Unit 4: NoSQL Database

NoSQL for Mere Mortals Dan Sullivan- 1st Edition, Pearson Education

Chapter 1 and 2

Mrs. Deepali Jaadhav

Unit 4: NoSQL Database

- Introduction to NoSQL Database
- Data management with distributed databases,
- ACID and BASE
- NoSQL Types:
  - Key-Value Database,
  - Document Database,
  - Column Family Database and
  - Graph Database
- Comparison of relational databases and NoSQL

Introduction of NoSQL Databases

- A database Management System provides the mechanism to store and retrieve the data.
- There are different kinds of database Management Systems:
  1. RDBMS (Relational Database Management Systems)
  2. OLAP (Online Analytical Processing)
  3. NoSQL (Not only SQL)

What are NoSQL databases ?

- “NoSQL refers to an ill-defined set of mostly open-source databases, mostly developed in the early 21st century, and mostly not using SQL”

What is a NoSQL database?

- NoSQL databases are different than relational databases like MySql.
- In relational database you need to create the table, define schema, set the data types of fields etc before you can actually insert the data.
- In NoSQL you don't have to worry about that, you can insert, update data on the fly.
- NoSQL database - really easy to scale and they are much faster in most types of operations that we perform on database.
- When you are dealing with huge amount of data then NoSQL database is your best choice.

Common Characteristics

- Not using the relational model
- Run well on clusters
- Can handle huge amount of data
- Open Source
- Build for 21st century web access
- Schema-less / schema-free
- BASE (not ACID)

Limitations of Relational databases

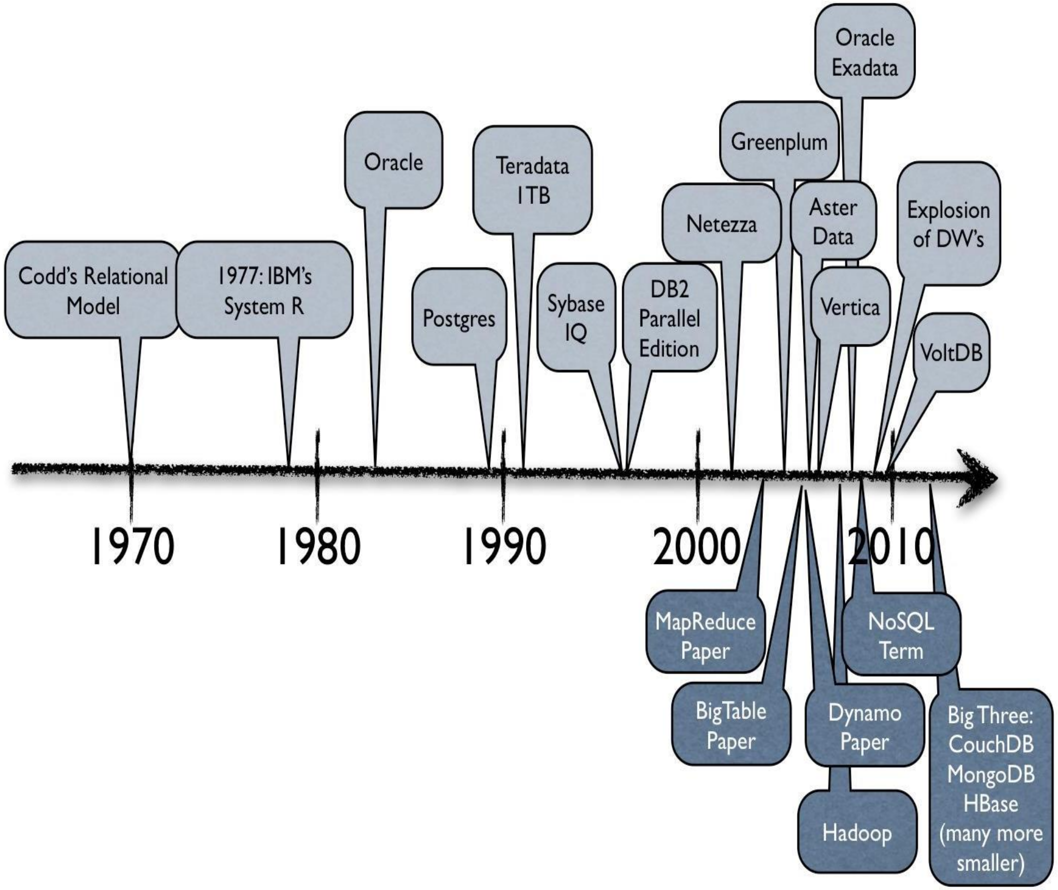
- In relational database we need to define structure and schema of data first and then only we can process the data.
- Relational database systems provides consistency and integrity of data by enforcing ACID properties (Atomicity, Consistency, Isolation and Durability).
- However in most of the other cases these properties are significant performance overhead and can make your database response very slow.
- RDBMS don't provide you a better way of performing operations such as create, insert, update, delete etc on this data.
- On the other hand NoSQL store their data in **JSON** format, which is compatible with most of the today's world application.

History of NoSQL Databases

- In the mid-1990s, the internet gained extreme popularity, and relational databases simply could not keep up with the flow of information demanded by users,
- As well as the larger variety of data types that occurred from this evolution.
- This led to the development of non-relational databases, often referred to as NoSQL.
- NoSQL databases can translate strange data quickly and avoid the rigidity of SQL by replacing "organized" storage with more flexibility.

History of NoSQL Databases

- 1998- Carlo Strozzi use the term NoSQL for his lightweight, open-source relational database
- 2000- Graph database Neo4j is launched
- 2004- Google BigTable is launched
- 2005- CouchDB is launched
- 2007- The research paper on Amazon Dynamo is released
- 2008- Facebooks open sources the Cassandra project
- 2009- The term NoSQL was reintroduced

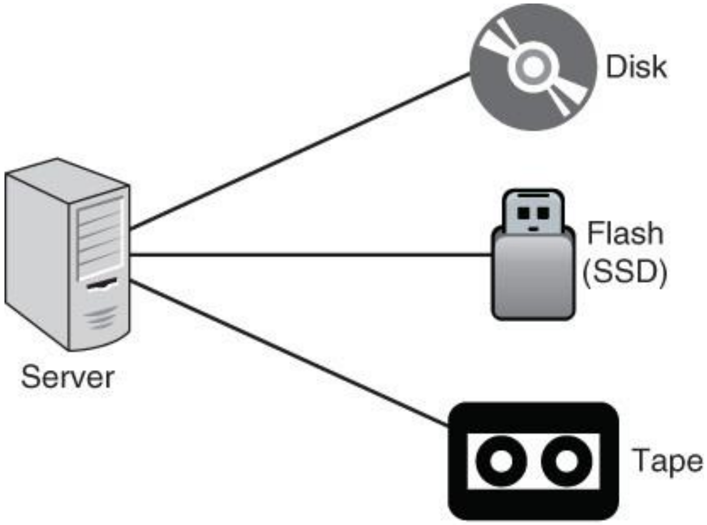


Data Management with Distributed Databases

- Before getting into the details of distributed databases, let's look at a simplified view of databases in general.
- Databases are designed to do two things: **store data and retrieve data**.
- To meet these objectives, the database management systems must do three things:
  - Store data persistently
  - Maintain data consistency
  - Ensure data availability

Store Data Persistently

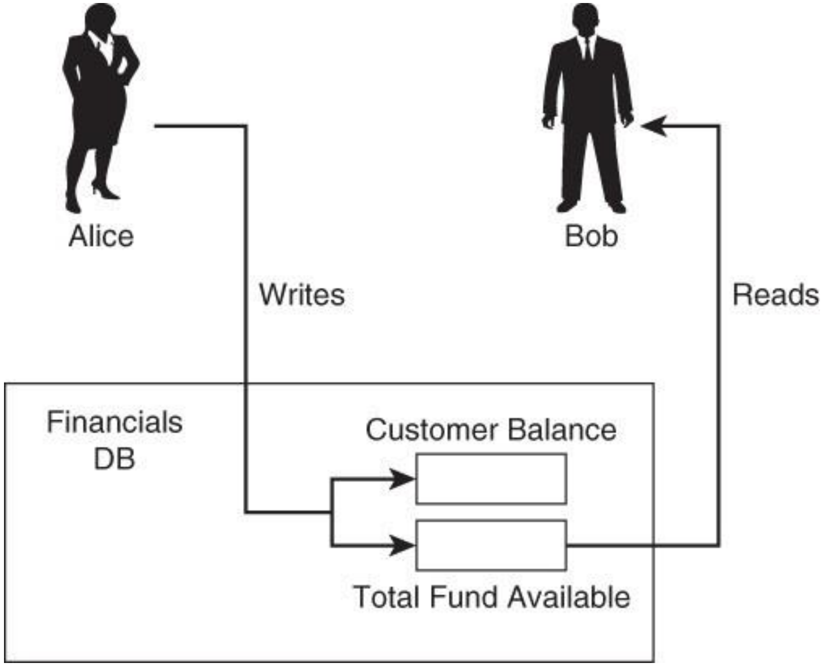
- Data must be stored persistently; that is, it must be stored in a way that data is not lost when the database server is shut down.
- If data were only stored in memory—that is, RAM —then it would be lost when power to the memory is lost.
- Only data that is stored on disk, flash, tape, or other long-term storage is considered persistently stored.
- Indices are used for faster retrieval of data.



Persistently Stored Data

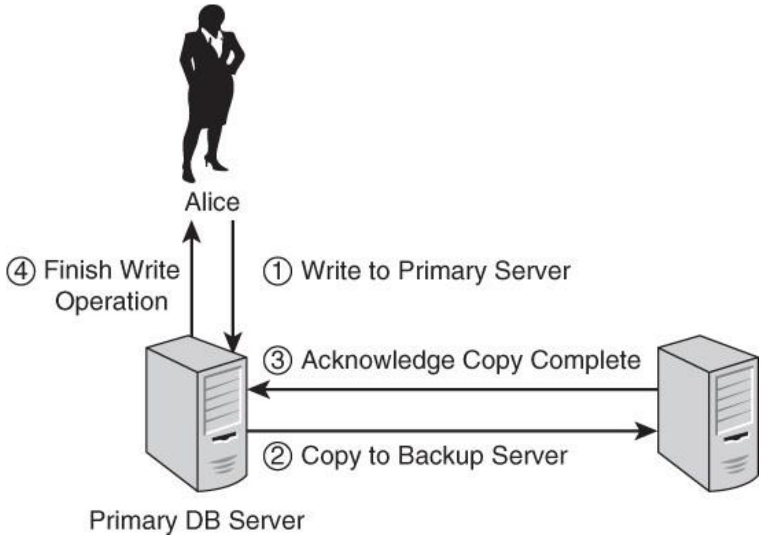
Maintain Data Consistency

- It is important to ensure that the correct data is written to a persistent storage device.
- If the write or read operation does not accurately record or retrieve data, the database will not be of much use.
- Concurrency control mechanism is used for data consistency.



Ensure Data Availability

- Data should be available whenever it is needed. This is difficult to guarantee.
- One way to avoid the problem of an unavailable database server is to have two database servers.
- One is used for updating data and responding to queries while the other is kept as a backup in case the first server fails. The server that is used for updating and responding to queries is called the primary server, and the other is the backup server



Ensure Data Availability

- With data consistent on two database servers, you can be sure that if the primary database fails, you can switch to using the backup database and know that you have the same data on both.
- When the primary database is back online, the first thing it does is to update itself so that all changes made to the backup database while the primary database was down are made to the primary database.
- The advantage of using two database servers is that it enables the database to remain available even if one of the servers fails.

Ensure Data Availability

- Because, in the case of a two-phase commit, a write operation is not complete until both databases are updated successfully, the speed of the updates depends on the amount of data written, the speed of the disks, the speed of the network between the two servers, and other design factors.

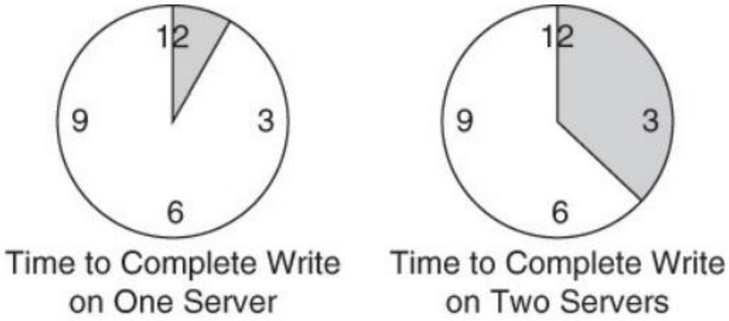

12
9
3
6
Time to Complete Write on One Server
12
9
3
6
Time to Complete Write on Two Servers

Figure 2.5 *Consistency and availability require more time to complete transactions in high-availability environments.*

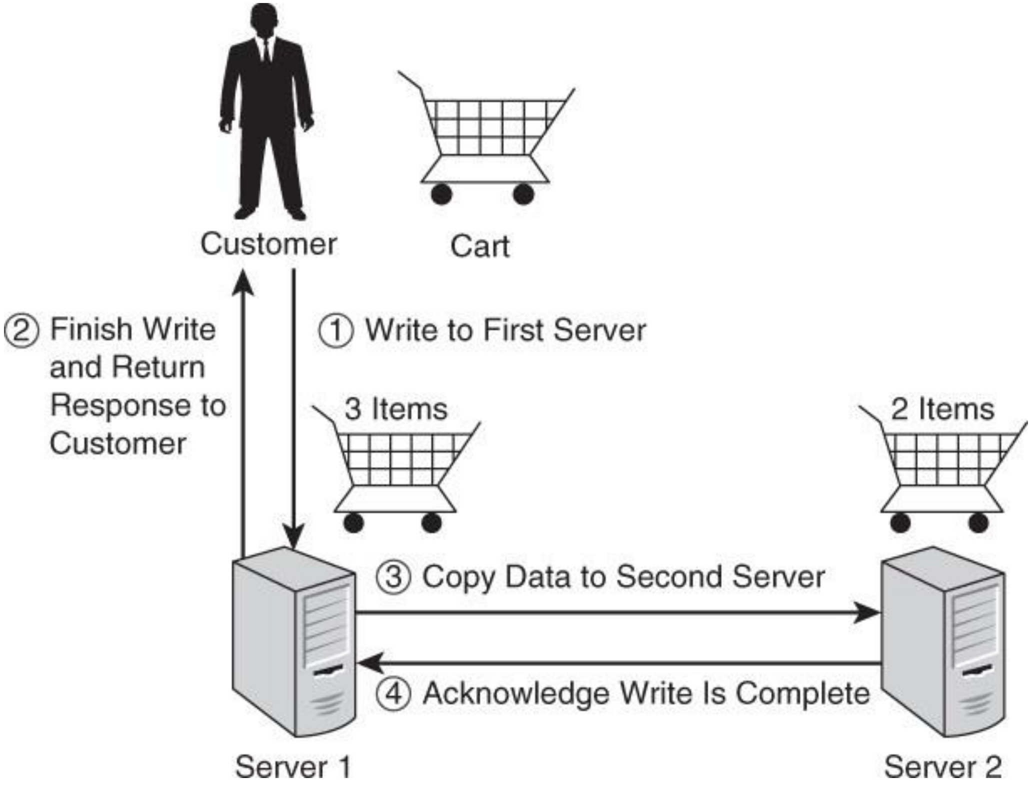
- You can have consistent data, and you can have a high-availability database, but transactions will require longer times to execute than if you did not have those requirements.

Availability and Consistency in Distributed Databases

- When two database servers must keep consistent copies of data, they incur longer times to complete a transaction.
- This is acceptable in applications that require both consistency and high availability at all times. Financial systems at a bank, for example, fall into this category.
- There are applications, in which the fast database operations are more important than maintaining consistency at all times. For example, an ecommerce site.
- Imagine you are programming the user interface for an e-commerce site. How long should the customer wait after clicking on an "Add to My Cart" button? Ideally, the interface would respond immediately so the customer could keep shopping.
- In this case, speed is more important than having consistent data at all times.

Availability and Consistency in Distributed Databases

- One way to deal with this problem is to write the updates to one database and then copy the data to another server. There is a brief period of time when the customer's cart on the two servers is not consistent, but the customer is able to continue shopping anyway.

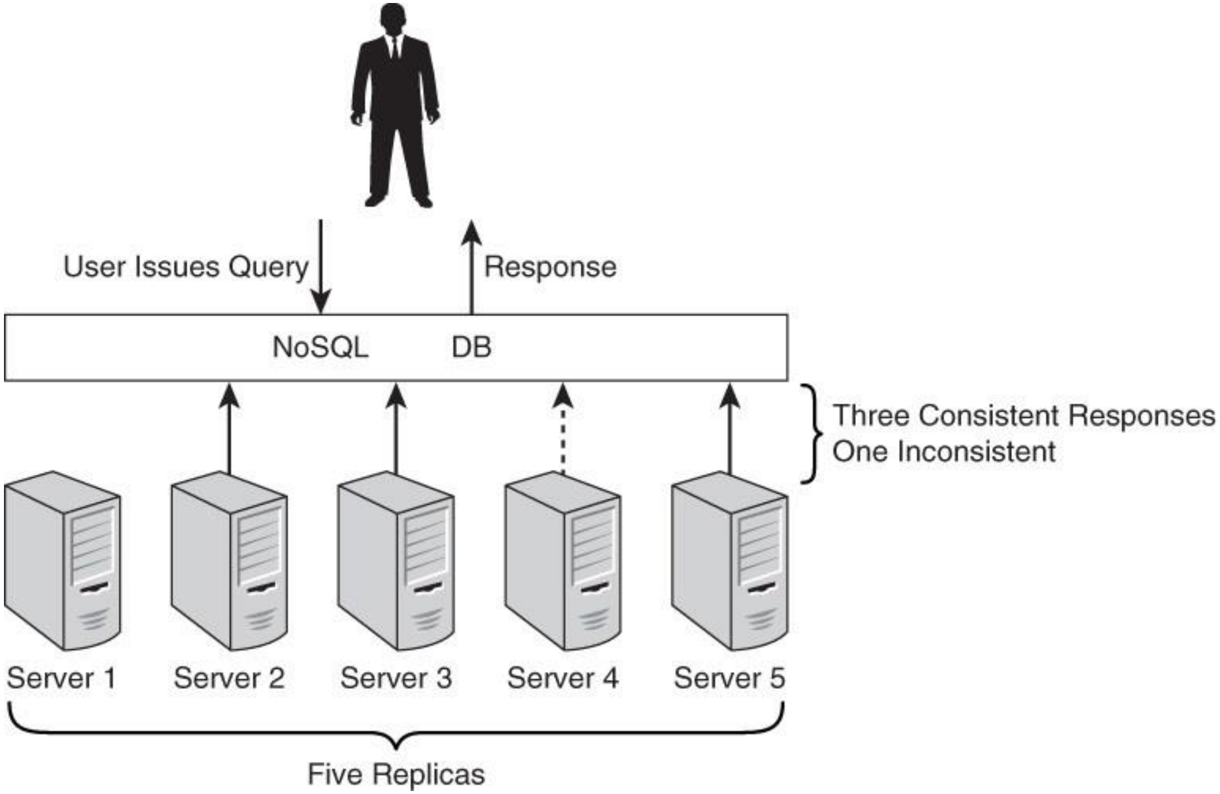

Customer
Cart
② Finish Write and Return Response to Customer
① Write to First Server
3 Items
2 Items
③ Copy Data to Second Server
④ Acknowledge Write Is Complete
Server 1
Server 2

Balancing Response Times, Consistency, and Durability

- NoSQL databases often implement eventual consistency; that is, there might be a period of time where copies of data have different values, but eventually all copies will have the same value.
- This raises the possibility of a user querying the database and getting different results from different servers in a cluster.
- NoSQL databases often use the concept of **quorums** when working with reads and writes.
- A ***quorum*** *is the number of servers that must respond to a read or write operation for the* operation to be considered complete.
- When a read is performed, the NoSQL database reads data from, potentially, multiple servers. Most of the time, all of the servers will have consistent data. While the database copies data from one of the servers to the other servers storing replicas, the replica servers may have inconsistent data.

Balancing Response Times, Consistency, and Durability

- One way to determine the correct response to any read operation is to query all servers storing that data. The database counts the number of distinct response values and returns the one that meets or exceeds a configurable threshold.



Balancing Response Times, Consistency, and Durability

- You can vary the threshold to improve response time or **consistency**. If the read threshold is set to 1, you get a fast response. The lower the threshold, the faster the response but the higher the risk of returning inconsistent data.
- Just as you can adjust a read threshold to balance response time and consistency, you can also alter a write threshold to balance response time and durability.
- ***Durability*** *is the* property of maintaining a correct copy of data for long periods of time.
- A write operation is considered complete when a minimum number of replicas have been written to persistent storage.

Brewer’s CAP Theorem

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

- Consistency
- Availability
- Partition tolerance

Proven by Nancy Lynch et al. MIT labs.

Consistency

- Consistency: Clients should read the same data. There are many levels of consistency.
    - Strict Consistency – RDBMS.
    - Tunable Consistency – Cassandra.
    - Eventual Consistency – Amazon Dynamo.
- Client perceives (realize) that a set of operations has occurred all at once – Pritchett
- More like Atomic in ACID transaction properties

Consistency

- **Strict consistency:** is the strongest consistency model. Under this model, a write to a variable by any processor needs to be seen instantaneously by all processors
- **Tunable Consistency:** means that you can set the consistency level for each read and write request. So, Cassandra gives you a lot of control over how consistent your data is.
  - You can allow some queries to be immediately consistent and other queries to be eventually consistent.
  - That means, in your application, the data that requires immediate consistency, you can create your queries accordingly and the data for which immediate consistency is not required, you can optimize for performance and choose eventual consistency.
- **Eventual Consistency**: is a guarantee that when an update is made in a distributed database, that update will eventually be reflected in all nodes that store the data, resulting in the same response every time the data is queried.

  Eventual consistency – data will become consistent at some point in time, with no guarantee when.

Consistency - Example

- Consistent databases should be used when the value of the information returned needs to be accurate.
- Financial data is a good example. When a user logs in to their banking institution, they do not want to see an error that no data is returned, or that the value is higher or lower than it actually is. Banking apps should return the exact value of a user's account information. In this case, banks would rely on consistent databases.
- Examples of a consistent database include:
  - Bank account balances
  - Text messages
- Database options for consistency:
  - MongoDB
  - Redis
  - HBase

Availability

- Availability: Data to be available.
- Node failures do not prevent survivors from continuing to operate
- Every operation must terminate in an intended response

Availability - Example

- Availability databases should be used when the service is more important than the information.
- An example of having a highly available database can be seen in e-commerce businesses. Online stores want to make their store and the functions of the shopping cart available 24/7 so shoppers can make purchases exactly when they need.
- Database options for availability:
  - Cassandra
  - DynamoDB
  - Cosmos DB

Partition Tolerance

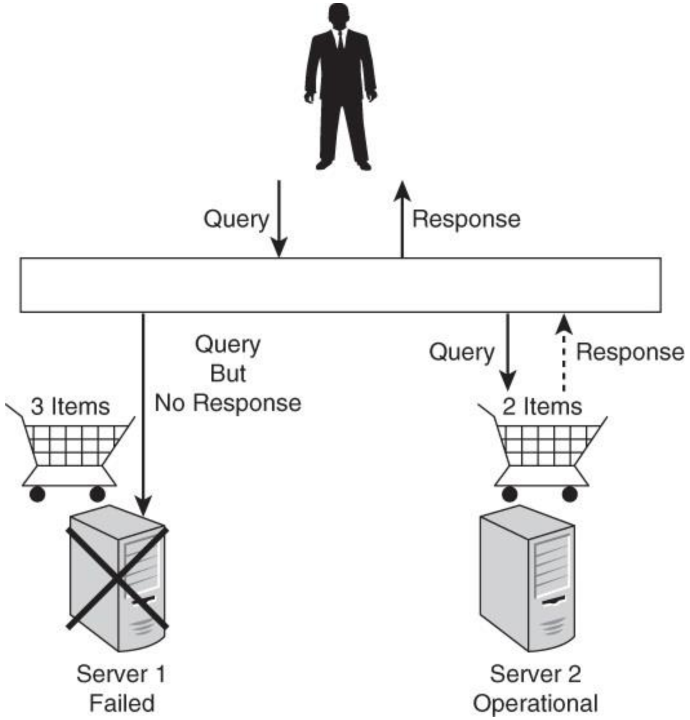
- A ***partition*** is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes.
- **Partition tolerance** means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.
- **Partial Tolerance**: Data to be partitioned across network segments due to network failures.
- The system continues to operate despite arbitrary message loss.
- Operations will complete, even if individual components are unavailable

Partition Tolerance

- In normal operations, your data store provides all three functions.
- But the CAP theorem maintains that when a distributed database experiences a network failure, you can provide either consistency or availability.
- All other times, all three can be provided. But, in the event of a network failure, a choice must be made.
- In the theorem, partition tolerance is a must. The assumption is that the system operates on a distributed data store, operates with network partitions.
- Network failures will happen, so to offer any kind of reliable service, partition tolerance is necessary—the P of CAP. That leaves a decision between the other two, C and A.
- When a network failure happens, one can choose to guarantee consistency or availability:
  - High consistency comes at the cost of lower availability.
  - High availability comes at the cost of lower consistency.

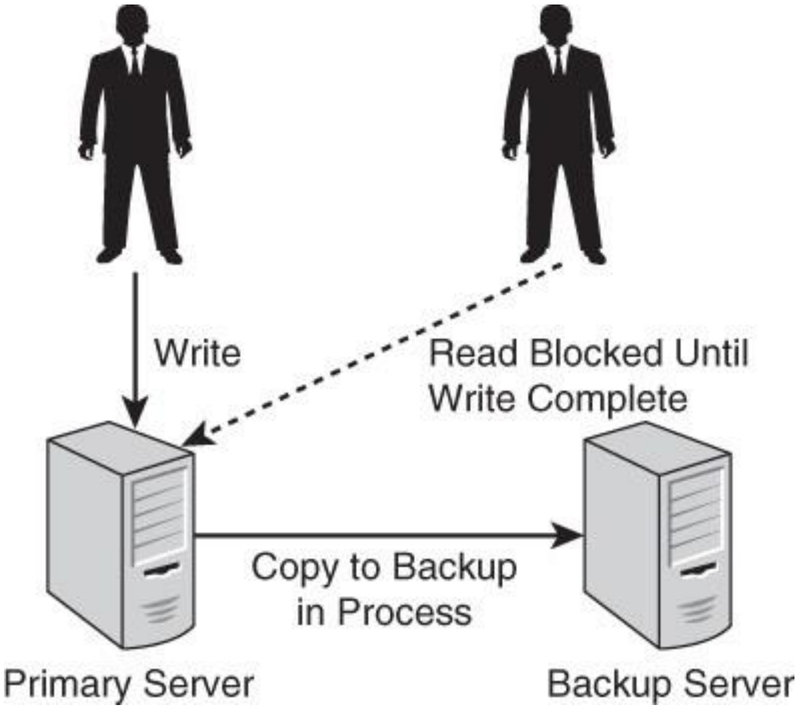
Example

- The e-commerce shopping cart that it is possible to have a backup copy of the cart data that is out of sync with the primary copy. The data would still be available if the primary server failed, but the data on the backup server would be inconsistent with data on the primary server if the primary server failed prior to updating the backup server.

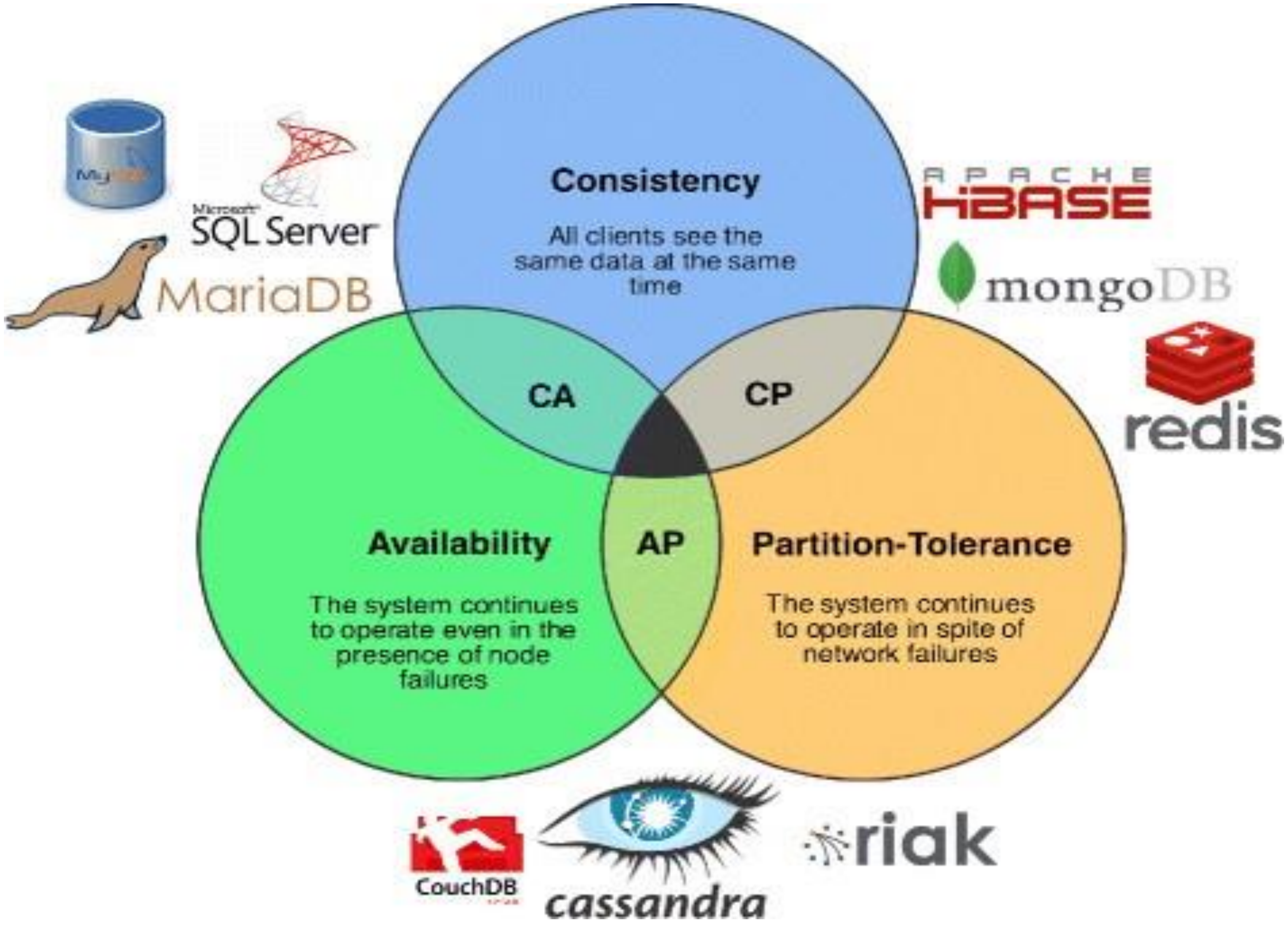


Example

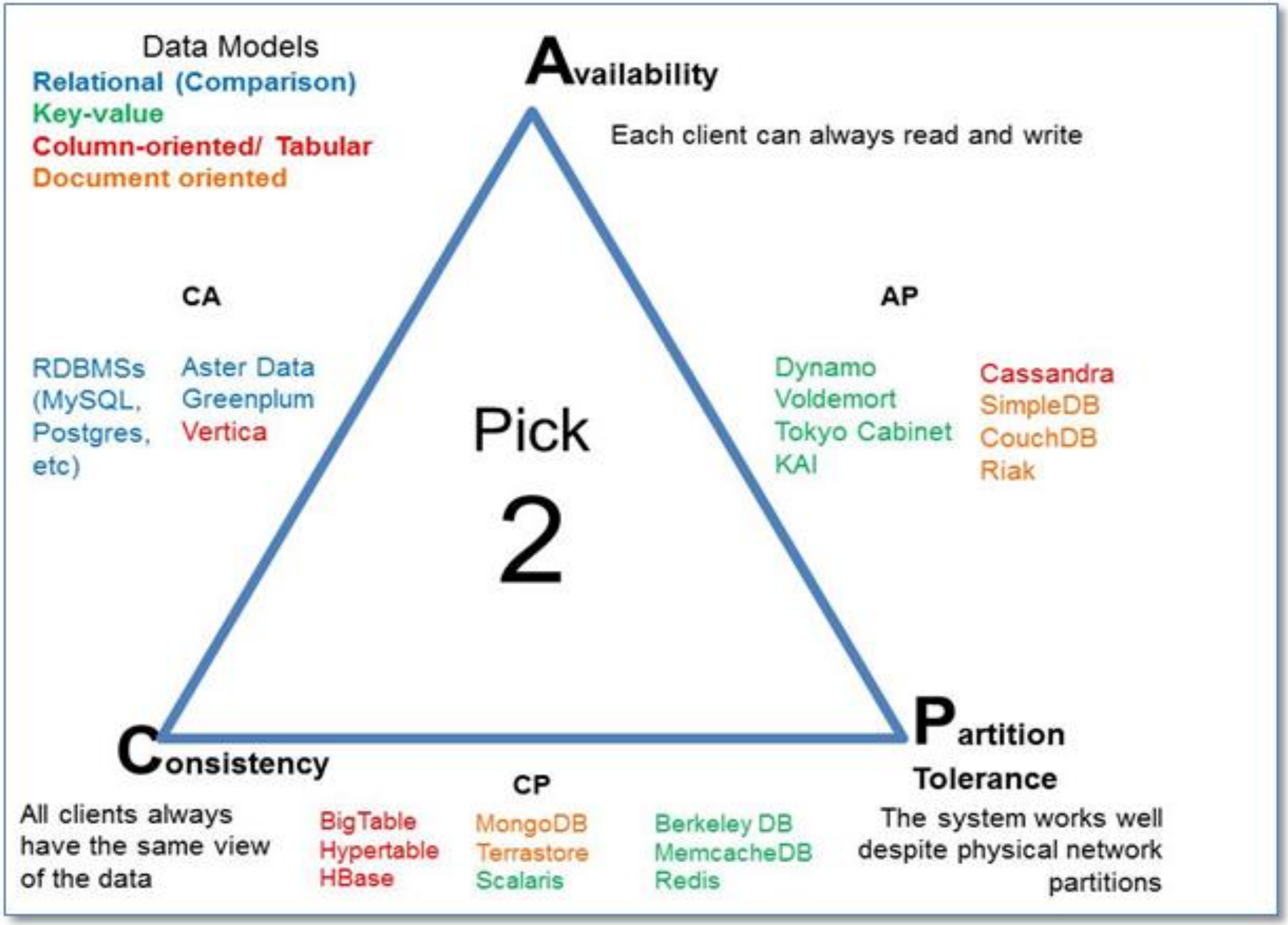
- Partition protection deals with situations in which servers cannot communicate with each other. This would be the case in the event of a network failure.
- This splitting of the network into groups of devices that can communicate with each other from those that cannot is known as partitioning.



CAP theorem NoSQL database types


Consistency
All clients see the same data at the same time
Availability
The system continues to operate even in the presence of node failures
Partition-Tolerance
The system continues to operate in spite of network failures
CA
CP
AP
MySQL
Microsoft SQL Server
MariaDB
APACHE HBASE
mongoDB
redis
CouchDB
cassandra
riak

CAP Theorem



Data Models
Relational (Comparison)
Key-value
Column-oriented/ Tabular
Document oriented

Availability

Each client can always read and write

CA

RDBMSs (MySQL, Postgres, etc)
Aster Data
Greenplum
Vertica

AP

Dynamo
Voldemort
Tokyo Cabinet
KAI

Cassandra
SimpleDB
CouchDB
Riak

Pick 2

Consistency

All clients always have the same view of the data

CP

BigTable
Hypertable
HBase

MongoDB
Terrastore
Scalaris

Berkeley DB
MemcacheDB
Redis

Partition Tolerance

The system works well despite physical network partitions

ACID and BASE

ACID: Atomicity, Consistency, Isolation, and Durability

- **A is for *atomicity*** Atomicity, as the name implies, describes a unit that cannot be further divided. The set of steps is indivisible. You have to complete all of them as a single indivisible unit, or you complete none of them.
- **C is for *consistency***. *In relational databases, this is known as strict consistency. In other* words, a transaction does not leave a database in a state that violates the integrity of data.
- **I is for *isolation***. *Isolated transactions are not visible to other users until transactions are* complete. For example, in the case of a bank transfer from a savings account to a checking account, someone could not read your account balances while the funds are being deducted from your savings account but before they are added to your checking account.
- **D is for durability**. This means that once a transaction or operation is completed, it will remain even in the event of a power loss. In effect, this means that data is stored on disk, flash, or other persistent media.

Relational database management systems are designed to support ACID transactions

ACID and BASE

BASE: Basically Available, Soft State, Eventually Consistent

- **BA is for basically available**. This means that there can be a partial failure in some parts of the distributed system and the rest of the system continues to function.
  - For example, if a NoSQL database is running on 10 servers without replicating data and one of the servers fails, then 10% of the users' queries would fail, but 90% would succeed.
  - NoSQL databases often keep multiple copies of data on different servers. This allows the database to respond to queries even if one of the servers has failed.
- **S is for soft state.** Usually in computer science, the term soft state means data will expire if it is not refreshed. In NoSQL operations, it refers to the fact that data may eventually be overwritten with more recent data. This property overlaps with the third property of BASE transactions, eventually consistent.

ACID and BASE

- **E is for eventually consistent**. This means that there may be times when the database is in an inconsistent state. For example, some NoSQL databases keep multiple copies of data on multiple servers. There is, however, a possibility that the multiple copies may not be consistent for a short period of time.

- The time it takes to update all copies depends on several factors, such as the load on the system and the speed of the network.

ACID and BASE

- **Types of Eventual Consistency:**
  1. Casual consistency
  2. Read-your-writes consistency
  3. Session consistency
  4. Monotonic read consistency
  5. Monotonic write consistency

ACID and BASE

1. Casual consistency

- Casual consistency ensures that the database reflects the order in which operations were updated.
- For example, if Alice changes a customer's outstanding balance to $1,000 and one minute later Bob changes it to $2,000, all copies of the customer's outstanding balance will be updated to $1,000 before they are updated to $2,000.

2. Read-Your-Writes Consistency

- Once you have updated a record, all of your reads of that record will return the updated value. You would never retrieve a value inconsistent with the value you had written.
- For example, Alice updates a customer's outstanding balance to $1,500. The update is written to one server and the replication process begins updating other copies. During the replication process, Alice queries the customer's balance. She is guaranteed to see $1,500 when the database supports read your-writes consistency.

ACID and BASE

3. Session Consistency

- Session consistency ensures read-your-writes consistency during a session. You can think of a session as a conversation between a client and a server or a user and the database.
- As long as the conversation continues, the database "remembers" all writes you have done during the conversation.
- If the session ends and you start another session with the same server, there is no guarantee it will "remember" the writes you made in the previous session.

4. Monotonic Read Consistency

- Monotonic read consistency ensures that if you issue a query and see a result, you will never see an earlier version of the value.
- For Example, Alice is updating customer balance. Currently it is $1500 she updates it to $2500. If Bob queries the database he will see the balance is $2,500 even if all the servers with copies of that customer's balance have not updated to the latest value.

ACID and BASE

5. Monotonic Write Consistency

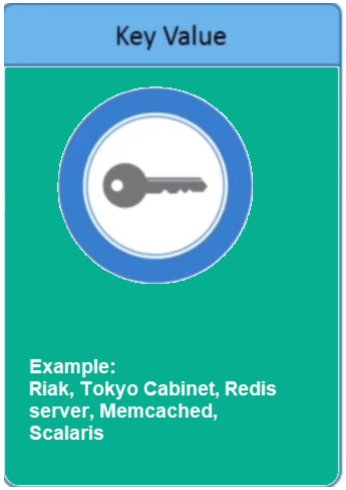
- Monotonic write consistency ensures that if you were to issue several update commands, they would be executed in the order you issued them.
- Let's consider a variation on the outstanding balance example. Alice decides to reduce all customers' outstanding balances by 10%.
- Charlie, one of her customers, has a $1,000 outstanding balance. After the reduction, Charlie would have a $900 balance.
- Charlie has just ordered $1,100 worth of material. His outstanding balance is now the sum of the previous outstanding balance ($900) and the amount of the new order ($1,100) or $2,000.

ACID and BASE

- **5. Monotonic Write Consistency (cont…)**
- Now consider what would happen if the NoSQL database performed Alice’s operations in a different order.
- Charlie started with a $1,000 outstanding balance. Next, instead of having the discount applied, his record was first updated with the new order ($1,100).
- His outstanding balance becomes $2,100. Now, the 10% discount operation is executed and his outstanding balance is set to $2,100–$210 or $1890.
- Monotonic write consistency is obviously an important feature. If you cannot guarantee the order of operations in the database, you would have to build features into your program to guarantee operations execute in the order you expect.

Types of NoSQL Databases

- The most widely used types of NoSQL databases are
  - **Key-value pair databases**
  - **Document databases**
  - **Column family store databases**
  - **Graph databases**
- Every category has its unique attributes and limitations. Users should select the database based on their product needs.

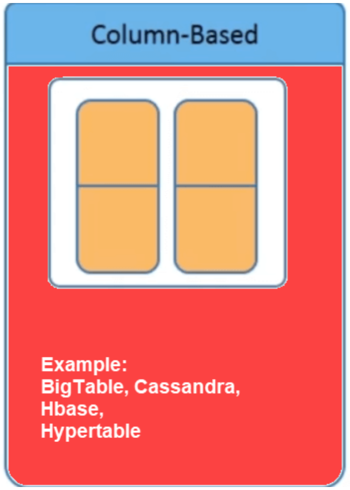


Key Value

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

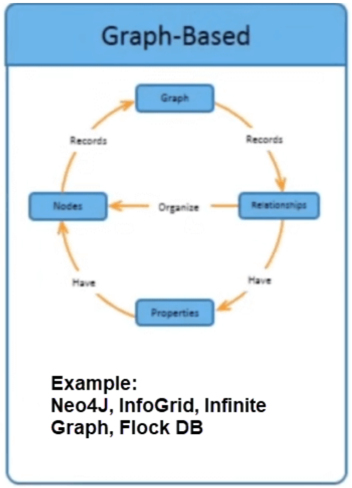
Document-Based

Example:
MongoDB, CouchDB, OrientDB, RavenDB



Column-Based

Example:
BigTable, Cassandra, Hbase, Hypertable

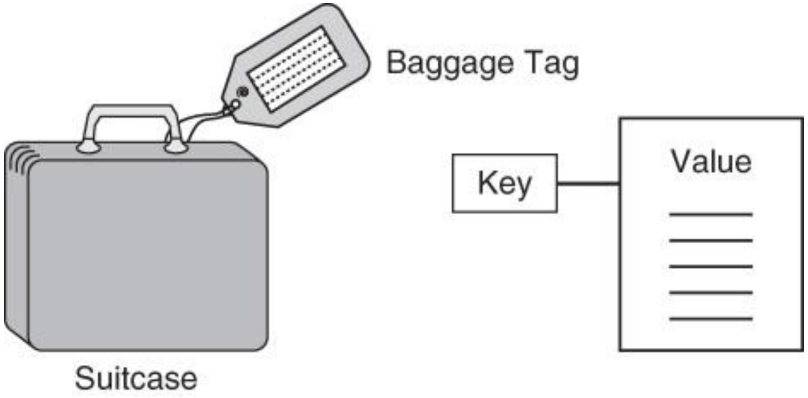


Graph-Based

Example:
Neo4J, InfoGrid, Infinite Graph, Flock DB

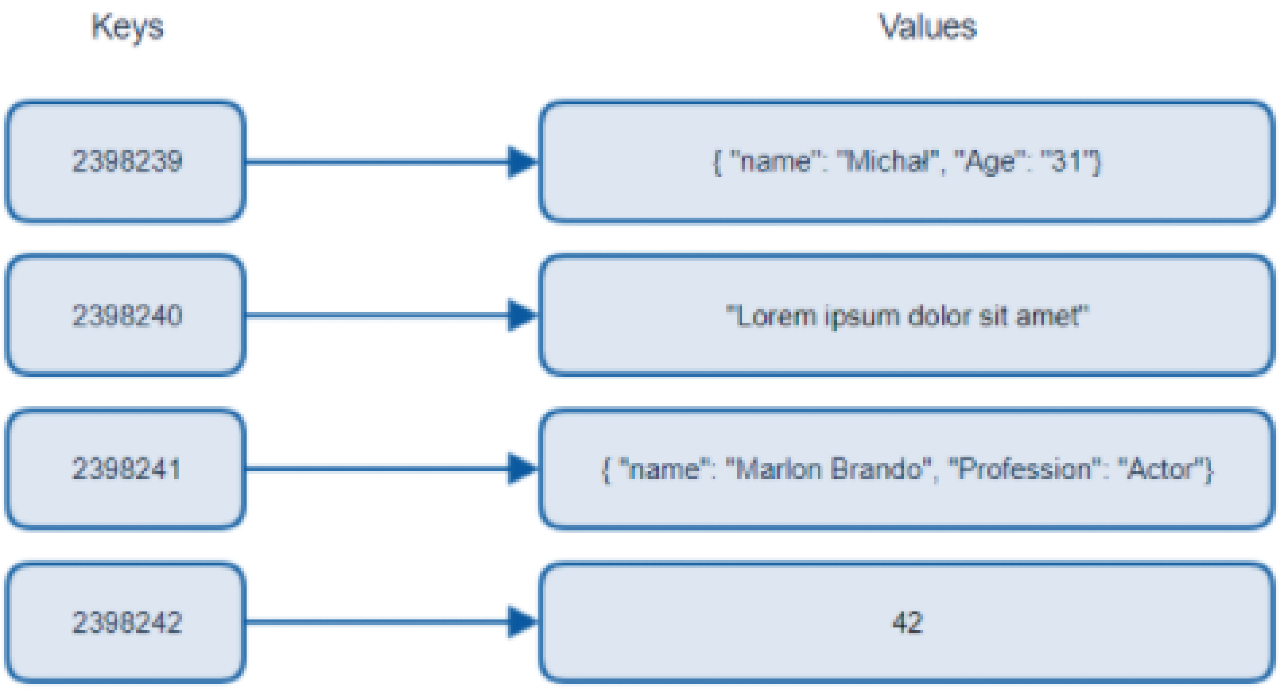
Key-Value Pair Databases

- Key-value pair databases are the simplest form of NoSQL databases. These databases are modeled on two components: **keys and values.**
- **Keys** - Keys are identifiers associated with values.
- For example The tag you receive has an identifier associated with your luggage. With your tag, you can find your luggage more efficiently than without it.
- The first customer checking bags on flight 1928 might be assigned ticket 1928.1 for her first bag and 1928.2 for her second bag. The second customer also has two bags and he is assigned 1928.3 and 1928.4



Key-Value Pair Databases

- Key-value databases are modeled on a simple, two-part data structure consisting of an identifier and a data value.
- The important principle to remember about keys is that they must be unique.



Keys	Values
2398239	{ "name": "Michal", "Age": "31"}
2398240	"Lorem ipsum dolor sit amet"
2398241	{ "name": "Marlon Brando", "Profession": "Actor"}
2398242	42

Key-Value Pair Databases

- **Values** - are data stored along with keys.
- Like luggage, values in a key-value database can store many different things.
- Values can be as simple as a string, such as a name, or a number, such as the number of items in a customer's shopping cart.
- You can store more complex values, such as images or binary objects, too.
- Key-value databases give developers a great deal of flexibility when storing values.
- For example, strings can vary in length. Values can also vary in type. An employee database might include photos of employees using keys such as Emp328.photo.

Key-value store Representatives

MemcachedDB

not open-source

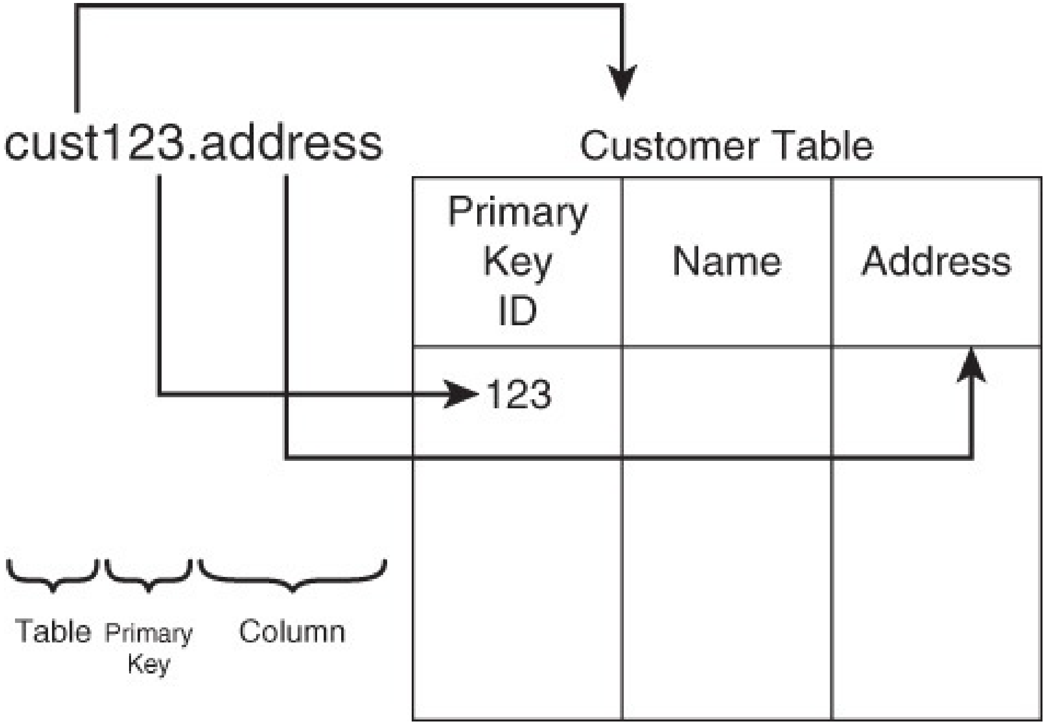
open-source version

Project Voldemort

Differences Between Key-Value and Relational Databases

- Key-value databases are modeled on minimal principles for storing and retrieving data.
- Unlike in relational databases, there are no tables, so there are no features associated with tables, such as columns and constraints on columns.
- There is no need for joins in key-value databases, so there are no foreign keys.
- Key-value databases do not support a rich query language such as SQL.
- Some key-value databases support buckets, or collections, for creating separate namespaces within a database. This can be used to implement something analogous to a relational schema.

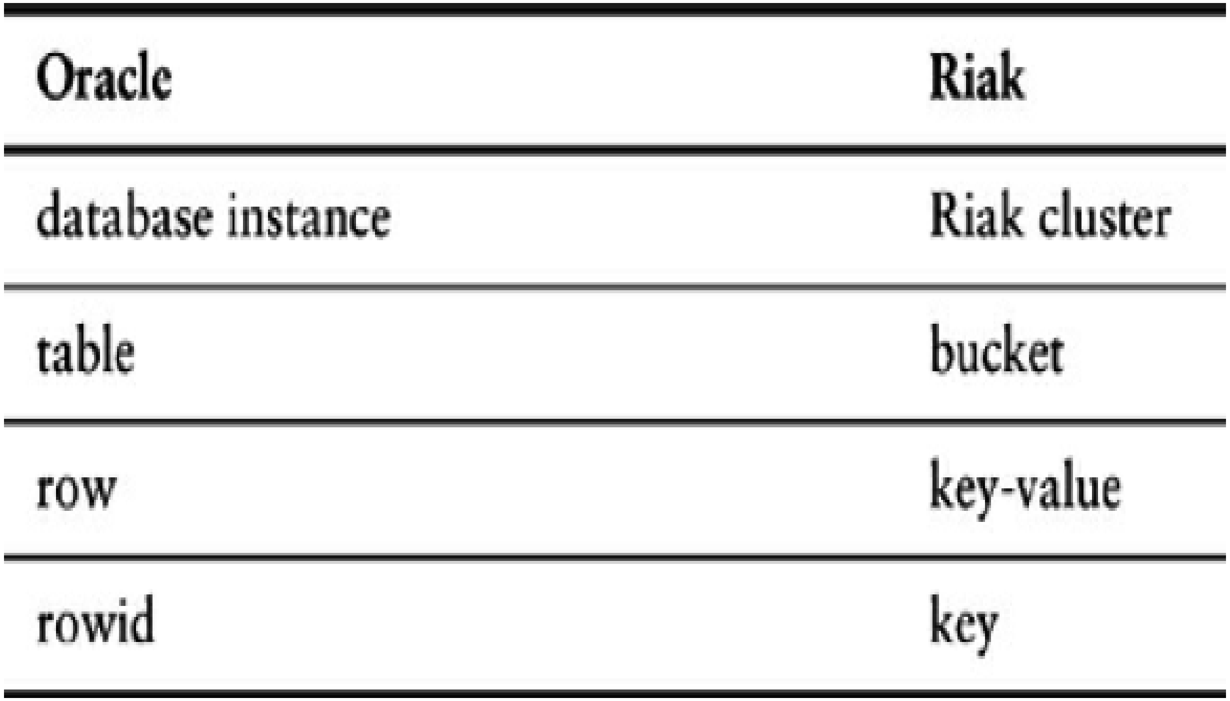
Differences Between Key-Value and Relational Databases



Customer Table

Primary Key ID	Name	Address
123		

RIAK Database



Oracle	Riak
database instance	Riak cluster
table	bucket
row	key-value
rowid	key

What is RIAK?

- Key Value store
- Distributed and horizontal Scalable
- Fault Tolerant
- Highly available
- Built for web
- Based on Amazon Dynamo

What is RIAK?

- Riak is a highly resilient NoSQL database.
- It ensures your most critical data is always available and that your Big Data applications can scale.
- Riak KV can be operationalized at lower costs than both relational and other NoSQL databases, especially at scale.
- Running on commodity hardware (inexpensive, widely available and basically interchangeable with other hardware of its type)
- Riak KV stores data as a combination of keys and values, and is a fundamentally content-agnostic database.
- You can use it to store anything you want – JSON, XML, HTML, documents, binaries, images, and more.
- Keys are simply binary values used to uniquely identify a value.

HORIZONTALLY SCALABLE

- Default configuration is optimized for a cluster
- Query load and data are spread evenly
- Add more nodes and get more:
  - operations/second
  - storage capacity
  - compute power (for Map/Reduce)

FAULT TOLERANT

- All nodes participate equally - no single point of failure (SPOF)
- All data is replicated
- Cluster transparently survives...
  - node failure
  - network partitions

HIGHLY AVAILABLE

- Any node can serve client requests
- Fall backs are used when nodes are down
- Always accepts read and write requests
- Per-request quorums

CORE CONCEPTS in Riak

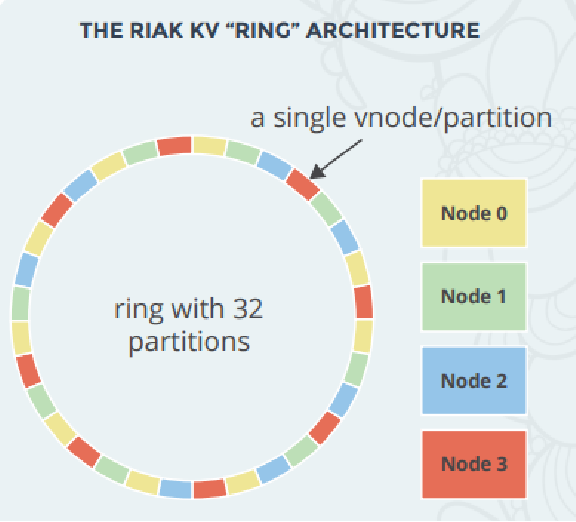
- Node - An instance of Riak
- Cluster – A collection of connected Riak nodes
- Bucket - Logical grouping of objects. Shared configuration
- Key - An identifier for a record/object
- Value - Opaque binary representation of data stored with key
- Metadata - Additional data linked to record, not part of value
- Riak Object - Bucket, Key, Value and Metadata. Unit of replication.

CORE CONCEPTS in Riak

- Partition - Logical division of storage
- Vnode - Process handling requests and managing a partition
- Ownership Handoff - Transfer of data on cluster change
- Hinted Handoff - Transfer of data on node/network failure
- Quorum - Set of nodes required to participate in transaction

THE RIAK RING

- The basis of Riak KV's masterless architecture, replication, and fault tolerance is the Ring.
- This Ring is a managed collection of partitions that share a common hash space.
- The hash space is called a Ring because the last value in the hash space is thought of as being adjacent to the first value in the space.
- Replicas of data are stored in the "next N partitions" of the hash spaces.



NODES AND VNODES

- Each node in a Riak KV cluster manages one or many virtual nodes, or vnodes.
- Each vnode is a separate process which is assigned a partition of the Ring, and is responsible for a number of operations in a Riak cluster, from the storing of objects, to handling of incoming read/write requests from other vnodes, to interpreting causal context metadata for objects.
- If your cluster has 64 partitions and you are running three nodes, two of your nodes will have 21 vnodes, while the third node holds 22 vnodes.
- The concept of vnodes is important as we look at data replication.
- No single vnode is responsible for more than one replica of an object.
- Each object belongs to a primary vnode, and is then replicated to neighboring vnodes located on separate nodes in the cluster.

CAP Theorem - AP

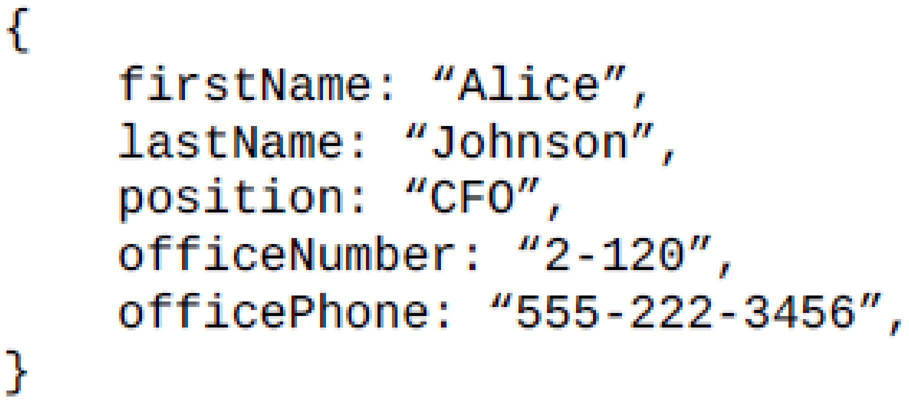
- Riak KV is a tunable AP system.
- By default, Riak KV replicas are “eventually consistent,” meaning that while data is always available, not all replicas may have the most recent update at the exact same time, causing brief periods—generally on the order of milliseconds—of inconsistency while all state changes are synchronized.
- Riak KV is designed to deliver maximum data availability, so as long as your client can reach one Riak KV server, it can write data.

Document Databases

- Document databases, also called document-oriented databases.
- It uses a key-value approach to storing data but with important differences from key-value databases. **A document database stores values as documents**.
- Documents are semi-structured entities, typically in a standard format such as JavaScript Object Notation (JSON) or Extensible Markup Language (XML).
- when the term ***document*** *is used in this* context, it does not refer to word processing or other office productivity files. It refers to data structures that are stored as strings or binar representations of strings.

Document Databases

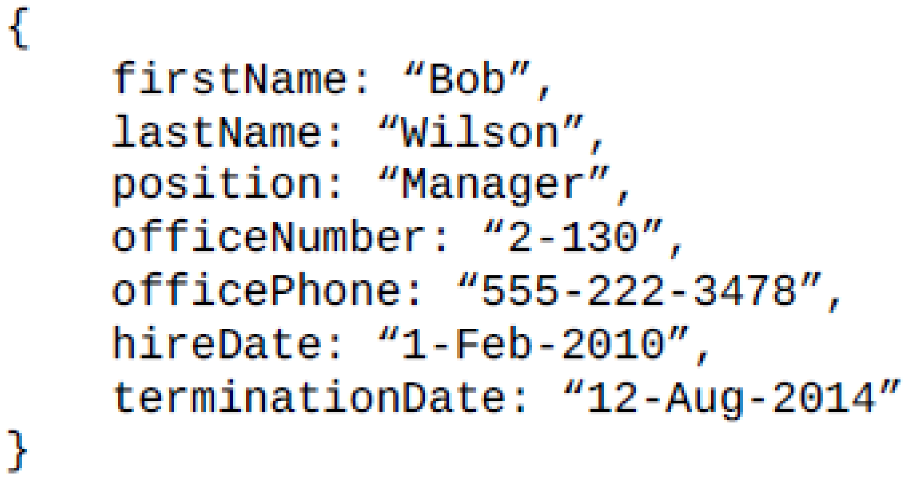
- **Documents**
- Instead of storing each attribute of an entity with a separate key, document databases store multiple attributes in a single document.
- One of the most important characteristics of document databases is you do not have to define a fixed schema before you add data to the database.



```
{
    firstName: “Alice”,
    lastName: “Johnson”,
    position: “CFO”,
    officeNumber: “2-120”,
    officePhone: “555-222-3456”,
}
```

Document Databases

- Simply adding a document to the database creates the underlying data structures needed to support the document.
- The lack of a fixed schema gives developers more flexibility with document databases than they have with relational databases.
- For example, employees can have different attributes than the ones listed above. Another valid employee document is



```
{
    firstName: “Bob”,
    lastName: “Wilson”,
    position: “Manager”,
    officeNumber: “2-130”,
    officePhone: “555-222-3478”,
    hireDate: “1-Feb-2010”,
    terminationDate: “12-Aug-2014”
}
```

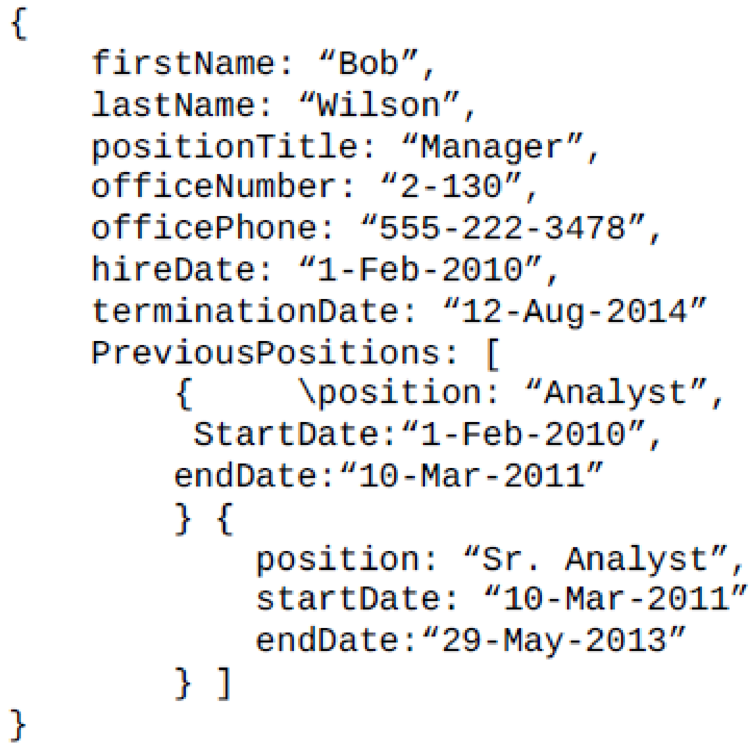
Document Databases

- **Querying Documents**
- Document databases provide application programming interfaces (APIs) or query languages that enable you to retrieve documents based on attribute values.
- For example, if you have a database with a collection of employee documents called “employees,” you could use a statement such as the following to return the set of all employees with the position Manager:

```
db.employees.find( { position:“Manager” })
```

Differences Between Document and Relational Databases

- A key distinction between document and relational databases is that document databases do not require a fixed, predefined schema.
- Another important difference is that documents can have embedded documents and lists of multiple values within a document.



```
{
    firstName: “Bob”,
    lastName: “Wilson”,
    positionTitle: “Manager”,
    officeNumber: “2-130”,
    officePhone: “555-222-3478”,
    hireDate: “1-Feb-2010”,
    terminationDate: “12-Aug-2014”
    PreviousPositions: [
        {     \position: “Analyst”,
         StartDate:“1-Feb-2010”,
        endDate:“10-Mar-2011”
        } {
            position: “Sr. Analyst”,
            startDate: “10-Mar-2011”
            endDate:“29-May-2013”
        } ]
}
```

Differences Between Document and Relational Databases

- Document databases are probably the most popular type of NoSQL database. They offer support for querying structures with multiple attributes, like relational databases, but offer more flexibility with regard to variation in the attributes used by each document.

Difference between RDBMS and MongoDB

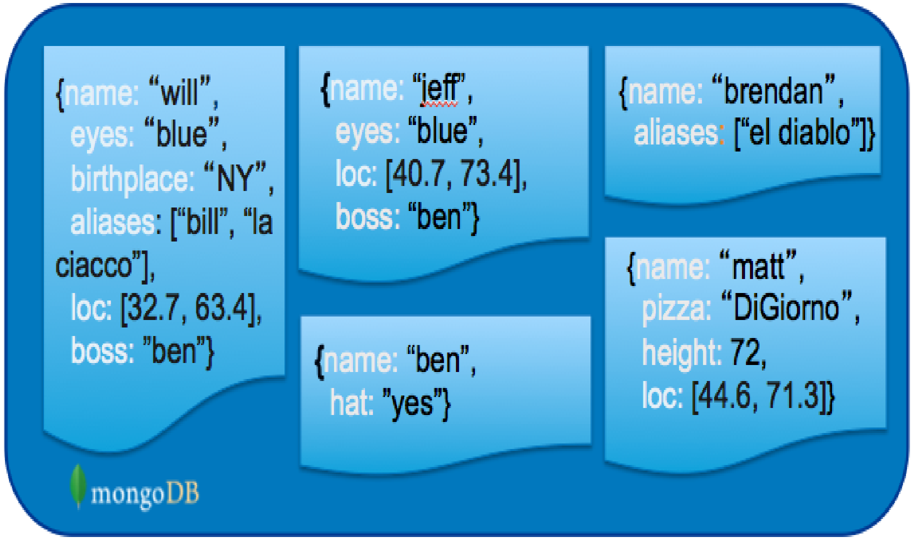
RDBMS	MongoDB
It is a relational database.	It is a non-relational and document-oriented database.
Not suitable for hierarchical data storage.	Suitable for hierarchical data storage.
It is vertically scalable i.e increasing RAM.	It is horizontally scalable i.e we can add more servers.
It has a predefined schema.	It has a dynamic schema.
It is quite vulnerable to SQL injection.	It is not affected by SQL injection.
It centers around ACID properties (Atomicity, Consistency, Isolation, and Durability).	It centers around the CAP theorem (Consistency, Availability, and Partition tolerance).
It is row-based.	It is document-based.
It is slower in comparison with MongoDB.	It is almost 100 times faster than RDBMS.
Supports complex joins.	No support for complex joins.
It is column-based.	It is field-based.
It does not provide JavaScript client for querying.	It provides a JavaScript client for querying.
It supports SQL query language only.	It supports JSON query language along with SQL.

What is MongoDB?

- **Defination:** MongoDB is an **open source**, **document-oriented** database designed with both scalability and developer agility in mind.
- Instead of storing your data in tables and rows as you would with a relational database, in MongoDB you store **JSON-like documents** with **dynamic schemas (schema-free, schemaless)**.

What is MongoDB? (Cont'd)

- **Document-Oriented DB**
  - Unit object is a document instead of a row (tuple) in relational DBs

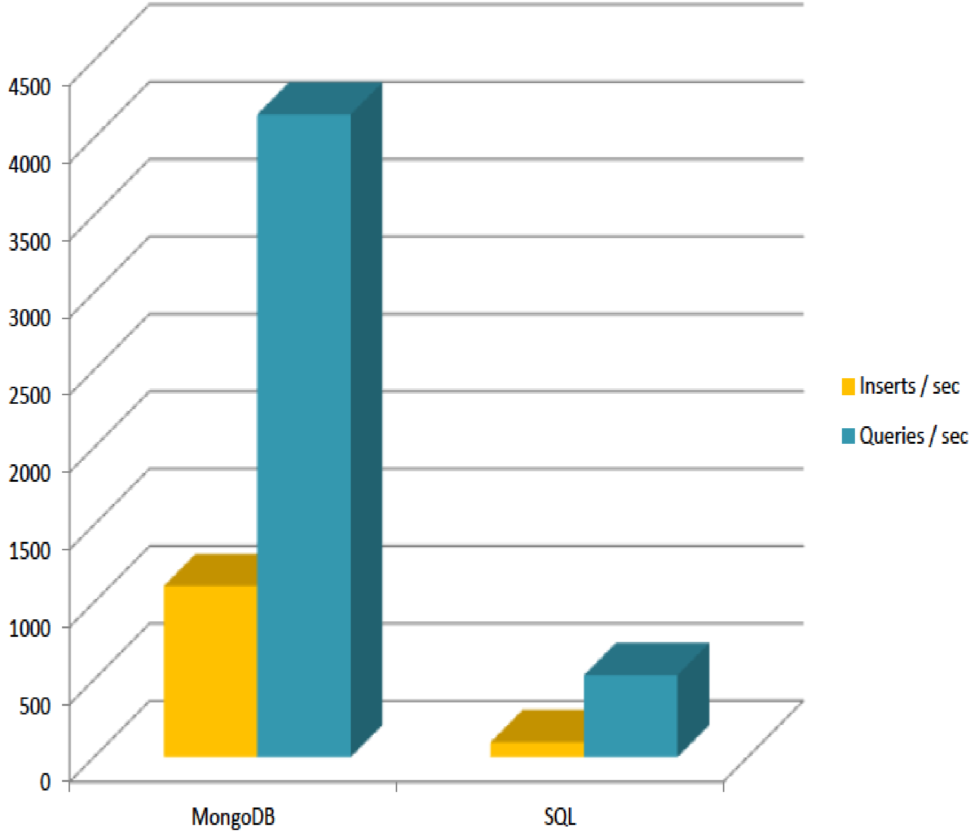


```
{name: "will",
 eyes: "blue",
 birthplace: "NY",
 aliases: ["bill", "la ciacco"],
 loc: [32.7, 63.4],
 boss: "ben"}

{name: "jeff",
 eyes: "blue",
 loc: [40.7, 73.4],
 boss: "ben"}

{name: "ben",
 hat: "yes"}

{name: "brendan",
 aliases: ["el diablo"]}

{name: "matt",
 pizza: "DiGiorno",
 height: 72,
 loc: [44.6, 71.3]}
```

```
> db.user.findOne({age:39})
{
    "_id" : ObjectId("5114e0bd42…"),
    "first" : "John",
    "last" : "Doe",
    "age" : 39,
    "interests" : [
        "Reading",
        "Mountain Biking ]
    "favorites": {
        "color": "Blue",
        "sport": "Soccer"}
}
```

Is It Fast?

- For semi-structured & complex relationships: Yes



Integration with Others

- C
- C++
- Erlang
- Haskell
- Java
- Javascript
- .NET (C# F#, PowerShell, etc)
- Node.js
- Perl
- PHP
- Python
- Ruby
- Scala

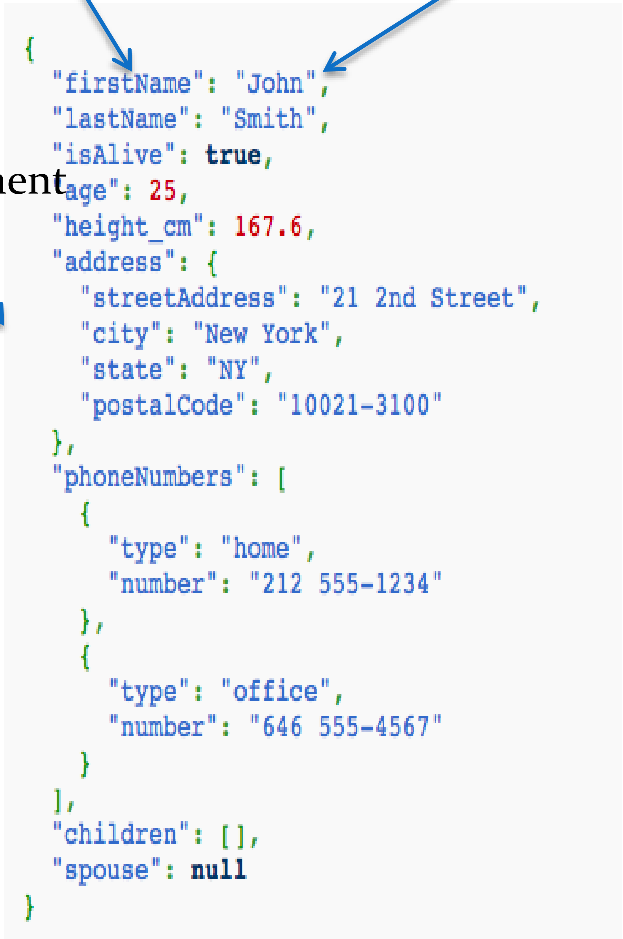
http://www.mongodb.org/display/DOCS/Drivers

JSON

Field Name

Field Value

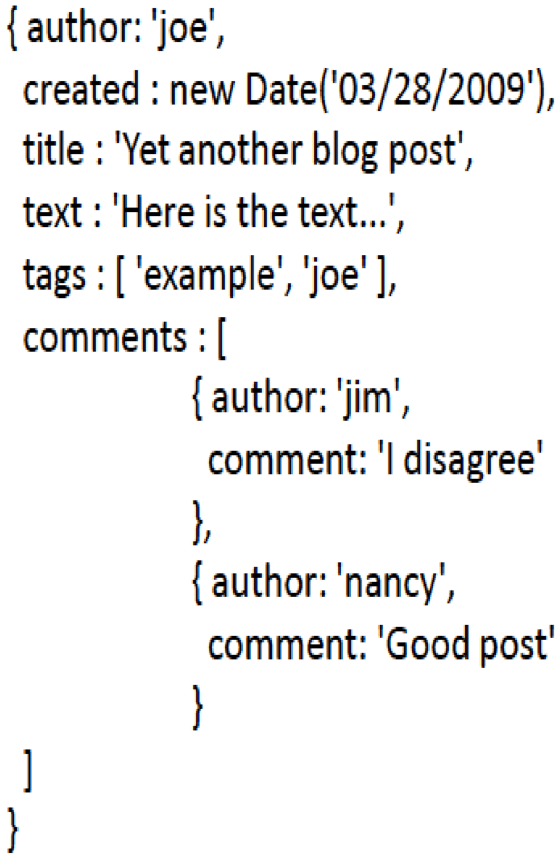
One document



```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "height_cm": 167.6,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

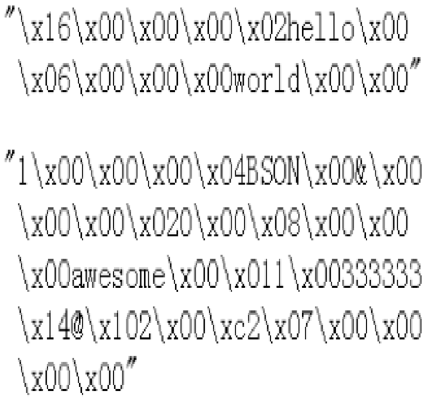
- ***Field Value***
  - Scalar (Int, Boolean, String, Date, …)
  - Document (Embedding or Nesting)
  - Array of JSON objects

Another Example



```
{ author: 'joe',
  created : new Date('03/28/2009'),
  title : 'Yet another blog post',
  text : 'Here is the text...',
  tags : [ 'example', 'joe' ],
  comments : [
              { author: 'jim',
                comment: 'I disagree'
              },
              { author: 'nancy',
                comment: 'Good post'
              }
  ]
}
```

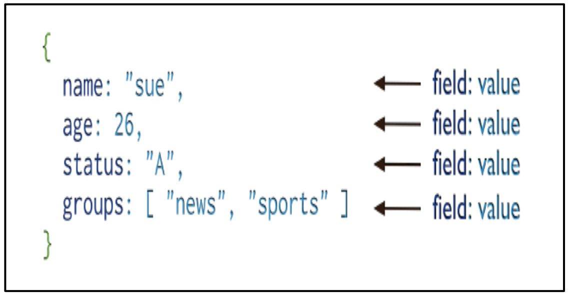
Remember it is stored in binary formats (BSON)



```
"\x16\x00\x00\x00\x02hello\x00
 \x06\x00\x00\x00world\x00\x00"

"1\x00\x00\x00\x04BSON\x00&\x00
 \x00\x00\x020\x00\x08\x00\x00
 \x00awesome\x00\x011\x00333333
 \x14@\x102\x00\xc2\x07\x00\x00
 \x00\x00"
```

MongoDB Model

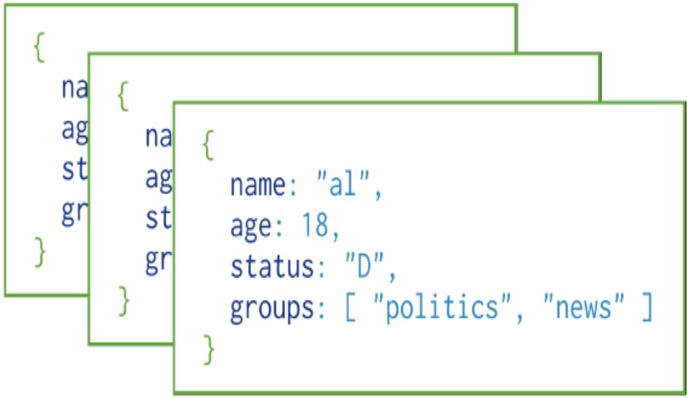
One ***document*** (e.g., one tuple in RDBMS)



```
{
  name: "sue",
  age: 26,
  status: "A",
  groups: [ "news", "sports" ]
}
```

- **Collection** is a group of similar documents
- Within a collection, each document must have a unique Id

One ***Collection*** (e.g., one Table in RDBMS)



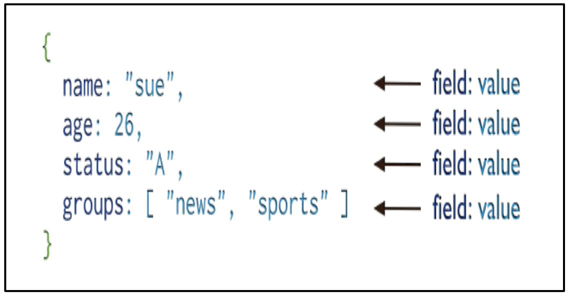
```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

Collection

Unlike RDBMS: No Integrity Constraints in MongoDB

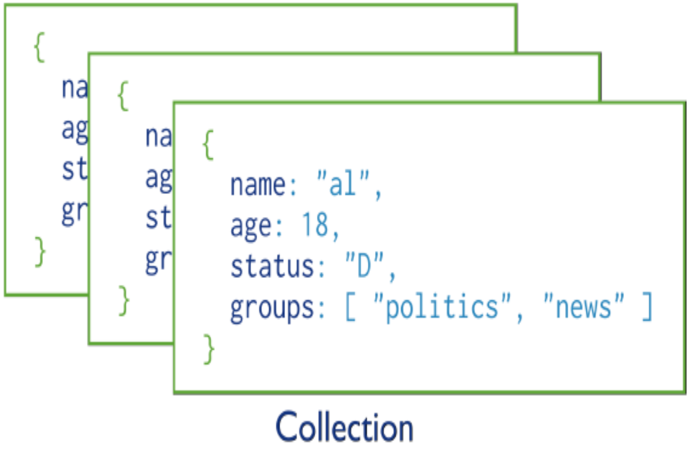
MongoDB Model

One ***document*** (e.g., one tuple in RDBMS)



```
{
  name: "sue",                         ← field: value
  age: 26,                             ← field: value
  status: "A",                         ← field: value
  groups: [ "news", "sports" ]         ← field: value
}
```

One ***Collection*** (e.g., one Table in RDBMS)

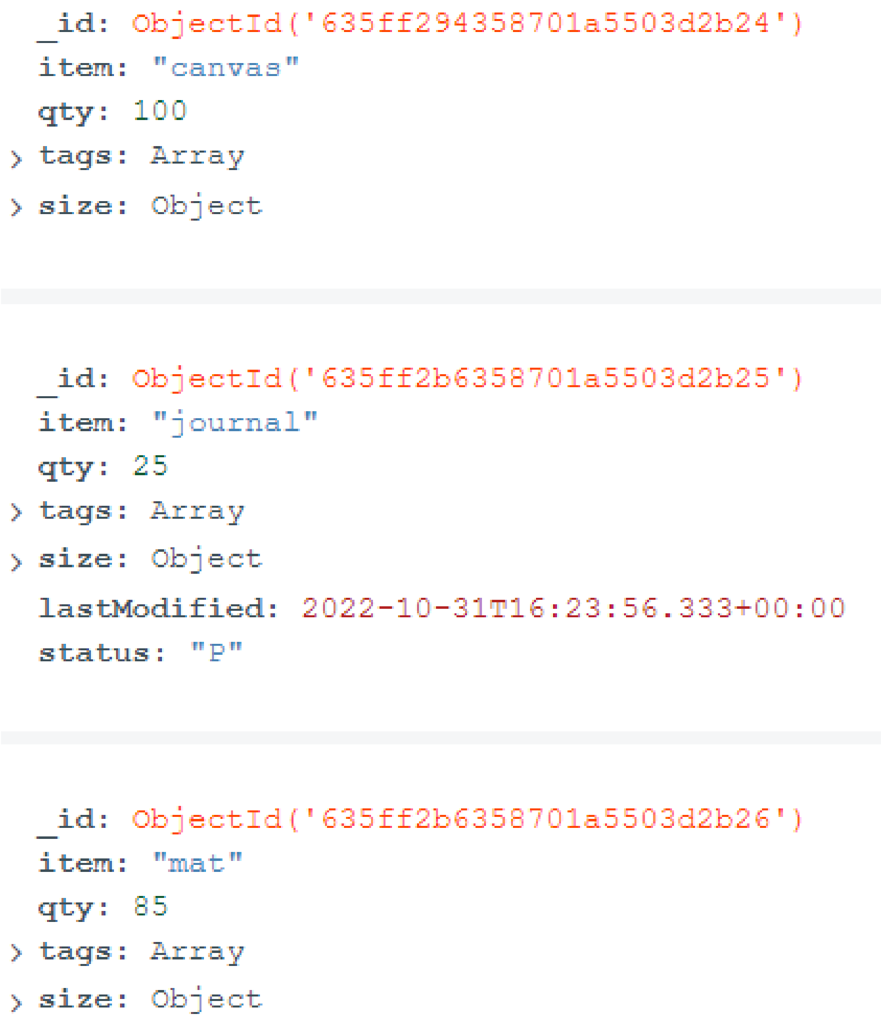


```
{
  name: "al",
  age: 18,
  status: "D",
  groups: [ "politics", "news" ]
}
```

- The field names **cannot** start with the **$** character
- The field names **cannot** contain the **.** character
- Max size of single document 16MB

Example Document in MongoDB

- _id is a special column in each document
- Unique within each collection
- _id ←→ Primary Key in RDBMS
- _id is 12 Bytes, you can set it yourself
- Or:
  - 1st 4 bytes → timestamp
  - Next 3 bytes → machine id
  - Next 2 bytes → Process id
  - Last 3 bytes → incremental values

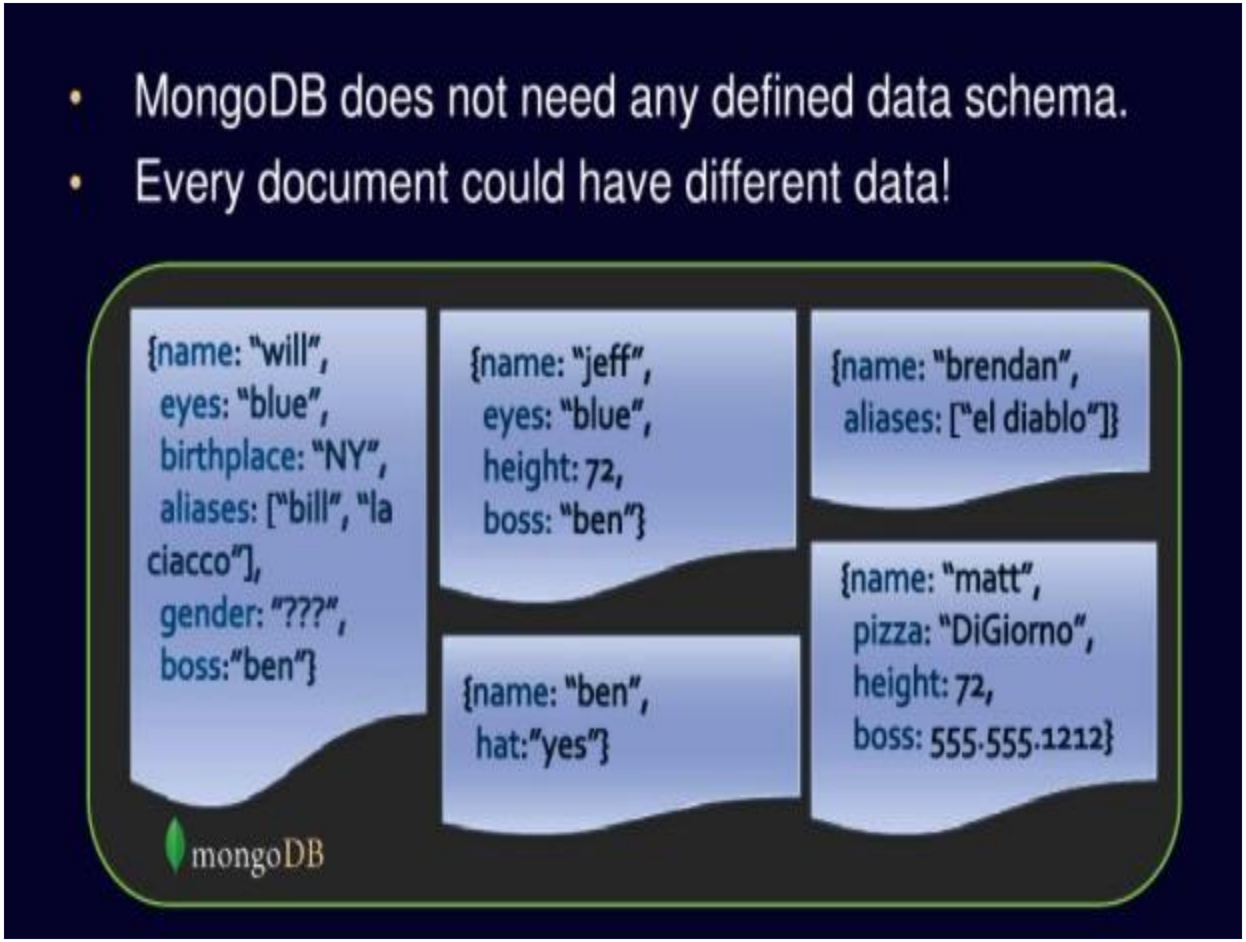


```
  _id: ObjectId('635ff294358701a5503d2b24')
  item: "canvas"
  qty: 100
› tags: Array
› size: Object
```

```
  _id: ObjectId('635ff2b6358701a5503d2b25')
  item: "journal"
  qty: 25
› tags: Array
› size: Object
  lastModified: 2022-10-31T16:23:56.333+00:00
  status: "P"
```

```
  _id: ObjectId('635ff2b6358701a5503d2b26')
  item: "mat"
  qty: 85
› tags: Array
› size: Object
```

No Defined Schema
(Schema-free Or Schema-less)



- MongoDB does not need any defined data schema.
- Every document could have different data!

```
{name: “will”,
  eyes: “blue”,
  birthplace: “NY”,
  aliases: [“bill”, “la
ciacco”],
  gender: ”???”,
  boss:”ben”}
```

```
{name: “jeff”,
  eyes: “blue”,
  height: 72,
  boss: “ben”}
```

```
{name: “brendan”,
  aliases: [“el diablo”]}
```

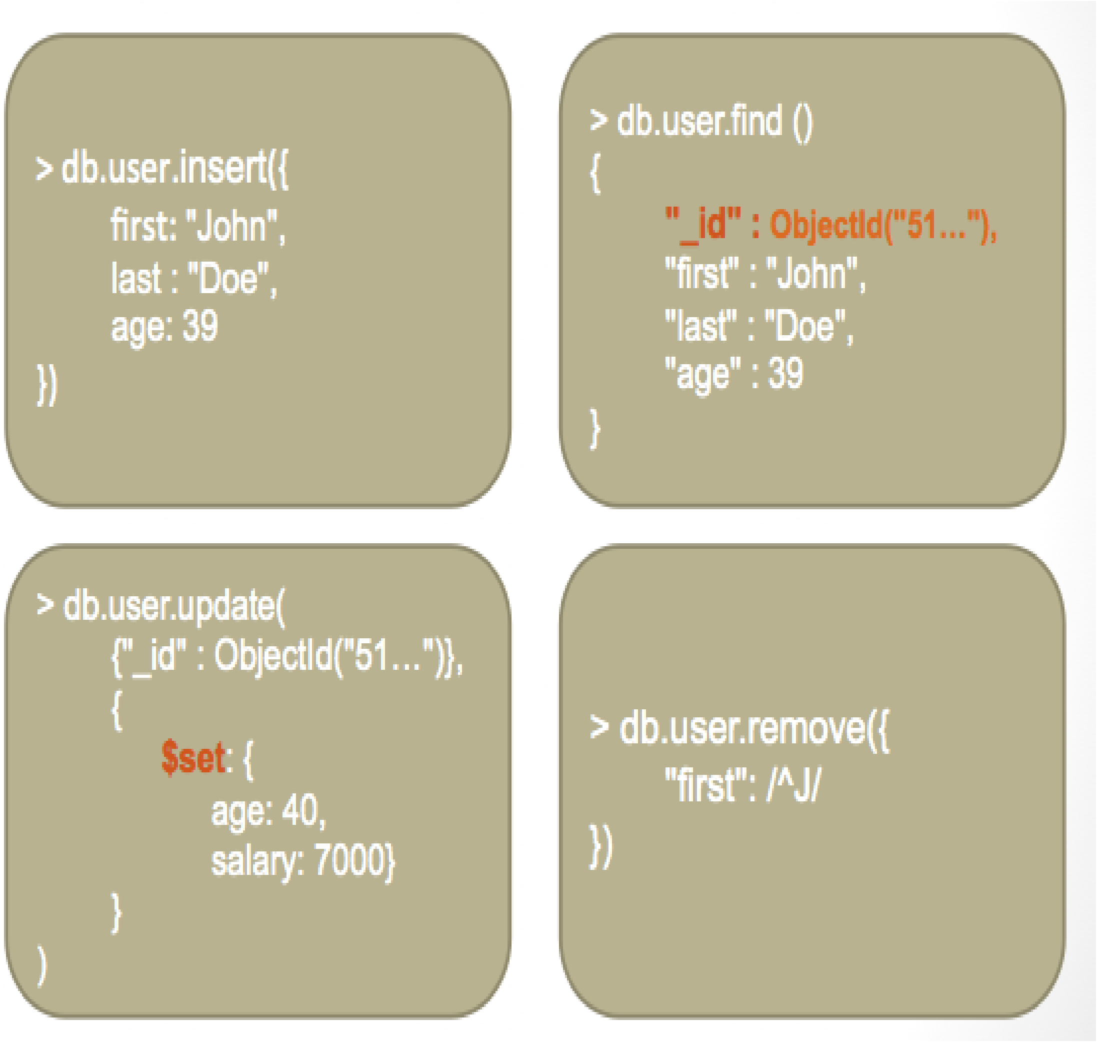
```
{name: “ben”,
 hat:”yes”}
```

```
{name: “matt”,
  pizza: “DiGiorno”,
  height: 72,
  boss: 555.555.1212}
```

CRUD

- Create
  - db.collection.insert(<document>)
  - db.collection.save(<document>)
  - db.collection.update(<query>, <update>, { upsert: true })
- Read
  - db.collection.find(<query>, <projection>)
  - db.collection.findOne(<query>, <projection>)
- Update
  - db.collection.update(<query>, <update>, <options>)
- Delete
  - db.collection.remove(<query>, <justOne>)

CRUD Examples



```
> db.user.insert({
        first: "John",
        last : "Doe",
        age: 39
})
```

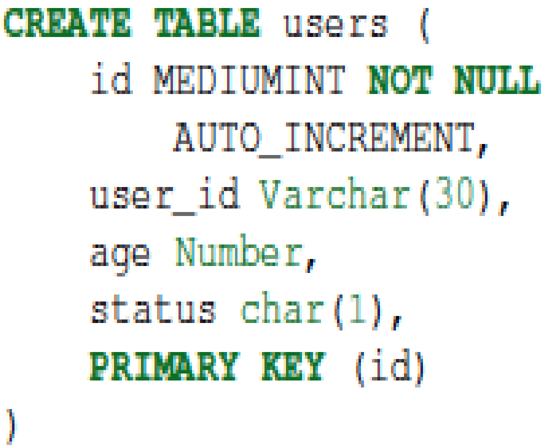
```
> db.user.find ()
{
        "_id" : ObjectId("51..."),
        "first" : "John",
        "last" : "Doe",
        "age" : 39
}
```

```
> db.user.update(
        {"_id" : ObjectId("51...")},
        {
            $set: {
                age: 40,
                salary: 7000}
        }
)
```

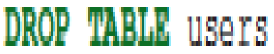
```
> db.user.remove({
        "first": /^J/
})
```

Examples

In RDBMS



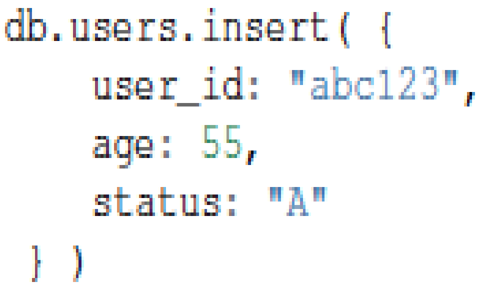
```
CREATE TABLE users (
    id MEDIUMINT NOT NULL
        AUTO_INCREMENT,
    user_id Varchar(30),
    age Number,
    status char(1),
    PRIMARY KEY (id)
)
```



```
DROP TABLE users
```

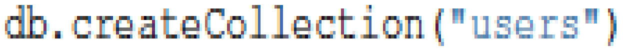
In MongoDB

Either insert the 1st docuement

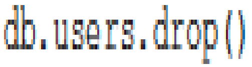


```
db.users.insert( {
    user_id: "abc123",
    age: 55,
    status: "A"
 } )
```

Or create "Users" collection explicitly

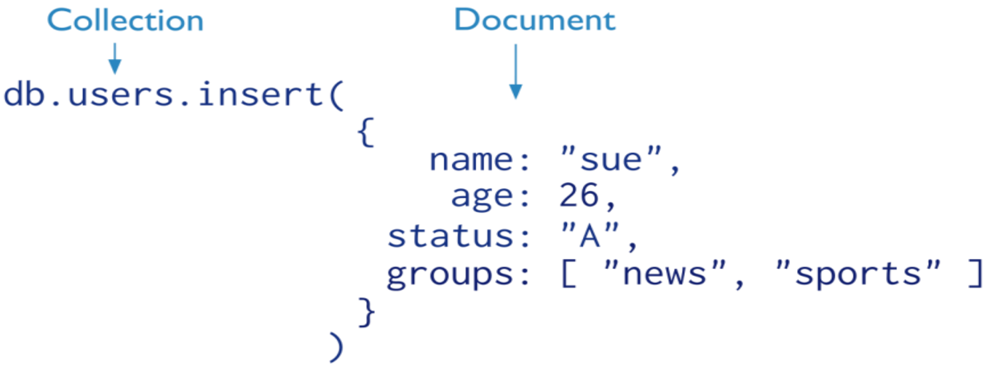


```
db.createCollection("users")
```

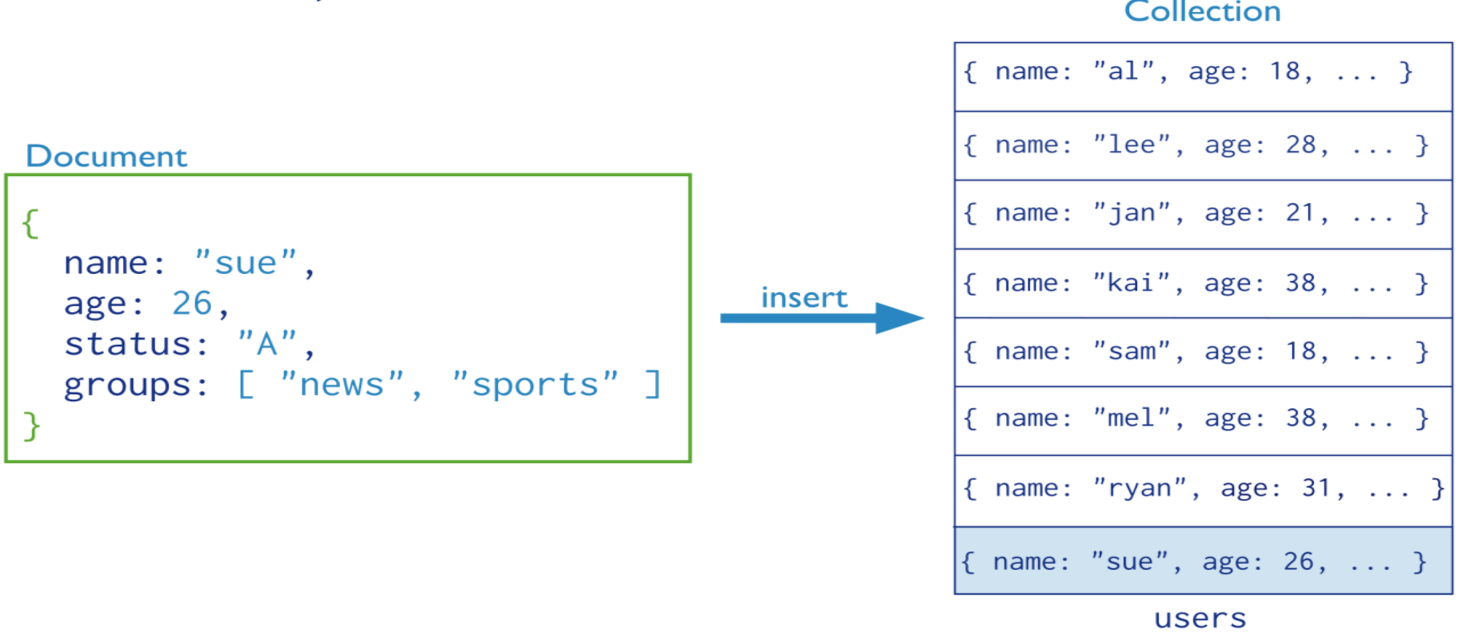


```
db.users.drop()
```

Insertion



```
db.users.insert(
                {
                  name: "sue",
                   age: 26,
                status: "A",
                groups: [ "news", "sports" ]
                }
               )
```

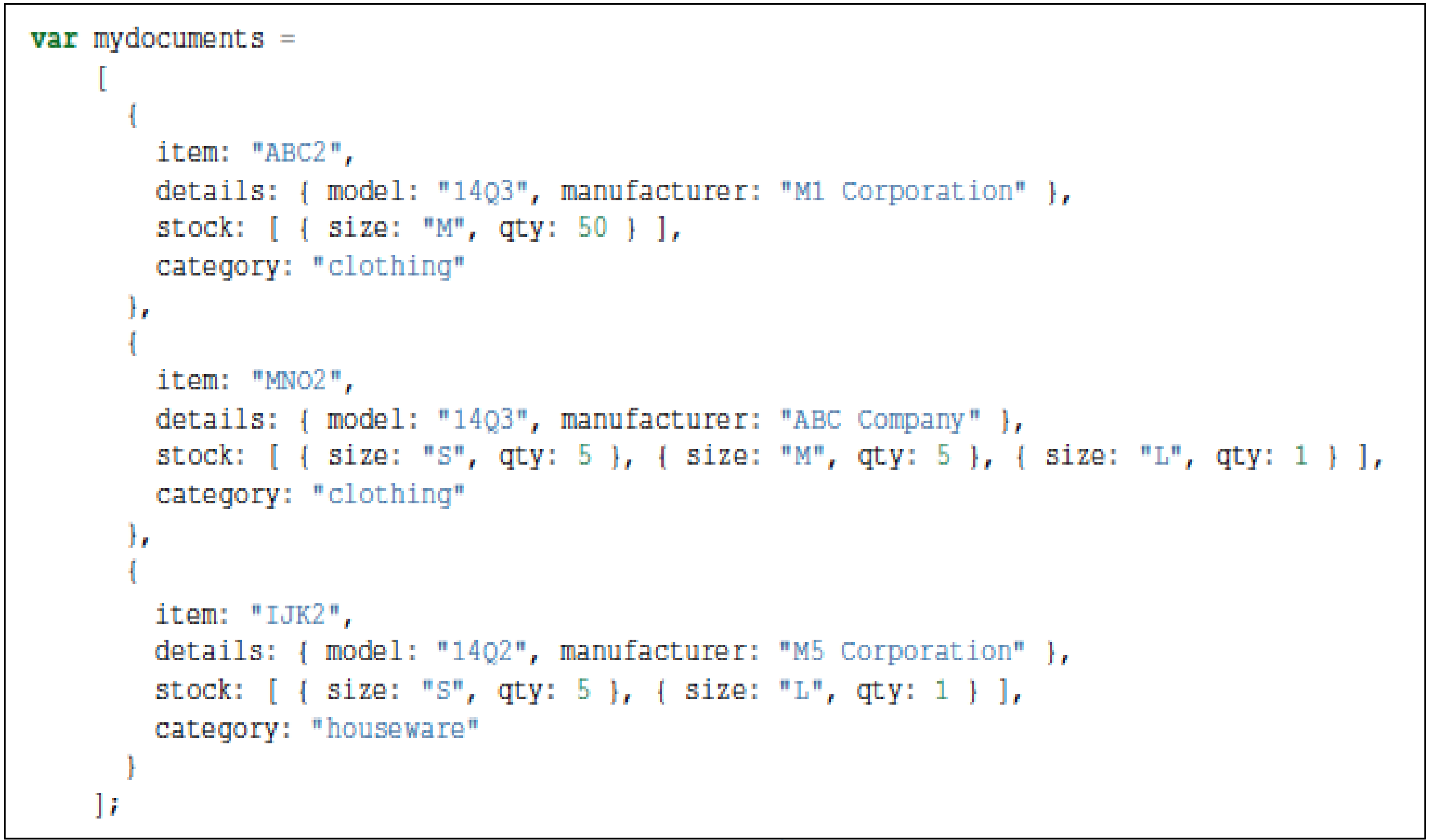


```
{
   name: "sue",
   age: 26,
   status: "A",
   groups: [ "news", "sports" ]
}
```

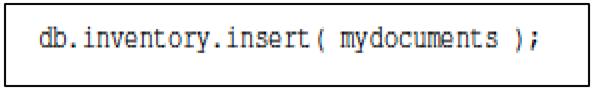
```
{ name: "al", age: 18, ... }
{ name: "lee", age: 28, ... }
{ name: "jan", age: 21, ... }
{ name: "kai", age: 38, ... }
{ name: "sam", age: 18, ... }
{ name: "mel", age: 38, ... }
{ name: "ryan", age: 31, ... }
{ name: "sue", age: 26, ... }
```

- The collection “users” is created automatically if it does not exist

Multi-Document Insertion
(Use of Arrays)



```
var mydocuments =
    [
      {
        item: "ABC2",
        details: { model: "14Q3", manufacturer: "M1 Corporation" },
        stock: [ { size: "M", qty: 50 } ],
        category: "clothing"
      },
      {
        item: "MNO2",
        details: { model: "14Q3", manufacturer: "ABC Company" },
        stock: [ { size: "S", qty: 5 }, { size: "M", qty: 5 }, { size: "L", qty: 1 } ],
        category: "clothing"
      },
      {
        item: "IJK2",
        details: { model: "14Q2", manufacturer: "M5 Corporation" },
        stock: [ { size: "S", qty: 5 }, { size: "L", qty: 1 } ],
        category: "houseware"
      }
    ];
```

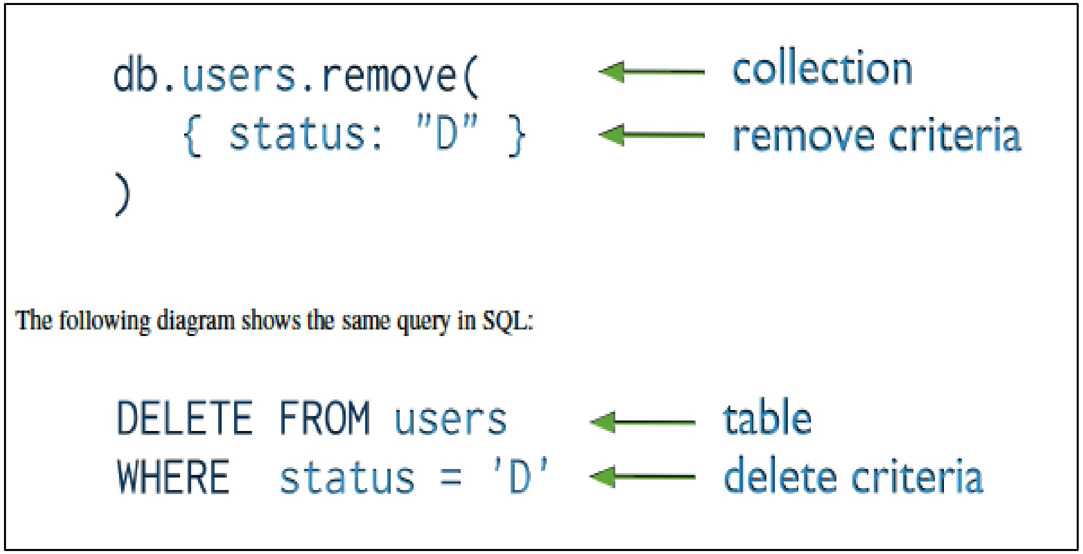


```
db.inventory.insert( mydocuments );
```

All the documents are inserted at once

Deletion
(Remove Operation)

- You can put condition on any field in the document (even ***_id***)



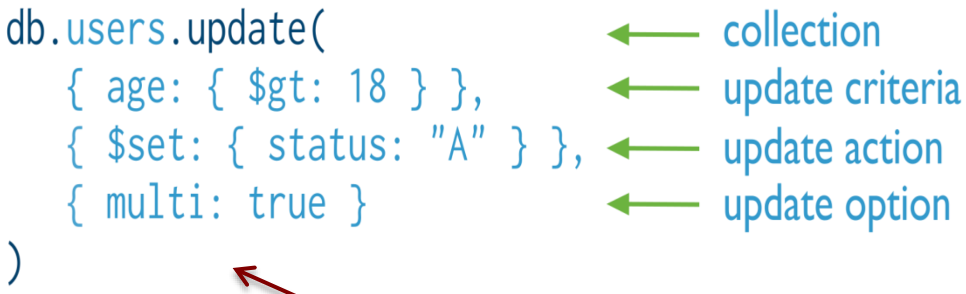
```
db.users.remove(
   { status: "D" }
)
```

The following diagram shows the same query in SQL:

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

db.users.remove ( ) → Removes all documents from *users* collection

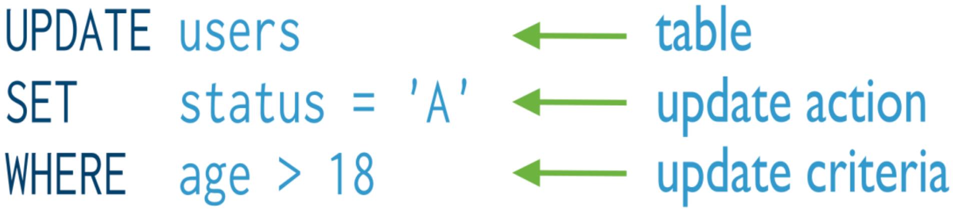
Update



```
db.users.update(
   { age: { $gt: 18 } },
   { $set: { status: "A" } },
   { multi: true }
)
```

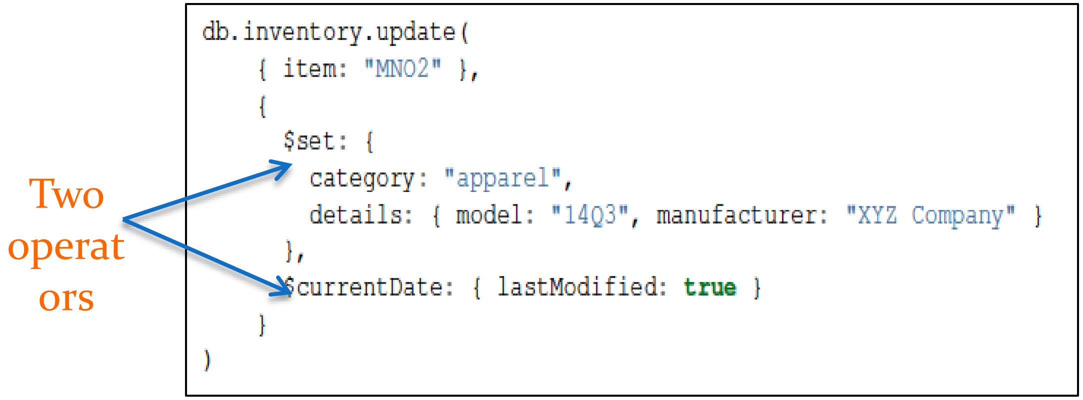
Otherwise, it will update only the 1st matching document

Equivalent to in SQL:



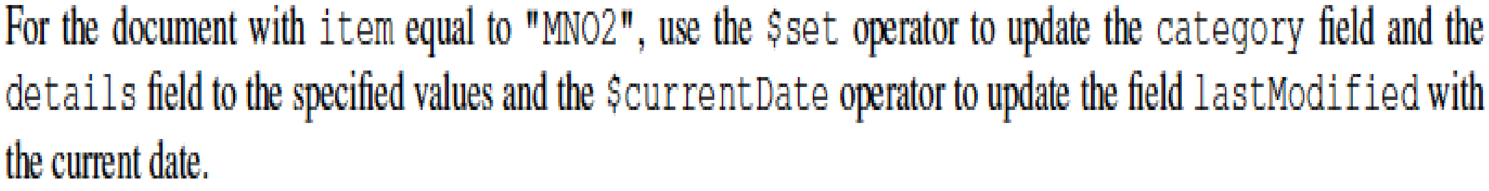
```
UPDATE users
SET    status = 'A'
WHERE  age > 18
```

Update (Cont’d)



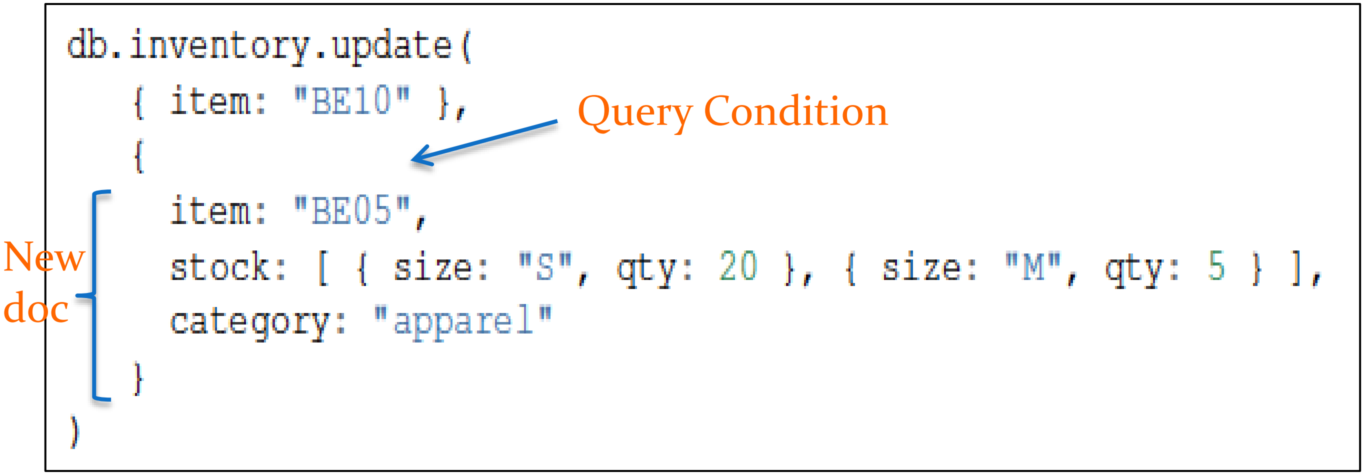
Two operators

```
db.inventory.update(
    { item: "MNO2" },
    {
      $set: {
        category: "apparel",
        details: { model: "14Q3", manufacturer: "XYZ Company" }
      },
      $currentDate: { lastModified: true }
    }
)
```



For the document with item equal to "MNO2", use the $set operator to update the category field and the details field to the specified values and the $currentDate operator to update the field lastModified with the current date.

Replace a document



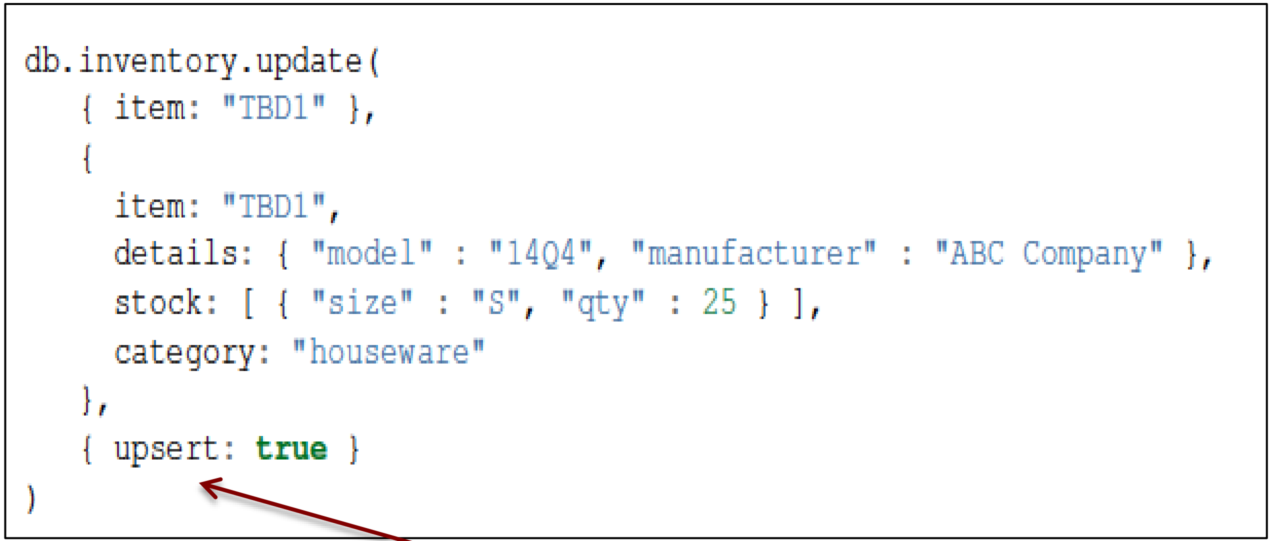
```
db.inventory.update(
   { item: "BE10" },
   {
     item: "BE05",
     stock: [ { size: "S", qty: 20 }, { size: "M", qty: 5 } ],
     category: "apparel"
   }
)
```

Query Condition

New doc

For the document having item = “BE10”, replace it with the given document

Insert or Replace



```
db.inventory.update(
   { item: "TBD1" },
   {
     item: "TBD1",
     details: { "model" : "14Q4", "manufacturer" : "ABC Company" },
     stock: [ { "size" : "S", "qty" : 25 } ],
     category: "houseware"
   },
   { upsert: true }
)
```

The ***upsert*** option

If the document having item = “TBD1” is in the DB, it will be replaced
Otherwise, it will be inserted.

Find queries

- SELECT * FROM inventory

  db.inventory.find{}
- SELECT * FROM inventory WHERE status = "D"

  db.inventory.find{ status: "D" }
- SELECT * FROM inventory WHERE status in ("A", "D")

  db.inventory.find{ status: { $in: ["**A**", "**D**"] } }
- SELECT * FROM inventory WHERE status = "A" AND qty < 30

  db.inventory.find{ status: "**A**", qty: { $lt: 30 } }
- SELECT * FROM inventory WHERE status = "A" OR qty < 30

  db.inventory.find{ $or: [{ status: "**A**" }, { qty: { $lt: 30 } }] }
- SELECT * FROM inventory WHERE status = "A" AND (qty < 30 OR item LIKE "p%")

  db.inventory.find{ status: "**A**", $or: [{ qty: { $lt: 30 } }, { item: /^p/ }] }

Must Practice It

Install it

Practice simple stuff

Move to complex stuff

Install it from here: http://www.mongodb.org

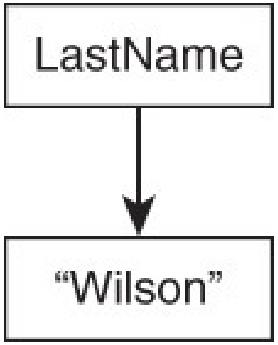
Manual: http://docs.mongodb.org/master/MongoDB-manual.pdf

Dataset: http://docs.mongodb.org/manual/reference/bios-example-collection/

Online Execution: https://docs.mongodb.com/manual/tutorial/insert-documents/

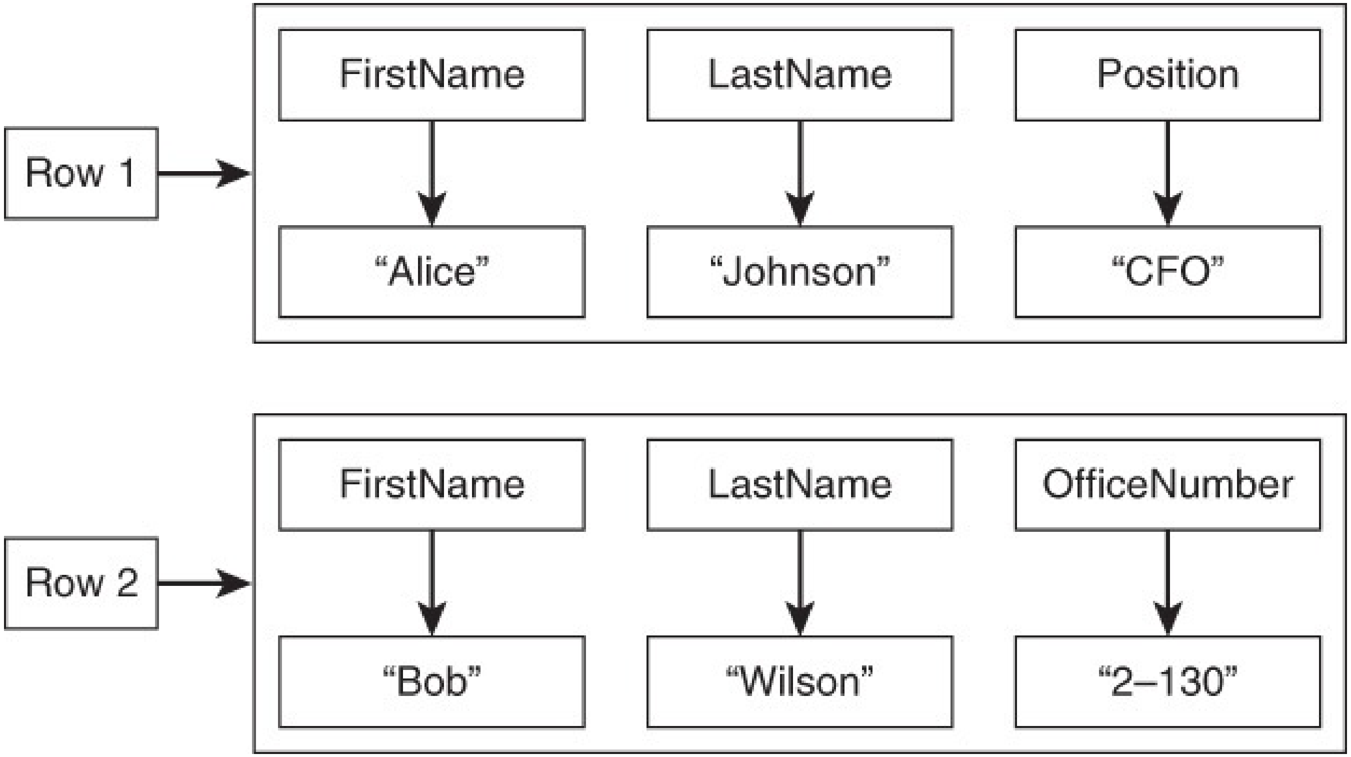
Column Family Databases

- Column family databases are perhaps the most complex of the NoSQL database types, at least in terms of the basic building block structures.
- Column family databases share some terms with relational databases, such as rows and columns.
- **Columns and Column Families**
- A column is a basic unit of storage in a column family database. A column is a name and a value.



Column Family Databases

- A set of columns makes up a row. Rows can have the same columns, or they can have different columns, as shown in Fig



Column Family Databases

- When there are large numbers of columns, it can help to group them into collections of related columns.
- For example, first and last name are often used together, and office numbers and office phone numbers are frequently needed together.
- These can be grouped in collections called **column families**.
- Column family databases do not require a predefined fixed schema.
- Developers can add columns as needed. Also, rows can have different sets of columns and super columns.
- Column family databases are designed for rows with many columns.

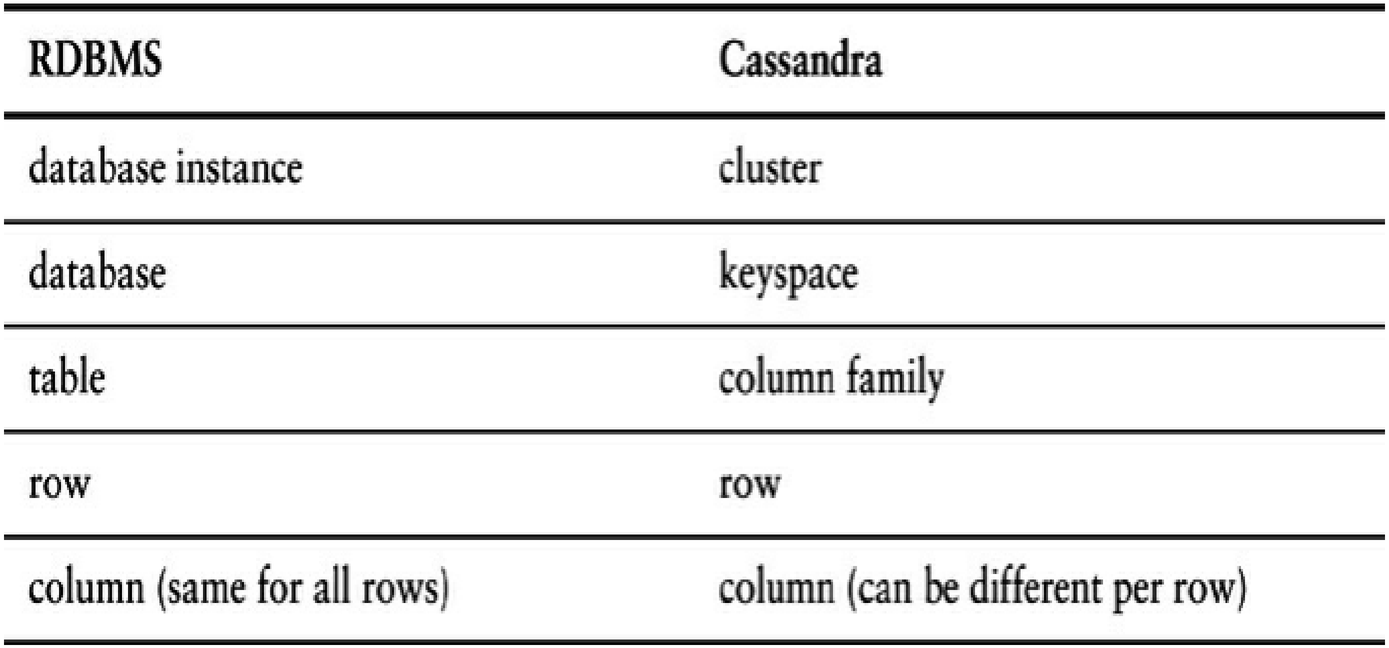
Column Family Databases

- Column-oriented databases work on columns and are based on BigTable paper by Google.
- Every column is treated separately.
- Values of single column databases are stored contiguously.
- They deliver high performance on aggregation queries like SUM, COUNT, AVG, MIN etc. as the data is readily available in a column.
- Column-based NoSQL databases are widely used to manage data warehouses, business intelligence, CRM, Library card catalogs,
- HBase, Cassandra, Hypertable are NoSQL query examples of column based database.
- Cassandra is more fast and scalable as compared to other column-family databases with write operations because data is spread across the cluster.

Differences Between Column Family and Relational Databases

- Column family databases and relational databases are superficially similar. They both make use of rows and columns.
- There are important differences in terms of data models and implementation details.
- One thing missing from column family databases is support for joining tables.
- Tables in relational databases have a relatively fixed structure, and the relational database management system can take advantage of that structure when optimizing the layout.
- Unlike a relational database table, the set of columns in a column family table can vary from one row to another.
- In relational databases, data about an object can be stored in multiple tables. Column family databases are typically denormalized, or structured so that all relevant information about an object is in a single, possibly very wide, row.

Column-Family Database - Cassandra



RDBMS	Cassandra
database instance	cluster
database	keyspace
table	column family
row	row
column (same for all rows)	column (can be different per row)

Cassandra

- Apache Cassandra is a highly scalable, high-performance distributed database designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.
- It is a type of NoSQL database.
- It is a column-oriented database.
- Its distribution design is based on Amazon's Dynamo and its data model on Google's Bigtable.
- Created at Facebook, it differs sharply from relational database management systems.
- Cassandra is being used by some of the biggest companies such as Facebook, Twitter, Cisco, Rackspace, eBay, Netflix, and more.

Cassandra

Features of Cassandra :

Cassandra has become so popular because of its outstanding technical features. Given below are some of the features of Cassandra:

- **Elastic scalability** – Cassandra is highly scalable; it allows to add more hardware to accommodate more customers and more data as per requirement.
- **Always on architecture** – Cassandra has no single point of failure and it is continuously available for business-critical applications that cannot afford a failure.
- **Fast linear-scale performance** – Cassandra is linearly scalable, i.e., it increases your throughput as you increase the number of nodes in the cluster. Therefore it maintains a quick response time.

Cassandra

- **Flexible data storage** – Cassandra accommodates all possible data formats including: structured, semi-structured, and unstructured. It can dynamically accommodate changes to your data structures according to your need.
- **Easy data distribution** – Cassandra provides the flexibility to distribute data where you need by replicating data across multiple data centers.
- **Transaction support** – Cassandra supports properties like Atomicity, Consistency, Isolation, and Durability (ACID).
- **Fast writes** – Cassandra was designed to run on cheap commodity hardware. It performs blazingly fast writes and can store hundreds of terabytes of data, without sacrificing the read efficiency.

Cassandra

History of Cassandra :

- Cassandra was developed at Facebook for inbox search.
- It was open-sourced by Facebook in July 2008.
- Cassandra was accepted into Apache Incubator in March 2009.
- It was made an Apache top-level project since February 2010.

Cassandra - Architecture

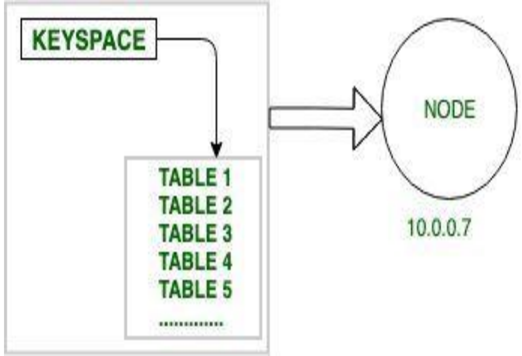
Components of Cassandra :

- **Basic Terminology:**
  - Node
  - Data center
  - Cluster
- **Operations:**
  - Read Operation
  - Write Operation
- **Storage Engine:**
  - CommitLog
  - Memtables
  - SSTables
- **Data Replication Strategies**
  - Simple Replication Strategy
  - Network Topology Strategy

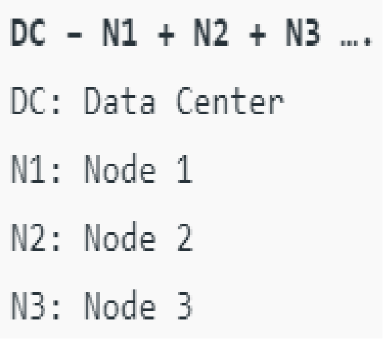
Cassandra - Architecture

Basic Terminology:–

- **Node** – Node is the basic component in Apache Cassandra. It is the place where actually data is stored. For Example: As shown in diagram node which has IP address 10.0.0.7 contain data (keyspace which contain one or more tables)..



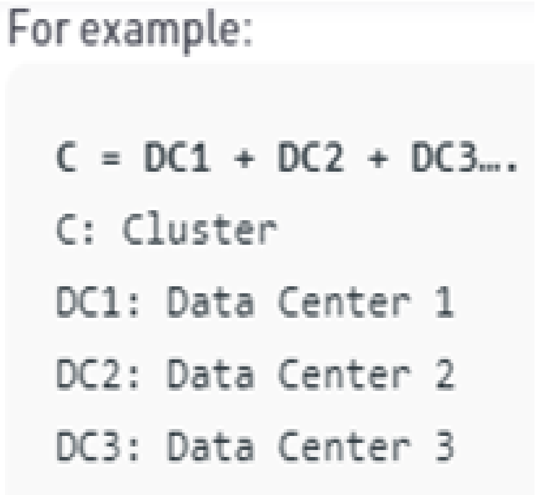
- **Data center** – It is a collection of related nodes.



```
DC - N1 + N2 + N3 ….
DC: Data Center
N1: Node 1
N2: Node 2
N3: Node 3
```

Cassandra - Architecture

- **Cluster** – A cluster is a component that contains one or more data centers.



For example:

```
C = DC1 + DC2 + DC3….
C: Cluster
DC1: Data Center 1
DC2: Data Center 2
DC3: Data Center 3
```

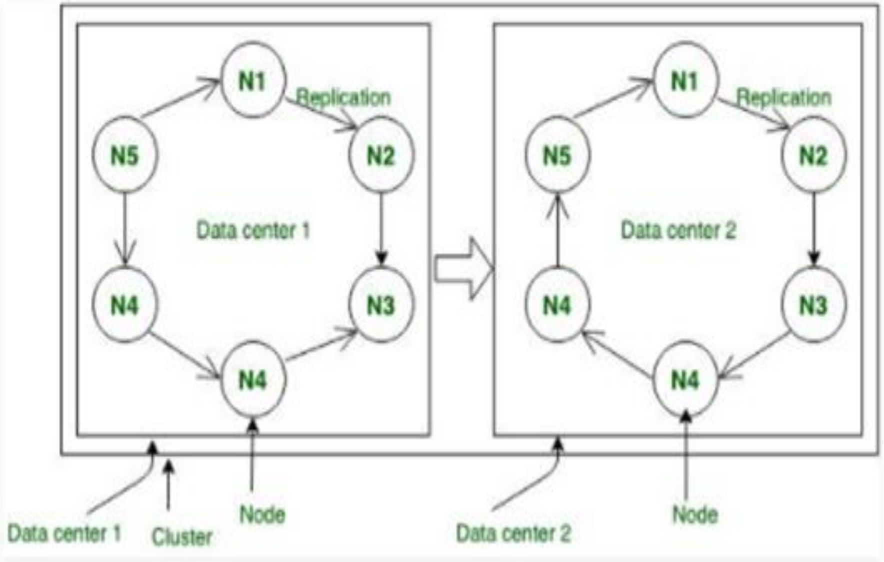
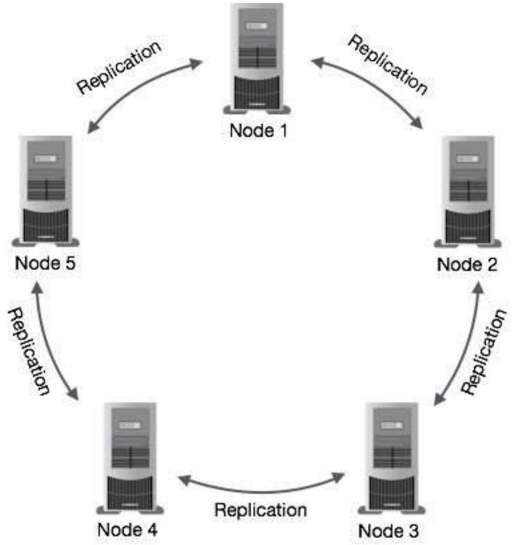


Figure - Node, Data center, Cluster

Cassandra - Architecture

Data Replication in Cassandra:

- In Cassandra, one or more of the nodes in a cluster act as replicas for a given piece of data.
- If it is detected that some of the nodes responded with an out-of-date value, Cassandra will return the most recent value to the client.
- After returning the most recent value, Cassandra performs a **read repair** in the background to update the stale values.



- The following figure shows a schematic view of how Cassandra uses data replication among the nodes in a cluster to ensure no single point of failure.

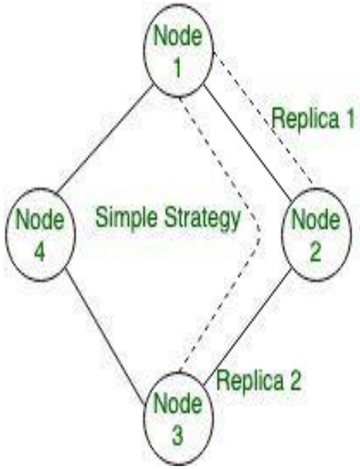
Note – Cassandra uses the **Gossip Protocol** in the background to allow the nodes to communicate with each other and detect any faulty nodes in the cluster.

Cassandra - Architecture

Data Replication Strategy:

- Basically it is used for backup to ensure no single point of failure.
- In this strategy Cassandra uses replication to achieve high availability and durability.
- Each data item is replicated at N hosts, where N is the replication factor configured \per-instance.
- There are two type of replication Strategy: Simple Strategy, and Network Topology Strategy.

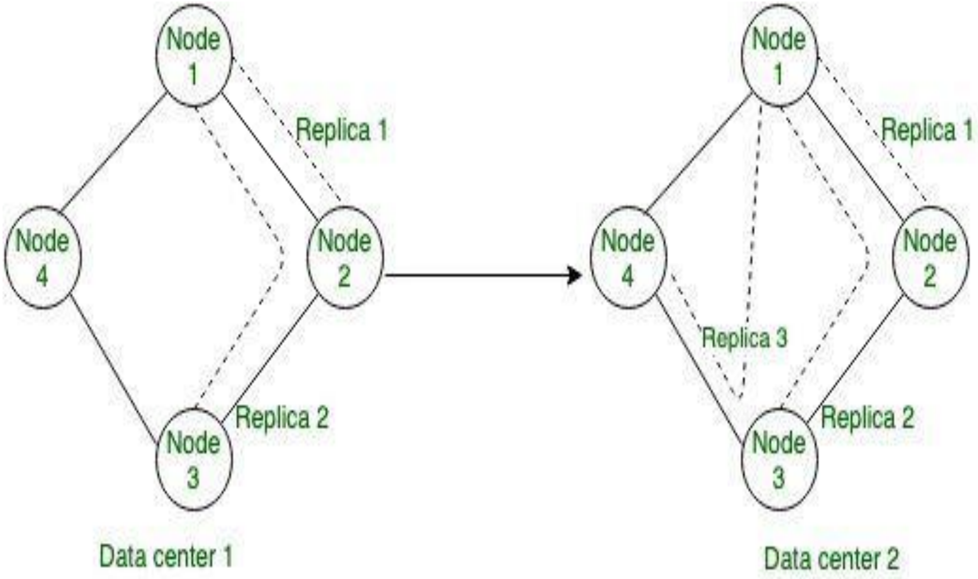
1. **Simple Strategy:**
   - ✓ In this Strategy it allows a single integer RF (replication_ factor) to be defined.
   - ✓ It determines the number of nodes that should contain a copy of each row. For example, if replication_ factor is 2, then two different nodes should store a copy of each row.
   - ✓ It treats all nodes identically.



Cassandra - Architecture

2. Network Topology Strategy:

- ✓ In this strategy it allows a replication factor to be specified for each datacenter in the cluster. Even if your cluster only uses a single datacenter.
- ✓ This Strategy should be preferred over Simple Strategy to make it easier to add new physical or virtual data centers to the cluster later.

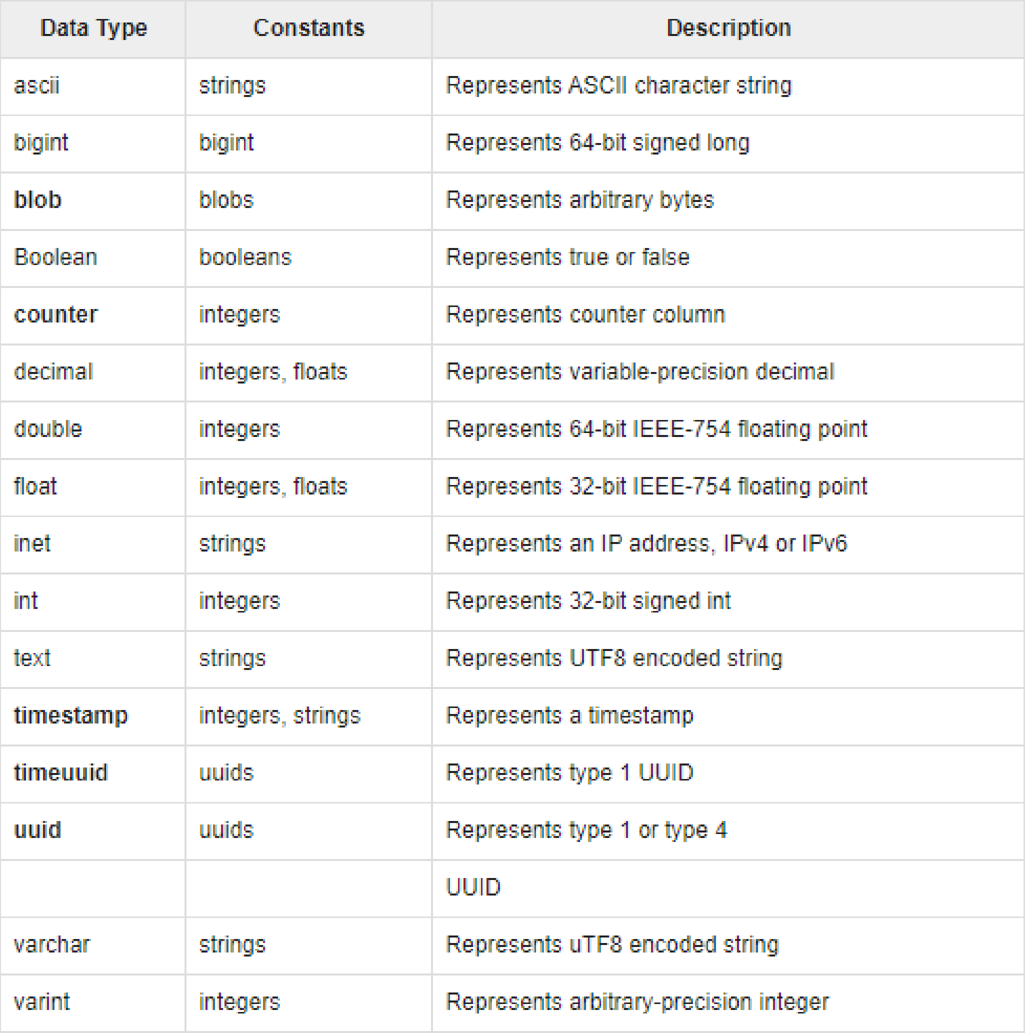


Application of Apache Cassandra

- Real-time, big data workloads
- Time series data management
- High-velocity device data consumption and analysis
- Media streaming management (e.g., music, movies)
- Social media (i.e., unstructured data) input and analysis
- Online web retail (e.g., shopping carts, user transactions)
- Real-time data analytics
- Online gaming (e.g., real-time messaging)
- Software as a Service (SaaS) applications that utilize web services
- Online portals (e.g., healthcare provider/patient interactions)
- Most write-intensive systems

Cassandra Query Language (CQL)

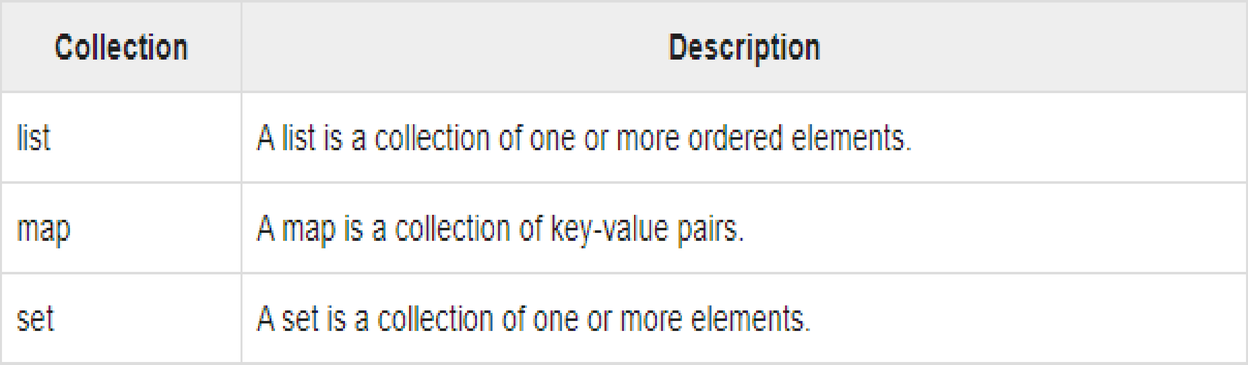
- CQL provides a rich set of built-in data types, including collection types. Along with these data types, users can also create their own custom data types. The following table provides a list of built-in data types available in CQL.



Data Type	Constants	Description
ascii	strings	Represents ASCII character string
bigint	bigint	Represents 64-bit signed long
blob	blobs	Represents arbitrary bytes
Boolean	booleans	Represents true or false
counter	integers	Represents counter column
decimal	integers, floats	Represents variable-precision decimal
double	integers	Represents 64-bit IEEE-754 floating point
float	integers, floats	Represents 32-bit IEEE-754 floating point
inet	strings	Represents an IP address, IPv4 or IPv6
int	integers	Represents 32-bit signed int
text	strings	Represents UTF8 encoded string
timestamp	integers, strings	Represents a timestamp
timeuuid	uuids	Represents type 1 UUID
uuid	uuids	Represents type 1 or type 4
		UUID
varchar	strings	Represents uTF8 encoded string
varint	integers	Represents arbitrary-precision integer

Cassandra Query Language (CQL)

- **Collection Types** : Cassandra Query Language also provides a collection data types. The following table provides a list of Collections available in CQL.



Collection	Description
list	A list is a collection of one or more ordered elements.
map	A map is a collection of key-value pairs.
set	A set is a collection of one or more elements.

- **User-defined data types** : Cqlsh provides users a facility of creating their own data types. Given below are the commands used while dealing with user defined datatypes.
  - **CREATE TYPE** – Creates a user-defined datatype.
  - **ALTER TYPE** – Modifies a user-defined datatype.
  - **DROP TYPE** – Drops a user-defined datatype.
  - **DESCRIBE TYPE** – Describes a user-defined datatype.
  - **DESCRIBE TYPES** – Describes user-defined datatypes.

Cassandra Keyspace Operations

- **Create Keyspace:**

```
CREATE KEYSPACE “KeySpace Name” WITH replication =
      {'class': Strategy name’, 'replication_factor‘:
      ‘No.Of replicas’};
```

Ex:

```
CREATE KEYSPACE emp WITH replication =
{'class':'SimpleStrategy', 'replication_factor' : 1};
```

  - ✓ The replication option specifies the replica placement strategy and the number of replicas wanted.

- ✓ You can verify whether the table is created or not using the command **Describe**.

```
DESCRIBE keyspaces;
```

- ✓ If you use this command over keyspaces, it will display all the keyspaces created

- **Using a Keyspace:**

You can use a created KeySpace using the keyword **USE**.

```
Ex: USE emp;
```

Cassandra Keyspace Operations

- **Altering a KeySpace :**

  ALTER KEYSPACE can be used to alter properties such as the number of replicas and the durable_writes of a KeySpace. Given below is the syntax of this command.

  ```
   ALTER KEYSPACE “KeySpace Name” WITH replication = {'class':
  ‘Strategy name’, 'replication_factor' : ‘No.Of replicas’};
  ```

  Ex: `ALTER KEYSPACE emp WITH replication =`

  ```
  {'class':'Network Topology Strategy', 'replication_factor' :
       3};
  ```

- **Dropping a Keyspace**

  You can drop a KeySpace using the command **DROP KEYSPACE**.

  Given below is the syntax for dropping a KeySpace.

  ```
  DROP KEYSPACE “KeySpace name”
  ```

  Ex: `DROP KEYSPACE emp;`

Cassandra Table Operations

- **Creating a Table:** You can create a table using the command **CREATE TABLE**

```
CREATE TABLE tablename(
       column1 name datatype PRIMARYKEY,
       column2 name data type,
       column3 name data type. ) OR
CREATE TABLE tablename( column1 name datatype,
       column2 name data type,
       column3 name data type,
       PRIMARY KEY (column1) )
```

- The primary key is a column that is used to uniquely identify a row. Therefore, defining a primary key is mandatory while creating a table. A primary key is made of one or more columns of a table.

```
USE emp;
cqlsh:emp>; CREATE TABLE emp_table( emp_id int PRIMARY KEY,
                                    emp_name text,
                                    emp_city text,
                                    emp_sal varint,
                                    emp_phone varint );
```

✓ The select statement will give you the schema.

```
cqlsh:emp> select * from emp_table;
```

Cassandra Table Operations

- **Altering a Table :** You can alter a table using the command **ALTER TABLE**.
- Using ALTER command, you can perform the following operations –
  - Add a column
  - Drop a column

  While adding columns, you have to take care that the column name is not conflicting with the existing column names.

```
ALTER TABLE table name ADD new column datatype;
Ex: ALTER TABLE emp_table ADD emp_email text;
```

- **Dropping a Column:** Using ALTER command, you can delete a column from a table.

```
ALTER table name DROP column name;
Ex: ALTER TABLE emp_table DROP emp_email;
```

Cassandra Table Operations

- **Dropping a Table:** You can drop a table using the command **Drop Table**. Its syntax is as follows –

```
DROP TABLE <tablename>
Ex: DROP TABLE emp_table;
```

- **Creating an Index**: You can create an index in Cassandra using the command **CREATE INDEX**

```
CREATE INDEX name ON emp1 (emp_name);
```

- **Dropping an Index:** You can drop an index using the command **DROP INDEX**

```
drop index name;
```

Cassandra CRUD Operations

- **Creating Data in a Table** : You can insert data into the columns of a row in a table using the command **INSERT**

```
INSERT INTO <tablename> (<column1 name>, <column2 name>....)
VALUES (<value1>, <value2>....) USING <option>
```

```
cqlsh:emp> INSERT INTO emp_table (emp_id, emp_name, emp_city,
emp_phone, emp_sal) VALUES(1,'ram', 'Hyderabad', 9848022338,
50000);
```

```
cqlsh:emp> INSERT INTO emp_table (emp_id, emp_name, emp_city,
emp_phone, emp_sal) VALUES(2,'robin', 'Hyderabad', 9848022339,
40000);
```

Cassandra CRUD Operations

- **UPDATE** is the command used to update data in a table. The following keywords are used while updating data in a table –

  Where – This clause is used to select the row to be updated.

  Set – Set the value using this keyword.

  Must – Includes all the columns composing the primary key.

- While updating rows, if a given row is unavailable, then UPDATE creates a fresh row.

```
UPDATE <tablename> SET <column name> = <new value>
<column name> = <value>.... WHERE <condition>
```

```
cqlsh:emp> UPDATE emp_table SET
emp_city='Delhi',emp_sal=50000 WHERE
           emp_id=2;
```

Cassandra CRUD Operations

- **Reading Data using Select Clause**: SELECT clause is used to read data from a table in Cassandra. Using this clause, you can read a whole table, a single column, or a particular cell.

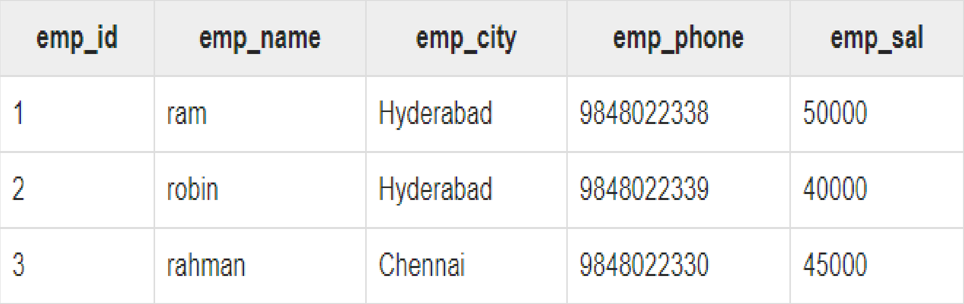
```
SELECT FROM <tablename>
Ex: cqlsh:emp> select * from emp_table;
Ex: cqlsh:emp> SELECT emp_name, emp_sal from
emp_table;
```

- **Where Clause:** Using WHERE clause, you can put a constraint on the required columns.

```
SELECT FROM <table name> WHERE <condition>;
Ex: cqlsh:emp> SELECT * FROM emp_table WHERE
emp_sal=50000;
```

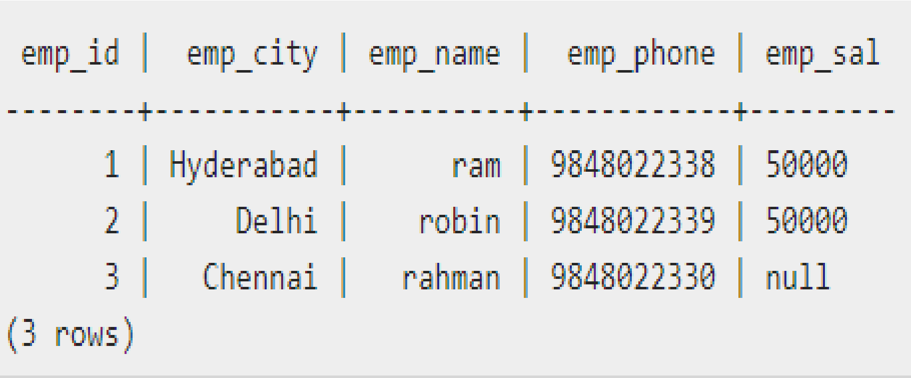
Cassandra CURD Operations

- **Deleting Data from a Table:** You can delete data from a table using the command **DELETE**.
- DELETE FROM <identifier> WHERE <condition>;



emp_id	emp_name	emp_city	emp_phone	emp_sal
1	ram	Hyderabad	9848022338	50000
2	robin	Hyderabad	9848022339	40000
3	rahman	Chennai	9848022330	45000

Ex: DELETE emp_sal FROM emp_table WHERE emp_id=3;

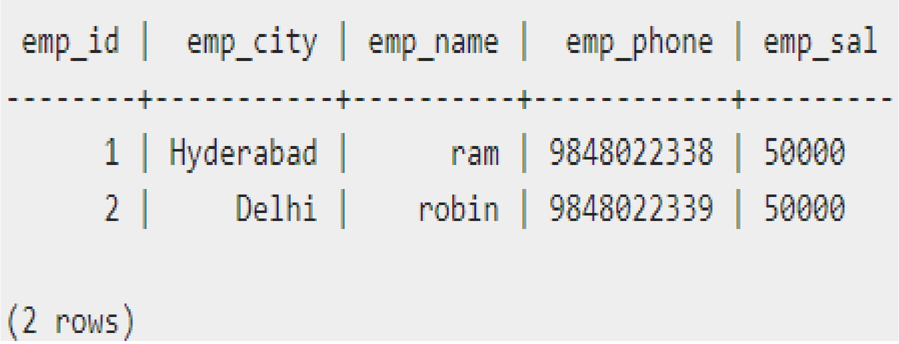


```
 emp_id |  emp_city | emp_name |  emp_phone | emp_sal
--------+-----------+----------+------------+---------
      1 | Hyderabad |      ram | 9848022338 | 50000
      2 |     Delhi |    robin | 9848022339 | 50000
      3 |   Chennai |   rahman | 9848022330 | null
(3 rows)
```

Cassandra CRUD Operations

- **Deleting an Entire Row**: The following command deletes an entire row from a table.

```
Ex: cqlsh:emp> DELETE FROM emp_table WHERE emp_id=3;
```



```
emp_id |  emp_city | emp_name |  emp_phone | emp_sal
--------+-----------+----------+------------+---------
      1 | Hyderabad |      ram | 9848022338 | 50000
      2 |     Delhi |    robin | 9848022339 | 50000

(2 rows)
```

Cassandra vs RDBMS

Cassandra	RDBMS
Cassandra is used to deal with unstructured data.	RDBMS is used to deal with structured data.
Cassandra has flexible schema.	RDBMS has fixed schema.
In Cassandra, a table is a list of "nested key-value pairs". (Row x Column Key x Column value)	In RDBMS, a table is an array of arrays. (Row x Column)
In Cassandra, keyspace is the outermost container which contains data corresponding to an application.	In RDBMS, database is the outermost container which contains data corresponding to an application.
In Cassandra, tables or column families are the entity of a keyspace.	In RDBMS, tables are the entities of a database.
In Cassandra, row is a unit of replication.	In RDBMS, row is an individual record.
In Cassandra, column is a unit of storage.	In RDBMS, column represents the attributes of a relation.
In Cassandra, relationships are represented using collections.	In RDBMS, there are concept of foreign keys, joins etc.

Cassandra

- https://cassandra.apache.org/_/index.html
- **How to Install Cassandra on Windows 10**
- Get Started with Apache Cassendra

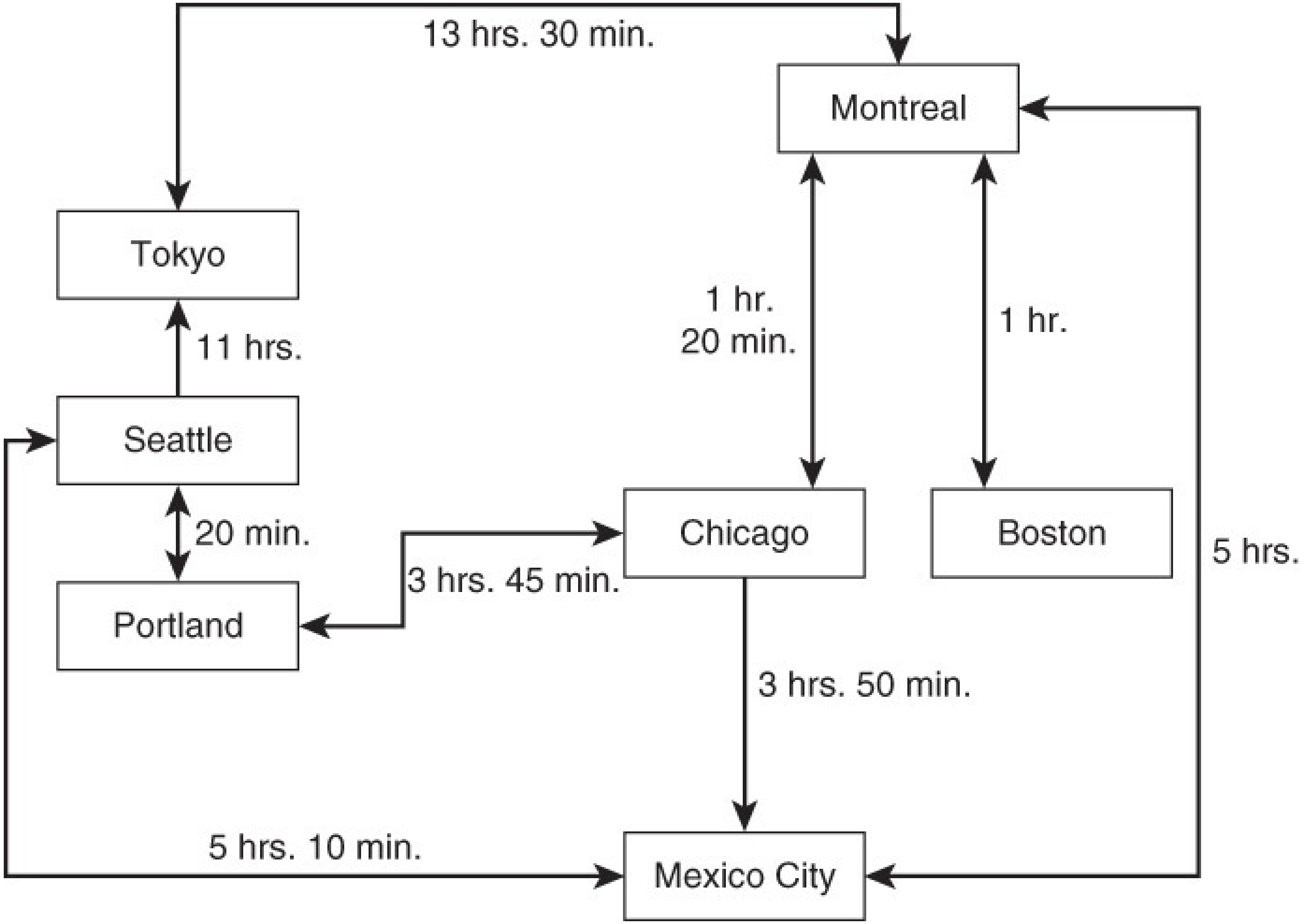
OR

- Cassandra using Astra Datastax

Graph Databases

- Graph databases are the most specialized of the four NoSQL databases.
- Instead of modeling data using columns and rows, a graph database uses structures called nodes and relationships.
- A ***relationship*** *is a* link between two nodes that contain attributes about that relation.
- **Nodes and Relationships**
- There are many ways to use graph databases. Nodes can represent people, and relationships can represent their friendships in social networks.

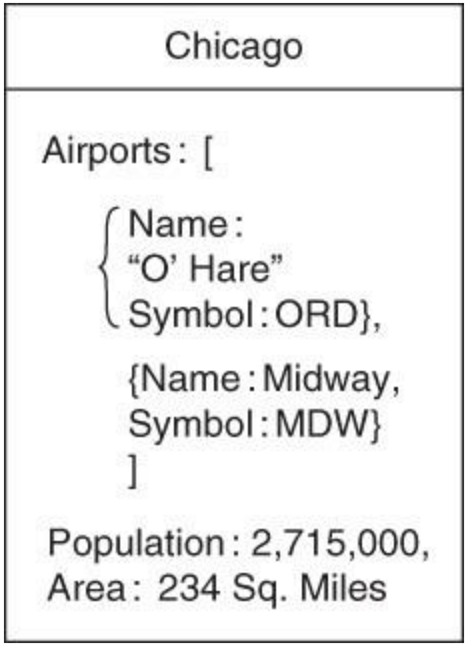
Graph Databases



Attributes include flying times between cities

Graph Databases

- Both the nodes and relationships can have complex structures.
- For example, each city can have a list of airports along with demographic and geographic data about the city, as shown in Fig.



Chicago

Airports : [
{Name : "O' Hare"
Symbol : ORD},
{Name : Midway,
Symbol : MDW}
]

Population : 2,715,000,
Area : 234 Sq. Miles

Graph Database vs. RDBMS

Graph Database	RDBMS
In graph database, data is stored in graphs.	In RDBMS, data is stored in tables.
In graph database there are nodes.	In RDBMS, there are rows.
In graph database there are properties and their values.	In RDBMS, there are columns and data.
In graph database the connected nodes are defined by relationships.	In RDBMS, constraints are used instead of that.
In graph database traversal is used instead of join.	In RDBMS, join is used instead of traversal.

Graph Database

- In the example graph in Figure, we see a bunch of nodes related to each other. Nodes are entities that have properties, such as name.
- The node of Martin is actually a node that has property of name set to Martin.

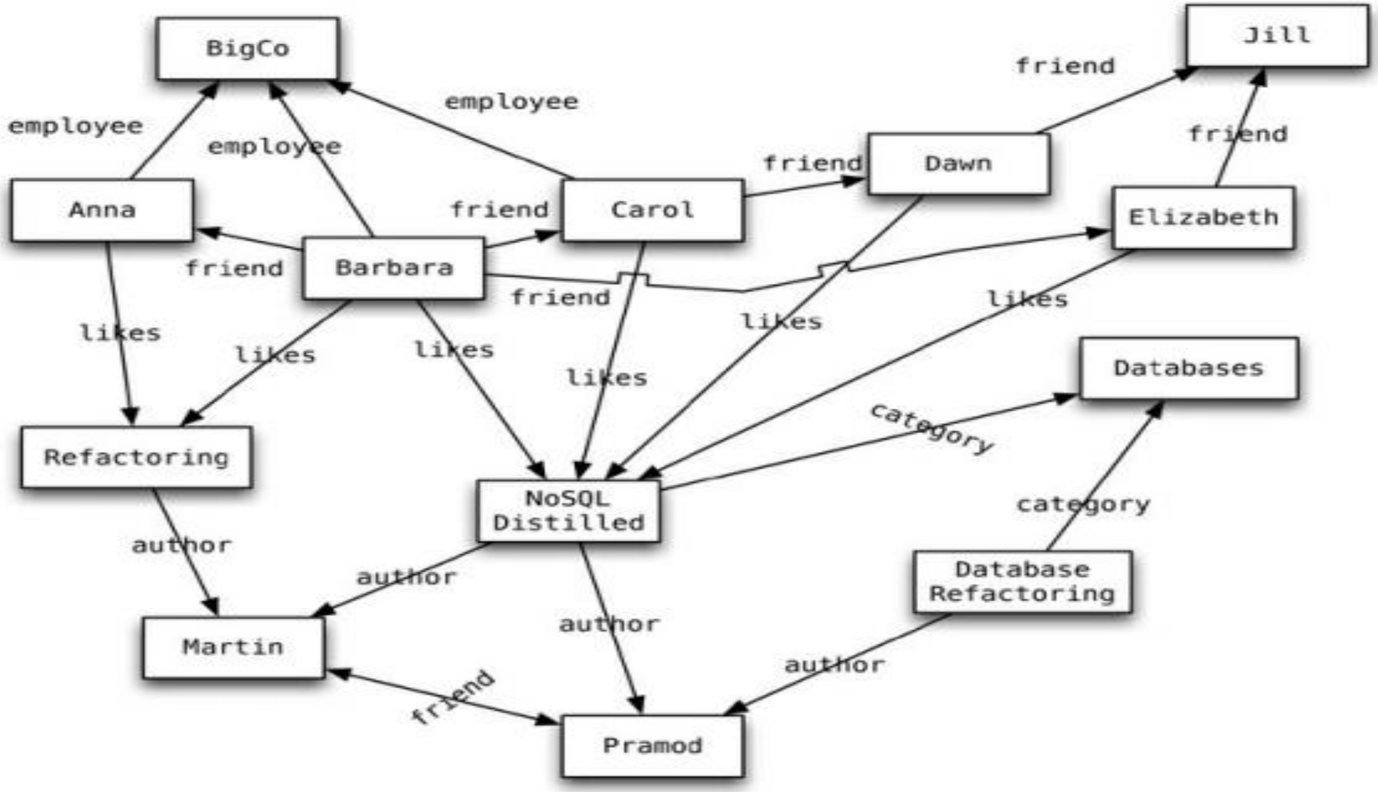


Figure 11.1. An example graph structure

Graph Database

- We also see that edges have types, such as likes, author, and so on.
- These properties let us organize the nodes; for example, the nodes Martin and Pramod have an edge connecting them with a relationship type of friend.
- Relationship types have directional significance; the friend relationship type is bidirectional but likes is not. When Dawn likes NoSQL Distilled, it does not automatically mean NoSQL Distilled likes Dawn.
- Once we have a graph of these nodes and edges created, we can query the graph in many ways, eg: "get all nodes employed by Big Co that like NoSQL Distilled."
- A query on the graph is also known as traversing the graph. An advantage of the graph databases is that we can change the traversing requirements without having to change the nodes or edges.

Application of Graph Database

- Graph databases are therefore highly beneficial to specific use cases:
  - Fraud Detection
  - 360 Customer Views
  - Recommendation Engines
  - Network/Operations Mapping
  - AI Knowledge Graphs
  - Social Networks
  - Supply Chain Mapping

Features of Graph Database

- There are many graph databases available, such as Neo4J [Neo4J], Infinite Graph [Infinite Graph], OrientB [OrientDB], or FlockDB [FlockDB].
- In Neo4J, creating a graph is as simple as creating two nodes and then creating a relationship. Let’s create two nodes, Martin and Pramod:

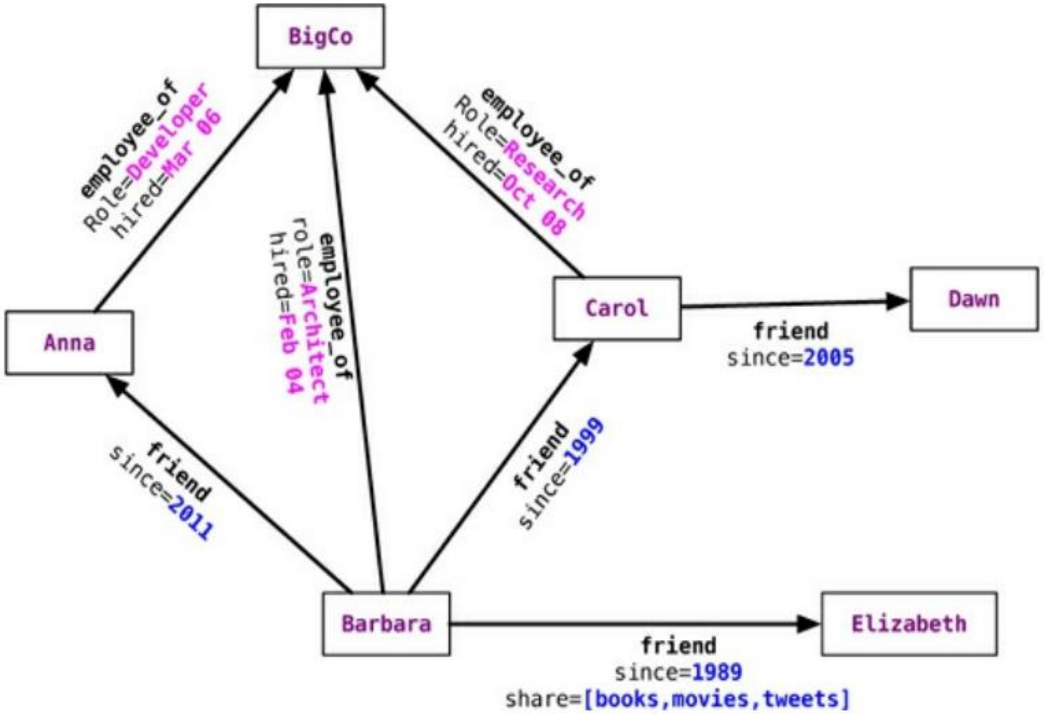
```
Node martin = graphDb.createNode();
martin.setProperty("name", "Martin");
Node pramod = graphDb.createNode();
pramod.setProperty("name", "Pramod");
```

- We have assigned the name property of the two nodes the values of Martin and Pramod. Once we have more than one node, we can create a relationship:

```
martin.createRelationshipTo(pramod,FRIEND);
pramod.createRelationshipTo(martin, FRIEND);
```

Features of Graph Database

- We have to create relationship between the nodes in both directions, for the direction of the relationship matters: For example, a product node can be liked by user but the product cannot like the user.
- This directionality helps in designing a rich domain model (Figure).
- Nodes know about INCOMING and OUTGOING relationships that are traversable both ways.



Neo4j Graph Database

- Neo4j is a NoSQL database.
- It is highly scalable and schema-free.
- It's world most popular graph database management system and Cypher Query Language (CQL).
- Neo4j is written in Java Language.
- Neo4j was developed by Neo technology and called an ACID-compliant transactional database with native graph storage and processing.

Features of Neo4j

- **Flexible Schema:** Neo4j follows a data model called graph model. The graph contains nodes and the nodes are connected to each other. Nodes and relationships store data in key-value pairs known as properties.
- **ACID Property:** Neo4j supports full ACID properties (Atomicity, Consistency, Isolation and Durability).
- **Scalability:** Neo4j facilitates you to scale the database by increasing the number of reads/writes, and the volume without affecting the data integrity and the speed of query processing.
- **Reliability:** Neo4j provides replication for data safety and reliability.
- **Cypher Query Language:** Neo4j provides a powerful declarative query language called Cypher Query language. It is used to create and retrieve relations between data without using the complex queries like Joins.
- **Built-in Web applications:** Neo4j also provides a built-in Neo4j browser web application which can be used to create and retrieve your graph data.
- **GraphDB:** Neo4j follows Property Graph Data Model.

Advantages of Neo4j

- **Highly scalable:** Neo4j is highly scalable. It provides a simple, powerful and flexible data model which can be changed according to applications and uses. It provides:
  - Higher vertical scaling.
  - Improved operational characteristics at scale.
  - Higher concurrency.
  - Simplified tuning.
- **Schema-free:** Neo4j is schema-free like other NoSQL databases.
- **High availability:** Neo4j provides high availability for large enterprise real-time applications with transactional guarantees.
- **Real-time data analysis:** Neo4j provides results based on real-time data.

Advantages of Neo4j

- **Easy representation:** Neo4j provides a very easy way to represent connected and semi-structured data.
- **Fast Execution:** Neo4j is fast because more connected data is very easy to retrieve and navigate.
- **Easy retrieval:** Neo4j facilitates you not only represent but also easily retrieve (**traverse/navigate**) connected data faster other databases comparatively.
- **Cypher Query language:** Neo4j provides CQL (**Cypher Query Language**) a declarative query language to represent the graph visually, using ASCII-art syntax. The commands of this language is very easy to learn and human readable.
- **No Join:** Neo4j doesn't require complex Joins to retrieve connected/related data as it is very easy to retrieve its adjacent node or relationship details without Joins or Indexes because it is a graph database and all nodes are already connected.

Neo4j CQL Functions

Index	Function	Usage
1.	STRING	They are used to work with string literals.
2.	Aggregation	They are used to perform some aggregation operations on CQL query results.
3.	Relationship	They are used to get details of relationships such as startnode, endnode, etc.

Neo4j CQL Data Types

Index	CQL Data Type	Usage
1.	Boolean	It is used to represent Boolean literals: True, False.
2.	byte	It is used to represent 8-bit integers.
3.	short	It is used to represent 16-bit integers.
4.	int	It is used to represent 32-bit integers.
5.	long	It is used to represent 64-bit integers.
6.	float	Float is used to represent 32-bit floating-point numbers.
7.	double	Double is used to represent 64-bit floating-point numbers.
8.	char	Char is used to represent 16-bit characters.
9.	String	String is used to represent strings.

Neo4j CQL Operators

- **Mathematical Operators:** i.e. +, -, *, /, %, ^
- **Comparison Operators:** i.e. =, <>, <, >, <=, >=
- **Boolean Operators:** i.e. AND, OR, XOR, NOT
- **String Operators:** i.e. =~
- **List Operators:** i.e. +, IN, [X], [X?..Y]
- **Regular Expression:** i.e. =-
- **String matching:** i.e. STARTS WITH, ENDS WITH, CONSTRAINTS

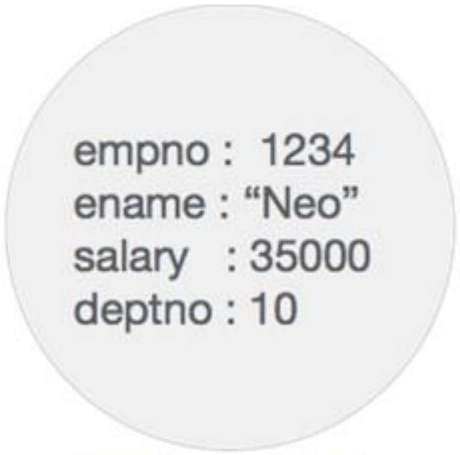
Neo4j - Building Blocks

- Neo4j Graph Database has the following building blocks –
  - Nodes
  - Properties
  - Relationships
  - Labels
  - Data Browser

Neo4j - Building Blocks

Node:

- Node is a fundamental unit of a Graph. It contains properties with key-value pairs as shown in the following image.



Employee Node

- Here, Node Name = "Employee" and it contains a set of properties as key-value pairs.

Neo4j - Building Blocks

Properties:

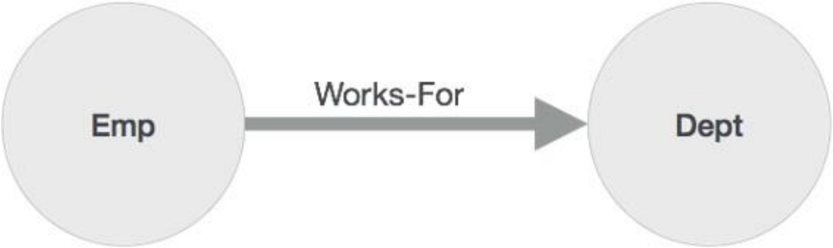
- Property is a key-value pair to describe Graph Nodes and Relationships.

  Key = value

- Where Key is a String and Value may be represented using any Neo4j Data types.

Relationships:

- Relationships are another major building block of a Graph Database. It connects two nodes as depicted in the following figure.



- Here, Emp and Dept are two different nodes. "WORKS_FOR" is a relationship between Emp and Dept nodes.

Neo4j - Building Blocks

Relationships:

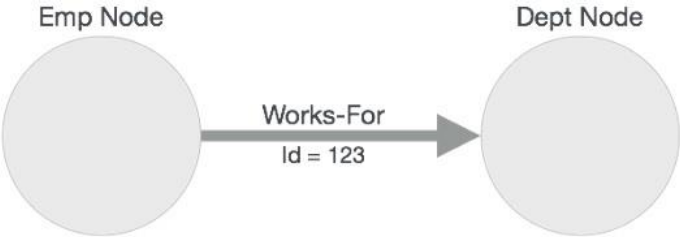
- As it denotes, the arrow mark from Emp to Dept, this relationship describes –

```
Emp WORKS_FOR Dept
```

- Each relationship contains one start node and one end node.
- Here, "Emp" is a start node, and "Dept" is an end node.
- As this relationship arrow mark represents a relationship from "Emp" node to "Dept" node, this relationship is known as an "Incoming Relationship" to "Dept" Node and "Outgoing Relationship" to "Emp" node.
- Like nodes, relationships also can contain properties as key-value pairs.
- Here, "WORKS_FOR" relationship has one property as key-value pair.

```
Id = 123
```

- It represents an Id of this relationship.



Neo4j - Building Blocks

Labels:

- Label associates a common name to a set of nodes or relationships.
- A node or relationship can contain one or more labels. We can create new labels to existing nodes or relationships. We can remove the existing labels from the existing nodes or relationships.
- From the previous diagram, we can observe that there are two nodes.
- Left side node has a Label: "Emp" and the right side node has a Label: "Dept".
- Relationship between those two nodes also has a Label: "WORKS_FOR".

Neo4j - Building Blocks

Neo4j Data Browser:

- Once we install Neo4j, we can access Neo4j Data Browser using the following URL

```
http://localhost:7474/browser/
```

- Neo4j Data Browser is used to execute CQL commands and view the output.
- Here, we need to execute all CQL commands at dollar prompt: "**$**"
- Type commands after the dollar symbol and click the "Execute" button to run your commands.
- It interacts with Neo4j Database Server, retrieves and displays the results just below the dollar prompt.
- Installation - Operations Manual (neo4j.com)

Neo4j CQL - Introduction

- CQL stands for Cypher Query Language. Like Oracle Database has query language SQL, Neo4j has CQL as query language.
- Neo4j CQL
  - Is a query language for Neo4j Graph Database.
  - Is a declarative pattern-matching language.
  - Follows SQL like syntax.
  - Syntax is very simple and in human readable format.
- Like Oracle SQL
  - Neo4j CQL has commands to perform Database operations.
  - Neo4j CQL supports many clauses such as WHERE, ORDER BY, etc., to write very complex queries in an easy manner.
  - Neo4j CQL supports some functions such as String, Aggregation. In addition to them, it also supports some Relationship Functions.

Neo4j CQL - Creating Nodes

- A node is a data/record in a graph database. You can create a node in Neo4j using the **CREATE** clause.
  - Create a single node
  - Create multiple nodes
  - Create a node with a label
  - Create a node with multiple labels
  - Create a node with properties
  - Returning the created node
- **Creating a Single node:**
  - You can create a node in Neo4j by simply specifying the name of the node that is to be created along with the CREATE clause.

```
CREATE (node_name);
```

```
CREATE (sample)
```

Neo4j CQL - Creating Nodes

- **Creating Multiple Nodes:**
  - The create clause of Neo4j CQL is also used to create multiple nodes at the same time. To do so, you need to pass the names of the nodes to be created, separated by a comma.

```
CREATE (node1),(node2)
```

```
CREATE (sample1),(sample2)
```

- **Creating a Node with a Label:**
  - A label in Neo4j is used to group (classify) the nodes using labels. You can create a label for a node in Neo4j using the CREATE clause.

```
CREATE (node:label)
```

```
CREATE (Dhawan:player)
```

Neo4j CQL - Creating Nodes

- **Creating a Node with Multiple Labels**
  - You can also create multiple labels for a single node. You need to specify the labels for the node by separating them with a colon " : ".

```
CREATE (node:label1:label2:. . . . labeln)
```

```
CREATE (Dhawan:person:player)
```

- **Create Node with Properties**
  - Properties are the key-value pairs using which a node stores data. You can create a node with properties using the CREATE clause. You need to specify these properties separated by commas within the flower braces "{ }".

```
CREATE (node:label { key1: value, key2: value, . . . . . . . . . })
```

```
CREATE (Dhawan:player{name: "Shikar Dhawan", YOB: 1985, POB: "Delhi"})
```

Neo4j CQL Write Clauses

- Neo4j - Merge Command
- Neo4j - Set Clause
- Neo4j - Delete Clause
- Neo4j - Remove Clause

Neo4j CQL Write Clauses

Neo4j - Merge Command:

- MERGE command is a combination of CREATE command and MATCH command.
- Neo4j CQL MERGE command searches for a given pattern in the graph. If it exists, then it returns the results.
- If it does NOT exist in the graph, then it creates a new node/relationship and returns the results.
  - Merge a node with label
  - Merge a node with properties
  - OnCreate and OnMatch
  - Merge a relationship

Neo4j CQL Write Clauses

```
MERGE (node: label {properties . . . . . . . })
```

- You can merge a node in the database based on the label using the MERGE clause. If you try to merge a node based on the label, then Neo4j verifies whether there exists any node with the given label. If not, the current node will be created.

```
MERGE (node:label) RETURN node
```

```
MERGE (Jadeja:player) RETURN Jadeja
```

Neo4j CQL Write Clauses

Neo4j - Set Clause:

- Using Set clause, you can add new properties to an existing Node or Relationship, and also add or update existing Properties values.
  - Set a property
  - Remove a property
  - Set multiple properties
  - Set a label on a node
  - Set multiple labels on a node

Setting a Property

- Using the SET clause, you can create a new property in a node.

```
MATCH (node:label{properties . . . . . . . . . . . . . . })
SET node.property = value
RETURN node
```

```
MATCH (Dhawan:player{name: "shikar Dhawan", YOB: 1985, POB:
"Delhi"})
SET Dhawan.highestscore = 187
RETURN Dhawan
```

Neo4j CQL Write Clauses

Removing a Property

- You can remove an existing property by passing **NULL** as value

```
MATCH (node:label {properties})
SET node.property = NULL
RETURN node
```

```
MATCH (Jadeja:player {name: "Ravindra Jadeja",
       YOB: 1988, POB: "NavagamGhed"})
SET Jadeja.POB = NULL
RETURN Jadeja
```

Neo4j CQL Write Clauses

Setting Multiple Properties

- In the same way, you can create multiple properties in a node using the Set clause. To do so, you need to specify these key value pairs with commas.

```
MATCH (node:label {properties})
SET node.property1 = value, node.property2 = value
RETURN node
```

```
MATCH (Jadeja:player {name: "Ravindra Jadeja", YOB: 1988})
SET Jadeja.POB: "NavagamGhed", Jadeja.HS = "90"
RETURN Jadeja
```

Neo4j CQL Write Clauses

Neo4j - Delete Clause:

- You can delete nodes and relationships from a database using the DELETE clause.

Deleting All Nodes and Relationships

```
MATCH (n) DETACH DELETE n
```

Deleting a Particular Node

- To delete a particular node, you need to specify the details of the node in the place of “n” in the above query.

```
MATCH (node:label {properties . . . . . . . . . . })
DETACH DELETE node
```

```
MATCH (Ishant:player {name: "Ishant Sharma",
          YOB: 1988, POB: "Delhi"})
DETACH DELETE Ishant
```

Neo4j CQL Write Clauses

Neo4j - Remove Clause:

- The REMOVE clause is used to remove properties and labels from graph elements (Nodes or Relationships).
- The main difference between Neo4j CQL DELETE and REMOVE commands is –
  - DELETE operation is used to delete nodes and associated relationships.
  - REMOVE operation is used to remove labels and properties.

Removing a Property

- You can remove a property of a node using MATCH along with the REMOVE clause.

```
MATCH (node:label{properties . . . . . . . })
REMOVE node.property
RETURN node
```

```
MATCH (Dhoni:player {name: "MahendraSingh Dhoni",
       YOB: 1981, POB: "Ranchi"})
REMOVE Dhoni.POB
RETURN Dhoni
```

Neo4j CQL Write Clauses

Removing a Label From a Node

- Similar to property, you can also remove a label from an existing node using the remove clause.

```
MATCH (node:label {properties . . . . . . . . . . . })
REMOVE node:label
RETURN node
```

```
MATCH (Dhoni:player {name: "MahendraSingh Dhoni",
       YOB: 1981, POB: "Ranchi"})
REMOVE Dhoni:player
RETURN Dhoni
```

Neo4j CQL Write Clauses

Removing Multiple Labels

- You can also remove multiple labels from an existing node.

```
MATCH (node:label1:label2 {properties . . . . . . . . })
REMOVE node:label1:label2
RETURN node
```

```
MATCH (Ishant:player:person {name: "Ishant Sharma",
       YOB: 1988, POB: "Delhi"})
REMOVE Ishant:player:person
RETURN Ishant
```

Neo4j CQL Read Clause

- Neo4j - Match Clause
- Neo4j - Optional Match Clause
- Neo4j - Where Clause
- Neo4j - Count Function

Neo4j CQL Read Clause

Neo4j - Match Clause:

Get All Nodes Using Match

- Using the MATCH clause of Neo4j you can retrieve all nodes in the Neo4j database.

```
MATCH (n)
RETURN n
```

Getting All Nodes Under a Specific Label

- Using match clause, you can get all the nodes under a specific label.

```
MATCH (node:label)
RETURN node
```

```
MATCH (n:player)
RETURN n
```

Neo4j CQL Read Clause

Match by Relationship

- You can retrieve nodes based on relationship using the MATCH clause.

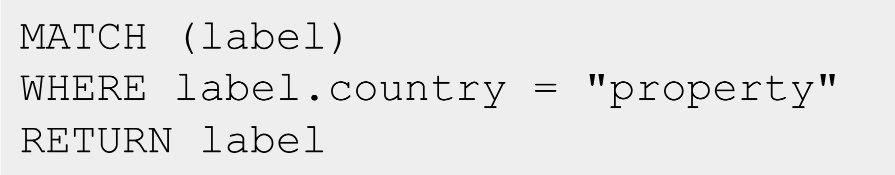
```
MATCH (node:label)<-[: Relationship]-(n)
RETURN n
```

```
MATCH (Ind:Country {name: "India", result: "Winners"})
                    <-[: TOP_SCORER_OF]-(n)
RETURN n.name
```

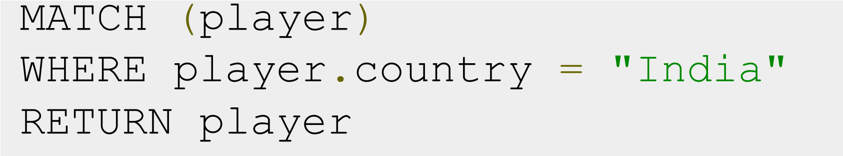
Neo4j CQL Read Clause

Neo4j - Where Clause:

- Like SQL, Neo4j CQL has provided WHERE clause in CQL MATCH command to filter the results of a MATCH Query.



```
MATCH (label)
WHERE label.country = "property"
RETURN label
```



```
MATCH (player)
WHERE player.country = "India"
RETURN player
```

Neo4j CQL Read Clause

WHERE Clause with Multiple Conditions

- You can also use the WHERE clause to verify multiple conditions.

```
MATCH (emp:Employee)
WHERE emp.name = 'Abc' AND emp.name = 'Xyz'
RETURN emp
```

```
MATCH (player)
WHERE player.country = "India" AND player.runs >=175
RETURN player
```

Neo4j General Clauses

- General clauses :
  - RETURN
  - ORDER BY
  - LIMIT
  - SKIP
  - WITH
  - UNWIND
  - UNION
  - CALL

Neo4j General Clauses

- **RETURN clause :**

  The RETURN clause is used return nodes, relationships, and properties in Neo4j.

  - Return nodes
  - Return multiple nodes
  - Return relationships
  - Return properties
  - Return all elements
  - Return a variable with column alias

```
Create (node:label {properties})
RETURN node
```

```
Create (Dhoni:player {name: "MahendraSingh Dhoni",
           YOB: 1981, POB: "Ranchi"})
RETURN Dhoni
```

Neo4J General Clauses

- **ORDER BY clause :**

ORDER BY is used to sort the output.

```
MATCH (n)
RETURN n.property1, n.property2 . . . . . . . .
ORDER BY n.property
```

Example –

```
MATCH (n)
RETURN n.name, n.age
ORDER BY n.name
```

Neo4J General Clauses

- **LIMIT :**
  - It is used to return a limited subset of the rows.
  - Example:

```
MATCH (n)
RETURN n.name
ORDER BY n.name
LIMIT 3
```

  - Limit accepts any expression that evaluates to a positive integer as long as it is not referring to any external variables

```
MATCH (n)
RETURN n.name
 ORDER BY n.name
 LIMIT 1 + toInteger(3 * rand())
```

Neo4J General Clauses

- **SKIP :**

SKIP defines from which row to start including the rows in the output.

Examples:

Skip first three rows:

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 3
```

Return middle two rows:

```
MATCH (n)
RETURN n.name
ORDER BY n.name
SKIP 1
LIMIT 2
```

Neo4j vs MySQL

Neo4J	MySQL
Neo4J contains vertices and edges. Each vertex or node represent a key value or attribute.	In MySQL, attributes are appended in plain table format.
In Neo4J, it is possible to store dynamic content like images, videos, audio etc.	In relational databases, such as MySQL, it is difficult to store videos, audios and images.
It provides the capability of deep search into the database without affecting the performance along with efficient timing.	It takes longer time for database search and also inconvenient compared to Neo4J.
In Neo4j, two or more objects can be related by making relationship between them.	It lacks relationship and very difficult to use them for connected graphs and data.

END OF UNIT 4