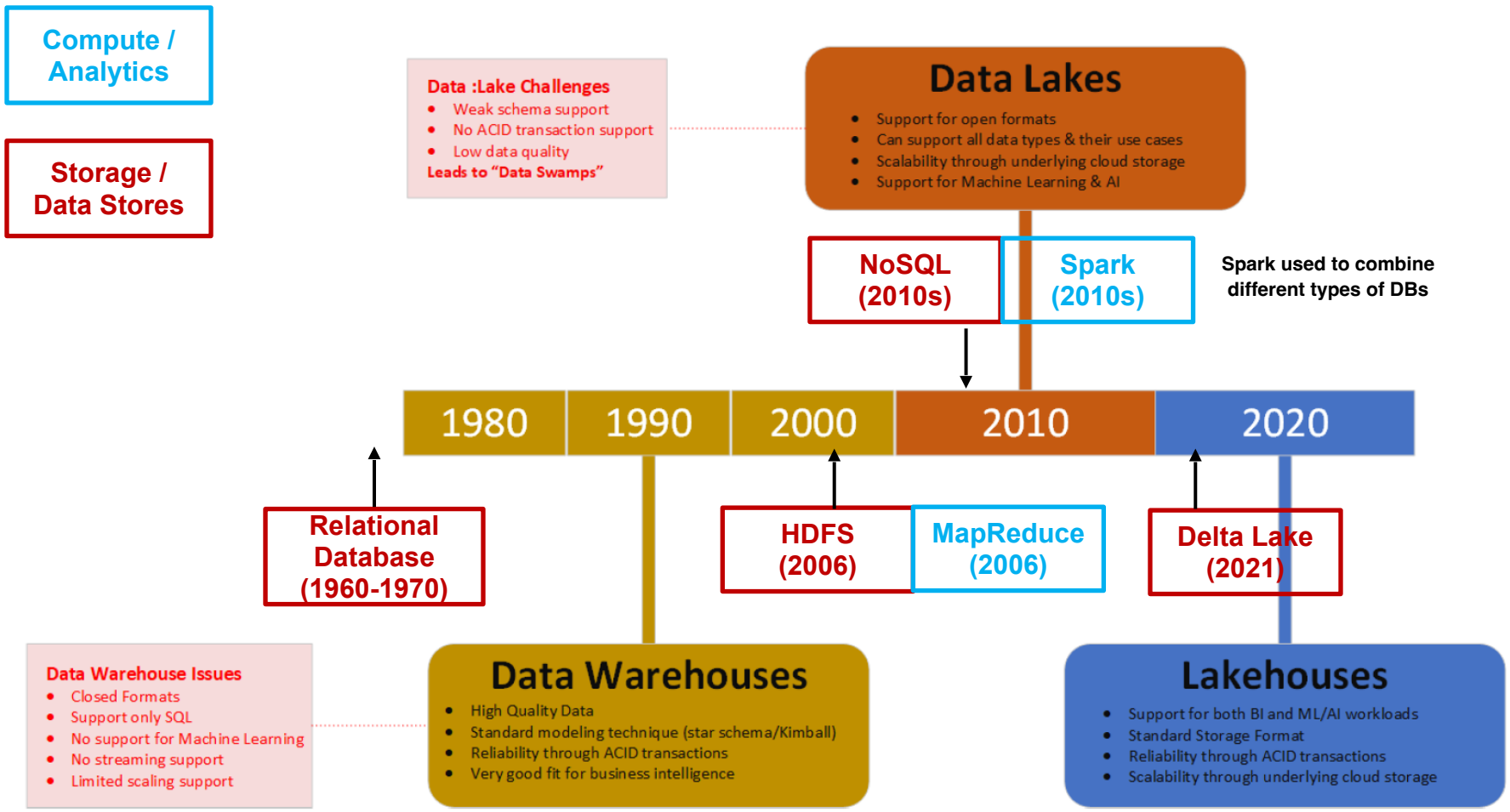
CS4225/CS5425 Big Data Systems for Data Science

NoSQL Overview

Ai Xin
School of Computing
National University of Singapore
aixin@comp.nus.edu.sg

School of Computing

Evolution of Data Architectures



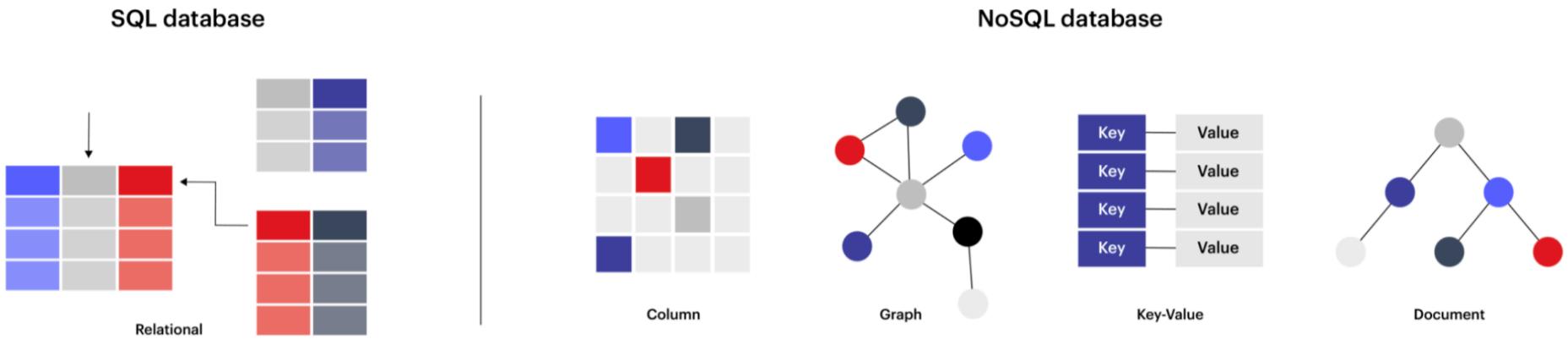
Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. Eventual Consistency
4. Duplication
5. Data Partitioning
6. Query Processing in NoSQL

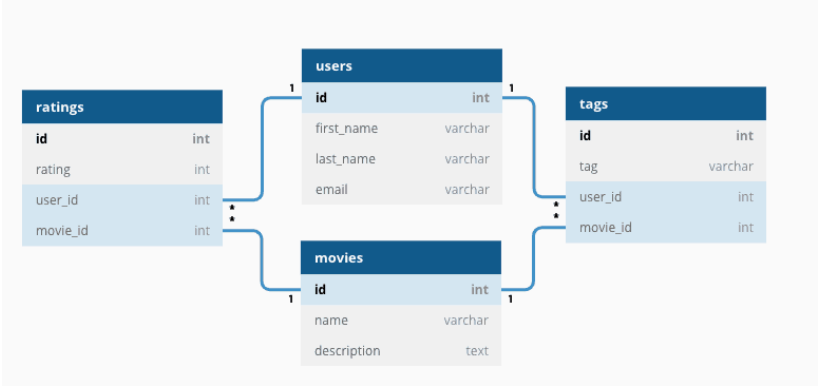
Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

What is NoSQL?

- NoSQL mainly refers to open source, distributed, **non-relational database**, i.e. it stores data in a format other than relational tables
- ”SQL” here refers to traditional relational database management systems (“DBMS”)
  - (This is a slight misnomer; here “SQL” is used to refer to relational databases, not the querying language. In fact, NoSQL systems can involve SQL-like querying languages in many cases.)
- NoSQL has come to stand for “Not Only SQL”, i.e. using relational and non-relational databases alongside one another, each for the tasks they are most suited for



What is NoSQL?



Traditional Relational Database Management System (DBMS)

```
>>> db.inventory.find()
[
  {
    _id: ObjectId("6138d8b1611d9dc433ad73dd"),
    item: 'canvas',
    qty: 100,
    tags: [ 'cotton' ],
    size: { h: 28, w: 35.5, uom: 'cm' }
  },
  {
    _id: ObjectId("6138d917611d9dc433ad73de"),
    item: 'table',
    qty: 5,
    price: 10
  }
]
```

mongoDB® (Document Store)

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

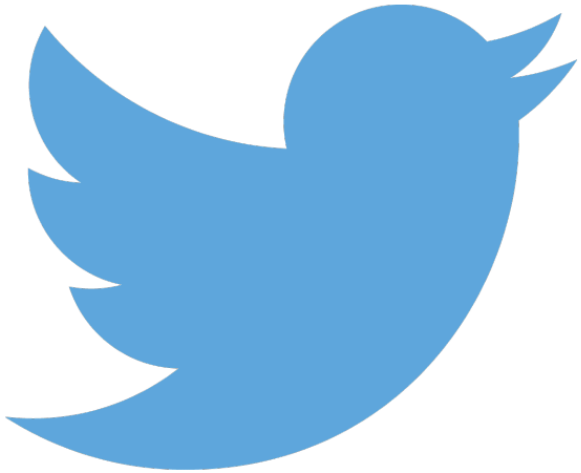
redis (Key-Value Store)

Metaphor: NoSQL = No Rules?

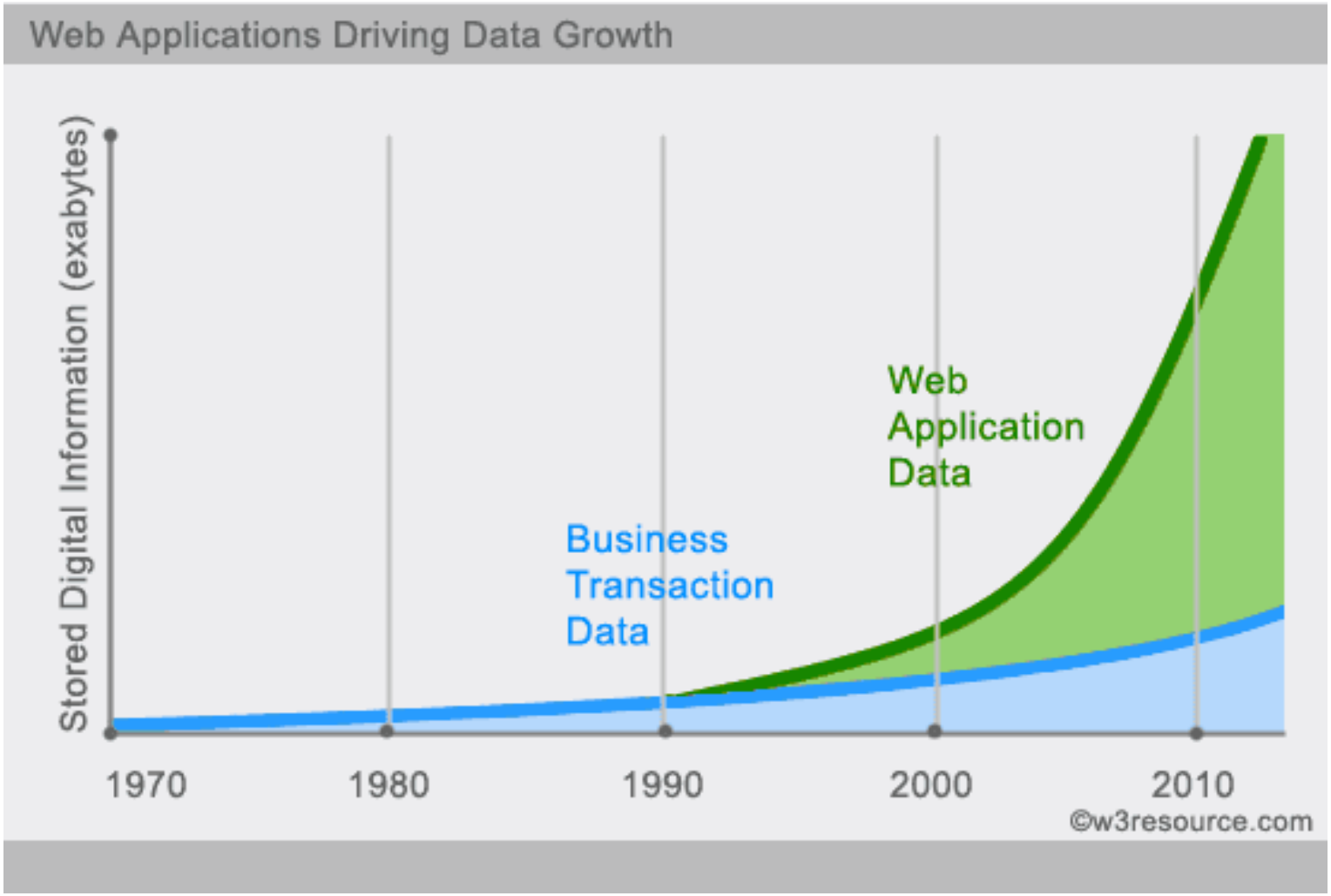
Relational Databases

Non-Relational Databases (NoSQL)

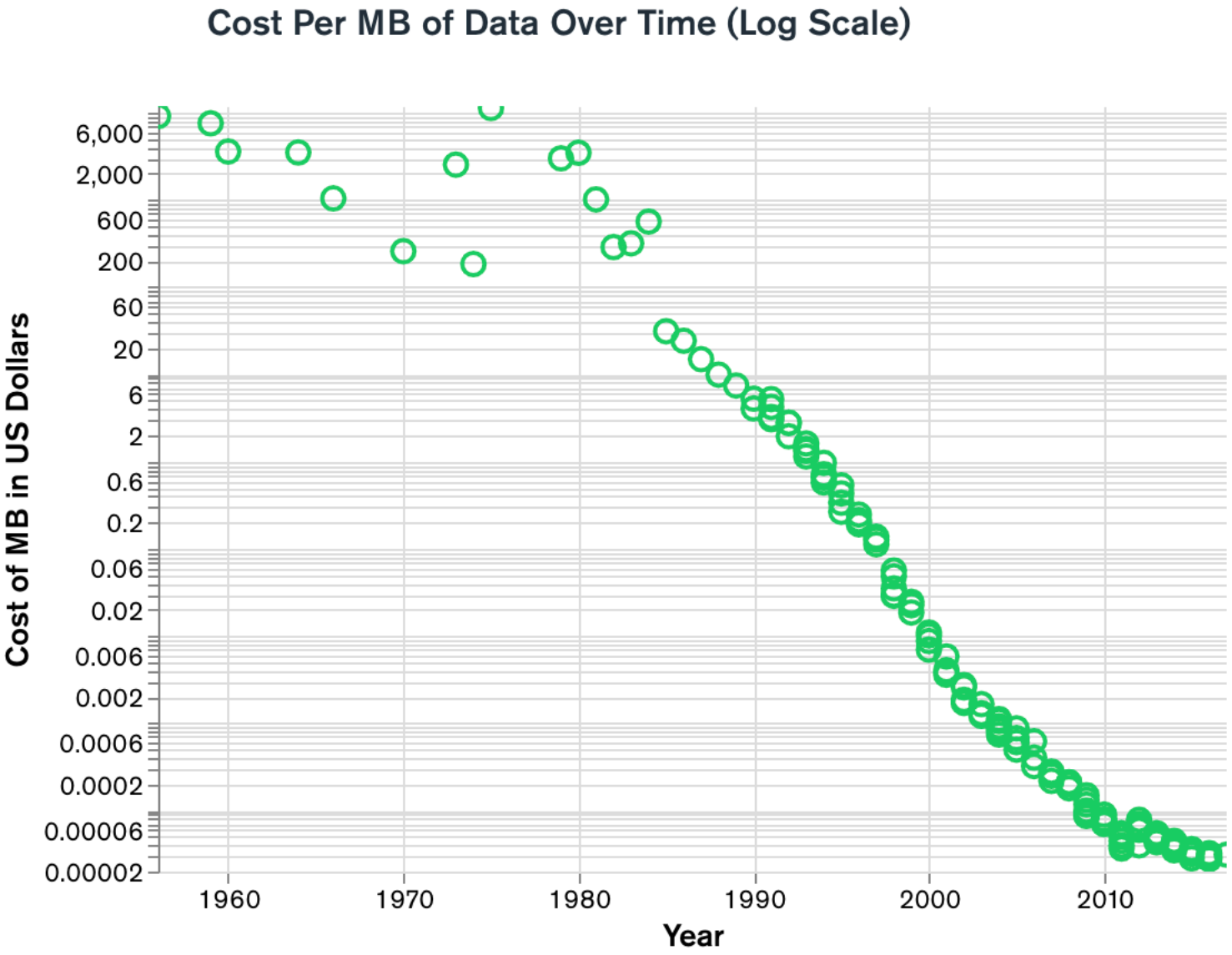
Who uses NoSQL?



Why NoSQL: Growth in Web Applications



Why NoSQL: Volume, Velocity



A Brief Overview of NoSQL Systems

1. Horizontally scalability
2. Replicate/distribute data over many servers

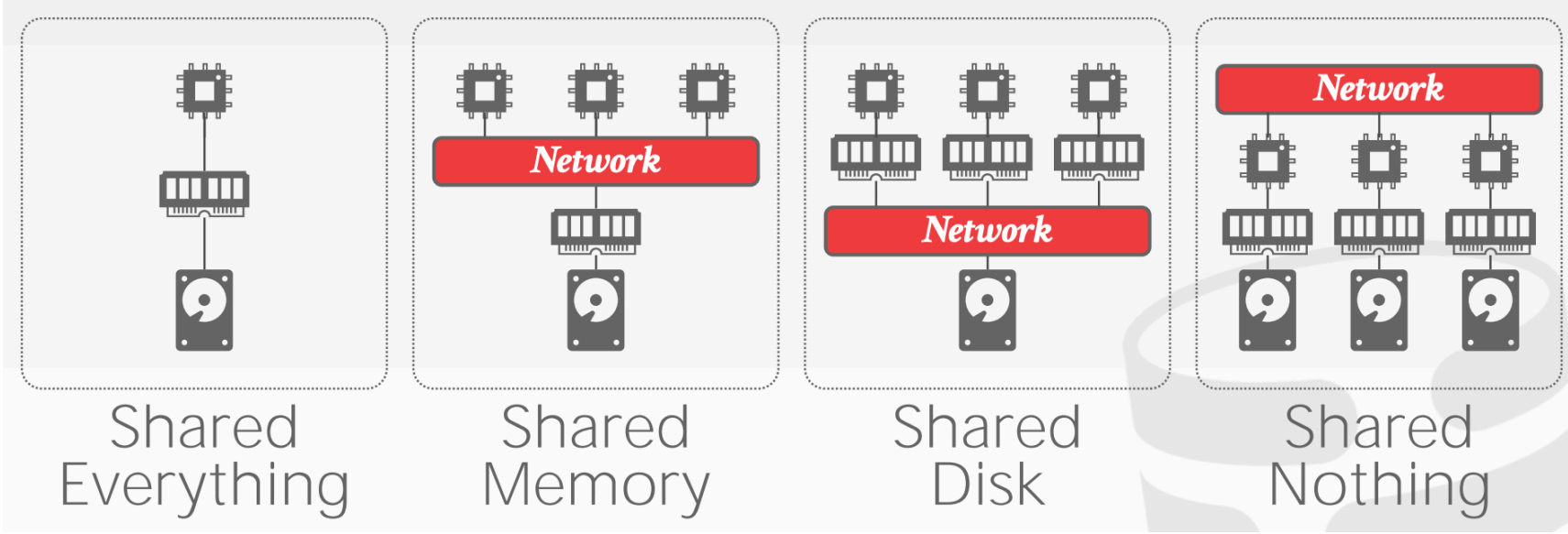
Volume, Velocity

3. Simple call interface
4. Often weaker concurrency model than RDBMS
5. Efficient use of distributed indexes and RAM
6. Flexible schemas

Variety

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Distributed Database Architectures



Examples:

Single node DBMS

Supercomputers

Cloud databases

NoSQL (mostly)

ORACLE RAC presto snowflake

Cockroach LABS H-Store redis cassandra mongoDB CouchDB relax MySQL Cluster

Credit: Andy Pavlo (CMU CS15-446)

Assumption of Distributed Database

- All nodes in a distributed database are well-behaved (i.e. they follow the protocol we designed for them; not 'adversarial' or trying to corrupt the database)
  - (Out of syllabus: If we cannot trust the other nodes, we need a **Byzantine Fault Tolerant** protocol; blockchains fall into this category)

Data Transparency

- Users should not be required to know how the data is physically distributed, partitioned, or replicated.
- A query that works on a single node database should still work on a distributed database.

Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. Eventual Consistency
4. Duplication
5. Data Partitioning
6. Query Processing in NoSQL

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

(Major) Types of NoSQL databases

- Key-value stores

- Wide column databases

- Document stores

- Graph databases

Key-Value Stores

Source: Wikipedia (Keychain)

Key-Value Stores: Data Model

Phone directory

Key	Value
Paul	(091) 9786453778
Greg	(091) 9686154559
Marco	(091) 9868564334

MAC table

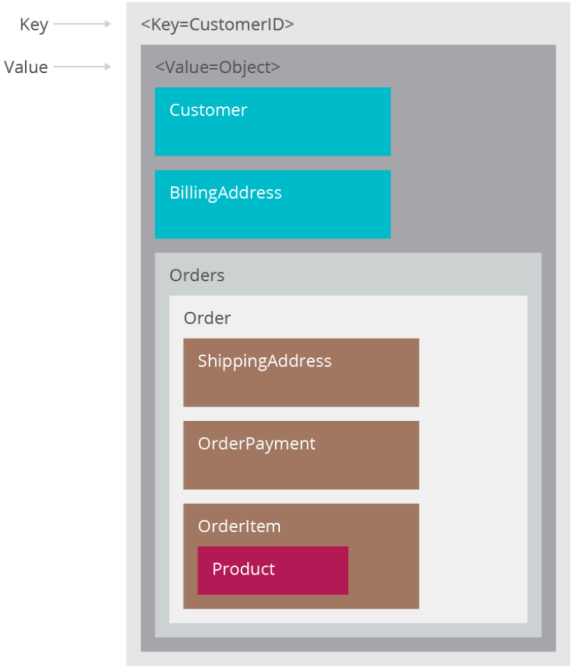
Key	Value
10.94.214.172	3c:22:fb:86:c1:b1
10.94.214.173	00:0a:95:9d:68:16
10.94.214.174	3c:1b:fb:45:c4:b1

- Stores associations between keys and values
  - Based on Amazon's Dynamo paper (2007)
- Keys are usually primitives and can be queried
  - For example, ints, strings, raw bytes, etc.
- Values can be primitive or complex; usually cannot be queried
  - Examples: ints, strings, lists, JSON, HTML fragments, BLOB (basic large object), etc.

DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6 (2007): 205-220.

Key-Value Stores: Operations

- Very simple API:
  - Get – fetch value associated with key
  - Put – set value associated with key
- Optional operations:
  - Multi-get
  - Multi-put
  - Range queries
- Suitable for:
  - Small continuous read and writes
  - Storing 'basic' information (e.g. raw chunks of bytes),
  - When complex queries are not required / rarely required
- Example Applications
  - Storing user sessions
  - Caches
  - User data that is often processed individually: e.g. patient medical data?
    - Only if no queries like 'How many patients who took X treatment recovered?'



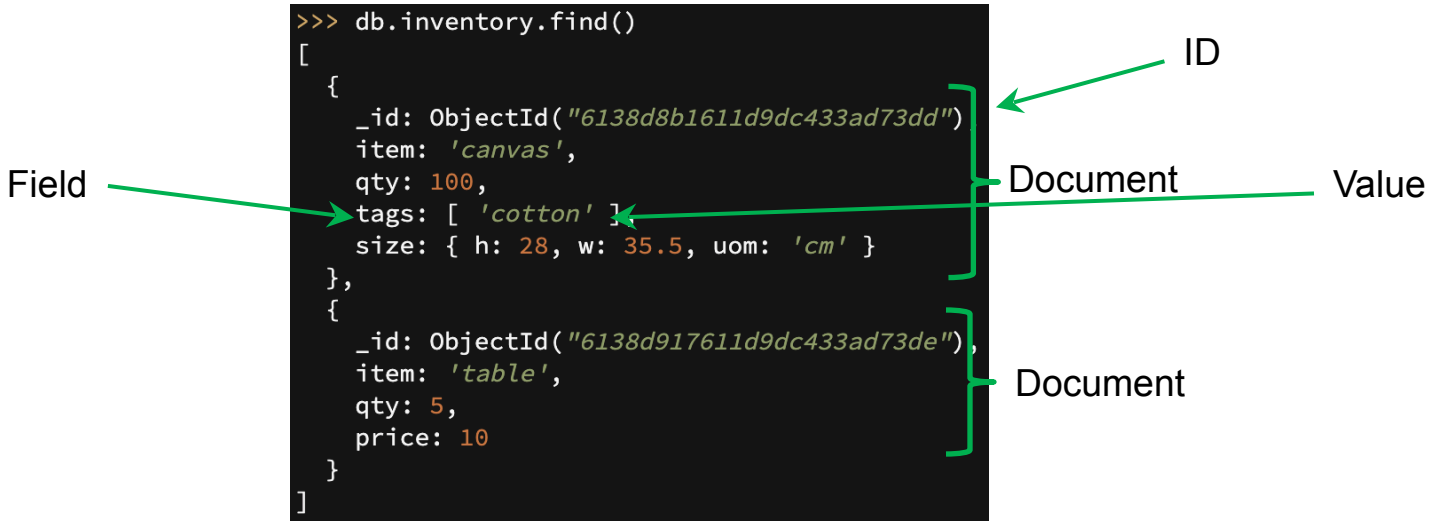
Key-Value Stores: Implementation

- Non-persistent:
  - Just a big in-memory hash table
  - Examples: Memcached, Redis
    - (But: these can also back up the data to disk periodically)
- Persistent
  - Data is stored persistently to disk
  - Examples: RocksDB, Dynamo, Riak

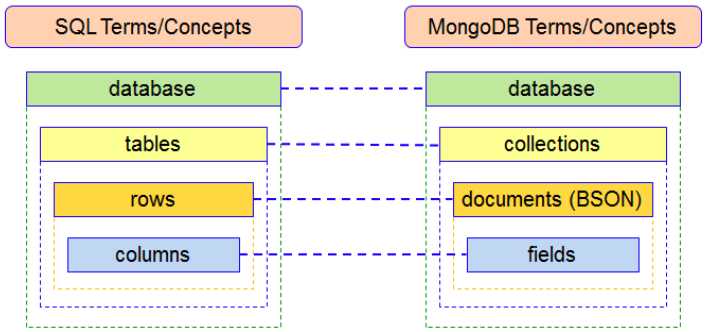
Document Stores

DeCandia, Giuseppe, et al. "Dynamo: Amazon's highly available key-value store." *ACM SIGOPS operating systems review* 41.6 (2007): 205-220.

Document Stores: Document Model



- A database can have multiple **collections**
- Collections have multiple **documents**
- A document is a JSON-like object: it has **fields and values**
  - Different documents can have different fields
  - Can be nested: i.e. JSON objects as values



Document Stores: Querying

- Unlike (basic) key value stores, document stores allow some querying based on the content of a document
- CRUD = Create, Read, Update, Delete

CRUD: Create

In MongoDB

```
db.users.insert (
   {
     name: "sue",
     age: 26,
     status: "A"
   }
)
```

collection

field: value

field: value

field: value

document

In SQL

```
INSERT INTO users
            ( name, age, status )
VALUES      ( "sue", 26, "A" )
```

table

columns

values/row

CRUD: Read

In MongoDB

```
db.users.find(                          <- collection
    { age: { $gt: 18 } },               <- query criteria
    { name: 1, address: 1 }             <- projection
).limit(5)                              <- cursor modifier
```

In SQL

```
SELECT _id, name, address   <- projection
FROM   users                <- table
WHERE  age > 18             <- select criteria
LIMIT  5                    <- cursor modifier
```

CRUD: Update

In MongoDB

```
db.users.update(                          <-- collection
    { age: { $gt: 18 } },                 <-- update criteria
    { $set: { status: "A" } },            <-- update action
    { multi: true }                       <-- update option
)
```

In SQL

```
UPDATE users          <-- table
SET    status = 'A'   <-- update action
WHERE  age > 18       <-- update criteria
```

CRUD: Delete

In MongoDB

```
db.users.remove(
    { status: "D" }
)
```

collection

remove criteria

In SQL

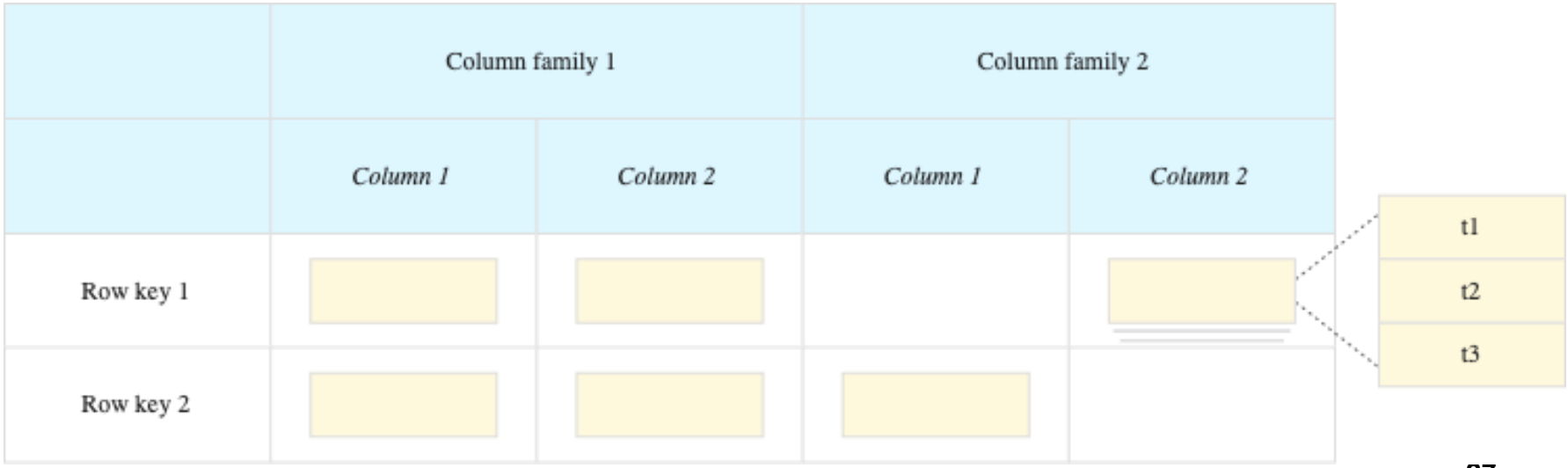
```
DELETE FROM users
WHERE  status = 'D'
```

table

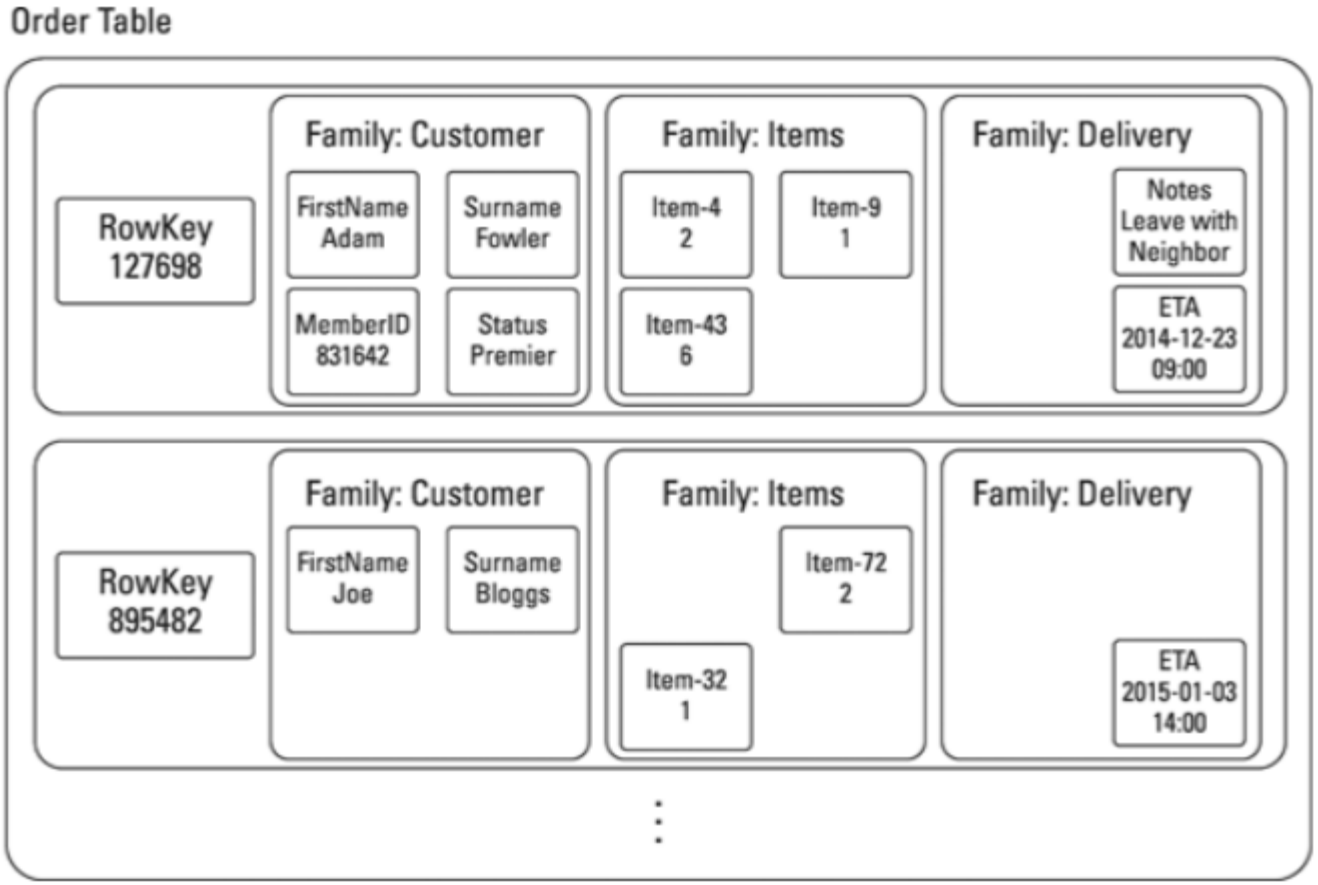
delete criteria

Wide Column Stores

- Rows describe entities
- Related groups of columns are grouped as **column families**
- **Sparsity**: if a column is not used for a row, it doesn't use space
- Examples: BigTable, Cassandra, HBase

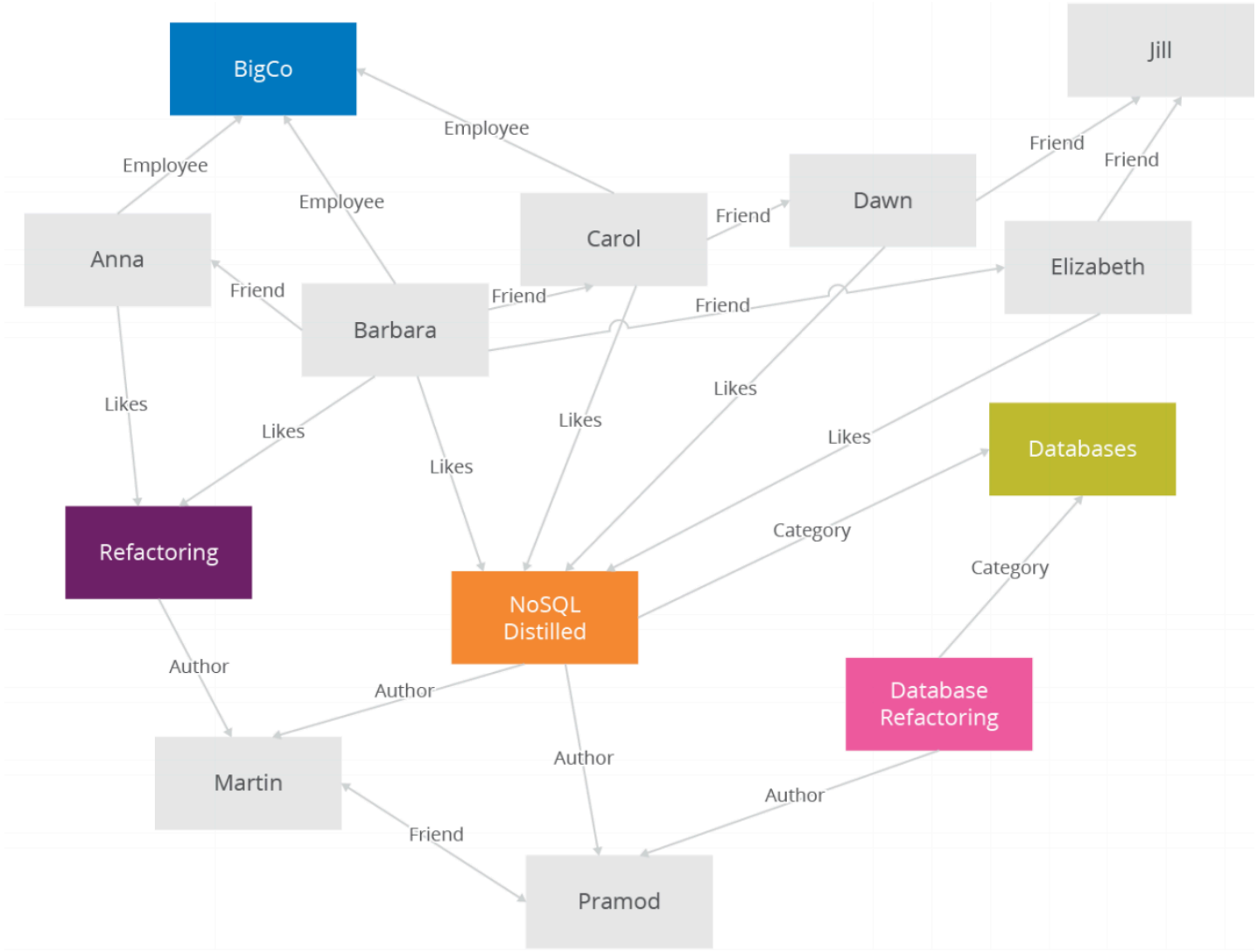


https://cloud.google.com/bigtable/docs/overview



An example of BigTable (an order table)

Graph Databases

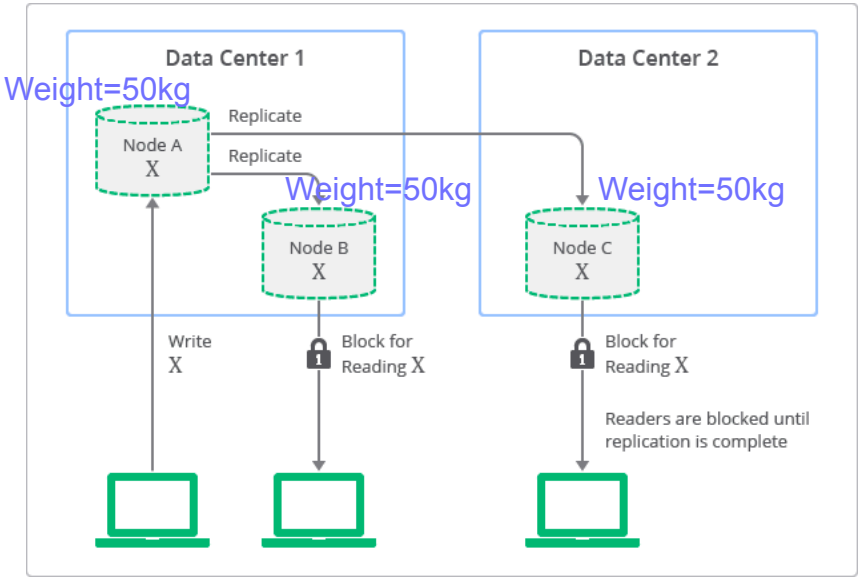


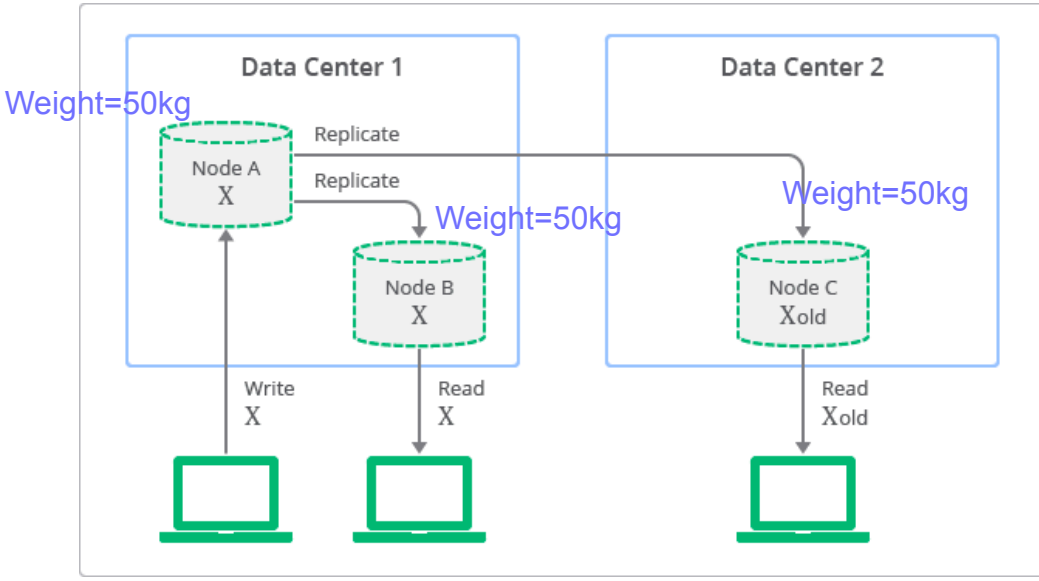
Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. ~~Eventual Consistency~~
4. ~~Duplication~~
5. Data Partitioning
6. Query Processing in NoSQL

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Strong vs Eventual Consistency

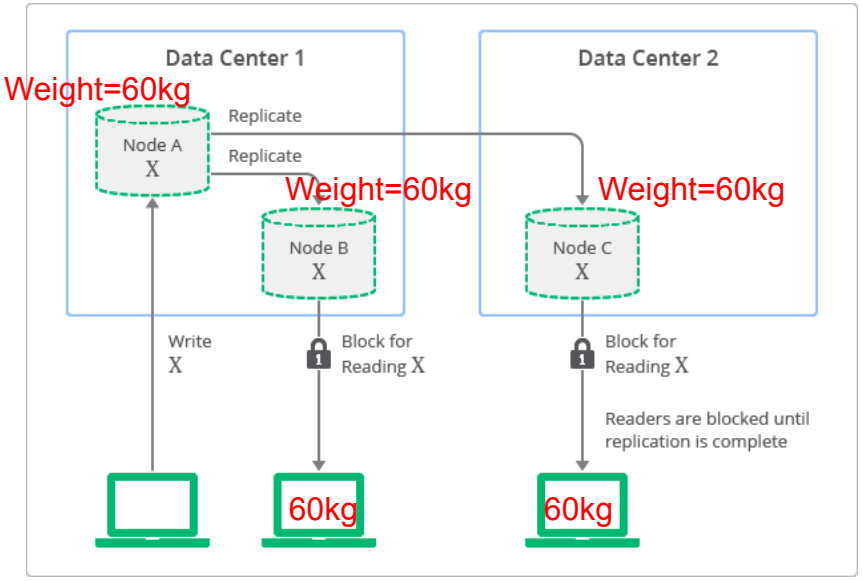


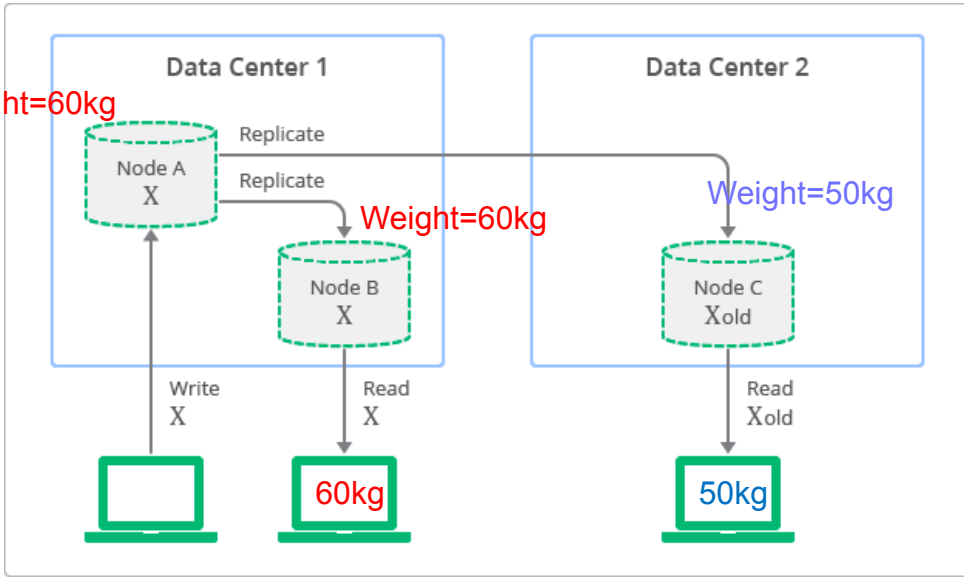


Strong consistency: any reads immediately after an update must give the same result on all observers

Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Strong vs Eventual Consistency

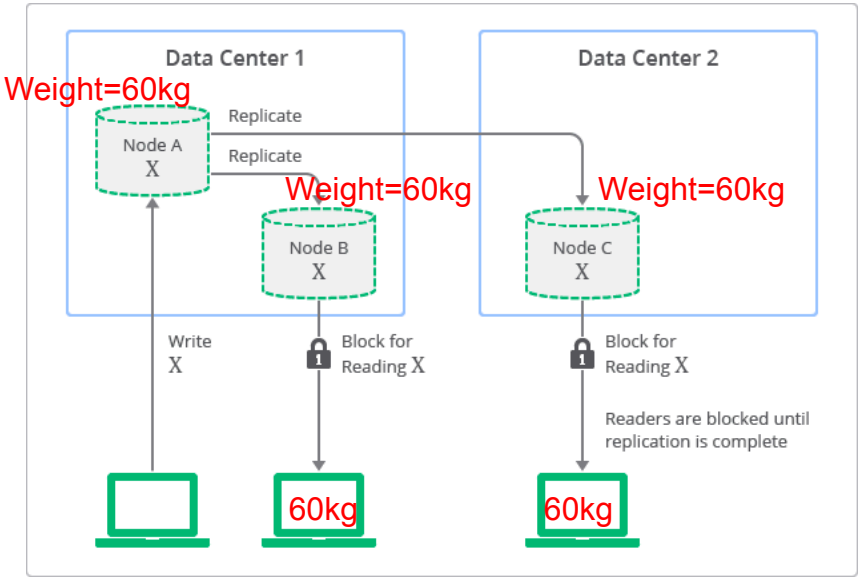


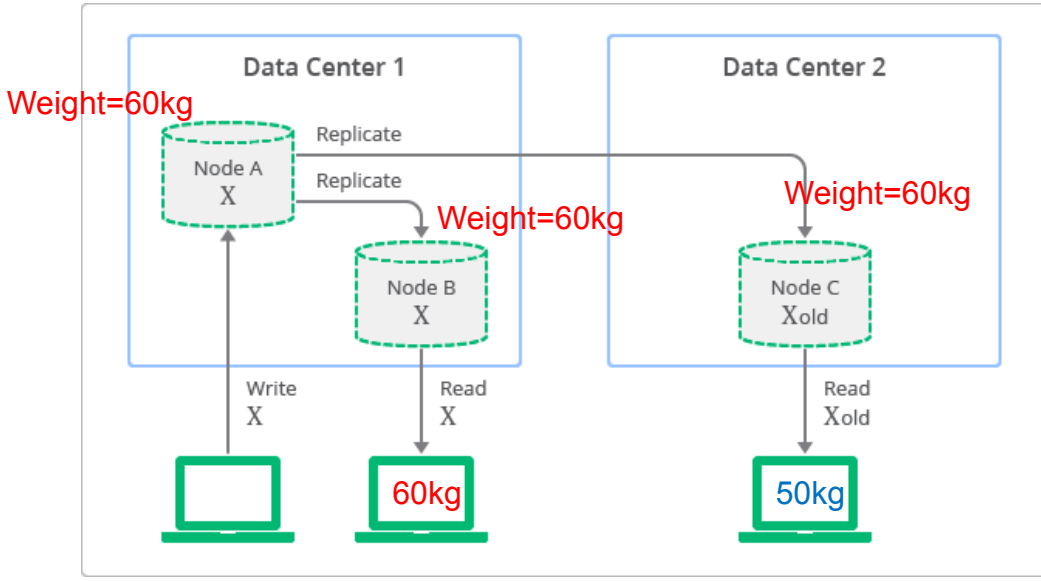


Strong consistency: any reads immediately after an update must give the same result on all observers

Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Strong vs Eventual Consistency





Strong consistency: any reads immediately after an update must give the same result on all observers

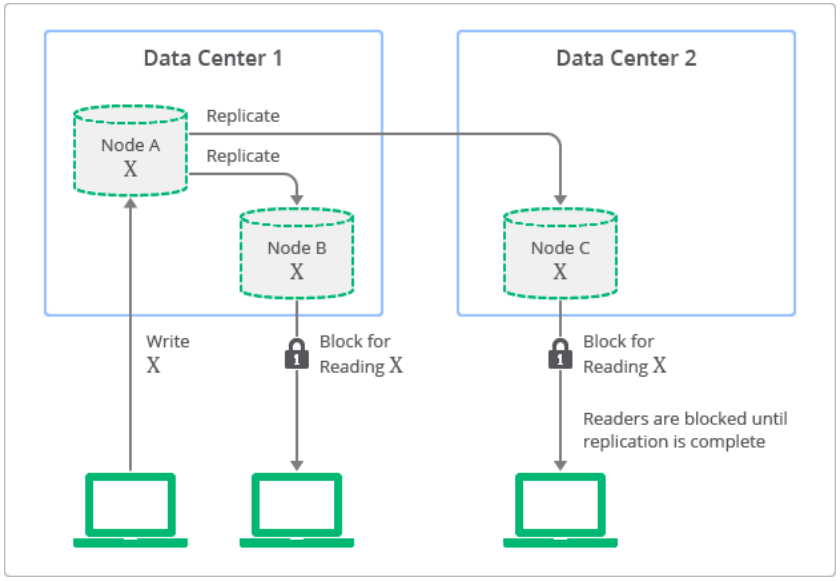
Eventual consistency: if the system is functioning and we wait long enough, eventually all reads will return the last written value

Eventual Consistency

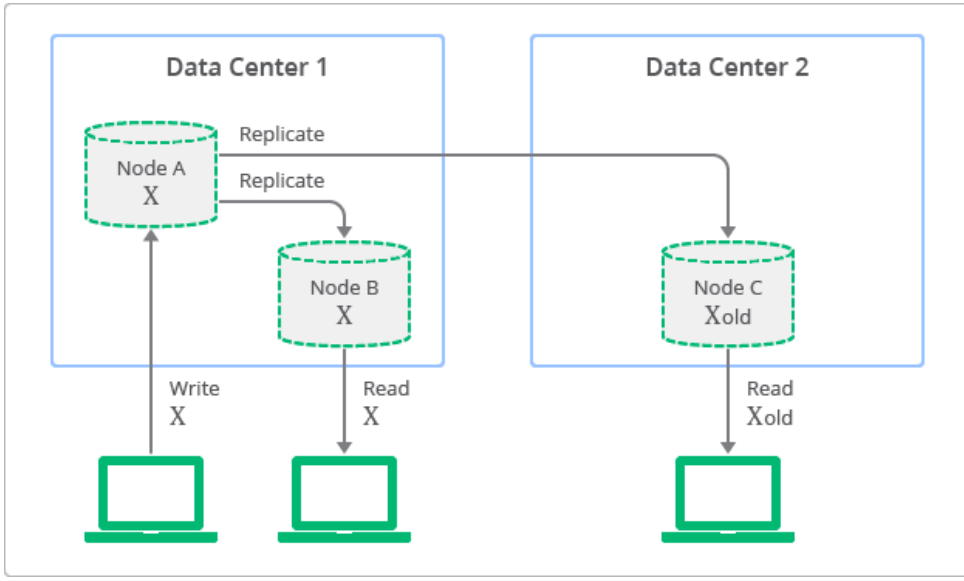
- Relational DBMS provide stronger (**ACID**) guarantees
  - **Atomicity**: all changes in one transaction either succeed or fail as a complete unit.
  - **Consistency**: database transitions from one consistent state at the beginning of the transaction to another consistent state at the end of the transaction.
  - **Isolation**: concurrent transactions are not affecting each other
  - **Durability**: persistence of committed transactions even under system failure
- But many NoSQL system relax this to weaker "**BASE**" approach:
  - **Basically Available**: basic reading and writing operations are available most of the time
  - **Soft State**: the state of the data could change without application interactions due to eventual consistency
  - **Eventually Consistent**: Contrast to **"strong consistency"**

Strong vs Eventual Consistency

Strong consistency



Eventual consistency



- **Implications**: eventual consistency offers better **availability** at the cost of a much weaker consistency guarantee. This may be acceptable for some applications (e.g. statistical queries, tweets, social network feed, …) but not for others (e.g. financial transactions)
  - Note that while NoSQL systems allow for weaker consistency guarantees, many more recent systems / versions are often configurable, i.e. can be configured for multiple different consistency levels (including strong) – *'tunable consistency'*

Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. Eventual Consistency
4. ~~Duplication~~
5. ~~Data Partitioning~~
6. Query Processing in NoSQL

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

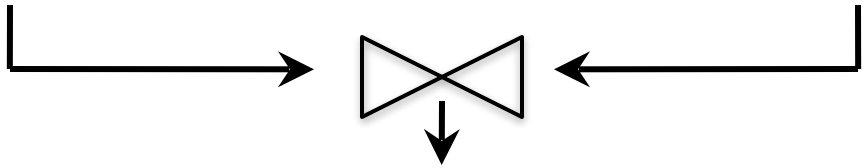
Recall: Joins in RDBMS

Users

user_id	name
100	Alice
101	Bob

Posts

post_id	post	user_id
1	Hello world	100
2	Bye world	101



post_id	post	user_id	name
1	Hello world	100	Alice
2	Bye world	101	Bob

What if we want to display posts with usernames?

Users

```
[
  {
    "user_id": 100,
    "name": "Alice"
  },
  {
    "user_id": 101,
    "name": "Bob"
  },
]
```

Posts

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101
  },
]
```

Posts (+ username)

```
[
  {
    "post_id": "1",
    "post": "Hello world",
    "user_id": 100,
    "name": "Alice",
  },
  {
    "post_id": "2",
    "post": "Bye world",
    "user_id": 101,
    "name: "Bob"
  },
]
```

- Answer 1: some NoSQL databases do support joins (e.g. later versions of MongoDB)
- Answer 2: Duplication (i.e. 'denormalization')
  - 'Storage is cheap: why not just duplicate data to improve efficiency?'
  - Tables are designed around the queries we expect to receive. This is beneficial especially when we mostly need to process a fixed type of queries
  - Leads to a new problem: what if user changes their name? (this needs to be propagated to multiple tables)

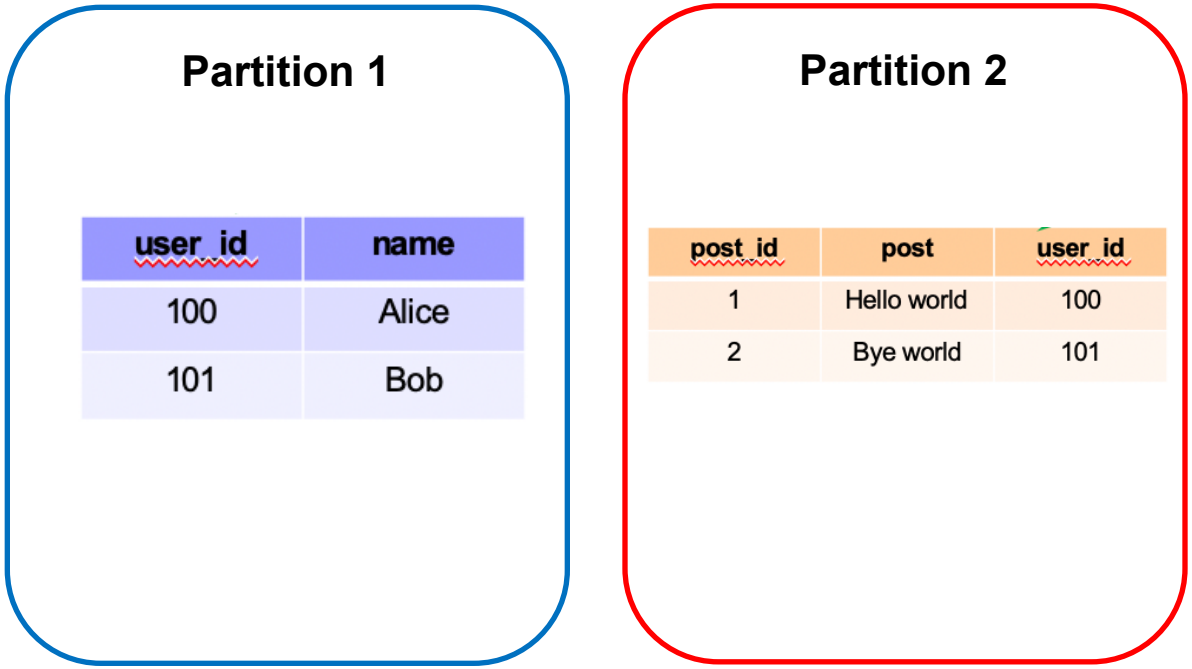
Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. Eventual Consistency
4. Duplication
5. Data Partitioning
6. Query Processing in NoSQL

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Table Partitioning

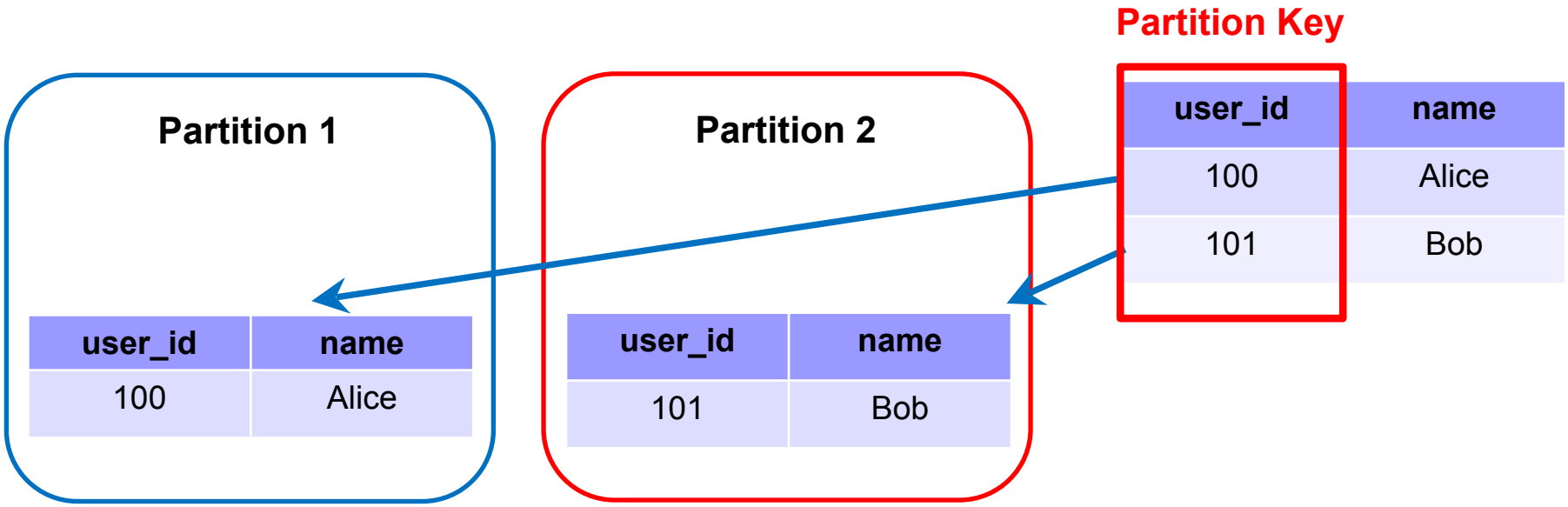
- Put different tables (or collections) on different machines



- **Problem**: scalability – each table cannot be split across multiple machines

Horizontal Partitioning

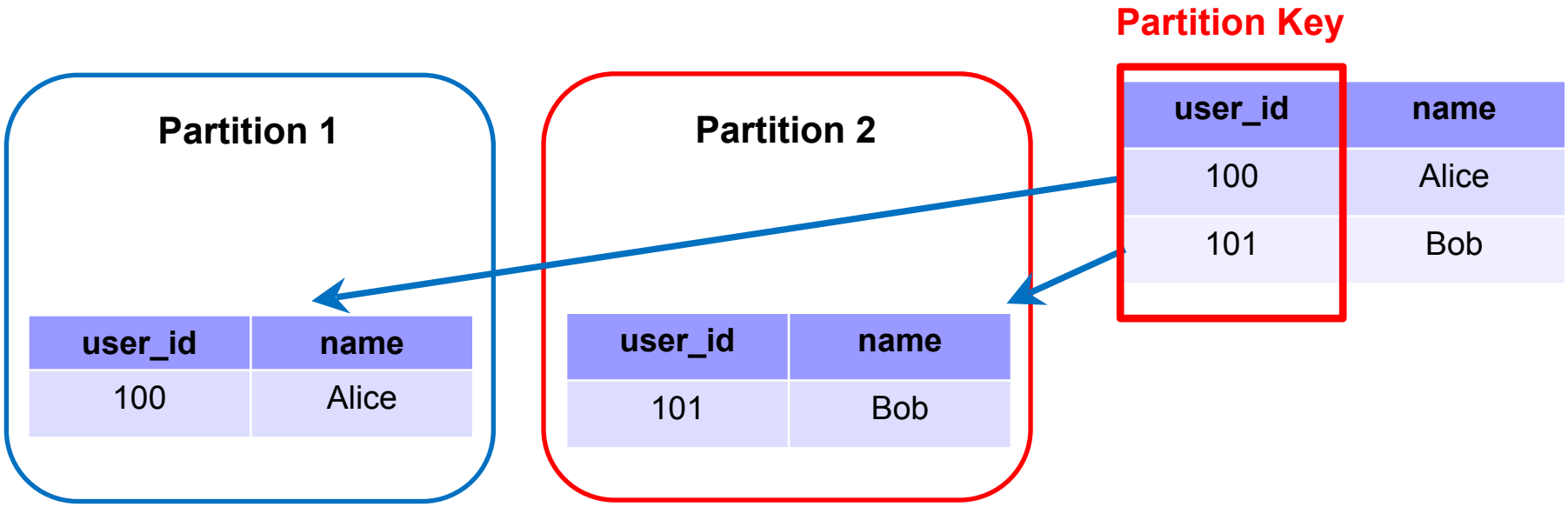
- Different tuples are stored in different nodes



- Also called "sharding"
- **Partition Key** (or "shard key") is the variable used to decide which node each tuple will be stored on: tuples with the same shard key will be on the same node
  - **How to choose partition key?** If we often need to filter tuples based on a column, or "group by" a column, then that column is a suitable partition key
  - Example: if we filter tuples by user_id=100, and user_id is the partition key, then all the user_id=100 data will be on the same partition. Data from other partitions can be ignored (called '**partition pruning**'), which saves time as we don't have to scan these tuples.

Horizontal Partitioning

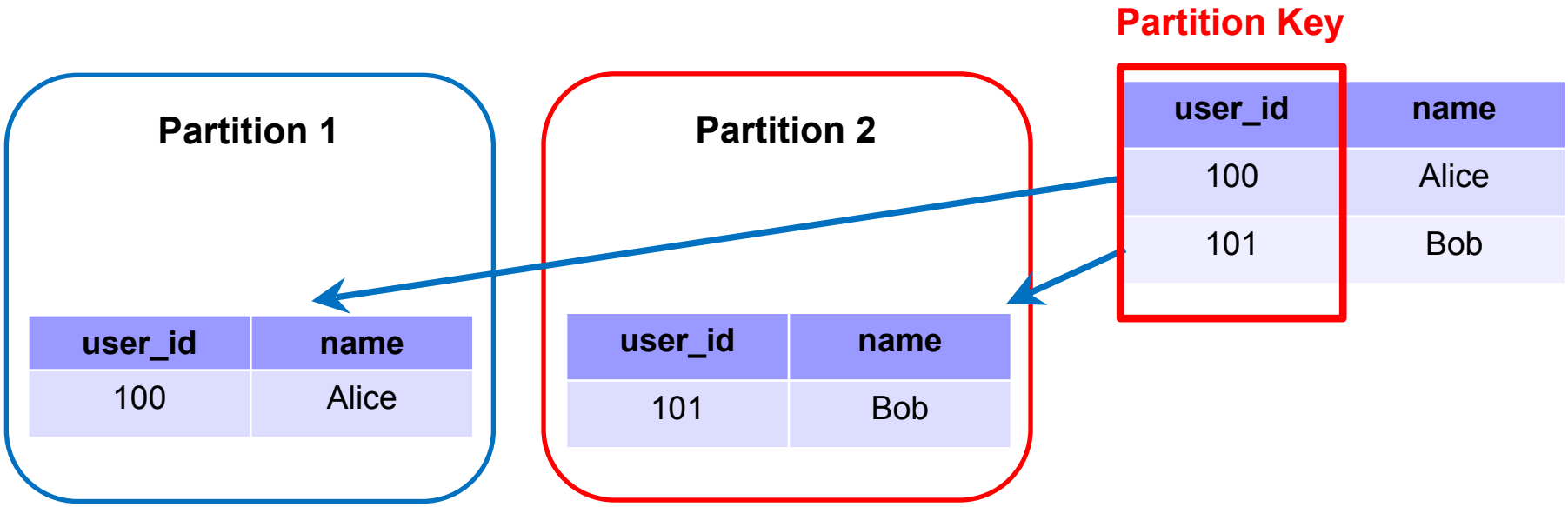
- Different tuples are stored in different nodes



- **Q**: imagine using each user's city, city_id as a partition key; when is this good / bad?

Horizontal Partitioning

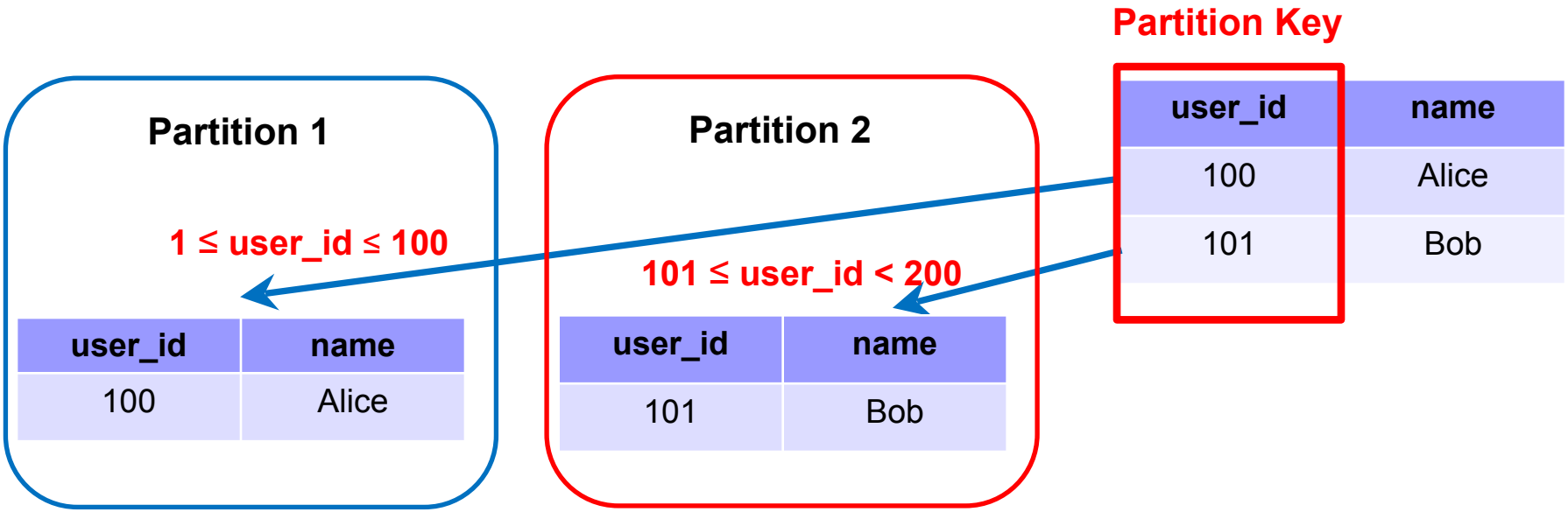
- Different tuples are stored in different nodes



- **Q**: imagine using each user's city, city_id as a partition key; when is this good / bad?
- **A**: good if we mostly aggregate data only within individual cities. Bad if there are too few cities (or cities are very imbalanced) and this causes a lack of scalability.

Horizontal Partitioning – Range Partition

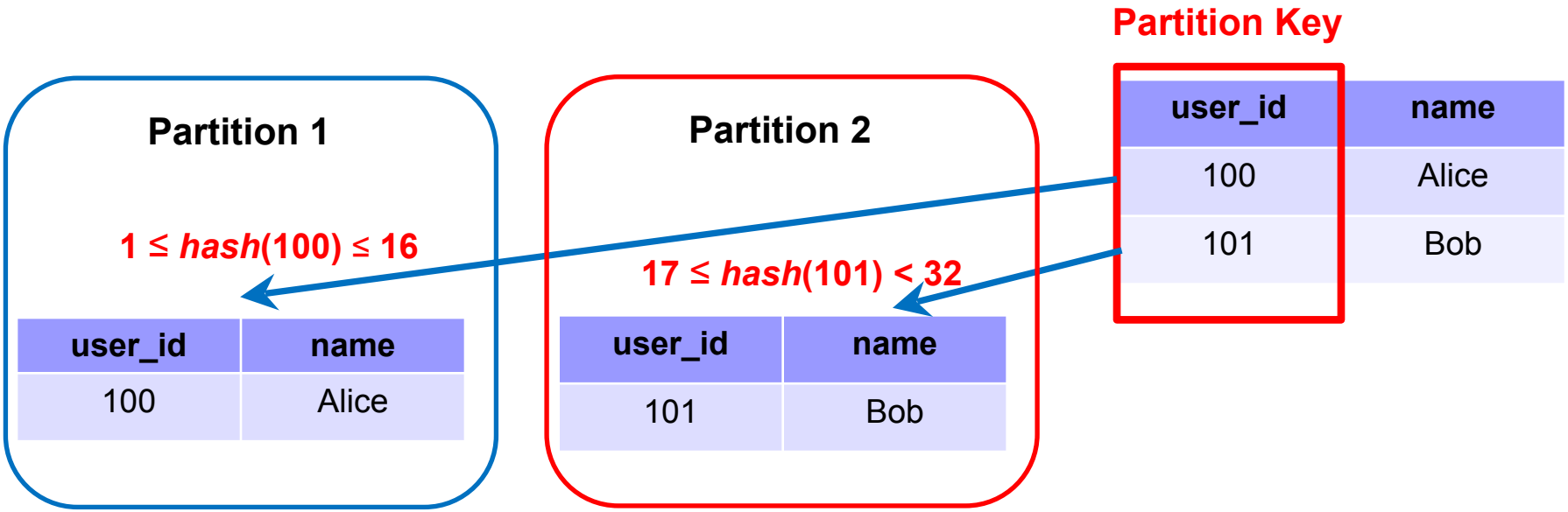
- Different tuples are stored in different nodes



- **Range Partition**: split partition key based on range of values
  - Beneficial if we need range-based queries. In the above example, if the user queries for user_id < 50, all the data in partition 2 can be ignored ('partition pruning'); this saves a lot of work
  - But: range partitioning can lead to imbalanced shards, e.g. if many rows have user_id = 0
  - Splitting the range is automatically handled by a balancer (it tries to keep the shards balanced)

Horizontal Partitioning – Hash Partition

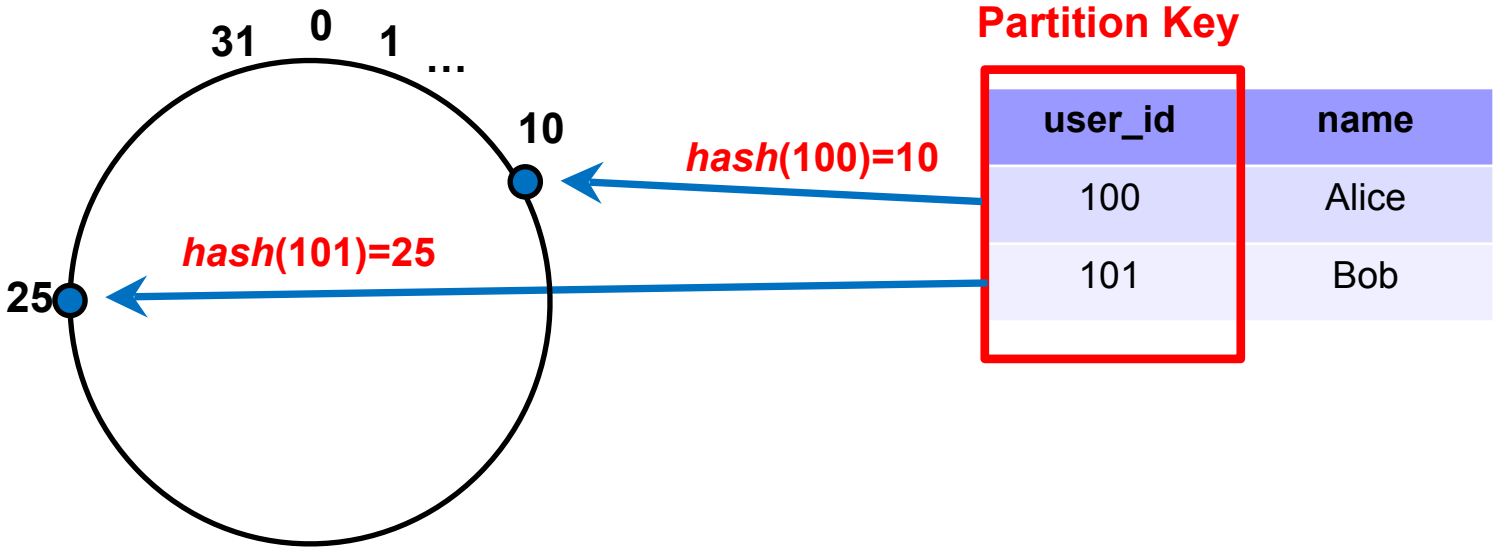
- Different tuples are stored in different nodes



- **Hash Partition**: hash partition key, then divide that into partitions based on ranges
  - Hash function automatically spreads out partition key values roughly evenly
  - **Question:** in previous approaches, how to add / remove a node? If we completely redo the partition, a lot of data may have to be moved around, which is inefficient.
  - **Answer**: consistent hashing

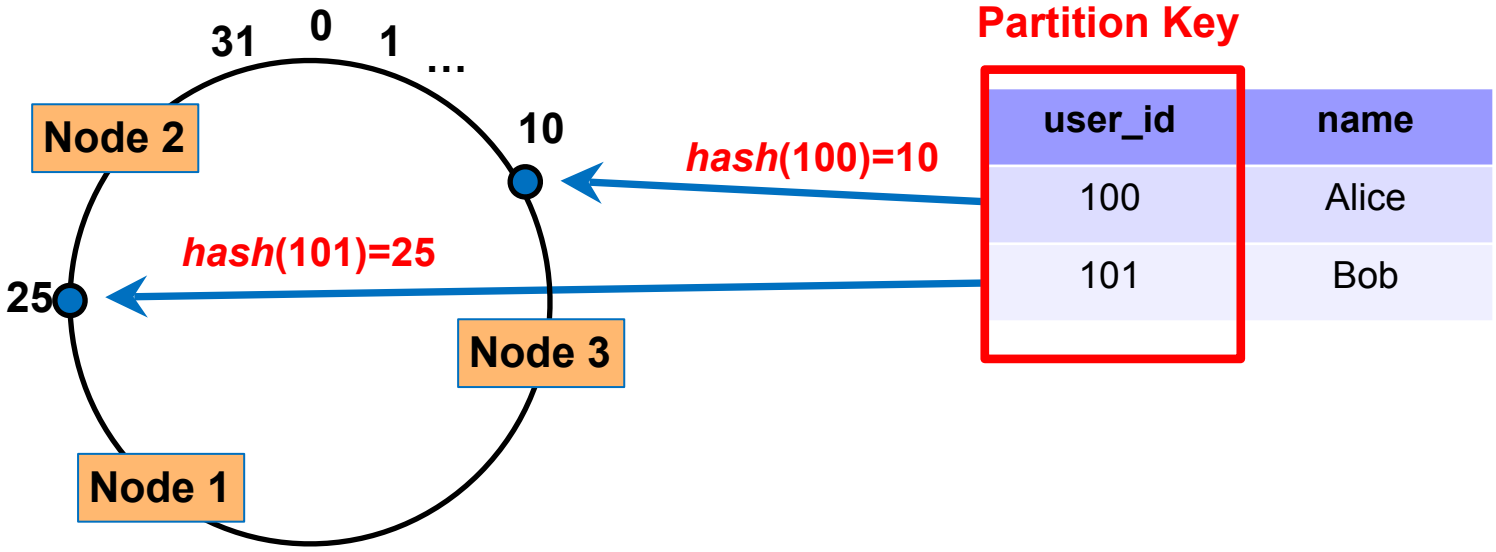
Consistent Hashing

- Think of the output of the hash function as lying on a circle:



Consistent Hashing

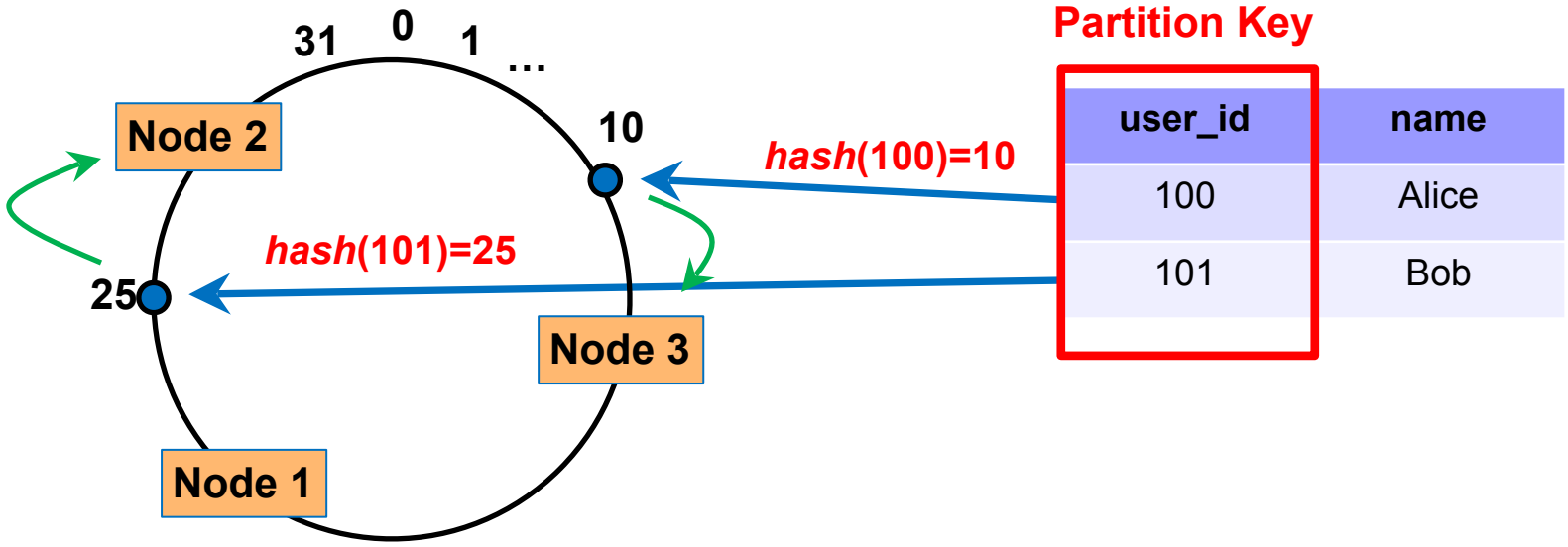
- Think of the output of the hash function as lying on a circle:



- **How to partition**: each node has a 'marker' (rectangles)

Consistent Hashing

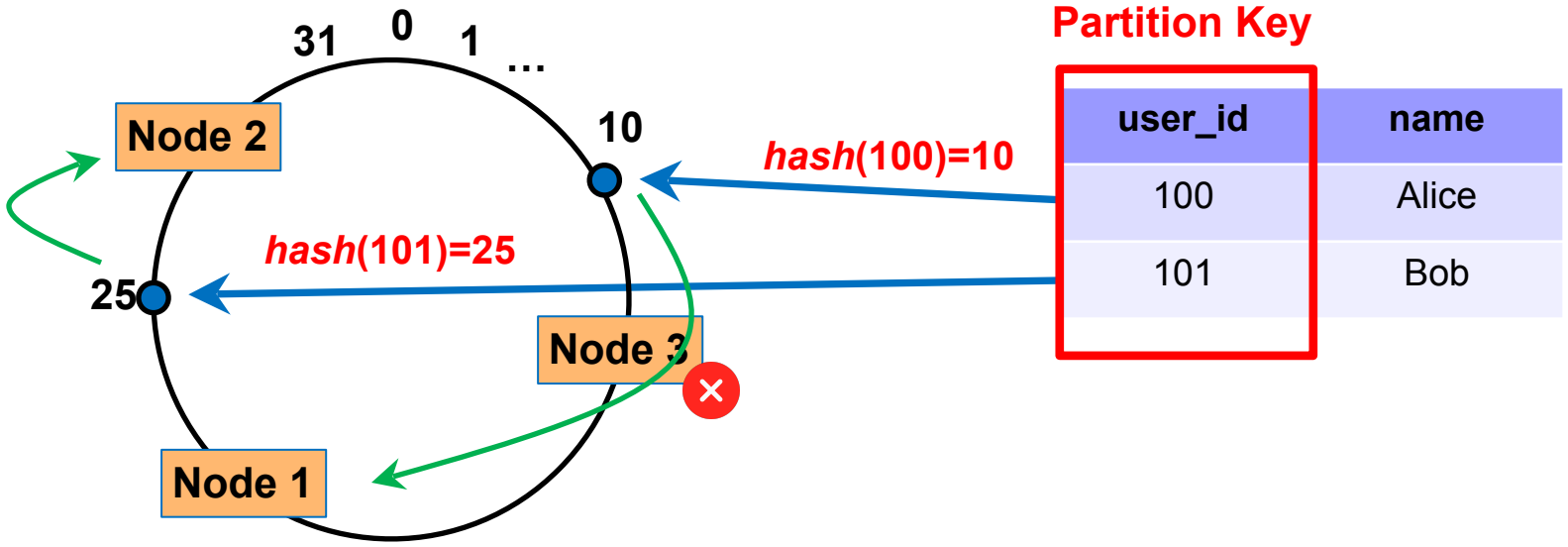
- Think of the output of the hash function as lying on a circle:



- **How to partition**: each node has a 'marker' (rectangles)
- Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it

Consistent Hashing

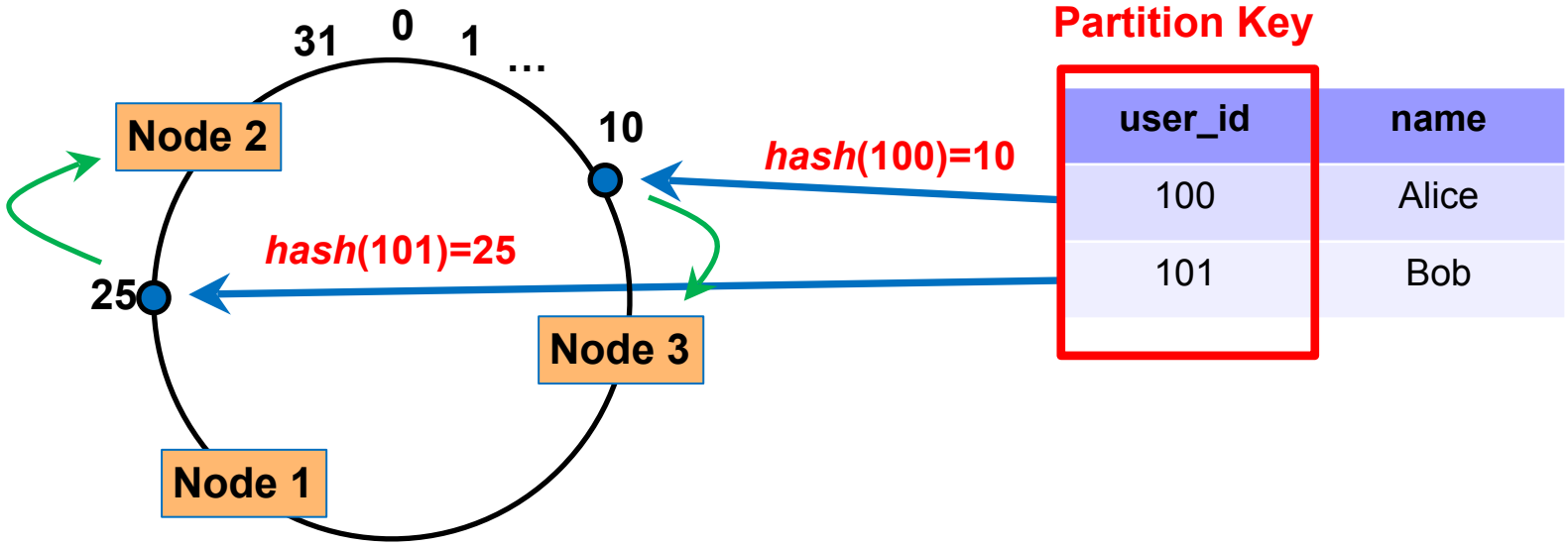
- Think of the output of the hash function as lying on a circle:



- **How to partition**: each node has a 'marker' (rectangles)
- Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it
- To delete a node, we simply re-assign all its tuples to the node clock-wise after this one

Consistent Hashing

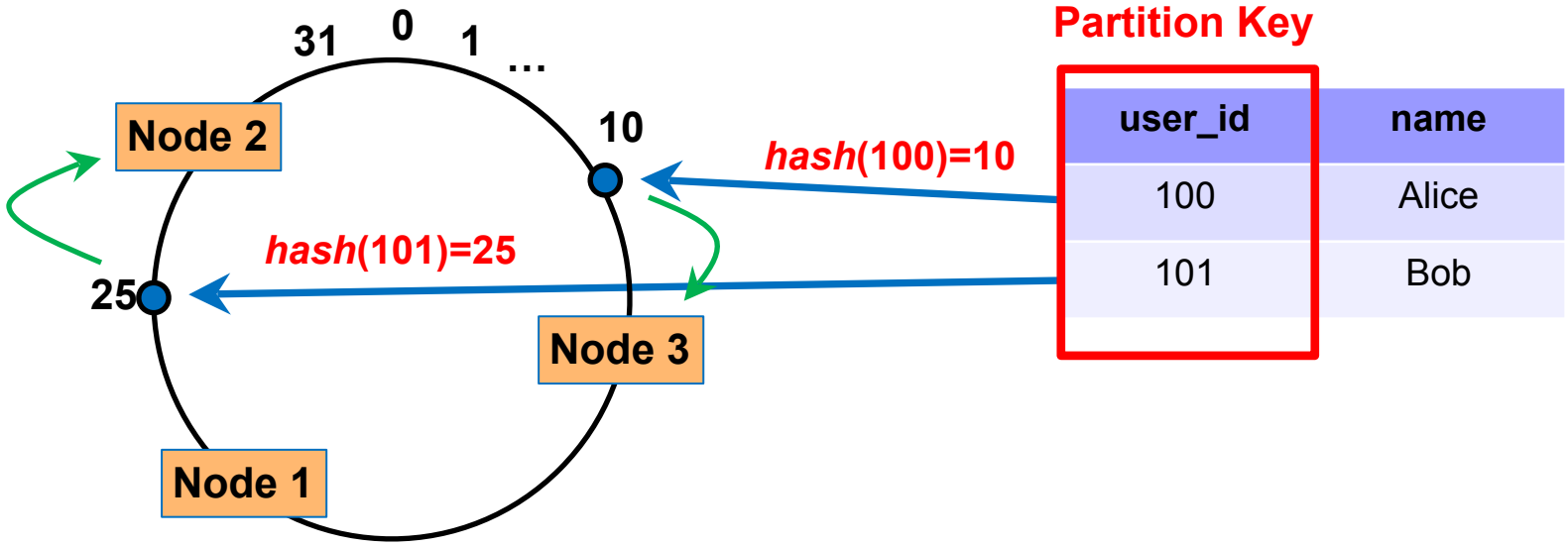
- Think of the output of the hash function as lying on a circle:



- **How to partition**: each node has a 'marker' (rectangles)
- Each tuple is placed on the circle, and assigned to the node that comes clockwise-after it
- To delete a node, we simply re-assign all its tuples to the node clock-wise after this one
- Similarly, nodes can be added by splitting the largest current node into two

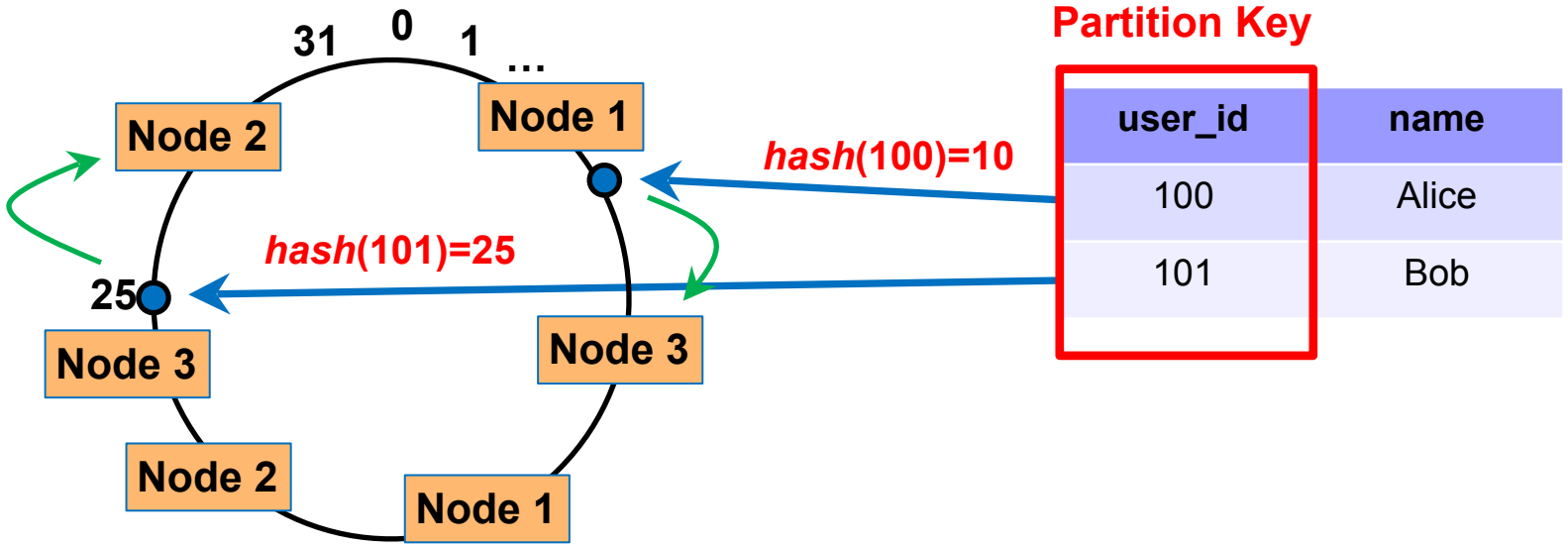
Consistent Hashing

- Think of the output of the hash function as lying on a circle:



- **Simple replication strategy**: replicate a tuple in the next few (e.g. 2) additional nodes clockwise after the primary node used to store it

Consistent Hashing



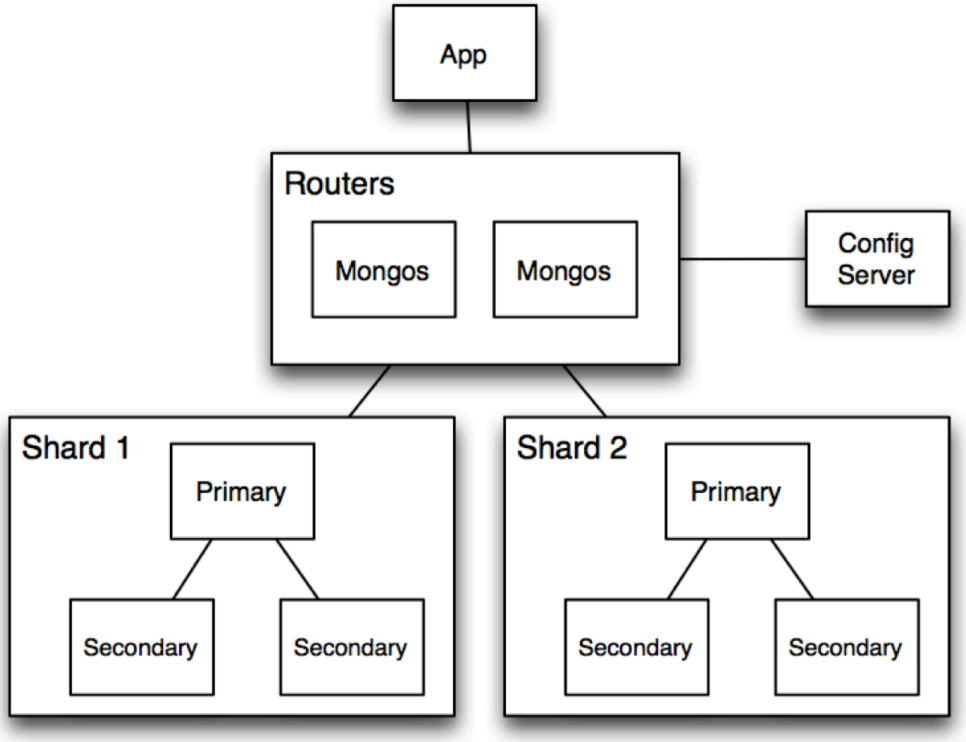
- **Multiple markers**: we can also have multiple markers per node. For each tuple, we still assign it to the marker nearest to it in the clockwise direction.
- Benefit: when we remove a node (e.g. node 1), its tuples will not all be reassigned to the same node. So, this can balance load better.

Today's Plan

1. What is NoSQL?
2. Major types of NoSQL systems
3. Eventual Consistency
4. Duplication
5. Data Partitioning
6. Query Processing in NoSQL

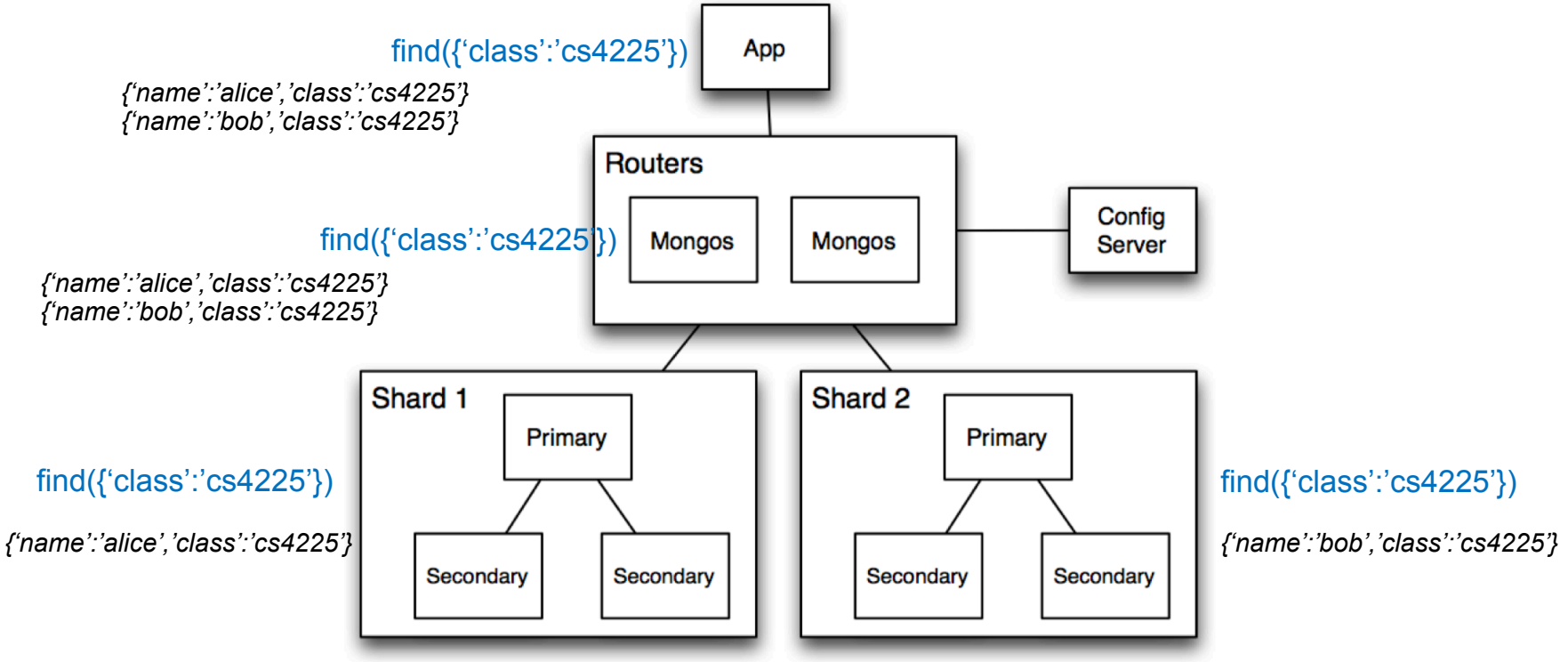
Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Architecture of MongoDB



- **Routers (mongos)**: handle requests from application and route the queries to correct shards
- **Config Server**: stores metadata about which data is on which shard
- **Shards**: store data, and run queries on their data

Example of Read or Write Query



1. Query is issued to a router (mongos) instance
2. With help of config server, mongos determines which shards should be queried
3. Query is sent to relevant shards
4. Shards run query on their data, and send results back to mongos
5. mongos merges the query results and returns the merged results

Conclusion: Key Features of NoSQL Systems

- **Horizontal partitioning**: as we get more and more data, we can simply partition it into more and more shards (even if individual tables become very large)
  - Horizontal partitioning improves speed due to parallelization.
- **Relaxed consistency guarantees**: prioritize availability over consistency – can return slightly stale data
- **Flexible schema:** make it easier to adapt applications to changing requirements

Conclusion: Pros & Cons of NoSQL Systems

Pros

Cons

+ **Flexible / dynamic schema**: suitable for less well-structured data
+ **Horizontal scalability**: improves speed due to parallelization
+ **High performance and availability**: due to their relaxed consistency model and fast reads / writes

- **No declarative query language**: query logic (e.g. joins) may have to be handled on the application side, which can add additional programming
- **Weaker consistency guarantees**: application may receive stale data that may need to be handled on the application side

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Conclusion: no one size fits all

When it comes to select a proper data store, it depends on

- needs of application: whether denormalization is suitable;
- complexity of queries (joins vs simple read/writes);
- importance of consistency (e.g. financial transactions vs tweets);
- data volume / need for availability / data relationships

In addition:

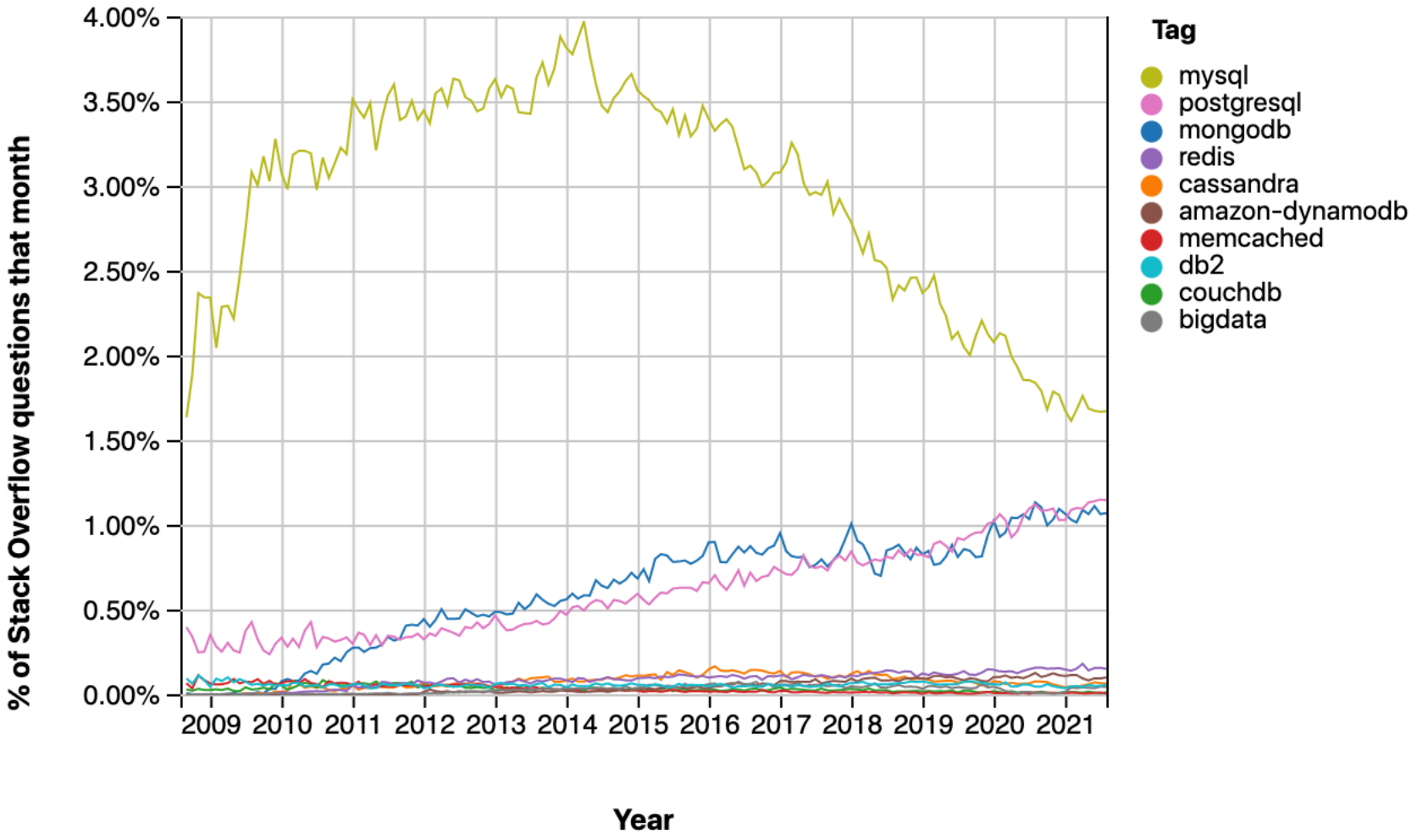
- Binary view is oversimplified: NoSQL databases often still have SQL-like query languages (e.g. Cassandra) and tunable consistency levels.
- Other alternative solutions: "NewSQL", distributed SQL (e.g. Presto), PostgreSQL, …
- Need to understand tradeoffs to make an informed decision!

Source: Cattell (2010). Scalable SQL and NoSQL Data Stores. *SIGMOD Record*.

Popularity of DBMS

Rank			DBMS	Database Model	Score		
Sep 2021	Aug 2021	Sep 2020			Sep 2021	Aug 2021	Sep 2020
1.	1.	1.	Oracle	Relational, Multi-model	1271.55	+2.29	-97.82
2.	2.	2.	MySQL	Relational, Multi-model	1212.52	-25.69	-51.72
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	970.85	-2.50	-91.91
4.	4.	4.	PostgreSQL	Relational, Multi-model	577.50	+0.45	+35.22
5.	5.	5.	MongoDB	Document, Multi-model	496.50	-0.04	+50.02
6.	6.	↑ 7.	Redis	Key-value, Multi-model	171.94	+2.05	+20.08
7.	7.	↓ 6.	IBM Db2	Relational, Multi-model	166.56	+1.09	+5.32
8.	8.	8.	Elasticsearch	Search engine, Multi-model	160.24	+3.16	+9.74
9.	9.	9.	SQLite	Relational	128.65	-1.16	+1.98
10.	↑ 11.	10.	Cassandra	Wide column	118.99	+5.33	-0.18
11.	↓ 10.	11.	Microsoft Access	Relational	116.94	+2.10	-1.51
12.	12.	12.	MariaDB	Relational, Multi-model	100.70	+1.72	+9.09
13.	13.	13.	Splunk	Search engine	91.61	+1.01	+3.71
14.	14.	↑ 15.	Hive	Relational	85.58	+1.64	+14.41
15.	15.	↑ 17.	Microsoft Azure SQL Database	Relational, Multi-model	78.26	+3.11	+17.81
16.	16.	16.	Amazon DynamoDB	Multi-model	76.93	+2.03	+10.75
17.	17.	↓ 14.	Teradata	Relational, Multi-model	69.68	+0.86	-6.72
18.	18.	↑ 21.	Neo4j	Graph	57.63	+0.68	+7.00
19.	19.	19.	SAP HANA	Relational, Multi-model	56.24	+0.66	+3.38
20.	↑ 21.	↑ 23.	FileMaker	Relational	52.32	+2.04	+4.75

Stack Overflow Trends



Source: https://insights.stackoverflow.com/

Acknowledgements

- CS4225 slides by He Bingsheng and Bryan Hooi
- Penn State CMPSC 431W Database Management Systems (Fall 2015) – McGrawHill, Wang-Chien Lee, Yu-San Lin
- mongodb.com, redis.com
- Otter video from https://www.reddit.com/r/singapore/comments/mezgg3/otters_on_the_grass_at_botanic_gardens_today/
- CMU CS15-445 slides by Andy Pavlo
- mongodb.com, https://learnmongodbthehardway.com/
- Martin Kleppmann, "Designing Data-Intensive Applications"