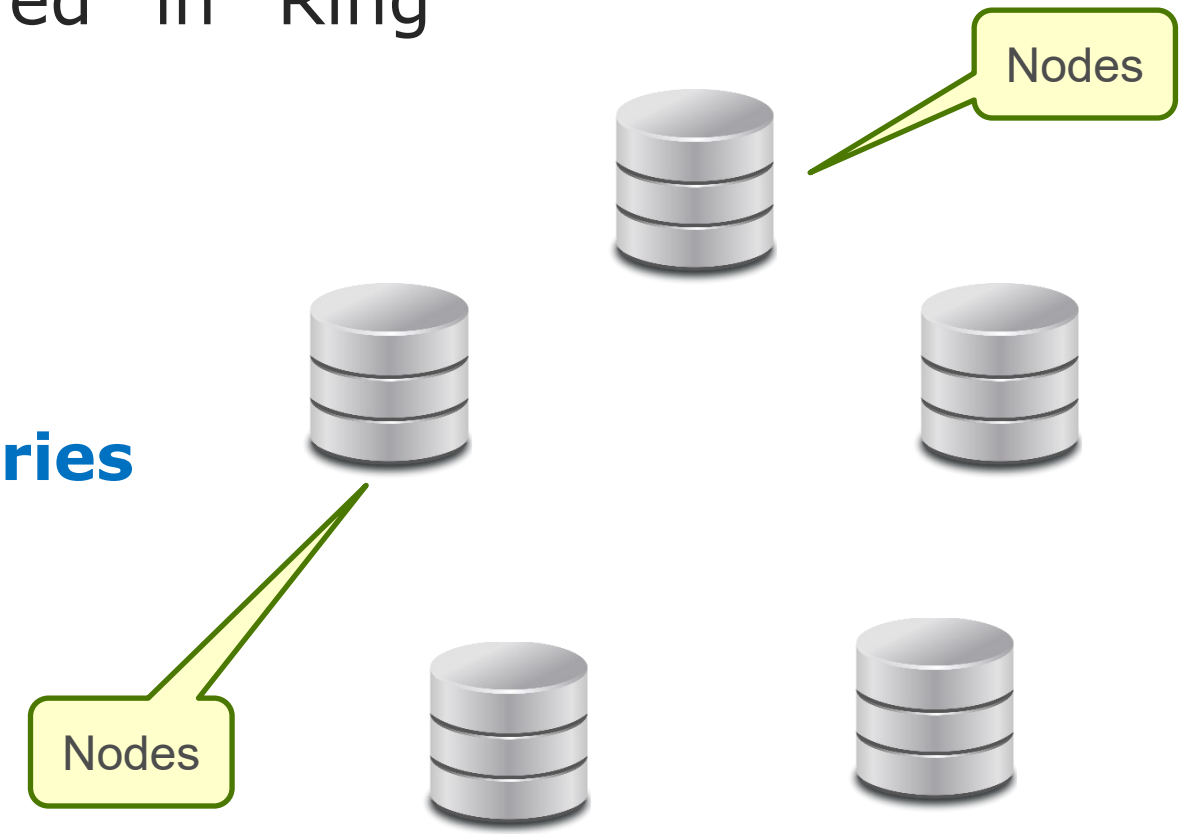
- Apache Cassandra was initially developed at Facebook and it came from Amazon's highly available Dynamo DB and Google's BigTable data model.
- It is a Column family NoSQL distributed database designed for scalability and high availability for Big Data.
- Unlimited elastically scalable
  - Installation is on multiple different nodes
- Always available (no downtime)
- Uses partitioning

When is Cassandra an option to use?

- You have a large amount of data that does not fit on one server
- The structure of the data changes often
- The system needs to be available all the time
  - No downtime
- Optimized read performance

Cassandra: Distributed Partitioned DB

- Nodes are logically structured in Ring Topology.
- No joins
- Query should run on 1 node
  - **Partition data based on queries**
- Denormalization
  - Optimized reads
  - Data duplication



Denormalization

- Denormalization is a strategy used on a database to increase performance.
- In computing, denormalization is the process of trying to improve the read performance of a DB, at the expense of losing some write performance, by adding redundant copies of data.
- Pros: Quick Read, simple queries
- Cons: Multiple writes, Manual integrity

Partitioning – Data is Distributed

customerNr	firstName	lastName
1	Frank	Brown
2	Bob	Johnson
3	John	Jackson
4	Frank	Young
5	Sue	Jones

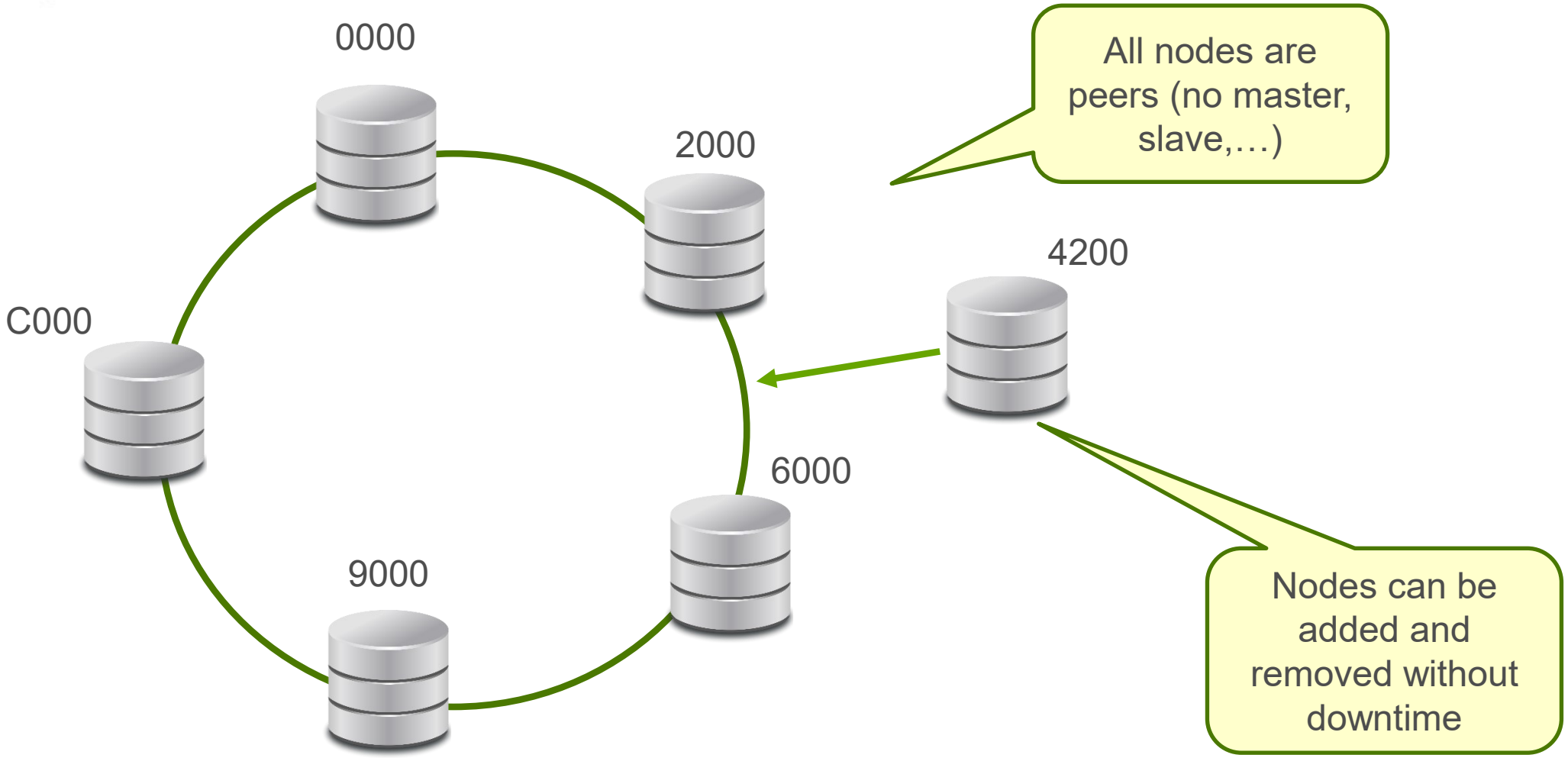
Hashed value of key associated with data partition is used to assign it to a node in the ring.

customerNr	firstName	lastName
1	Frank	Brown
2	Bob	Johnson

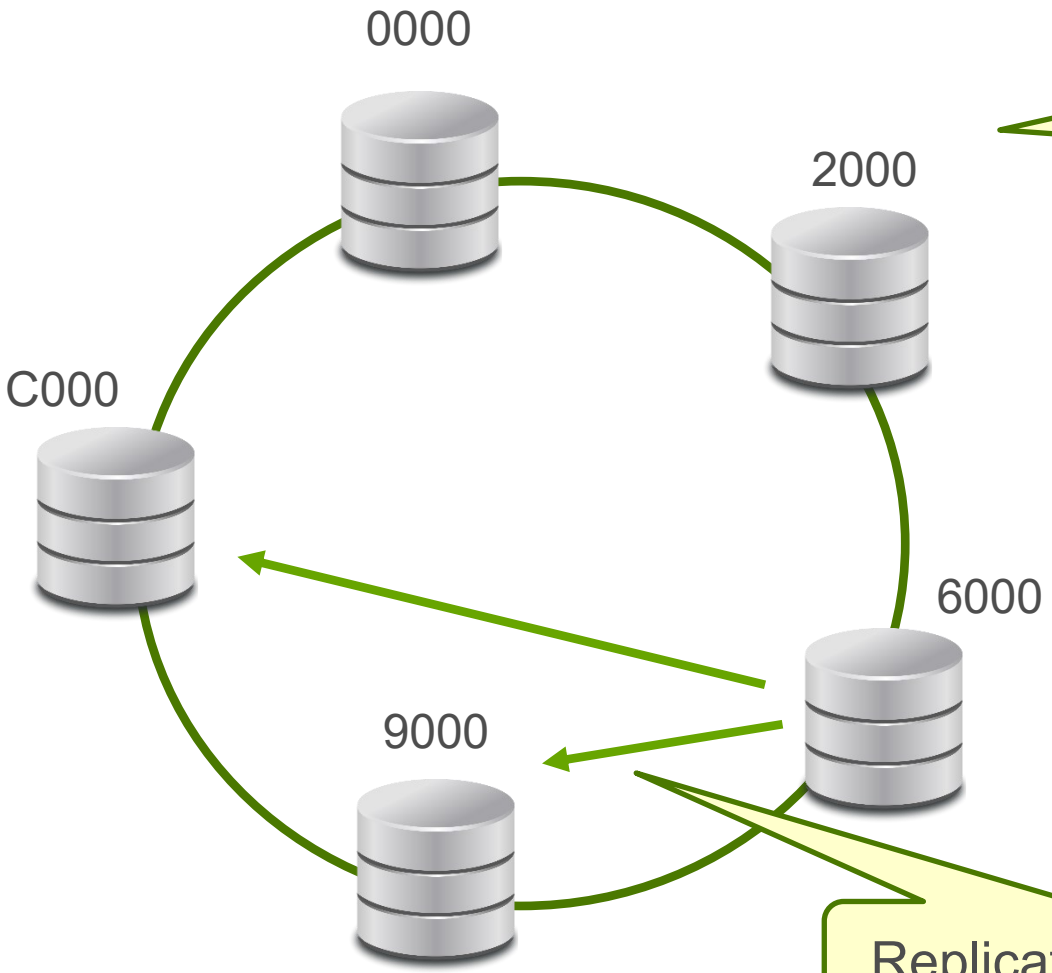
customerNr	firstName	lastName
3	John	Jackson
5	Sue	Jones

customerNr	firstName	lastName
4	Frank	Young

Cassandra Architecture



Data Replication for Fault Tolerance



Consistency can be tuned for read and write. You can choose consistency level: ALL, Quorum, ONE

Consistency level typically represents a fraction of nodes within the cluster, which are required to acknowledge the operation for it to be considered successful.

Replication factor = 3

Quorum Consistency Level

- **Quorum** consistency helps in preventing inconsistencies and conflicts in a distributed database.
  - For example, if you have a replication factor of 3 (meaning same data is stored on 3 nodes), you can use a quorum consistency level of QUORUM, which requires acknowledgment from at least 2 replicas for both reads and writes.
  - This means that you achieve a majority vote among the replicas for a read or write operation, which ensures strong consistency.
- There are more consistency levels other than ALL, Quorum and ONE

Cassandra Data Model

Keyspace population

Table *population_by_country_city*

Country	City	Population
USA	Los Angeles	4,000,000
	New York	8,000,000
DE	Berlin	3,350,000
	Nuremberg	500,000
UK	London	9,200,000
FR	Paris	2,230,000
	Toulouse	1,100,000
CA	Montreal	4,200,000
	Toronto	6,200,000
AU	Sydney	4,900,000

columns

partitions

rows

partition key

clustering column

Data columns

Cassandra Data Model contd..

- Tabular data model where data is organized in rows and columns.
- Keyspace is a container of tables. Like a database or schema in RDBMS but is more flexible/dynamic.
- Groups of related rows called *partitions* are stored together on the same node (or nodes).
- Each row contains a partition key.

Creating a Table in CQL (Cassandra Query Language)

```
CREATE TABLE my_keyspace.my_table (
    partition_key_column_1 data_type,
    partition_key_column_2 data_type,
    clustering_column data_type,
    other_columns data_type,
    PRIMARY KEY ((partition_key_column_1,
                  partition_key_column_2), clustering_column)
);
```

Subset of SQL

Creating a Table in CQL

keyspace

table

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

column definitions

Primary key

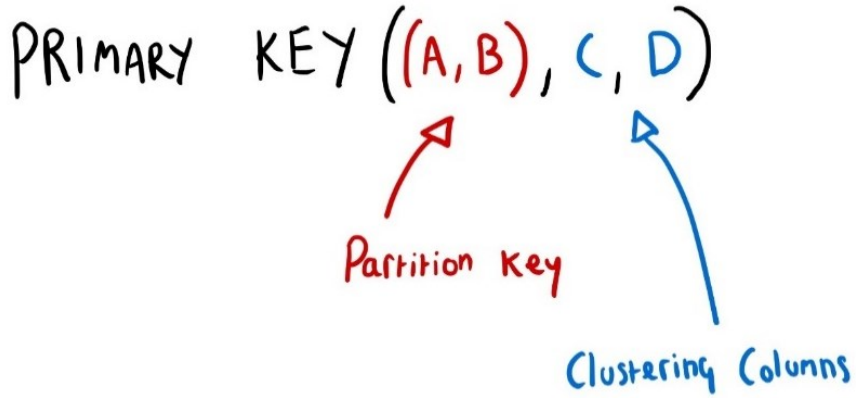
Partition key

Clustering columns

Clustering columns naturally order the data.

Primary Key

- **Primary Key**
  - An identifier for a row. Consists of
    - at least one **Partition Key**
    - and zero or more **Clustering Columns**
  - Must ensure uniqueness
  - May define sorting through clustering columns



Good Examples:

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

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

Bad Example:

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

Partition key

- An identifier for a partition which **partitions rows**.
- Decides where to place this record
- Consists of at least one column, may have more if needed
- Needs to be mentioned while querying the table

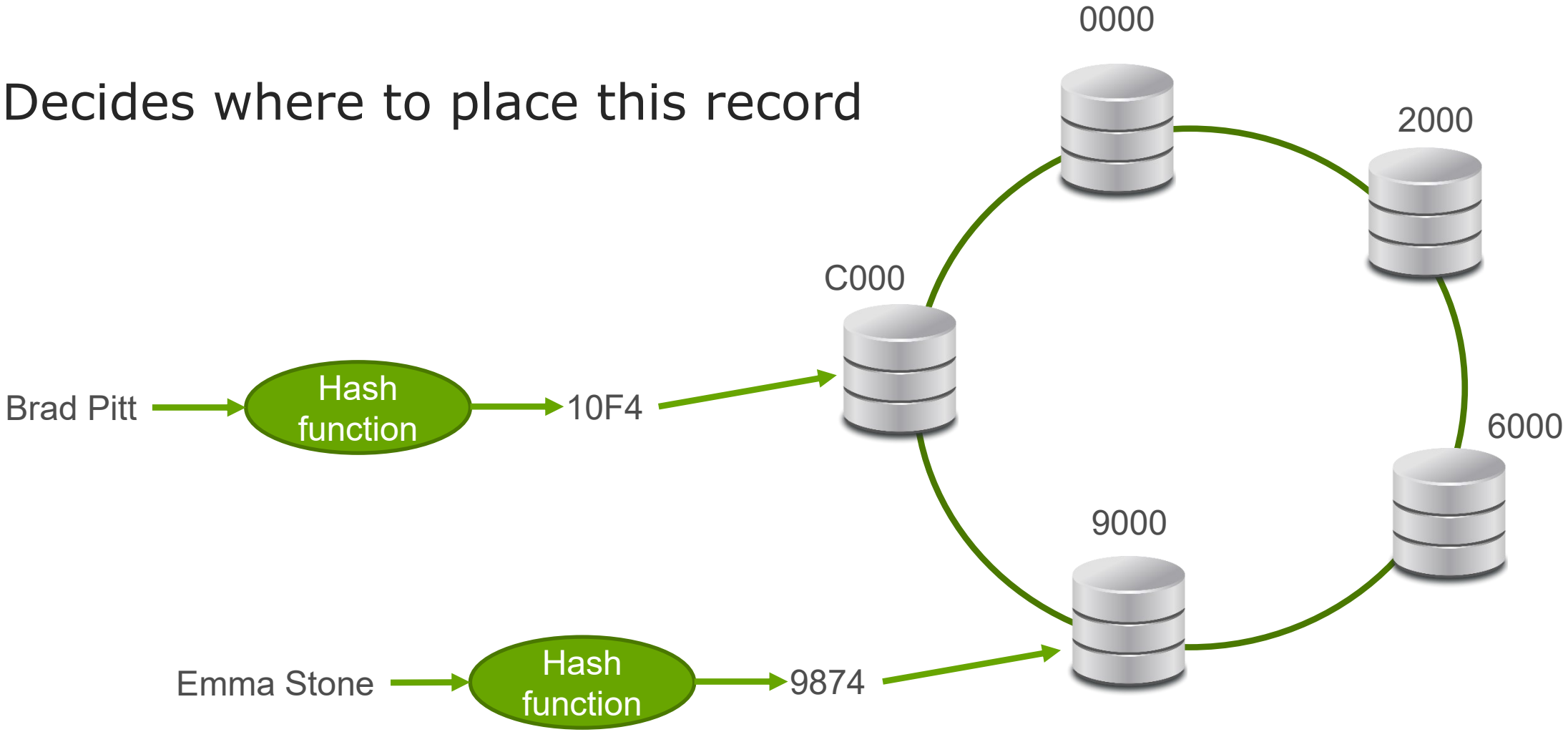
```
CREATE TABLE killrvideo.users_by_city (
      city text,
      last_name text,
      first_name text,
      address text,
      email text,
      PRIMARY KEY ((city), last_name, first_name, email));
```

Partition key

Clustering columns

Partition key

- Decides where to place this record



Clustering Columns

- Used either to ensure uniqueness or sorting order for rows.
- Group rows together so they can be found fast in a query
- Attributes you want to query on besides the partition key
- Optional to add

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

Partition key

Clustering columns

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

✗ Not Unique

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

✓

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

✗ Not Sorted

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

✓

Primary Key cannot be changed!

- Once you create a table with a Primary Key, you cannot change that Primary Key. You can add or remove as many data columns as you want but you cannot change the Primary Key.
- Why?
- Because, the Primary Key is defining the physical location of the data around the cluster (which row should go to which node), so if you change that, you will need to change the physical location of all the data.

Rules of a Good Partition

1. Store together what you retrieve together

Example: open a video? Get the comments in a single query!

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

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

Rules of a Good Partition contd..

2. Avoid big and constantly growing partitions

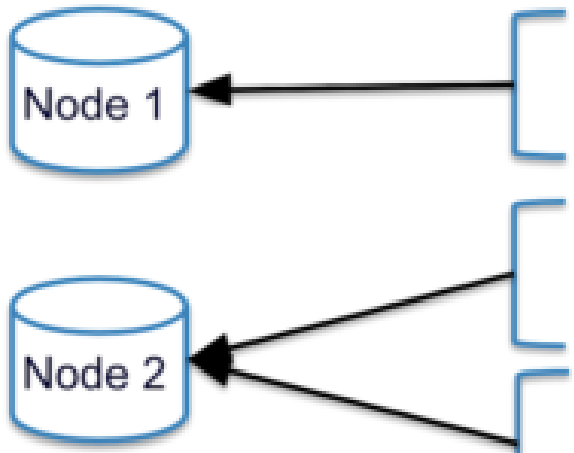
- Big partitions will increase read times because you'll need to query multiple servers.
- Up to 2 billion cells per partition, ~100K rows or ~100MB is ideal

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

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

BUCKETING

Bucketing Example



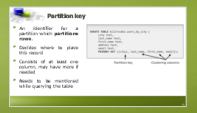
Sensor #	Date	Timestamp	Metric1	Metric2	Metric3
1	2015-01-01	20150101-000000	5.01	5.67	0.678
1	2015-01-01	20150101-000010	5.01	5.67	0.678
1	2015-01-01	20150101-000020	5.05	5.8	0.678
1	2015-01-02	20150102-000000	5.01	5.67	0.678
1	2015-01-02	20150102-000010	5.01	5.67	0.678
1	2015-01-02	20150102-000020	5.05	5.8	0.678
2	2015-01-02	20150102-000000	6.01	7.67	0.978
2	2015-01-02	20150102-000010	6.01	7.67	0.698
2	2015-01-02	20150102-000020	6.05	8.8	0.679

Partition Key

Clustering Key

Primary Key

PRIMARY KEY ((Sensor,Date),Timestamp)



Rules of a Good Partition contd..

3. Avoid hot partitions

- Hot partition is a partition that is getting hit a lot more frequently compared to other partitions.
- Good data modelling is spreading the data across all the partitions evenly so that all the nodes are hit at an equal rate.

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

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

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

Data types

- TEXT, INT, BIGINT, FLOAT, DOUBLE, BOOLEAN, DECIMAL
- UUID
- TIMEUUID
- Collection
  - SET
  - LIST
  - MAP

Always use UUID (Universally Unique Identifier) for generated unique key

Always use TIMEUUID for generated unique key alongwith a timestamp

```
cqlsh> Create table University.Teacher
   ... (
   ...  id int,
   ... Name text,
   ... Email set<text>,
   ... Primary key(id)
   ... );
cqlsh>
```

collection of emails

```
cqlsh> insert into University.Teacher(id,Name,Email) values(1,'Guru99',{'abc@gma
il.com','xyz@hotmail.com'});
cqlsh>
```

insertion in collection

Naming Convention

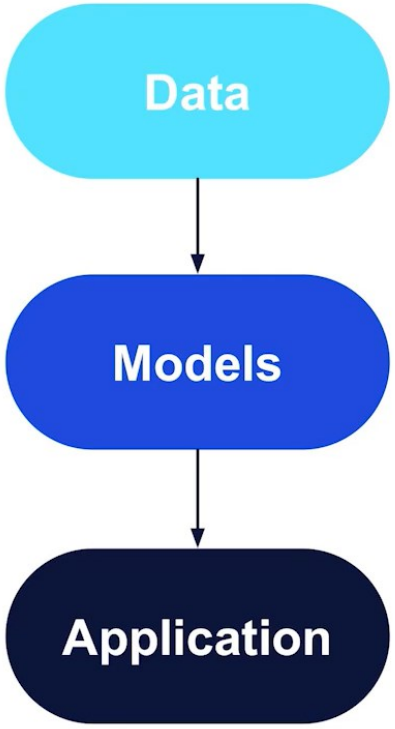
- Snake_case is mostly used
- You can add the partition column name after “by” to make the table name more meaningful.
  - users_by_rating
  - population_by_country

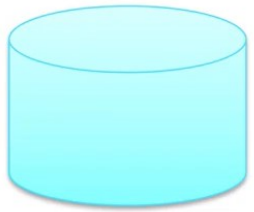
Data modeling

- When migrating a relational database to Cassandra, it's important to understand that Cassandra is a NoSQL database designed for **scalability** and **high availability**. As a result, the data modeling approach is quite different from that of a traditional relational database.
- In Cassandra, you need to design your tables based on your specific query patterns.
- This approach is known as "**query-driven data modeling**" in Cassandra.

Data modeling

- RDBMS





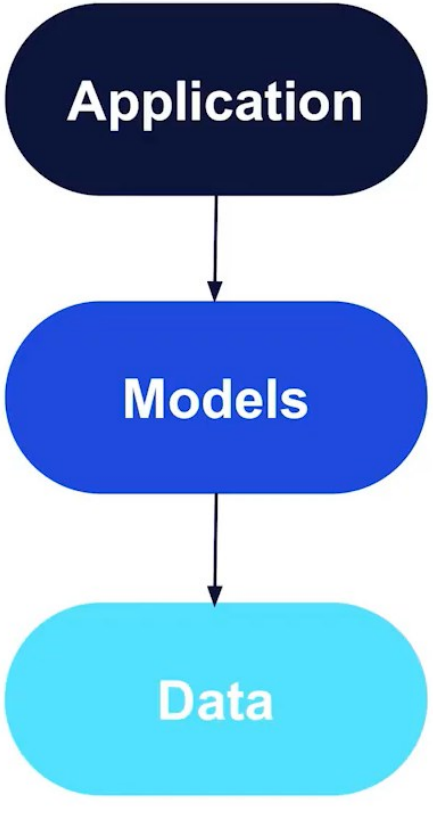
Employees

userId	firstName	lastName
1	Edgar	Codd
2	Raymond	Boyce

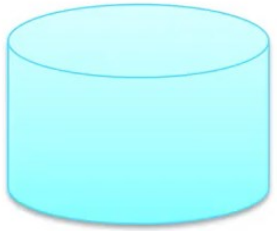
Department

departmentId	department
1	Engineering
2	Math

- Cassandra



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



Data modeling

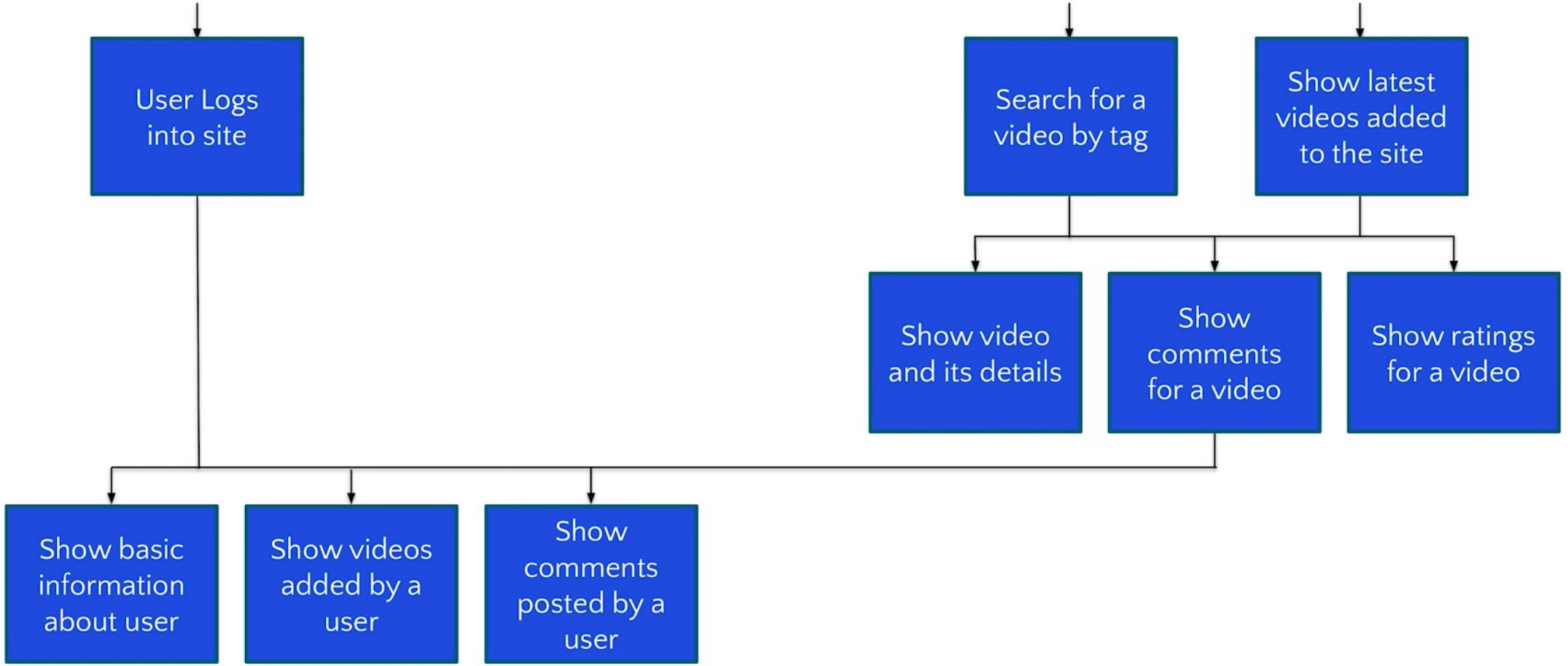
Relational data modelling:

1. Analyze raw data
2. Identify entities, their poroperties and relations
3. Design tables, using normalization and FKs.
4. Use JOIN when doing queries to join normalizaed data from multiple tables.

NoSQL Data Modelling:

1. Analyze user behavior (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.

Video Application Workflow



Queries that support the workflow

Users

- User Logs into site — Find user by email address
- Show basic information about user — Find user by id

Comments

- Show comments for a video — Find comments by video (latest first)
- Show comments posted by a user — Find comments by user (latest first)

Ratings

- Show ratings for a video — Find ratings by video

Videos

- Search for a video by tag — Find video by tag
- Show latest videos added to the site — Find videos by date (latest first)
- Show video and its details — Find video by id
- Show videos added by a user — Find videos by user (latest first)

Relational way

User Logs into site

Find user by email address

Show basic information about user

Find user by id

- Single Users table with all user data and PK as userID.

```
CREATE TABLE users (
        userID int,
        firstName text,
        lastName text,
        email text,
        createdDate timestamp,
        PRIMARY KEY (userID)
);
```

Cassandra way

User Logs into site

Find user by email address

```
CREATE TABLE users_by_email (
       email text,
       password text,
       userid uuid,
       PRIMARY KEY (email)
);
```

Show basic information about user

Find user by id

```
CREATE TABLE users_by_id (
       userid uuid,
       firstname text,
       lastname text,
       email text,
       created_date timestamp,
       PRIMARY KEY (userid)
);
```

In the Cassandra way, you'd have multiple tables optimized for specific query patterns, and data duplication is common to support various access paths efficiently.

Video queries

Show video and its details

Find video by id

```
CREATE TABLE video_by_id (
        videoid uuid,
        userid uuid,
        name text,
        description text,
        location text,
        location_type int,
        preview_image_location text,
        tags set<text>,
        added_date timestamp,
        PRIMARY KEY (videoid)
);
```

Show videos added by a user

Find videos by user (latest first)

```
CREATE TABLE videos_by_user (
        userid uuid,
        added_date timestamp,
        videoid uuid,
        name text,
        preview_image_location text,
        PRIMARY KEY ((userid), added_date, videoid)
)
WITH CLUSTERING ORDER BY (
        added_date DESC,
        videoid ASC);
```

CQL Cassandra Query Language:
Create Table

ratings_by_user

email	title	year	rating
joe@datastax.com	Alice in Wonderland	2010	9
joe@datastax.com	Edward Scissorhands	1990	10
jen@datastax.com	Alice in Wonderland	1951	8
jen@datastax.com	Alice in Wonderland	2010	10

```
CREATE TABLE ratings_by_user (
  email TEXT,
  title TEXT,
  year INT,
  rating INT,
  PRIMARY KEY ((email), title, year)
); ↩
```

Partition key

Clustering columns

CQL Cassandra Query Language: Insert

email	title	year	rating
joe@datastax.com	Alice in Wonderland	2010	9
joe@datastax.com	Edward Scissorhands	1990	10
jen@datastax.com	Alice in Wonderland	1951	8
jen@datastax.com	Alice in Wonderland	2010	10

```
CREATE TABLE ratings_by_user (
  email TEXT,
  title TEXT,
  year INT,
  rating INT,
  PRIMARY KEY ((email), title, year)
); ↩
```

```
INSERT INTO ratings_by_user (email, title, year,
rating)
VALUES ('joe@datastax.com', 'Alice in Wonderland',
2010, 9);
INSERT INTO ratings_by_user (email, title, year,
rating)
VALUES ('joe@datastax.com', 'Edward Scissorhands',
1990, 10);
INSERT INTO ratings_by_user (email, title, year,
rating)
VALUES ('jen@datastax.com', 'Alice in Wonderland',
2010, 10);
INSERT INTO ratings_by_user (email, title, year,
rating)
VALUES ('jen@datastax.com', 'Alice in Wonderland',
1951, 8); ↩
```

Ratings by movie

title	year	email	rating
Alice in Wonderland	2010	jen@datastax.com	10
Alice in Wonderland	2010	joe@datastax.com	9
Alice in Wonderland	1951	jen@datastax.com	8
Edward Scissorhands	1990	joe@datastax.com	10

```
CREATE TABLE ratings_by_movie (
  title TEXT,
  year INT,
  email TEXT,
  rating INT,
  PRIMARY KEY ((title, year), email)
); ↩
```

```
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Alice in Wonderland', 2010,
'jen@datastax.com', 10);
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Alice in Wonderland', 2010,
'joe@datastax.com', 9);
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Alice in Wonderland', 1951,
'jen@datastax.com', 8);
INSERT INTO ratings_by_movie (title, year, email,
rating)
VALUES ('Edward Scissorhands', 1990,
'joe@datastax.com', 10); ↩
```

Actors by movie

title	year	first_name	last_name
Alice in Wonderland	2010	Anne	Hathaway
Alice in Wonderland	2010	Johnny	Depp
Edward Scissorhands	1990	Johnny	Depp

```
CREATE TABLE actors_by_movie (
  title TEXT,
  year INT,
  first_name TEXT,
  last_name TEXT,
  PRIMARY KEY ((title, year), first_name, last_name)
); ↩
```

Movies by actor

first_name	last_name	title	year
Johnny	Depp	Alice in Wonderland	2010
Johnny	Depp	Edward Scissorhands	1990
Anne	Hathaway	Alice in Wonderland	2010

```
CREATE TABLE movies_by_actor (
  first_name TEXT,
  last_name TEXT,
  title TEXT,
  year INT,
  PRIMARY KEY ((first_name, last_name), title, year)
); ↩
```

Movies by user

email	watched_on	title	year
joe@datastax.com	2020-04-28	Edward Scissorhands	1990
joe@datastax.com	2020-03-08	Alice in Wonderland	2010
joe@datastax.com	2020-02-13	Toy Story 3	2010
joe@datastax.com	2020-01-22	Despicable Me	2010
joe@datastax.com	2019-12-30	Alice in Wonderland	1951
jen@datastax.com	2011-10-01	Alice in Wonderland	2010

```
CREATE TABLE movies_by_user (
  email TEXT,
  title TEXT,
  year INT,
  watched_on DATE,
  PRIMARY KEY ((email), watched_on, title, year)
) WITH CLUSTERING ORDER BY (watched_on DESC); ↩
```

Data duplication

email	title	year	rating
joe@datastax.com	Alice in Wonderland	2010	9
joe@datastax.com	Edward Scissorhands	1990	10
jen@datastax.com	Alice in Wonderland	1951	8
jen@datastax.com	Alice in Wonderland	2010	10

title	year	email	rating
Alice in Wonderland	2010	jen@datastax.com	10
Alice in Wonderland	2010	joe@datastax.com	9
Alice in Wonderland	1951	jen@datastax.com	8
Edward Scissorhands	1990	joe@datastax.com	10

title	year	first_name	last_name
Alice in Wonderland	2010	Anne	Hathaway
Alice in Wonderland	2010	Johnny	Depp
Edward Scissorhands	1990	Johnny	Depp

email	watched_on	title	year
joe@datastax.com	2020-04-28	Edward Scissorhands	1990
joe@datastax.com	2020-03-08	Alice in Wonderland	2010
joe@datastax.com	2020-02-13	Toy Story 3	2010
joe@datastax.com	2020-01-22	Despicable Me	2010
joe@datastax.com	2019-12-30	Alice in Wonderland	1951
jen@datastax.com	2011-10-01	Alice in Wonderland	2010

Queries

Never do this type of a query on real production data in Cassandra!

Q1. Retrieve all rows:

```
SELECT * FROM users; ↩
```

Q2. Retrieve one row/partition:

```
SELECT * FROM users
WHERE email = 'joe@datastax.com'; ↩
```

Q3. Retrieve two rows/partitions:

```
SELECT * FROM users
WHERE email IN ('joe@datastax.com',
                'jen@datastax.com'); ↩
```

Queries

title	year	email	rating
Alice in Wonderland	2010	jen@datastax.com	10
Alice in Wonderland	2010	joe@datastax.com	9
Alice in Wonderland	1951	jen@datastax.com	8
Edward Scissorhands	1990	joe@datastax.com	10

```
CREATE TABLE ratings_by_movie (
  title TEXT,
  year INT,
  email TEXT,
  rating INT,
  PRIMARY KEY ((title, year), email)
); ↩
```

A query can only filter on fields in the primary key

You cannot directly filter on rating

Queries

```
SELECT COUNT(rating) AS count,
       SUM(rating) AS sum,
       AVG(CAST(rating AS FLOAT)) AS avg,
       MIN(rating) AS min,
       MAX(rating) AS max
FROM   ratings_by_movie
WHERE  title = 'Alice in Wonderland'
  AND  year  = 2010; ↩
```

```
SELECT * FROM ratings_by_user
WHERE email = 'jim@datastax.com'
ORDER BY title ASC, year DESC;
```

```
SELECT   title, year,
         AVG(CAST(rating AS FLOAT)) AS avg_rating
FROM     ratings_by_movie
GROUP BY title, year; ↩
```