Distributed Databases

An introduction to NoSQL

History

70’s-90’s

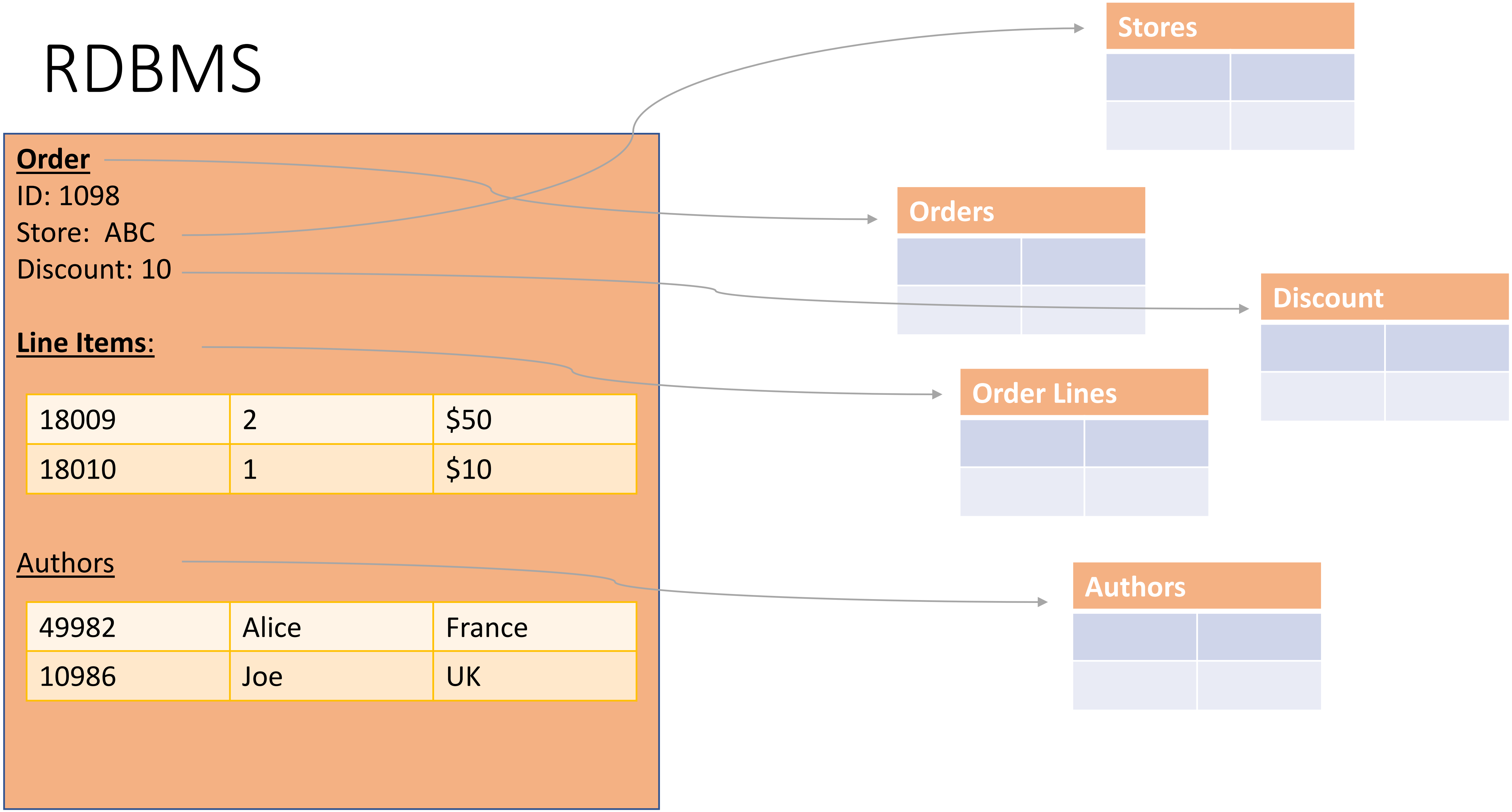
RDBMS

- Structured and organized data
- Structured Query Language SQL
- Data and its relationships are stored in separate tables
- Tight Consistency

2000’s

2005-2010

RDBMS


Order
ID: 1098
Store: ABC
Discount: 10
Line Items:
18009
2
$50
18010
1
$10
Authors
49982
Alice
France
10986
Joe
UK
Stores
Orders
Discount
Order Lines
Authors

NoSQL Evolution

70's-90's

RDBMS

- Structured and organized data
- Structured Query Language SQL
- Data and its relationships are stored in separate tables
- Tight Consistency

2000's

CAP Theorem

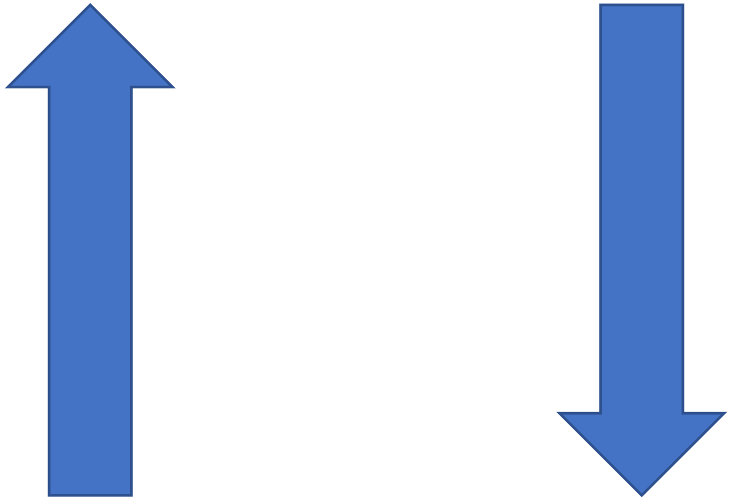
2005 – 2010

NoSQL Evolution

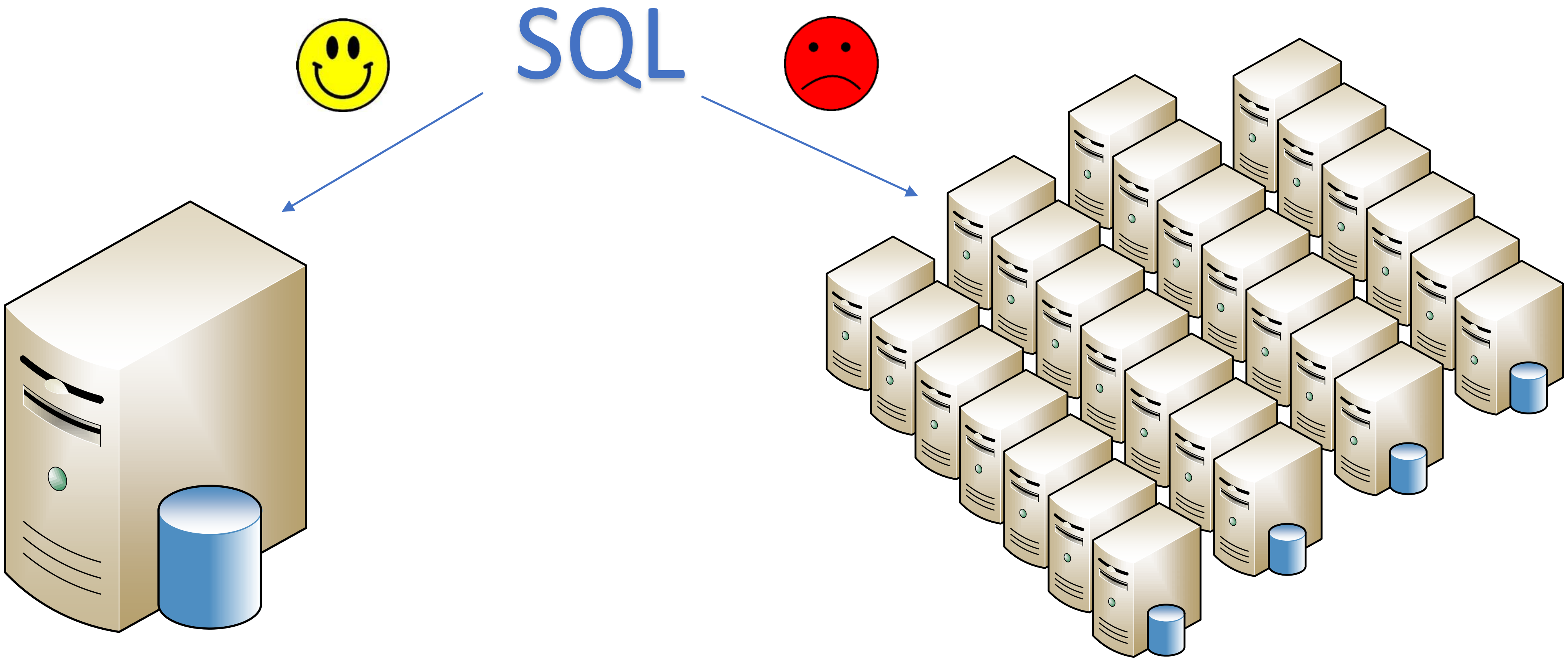
70's-90's	2000's	2005 – 2010
RDBMS	**CAP Theorem**	**NoSQL**
Structured and organized data		Schema-less
Structured Query Language SQL		Unstructured and unpredictable data
Data and its relationships are stored in separate tables		high performance, high availability and scalability
Tight Consistency		

The world has changed

Data / Risk **Time / Cost**



NoSQL Evolution



NoSQL Evolution

2006 - BigTable paper

2007 - Dynamo paper/AWS SimpleDB

2008 - Cassandra

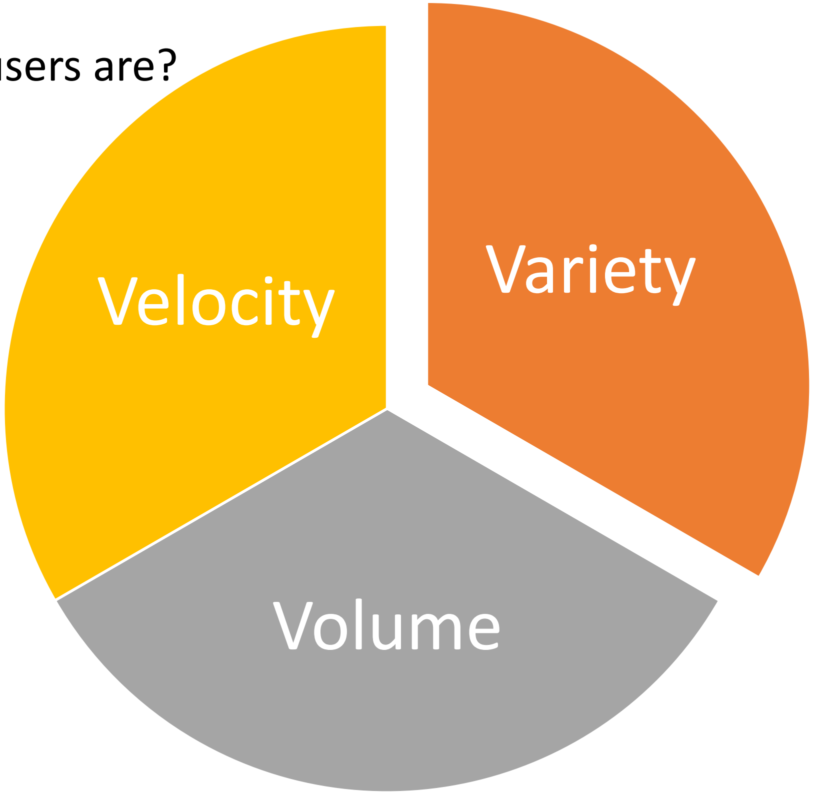
2009 - MongoDB/Riak/Neo4J

2012 - AWS DynamoDB

2014 - Azure DocumentDB

3Vs concept

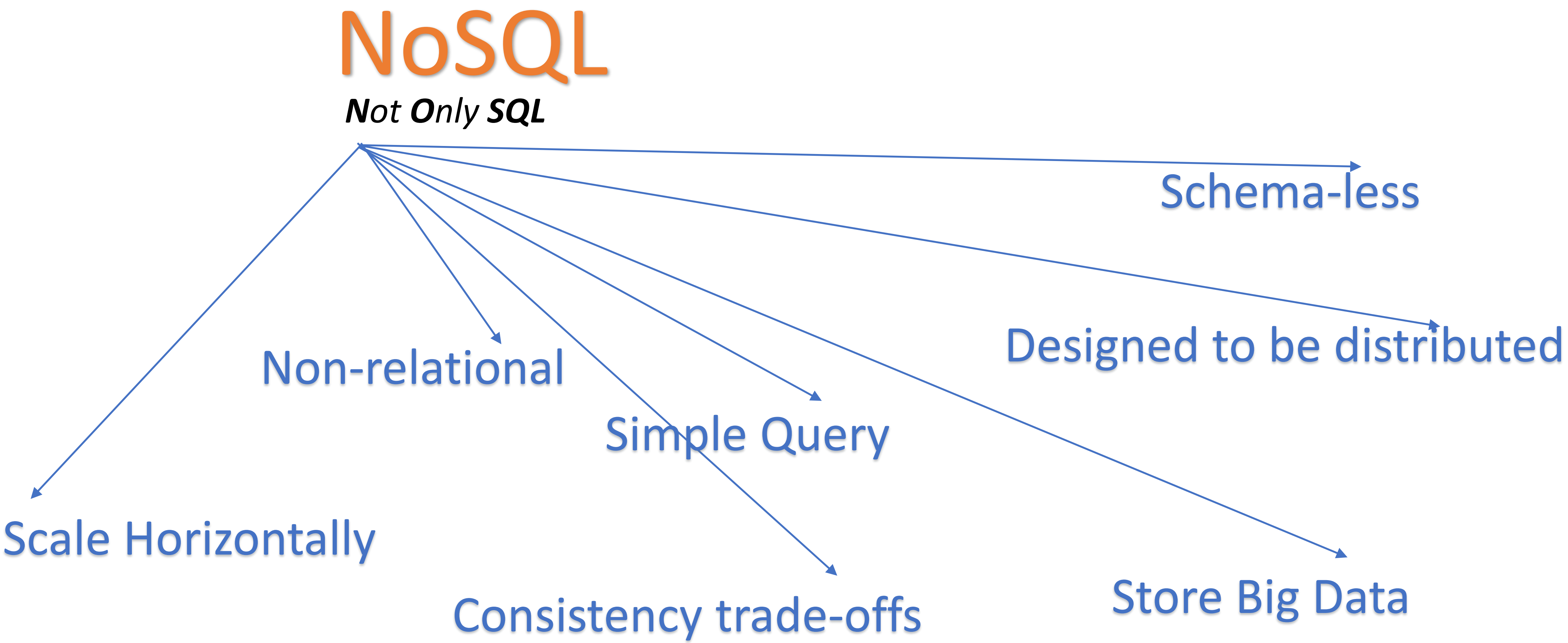
- How to make data available where users are?
- How to write highly available apps?



- How to deal with schema changes?
- How to iterate rapidly?
- What data models work at scale?

- How to deal with massive volumes of data?
- How to elastically scale?

NoSQL Characteristics



CAP Theorem

- Conjectured by Eric Brewer in 2000
- Proved by Seth Gilbert and Nancy Lynch in 2002
- It is impossible for a distributed system to provide all three of the following guarantees:

Consistency: All nodes see the same data at the same time

Availability: The system is always on

Partition Tolerance: The system continues to function even if the communication among the servers is unreliable

CAP Theorem

- Increased globalization has led to the requirement to place data near clients who are spread across the world
- CAP theorem describes the **trade-offs** involved in distributed systems
- The Relational Database is built on the principle of **ACID** (Atomicity, Consistency, Isolation, Durability)
- It implies that a truly distributed relational database should have **availability, consistency and partition tolerance**.
- Which is **impossible** …
- Distributed databases should choose between consistency and availability ➔ You can't trade partition tolerence

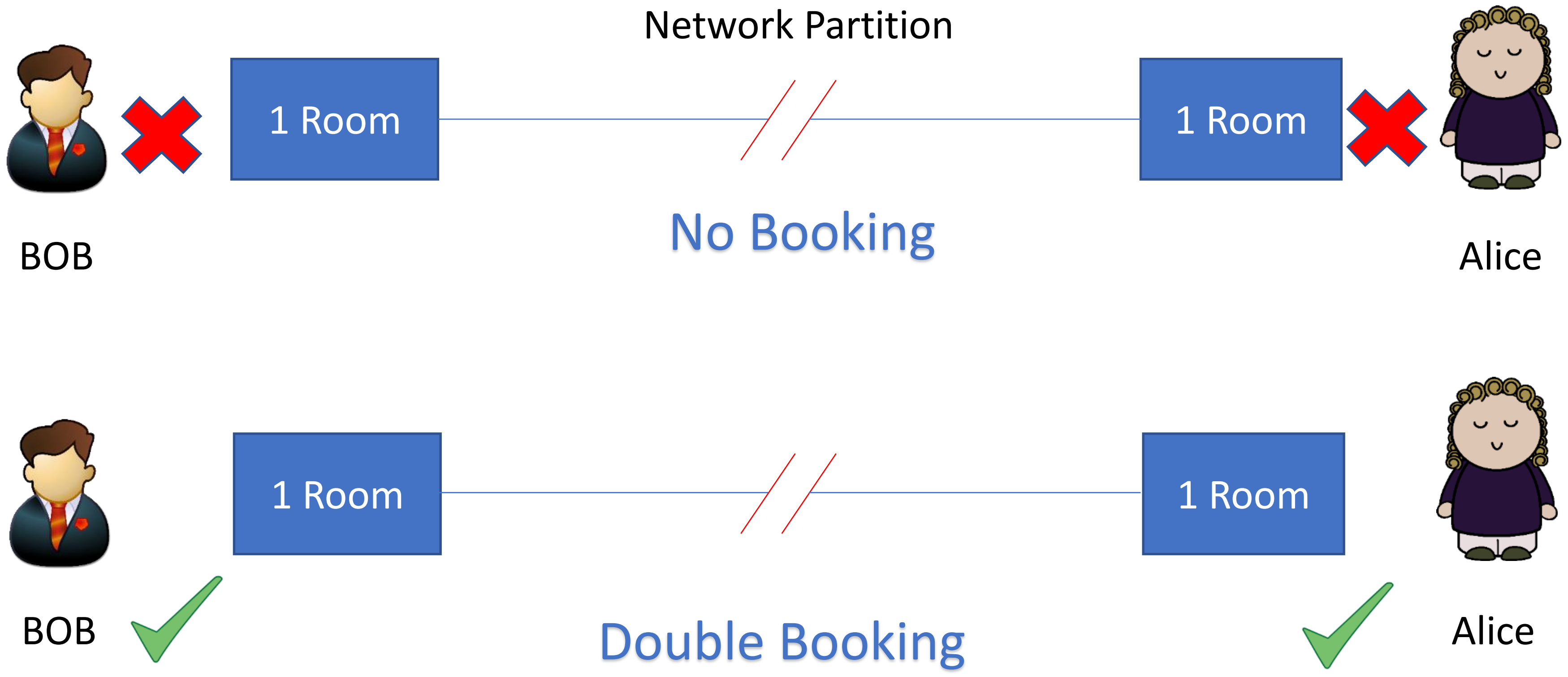
CAP Theorem

Example: Hotel Booking

Consistency

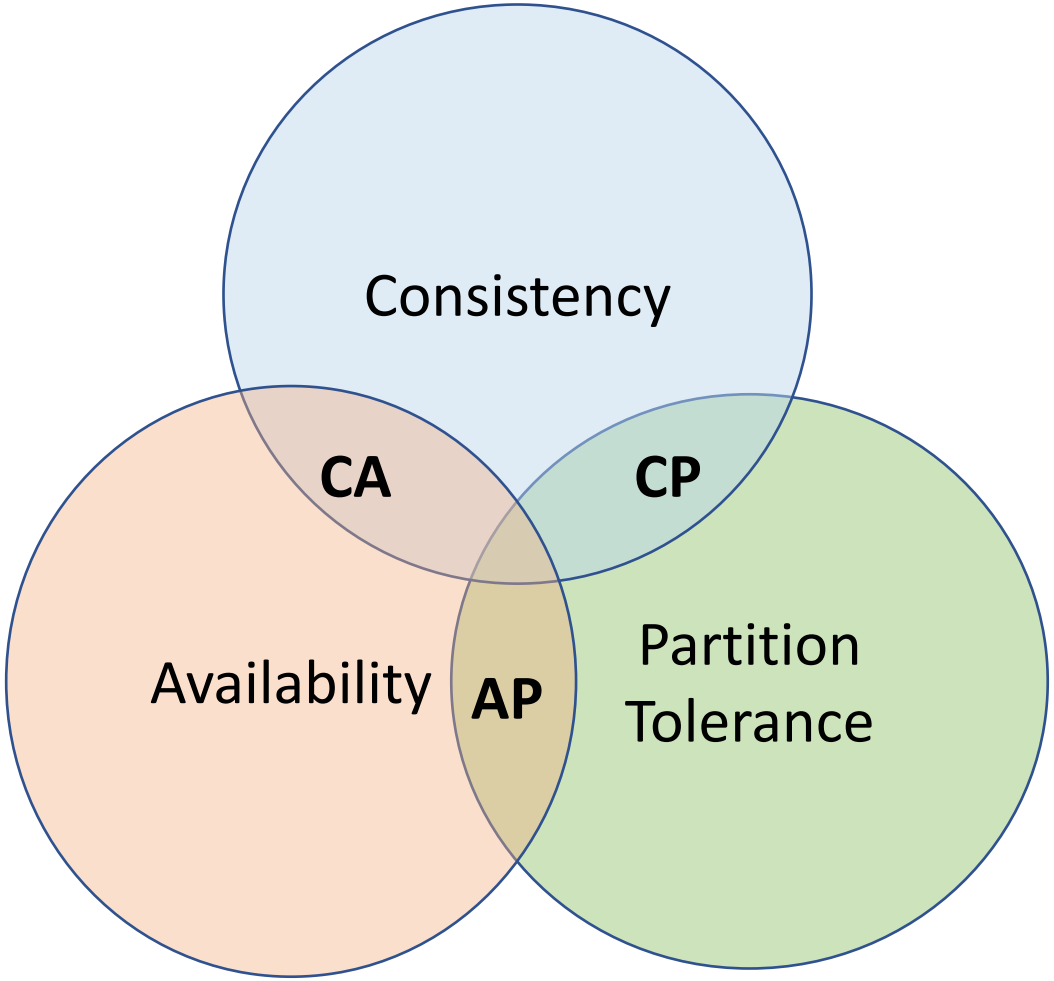
VS

Availability



CAP Theorem

- **CA**: single node system (*Not a distributed system*)
- **CP**: some data may not be accessible, but the rest is consistent
- **AP**: System is still available under partitioning, but some of the data returned may be inaccurate



CAP Theorem revisited

- Prof. Eric Brewer: father of CAP theorem (2012)
  - “The “2 of 3” formulation was always **misleading** because it tended to oversimplify the tensions among properties. ...
  - **CAP prohibits only a tiny part of the design space**: *perfect availability and consistency in the presence of partitions*, which are rare.”

  http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Consistency or Availability

- Consistency and Availability is not “binary” decision
- AP systems sacrifice consistency for availability, but are not inconsistent
- CP systems sacrifice availability for consistency, but are not unavailable
- AP and CP systems can offer a degree of consistency, and availability, in addition to a partition tolerance

Types of consistency

- Strong Consistency
  - After the update completes, **any subsequent access** will return the **same** updated value.
- Weak Consistency
  - It is **not guaranteed** that subsequent accesses will return the updated value.
- Eventual Consistency
  - If no new updates are made to a given data item, eventually all accesses to that item will return the last updated value

Eventual Consistency

Eventually consistent services provide weak consistency using **BASE**
Basically Available, Soft state, Eventual consistency

- Basically available: indicates that the system guarantees availability (CAP theorem)
- Soft state: indicates that the state of the system may change over time, even without input
- Eventual consistency: indicates that the system will become consistent over time

CAP / PACELC

- Many designers incorrectly conclude that the CAP theorem imposes certain restrictions on a DDBS during normal system operation.
- CAP only posits limitations in the face of certain types of failures, and does not constrain any system capabilities during normal operation.
- CAP allows the system to make the complete set of ACID (atomicity, consistency, isolation, and durability) guarantees alongside high availability when there are no partitions

PACELC

- Proposed by Daniel Abadi, Yale University, 2010

- Introduces another tradeoff to the CAP theorem between **consistency** and **latency**
- *if there is a **partition** (P), how does the system trade off **availability** and **consistency** (A and C);*
- *else (E), when the system is running normally in the absence of partitions, how does the system trade off **latency** (L) and **consistency** (C)?*

PACELC - Data Replication

- High Availability requires some degree of data replication during normal system operation.
- As soon as a DDBS replicates data, a tradeoff between consistency and latency arises.

<u>Data Replication Options:</u>

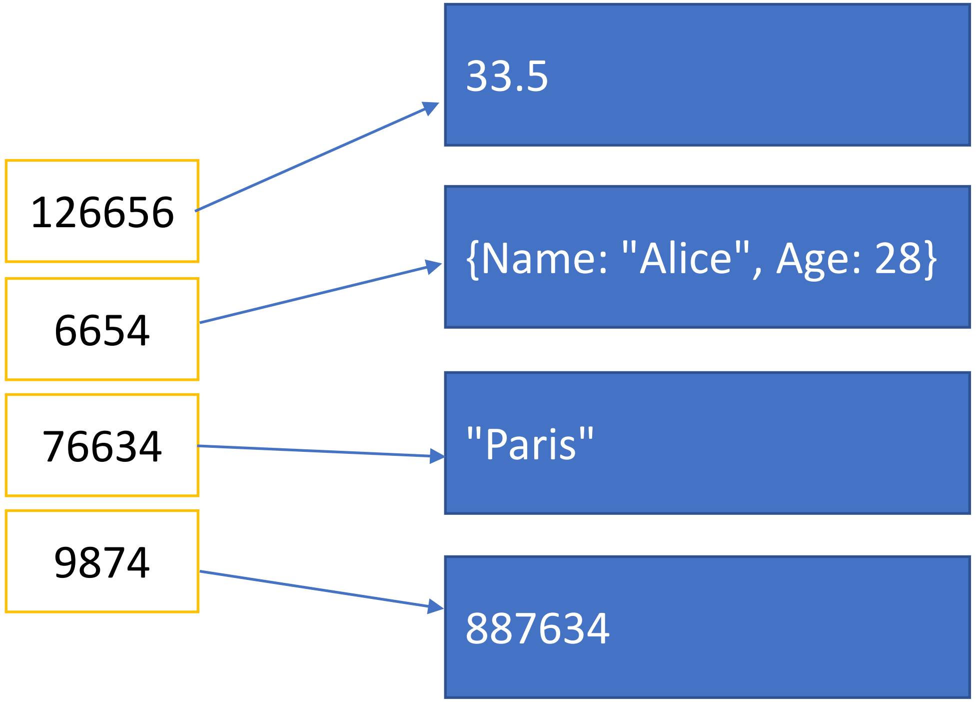
- How data updates are **sent to replicas**
  - Synchronous → Consistency++, latency++
  - Asynchronous → Latency--, Consistency--
  - Combination → Configurable

Classes of NoSQL Databases

NoSQL Data Models

Key-Value Store

- Big Hash table where key is unique and value can be string, JSON, BLOB...
- Single key lookup
- Very fast single key lookup
- Not so fast for reverse lookups
- Typical use cases: session data, user profiles, shopping card contents
- Examples: Redis, Dynamo, Riak



NoSQL Data Models

Document Store

- Extends Key-Value NoSQL store functionality
- Operations and queries can be performed based on keys and values
- Internal structure matters
- Encoded using XML, JSON, BSON
- Typical use cases: blogging platforms, e-commerce applications (flexible schema for product and order data)
- Examples: MongoDB, OrientDB, Cloudant

Implicit Schema – No Explicit Schema

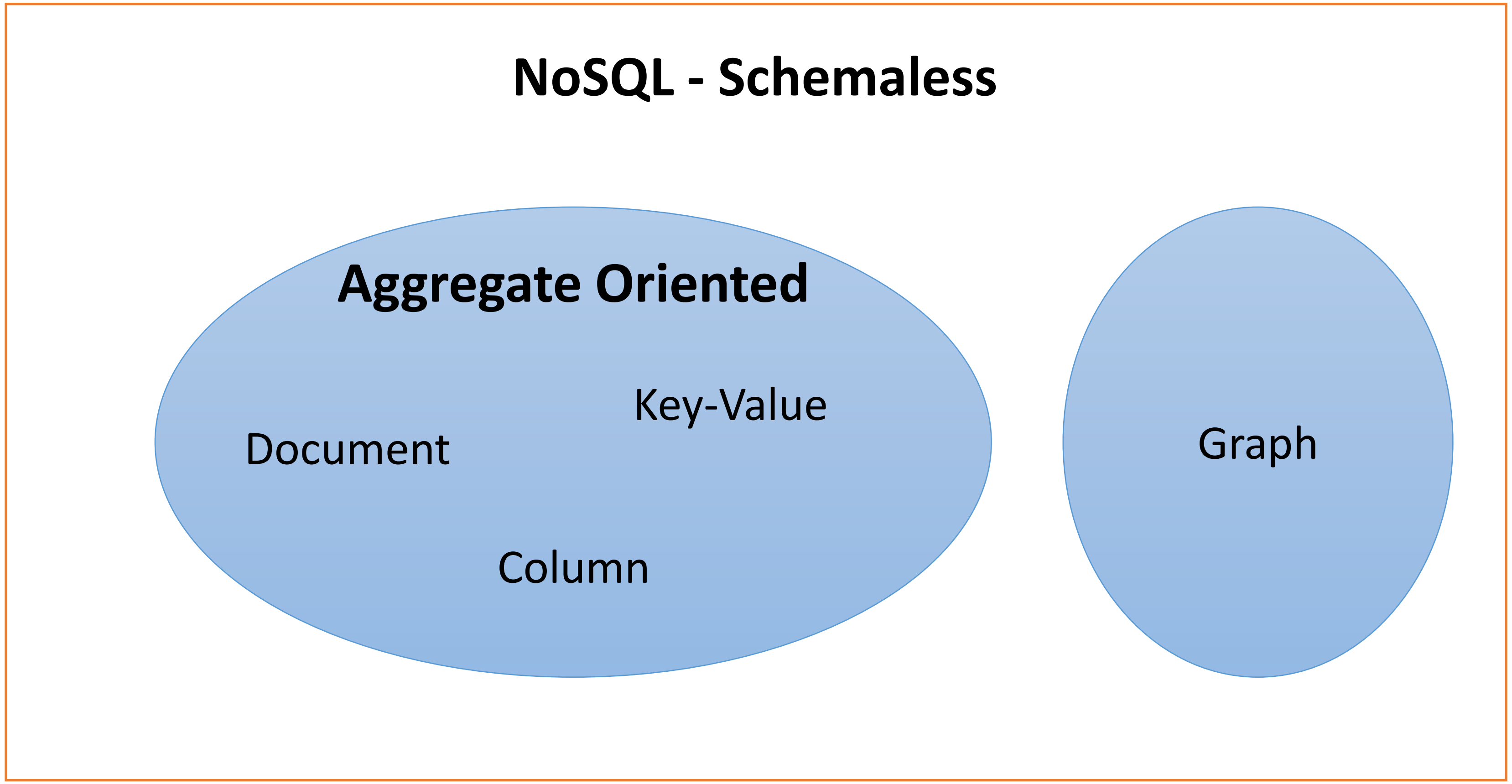
```
{
  "Id": 5000,
  "CustomerId": 4453,
  "StoreId": 321234,
  "LineItems": [
    {"BookId": 1001, "Price": 50},
    {BookId": 1010, "Price": 120}
  ]
},
{
  "id": 5001,
  "CusotmerId": 4500,
  "StoreId": 321250,
  "LineItems": [
    {"BookId": 1001, "Price": 50},
    {BookId": 1010, "Price": 120}
  ],
  "Discount": "20“
}
```

NoSQL Data Models

Column-oriented stores

- Each storage block contains data from one column
- Read and write is done using columns
- Allows to get data from a single column
- High performance on aggregation query (e.g. sum, count, avg, min, max)
- Disadvantage when it's required to change the aggregation structure: example per store/author instead of order
- Examples: BigTable, Cassandra, SimpleDB, HBase, Hypertable

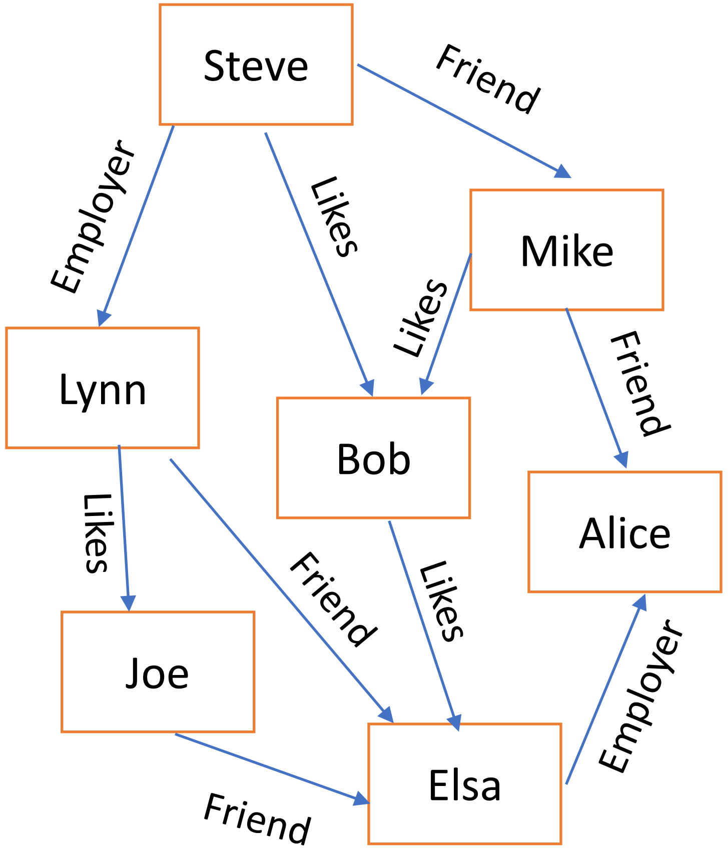
NoSQL Data Models



NoSQL Data Models

Graph

- Network database using graphs with node and edges for storage
- Nodes represent entities, edges represent their relationships
- Each node knows its adjacent nodes
- As the number of nodes increases, the cost of a local step (or hop) remains the same
- Typical use cases: social networks, location-based services, recommendation systems: frequently bought together, often-visited attractions…
- Examples: OrientDB, Neo4J, Titan



DDBS in PACELC

DDBS	A	C	L	C
Dynamo	x		x	
Cassandra	x		x	
Riak	x		x	
BigTable (ACID)		x		x
HBase (ACID)				
MongoDB	x			x
PNUTS		x	x	

A Gentle Introduction to MongoDB

What is MongoDB

- NoSQL Database
  - No constraints on data schema
  - Open-source, cross-platform database
  - Written in C++ and created by 10Gen

- Document Oriented
  - Database is organized into collections and documents
  - Stores documents in a JSON-style syntax called BSON (binary JSON)

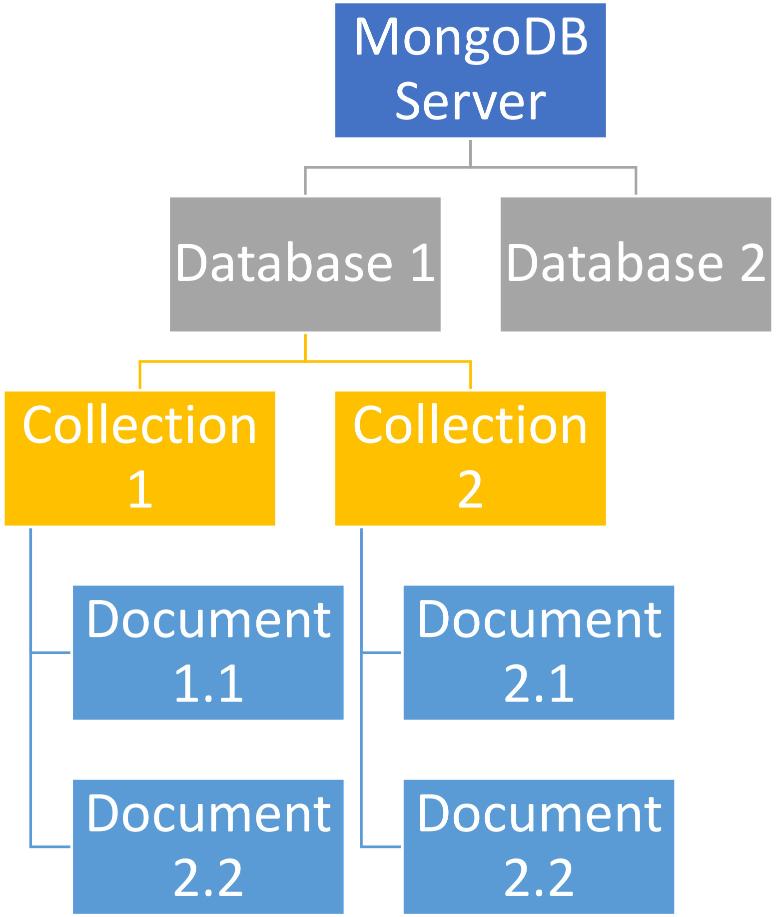
- Designed with Scalability in Mind
  - Auto-Sharding in order to scale horizontally

MongoDB Overview

- **Database**: Physical container for collections

- **Collection**: group of documents with similar or related purpose

- **Document**: set of key-value pairs with dynamic schema



MongoDB vs RDBMS

RDBMS	MongoDB
• Database	• Database
• Table	• Collection
• Row	• JSON Document
• Index	• Index
• Join	• Lookup
• Foreign Key	• Reference
• Multi-table transaction	• Single document transaction*

* multi-document transaction coming soon in Mongo v4

MongoDB

Why use MongoDB?

- Index on any attribute
- Replication and high availability
- Auto-sharding
- Rich queries
- Fast in-place updates

Where to use MongoDB?

- Big Data
- Content Management and Delivery
- Mobile and Social Infrastructure
- User Data Management
- Data Hub

Tabular vs Document Oriented


```
{
  _id: ObjectId(8eb39ab1028d)
  title: 'Introduction to NoSQL',
  tags: ['schemaless', 'database', 'NoSQL', 'CAP'],
  likes: 505,
  comments: [
    {
      user:'Alice',
      message: 'Nice tutorial',
      dateCreated: new Date(2018,4,12,1,10),
      likes: 3
    },
    {
      user:'Bob',
      message: 'Good job',
      dateCreated: new Date(2018,4,10,3,15),
      likes: 10
    }  ]
}
```

MongoDB Documents are Typed

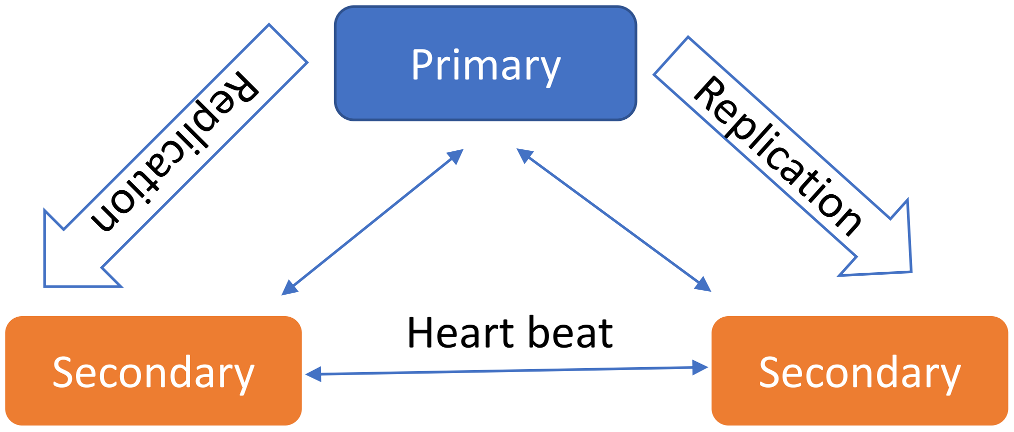
```
{
  title: 'Introduction to NoSQL',
  tags: ['schemaless', 'database', 'NoSQL', 'CAP'],
  likes: 505,
  comments:
          [
              { user:'Alice',
                message: 'Nice tutorial',
                dateCreated: new Date(2018,4,12,1,10),
                likes: 3   },
              { user:'Bob',
                message: 'Good job',
                dateCreated: new Date(2018,4,10,3,15),
                likes: 10    }  ]
}
```

String
Array
Integer

Array of Nested Documents

Replication

- Replication is the process of synchronizing data across multiple servers.
- Redundancy and data availability with multiple copies of data on different database servers
- Replication allows to recover from hardware failure and service interruptions



Why Replication

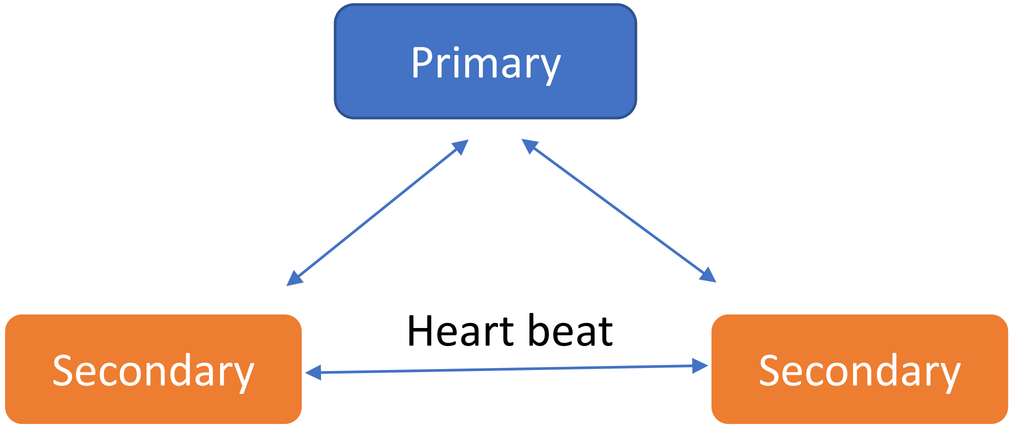
- To keep your data safe
- High availability and durability of data
- Protection against hardware unavailability or failure
  - Disaster recovery
  - No downtime for maintenance (like backups, index rebuilds, compaction)
- Read scaling (extra copies to read from)
- Replica set is transparent to the application
  - From application point of view, it is as if connected to a non replicated database

Replication Concept

- A replica set is a group of **mongod** instances that host the same data set.
- One primary node and 2 or more secondary nodes
- All write operations go to primary
- All changes are recorded into operations log
- Asynchronous replication to secondary
- Automatic failover
- Automatic recovery
- At the time of automatic failover or maintenance, election establishes for primary and a new primary node is elected.

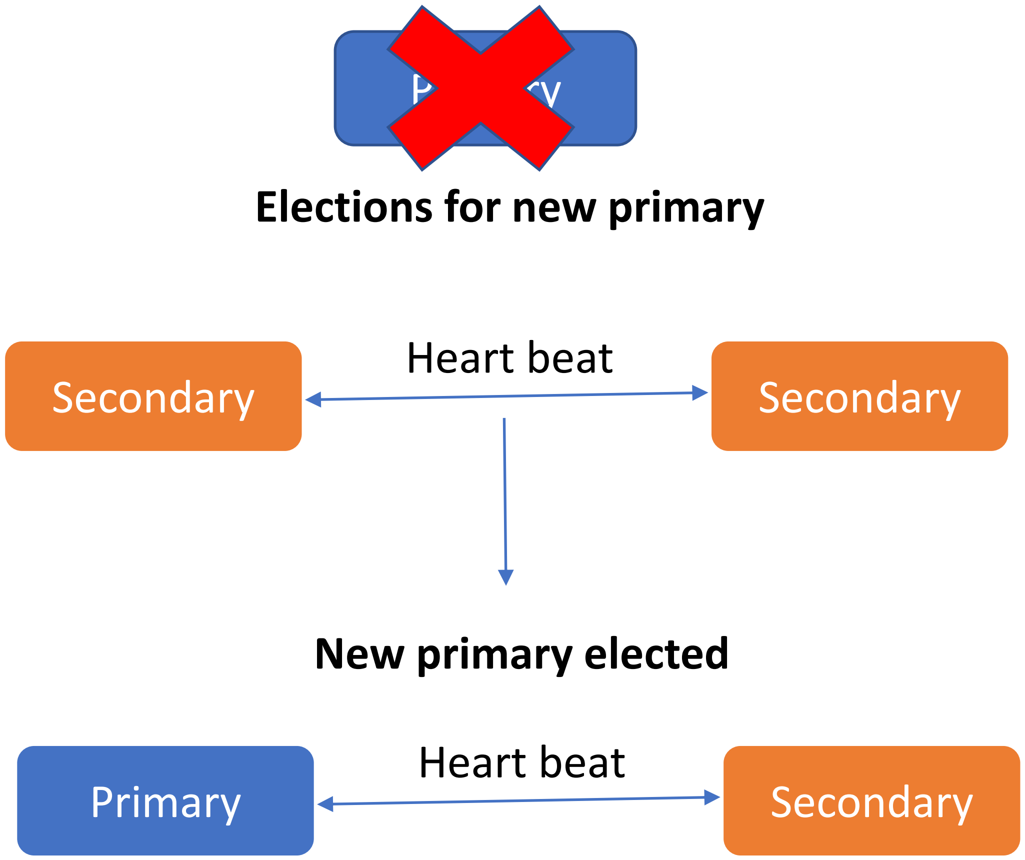
Automatic Failover

- When primary node stops
- Another member is selected as the new primary
- Selection happens through an election process
- The secondary node that gets majority of votes become the primary



Automatic Failover

- When primary node stops
- Another member is selected as the new primary
- Selection happens through an election process
- The secondary node that gets majority of votes become the primary



MongoDB Data Sharding

Sharding is the process of storing data records across multiple machines to enable:

Geo-Locality: for geographically distributed deployments to support optimal UX for customers across vast geographies.

Scale: To support massive workloads and data volumes

Hardware optimizations: on performance vs. Cost

Lower recovery times: to make recovery time objectives feasible (RTO)

What is sharding?

- Sharding Adds more servers to a database and automatically balances data and load across various servers
- Sharding provides additional write capacity by distributing the write load over a number of mongodb instances
- Sharding splits the data set and distributes them across multiple databases, or shards. Each shard serves as an independent database, and together, shards make a single logical database.
- Sharding reduces the number of operations each shard handles.

If a database has 1 TB data set distributed amongst 4 shards, then each shard may hold only 256 GB of data.

When to use Sharding?

- The data set outgrows the storage capacity of a single MongoDB instance
- The size of the active working set exceeds the capacity of the maximum available RAM
- A single MongoDB instance is unable to manage write operations

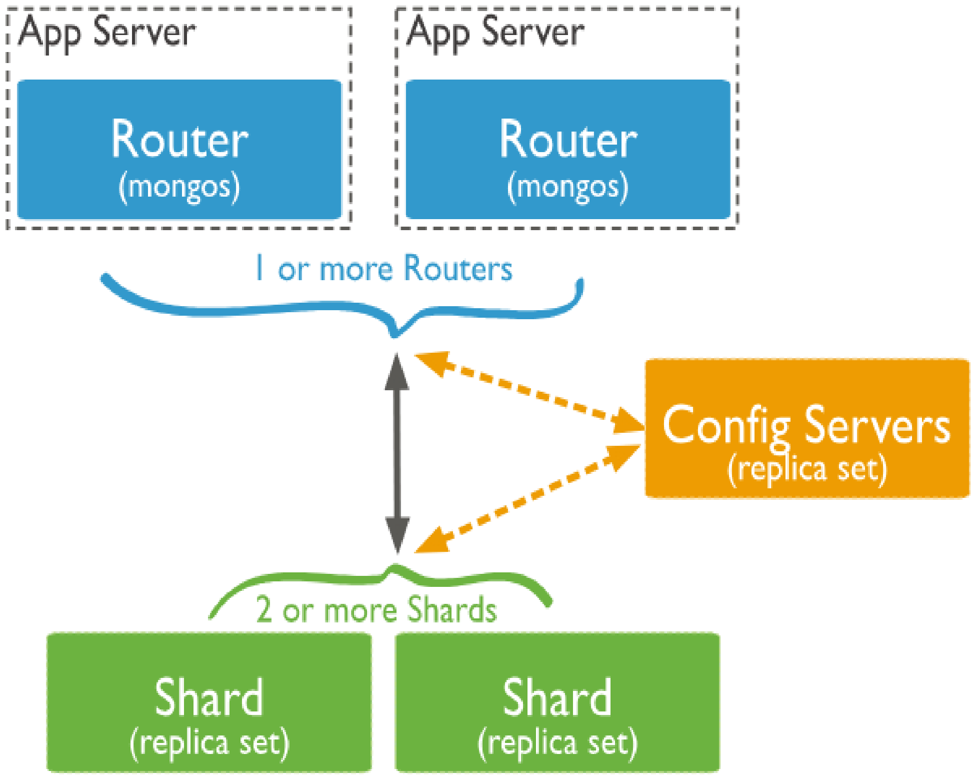
Deploy sharding if you expect that the read and write operations in your database are going to be increased in the future.

What is a shard?

- A shard is a replica set or a single mongod instance that holds the data subset used in a shared cluster. Each shard is a replica set that provides redundancy and high availability for the data it holds.
- MongoDB shards data on a per collection basis.
- When directly connected to a shard, you will be able to view only a fraction of the data contained in a cluster.
- Data is not organized in any particular order in a shard
- There in so guarantee that two contiguous data chunk will reside on any particular shard

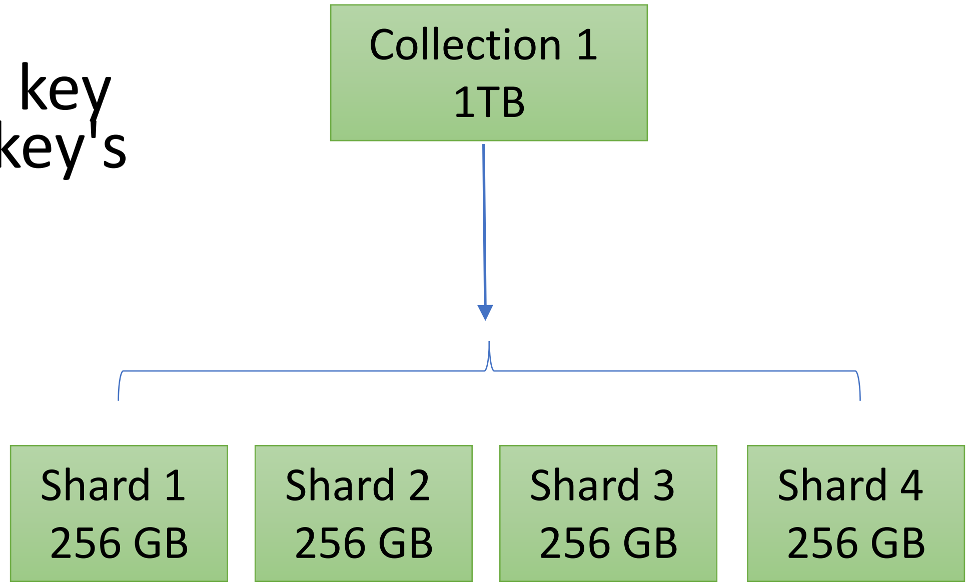
Sharded Cluster

- **Shard**: Each shard contains a subset of the sharded data. Each shard can be deployed as a replica set
- **Mongos**: The mongos acts as a query router, providing an interface between client applications and the sharded cluster.
- **Config servers**: Config servers store metadata and configuration settings for the cluster.



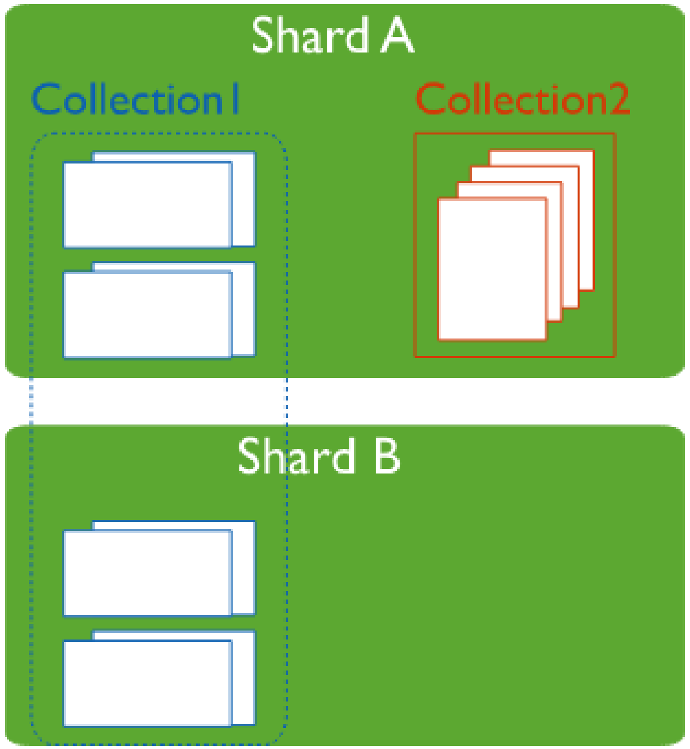
What is a Shard Key

- Wen deploying sharding, you need to choose a key from a collection and split the data using they key's value. The characteristics of a shard key are as follows:
- Determines document distribution among the different shards in a cluster.
- Is a field that exists in every document in the collection and can be an indexed or indexed compound field
- Performs data partitions in a collection.
- Helps distribute documents according to its range values.



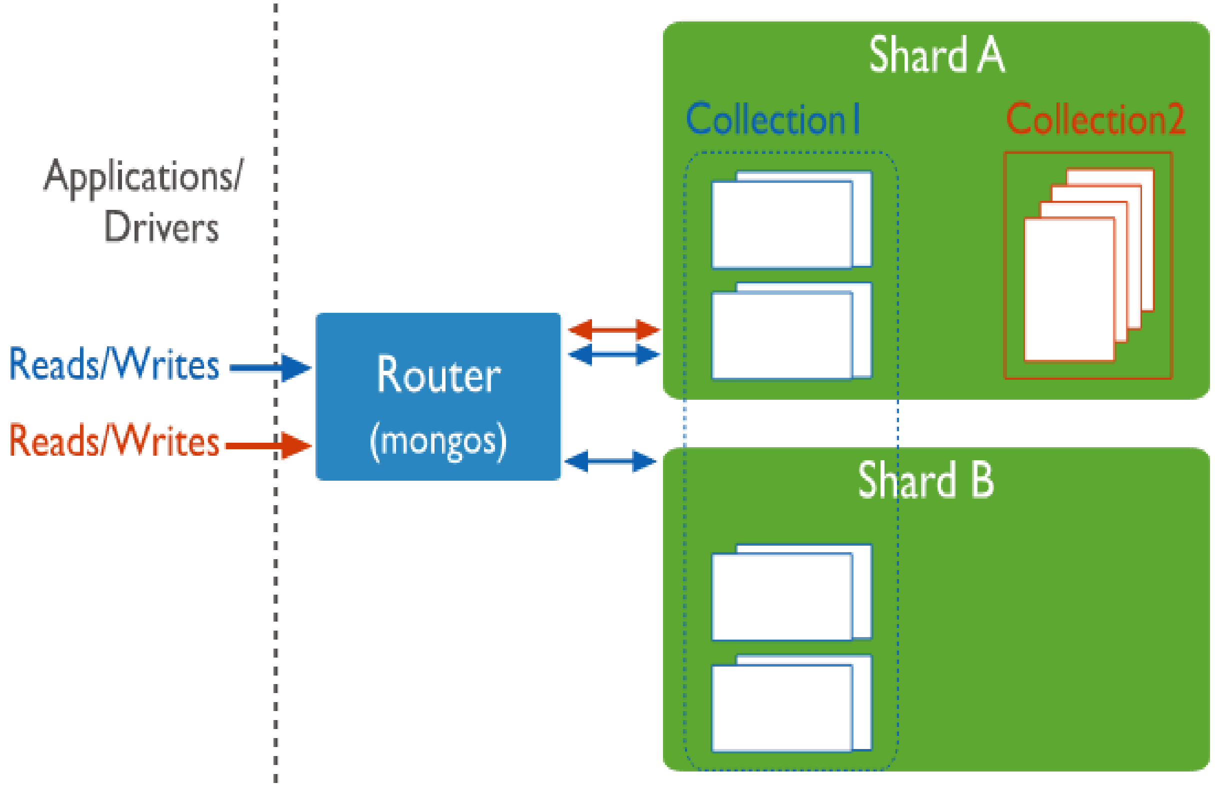
Sharded and non sharded collections

- A database can have a mixture of sharded and unsharded collections.
- Sharded collections are partitioned and distributed across the shards in the cluster.
- Unsharded collections are stored on a primary shard. Each database has its own primary shard.

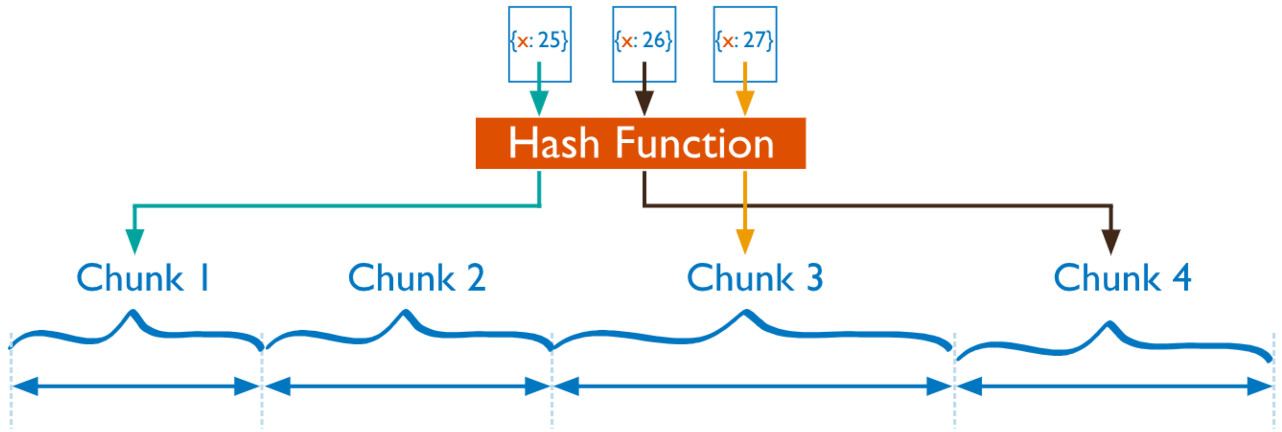


Connecting to a Shard cluster

- You must connect to a mongos router to interact with any collection in the sharded cluster. This includes sharded *and* unsharded collections
- Clients should *never* connect to a single shard in order to perform read or write operations.



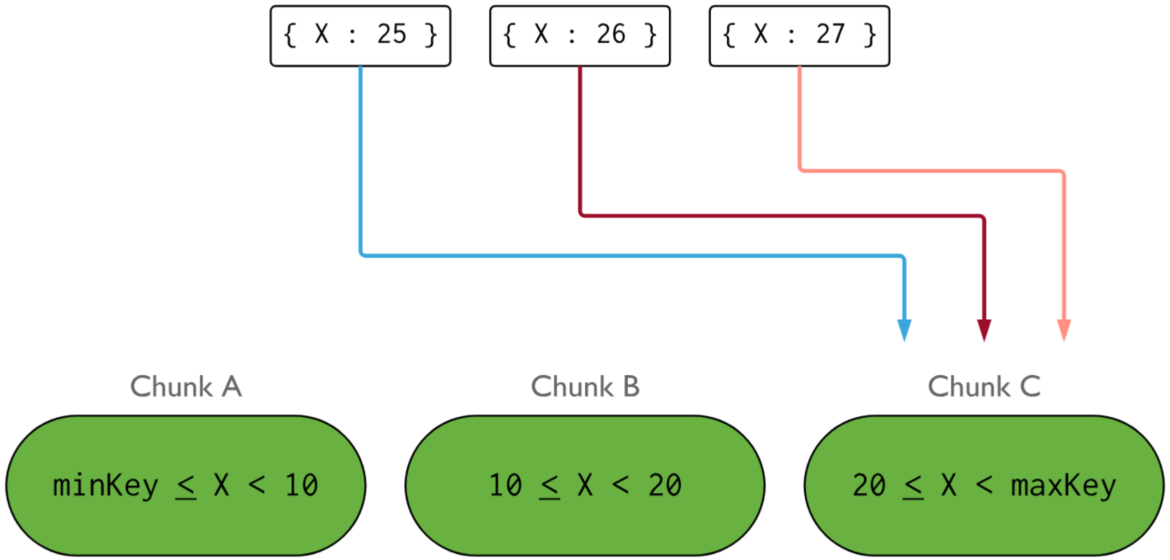
Hashed Sharding



- Hashed Sharding involves computing a hash of the shard key field’s value. Each chunck is then assigned a range based on the hashed shard key values
- A range of shard keys may be “close”, their hashed values are unlikely to be on the same **chunk**
- Data distribution based on hashed values facilitates more even data distribution.
- However, hashed distribution means that ranged-based queries on the shard key are less likely to target a single shard, resulting in more cluster wide **broadcast operations**

Ranged Sharding

- Data is divided into ranges based on the shard key values.
- A range of shard keys whose values are “close” are more likely to reside on the same **chunk**.
- This allows for **targeted operations** as a mongos can route the operations to only the shards that contain the required data.
- Poorly considered shard keys can result in uneven distribution of data, which can cause performance bottlenecks.



References

- http://milinda.pathirage.org/kappa-architecture.com/
- https://www.youtube.com/watch?v=fU9hR3kiOK0
- https://www.youtube.com/watch?v=aJuo_bLSW6s&feature=youtu.be
- https://www.oreilly.com/ideas/questioning-the-lambda-architecture
- https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying
- https://ferd.ca/beating-the-cap-theorem-checklist.html
- https://martinfowler.com/eaaDev/EventSourcing.html
- https://martinfowler.com/bliki/CQRS.html
- http://lambda-architecture.net/stories/
- http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html
- https://codahale.com/you-cant-sacrifice-partition-tolerance/
- https://www.voltdb.com/blog/2010/10/21/clarifications-cap-theorem-data-related-errors/
- http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/
- https://docs.microsoft.com/en-us/azure/architecture/patterns/event-sourcing
- https://mapr.com/developercentral/lambda-architecture/
- http://cs-www.cs.yale.edu/homes/dna/papers/abadi-pacelc.pdf