CET4001B Big Data Technologies

School of Computer Engineering and Technology

CET4001B Big Data Technologies

Teaching Scheme
Credits: 03 + 01

Theory: 3 Hrs / Week
Practical: 2Hrs/Week

Course Objectives:

- Understand the various aspects and life cycle of Big Data
- Learn the concepts of NoSQL for Big Data
- Design an application for distributed systems on Big Data.
- To understand and analyse different storage technologies required for Big Data
- To explore the technological foundations of Big Data Analytics
- To understand the role of various visualization techniques and explore the various Big Data visualization tools.

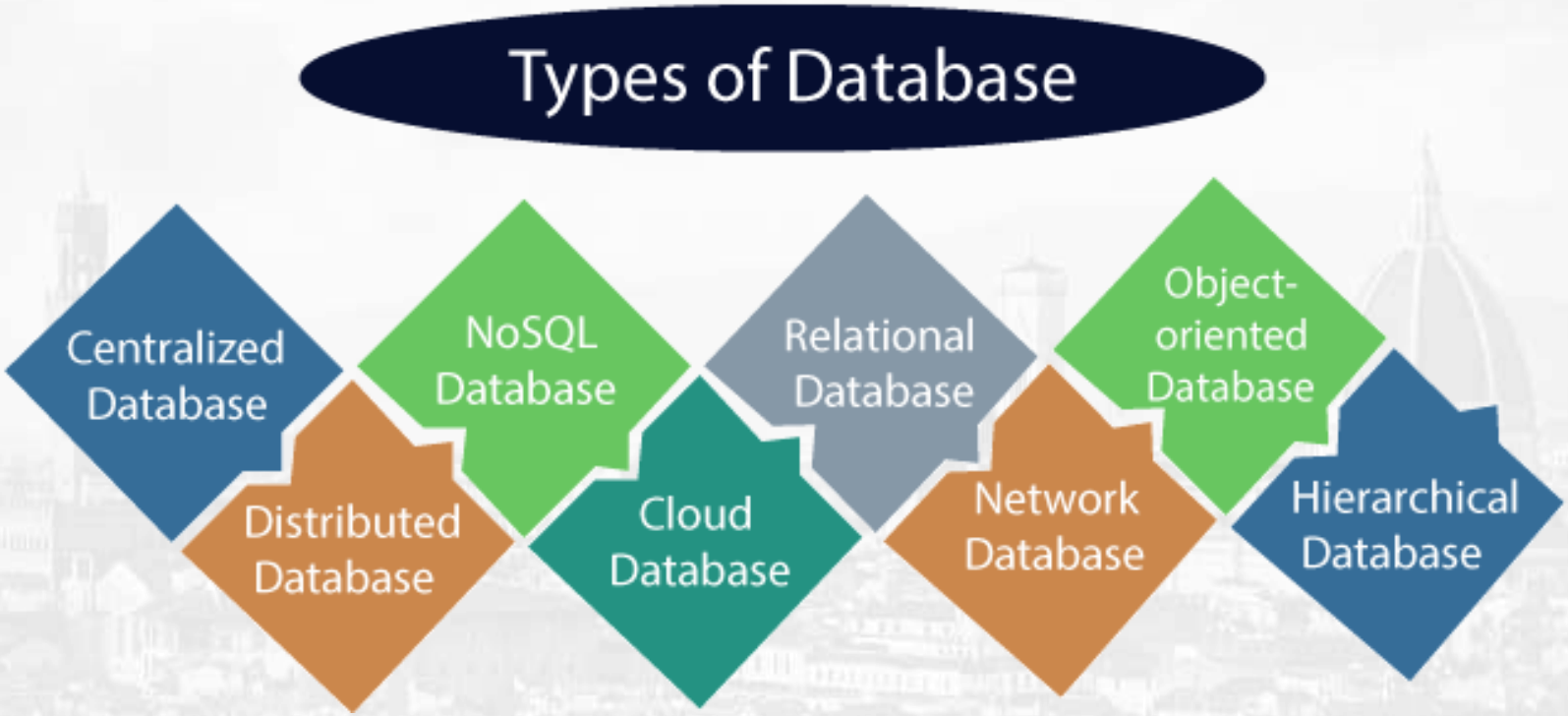
Course Outcomes:

- Recognize the characteristics of Big Data
- Ability to demonstrate information retrieval of Big Data
- Analyse the HADOOP and Map Reduce technologies associated with big data
- Perform analytics to learn the usage of distributed processing framework
- To investigate the impact of different visualizations for real world applications

Unit-II: **NoSQL databases for Big Data**

- Types of databases
- structured versus unstructured data
- NoSQL movement and concept of NoSQL database
- comparative study of SQL and NoSQL
- Types and examples of NoSQL database
  - key value store, document store, columnar databases, graph databases.
- Characteristics of NoSQL
- NoSQL data modelling
- Advantages of NoSQL
- CAP theorem
- BASE properties
- Sharding
  - characteristics, advantages, types.
- NoSQL using MongoDB
  - - MongoDB shell, data types, CRUD operations, querying, aggregation framework operators, indexing.

Types of Databases



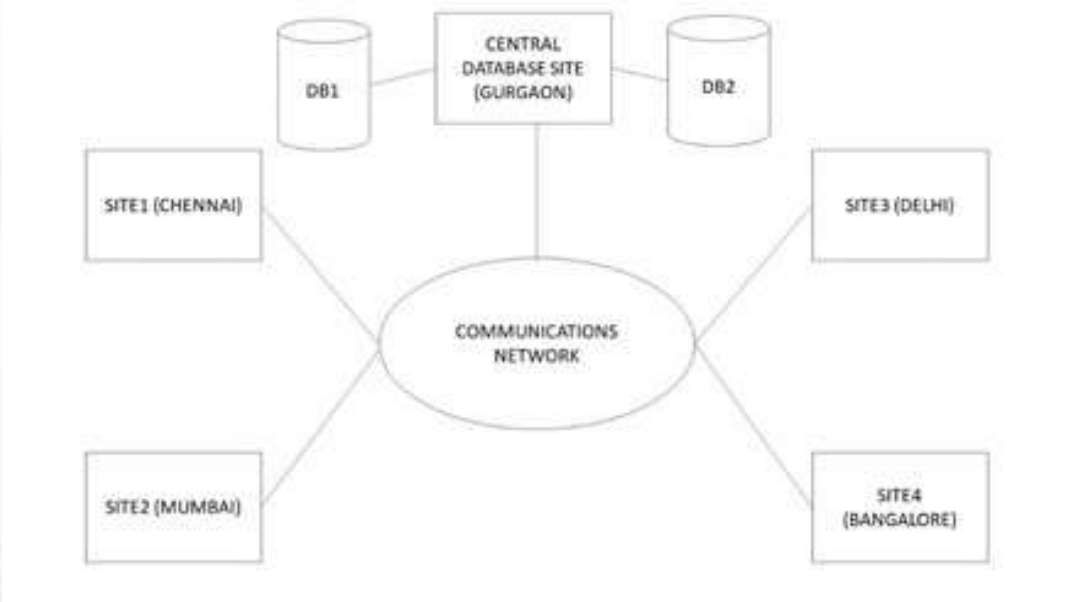
Centralized Database

A type of database that stores data at a centralized database system.

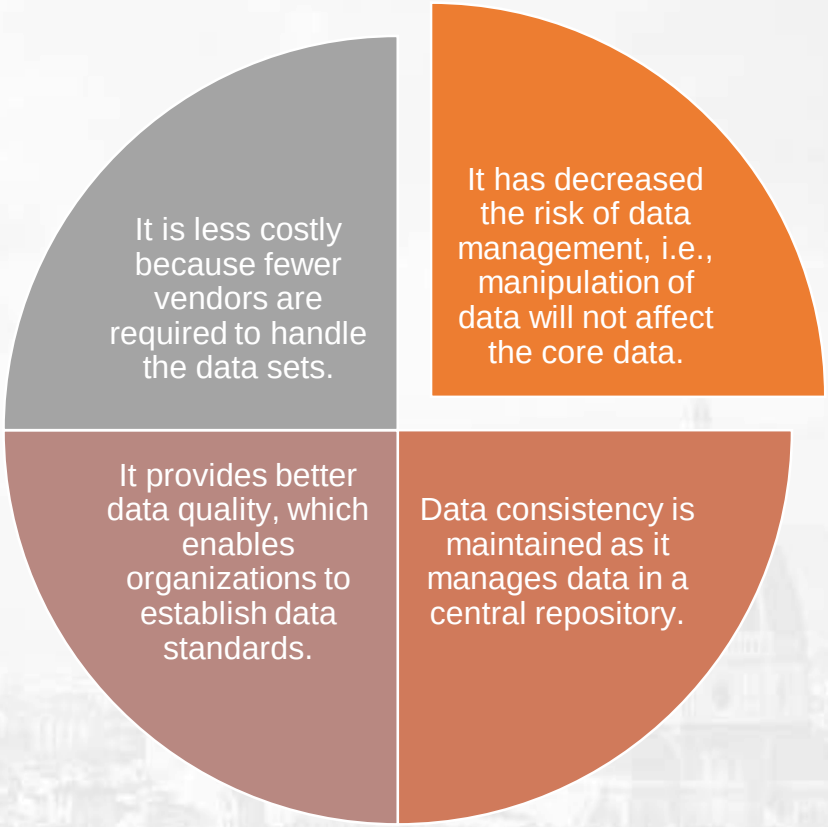
Comforts the users to access the stored data from different locations through several applications that contain the authentication process to let users access data securely.

Example of a Centralized database : a Central Library that carries a central database of each library in a college/university.

Centralized Database



Advantages of Centralized Database



Limitations of Centralized Database

The size of the centralized database is large, which increases the response time for fetching the data.

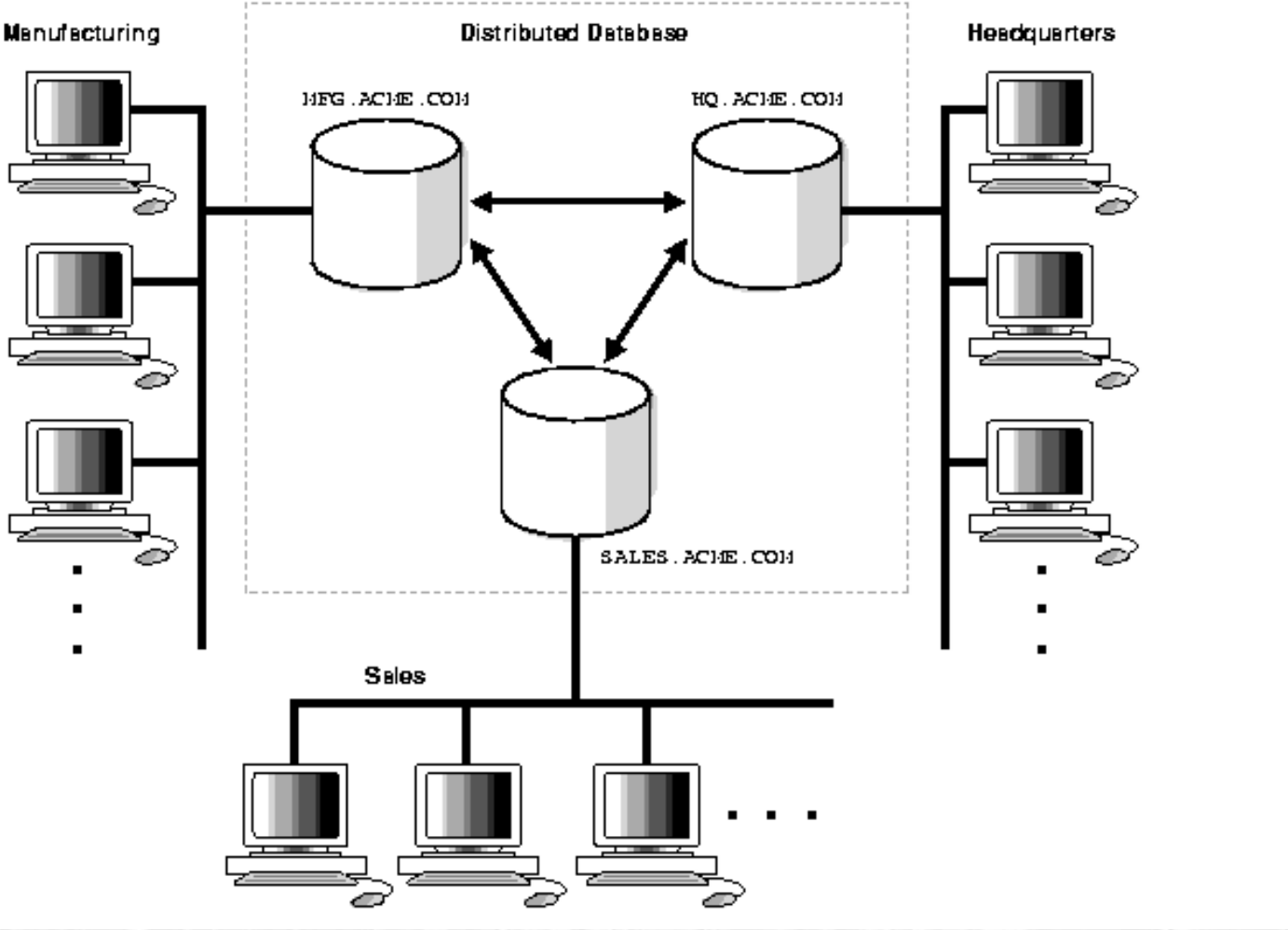
not easy to update such an extensive database system.

If any server failure occurs, entire data will be lost, which could be a huge loss.

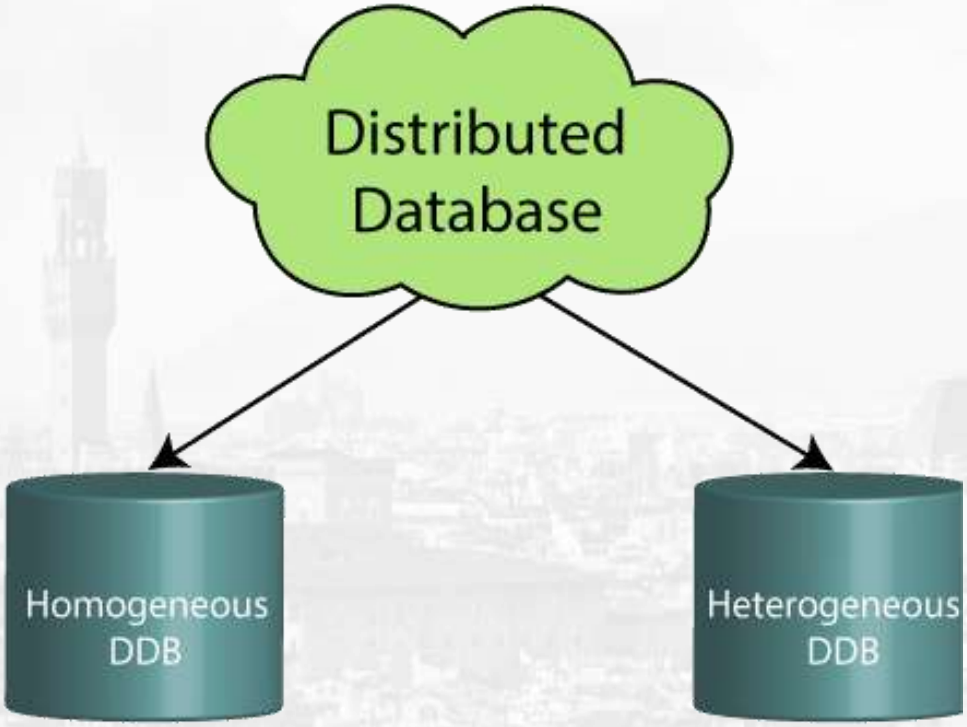
Distributed Database System

- Data stored at a number of sites each site *logically* consists of a single processor.
- Processors at different sites are interconnected by a computer network no multiprocessors
  - parallel database systems
- Distributed database is a database, not a collection of files data logically related as exhibited in the users' access patterns
  - relational data model
- D-DBMS is a full-fledged DBMS
  - not remote file system, not a TP system

Distributed Database



Distributed Database



- Unlike a centralized database system, data is distributed among different database systems of an organization.
- are connected via communication links. which help the end-users to access the data easily.
- Examples : Apache Cassandra, HBase, Ignite, etc.

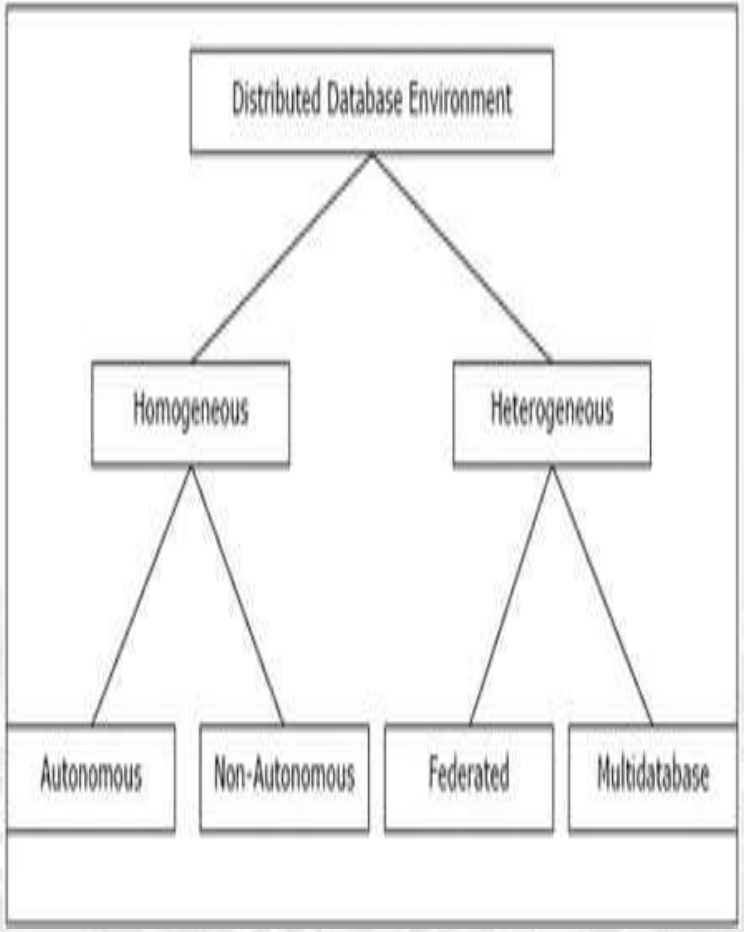
Distributed Database Types

Homogeneous DDB:

- Execute on the same operating system
- use the same application process

Heterogeneous DDB:

- execute on different operating systems
- under different application procedures,
- carries different hardware devices.



Advantages

- **Increased Reliability and availability**
  - Reliability is basically defined as the probability that a system is running at a certain time whereas Availability is defined as the probability that the system is continuously available during a time interval.
- **Easier Expansion**
  - In a distributed environment expansion of the system in terms of adding more data, increasing database sizes, or adding more processor is much easier.
- **Improved Performance**
  - We can achieve inter-query and intra-query parallelism by executing multiple queries at different sites by breaking up a query into a number of sub-queries that basically executes in parallel which basically leads to improvement in performance

Advantages of Distributed Database

- Modular development is possible i.e., the system can be expanded by including new computers and connecting them to the distributed system.
- One server failure will not affect the entire data set.
- A user doesn't know where the data is located physically.
- the data presented to the user as if it were located locally.
- Data can be joined and updated from different tables which are located on different machines.
- It is secure.

Disadvantages

- The distributed database is quite complex
- This database is more expensive as it is complex and hence, difficult to maintain.
- As it is distributed system it requires database to be more secure and each and every node should be secure as well.
- Data integrity and Data Redundancy will not be maintained.

Limitations of Distributed Databases

Network traffic is increased in a distributed database.

Different data formats are used in different systems.

While recovering a failed system, the DBMS has to make sure that the recovered system is consistent with other systems.

Managing distributed deadlock is a difficult task.

Relational Database

based on the relational data model,

which stores data in the form of rows (tuple) and columns(attributes), and together forms a table(relation).

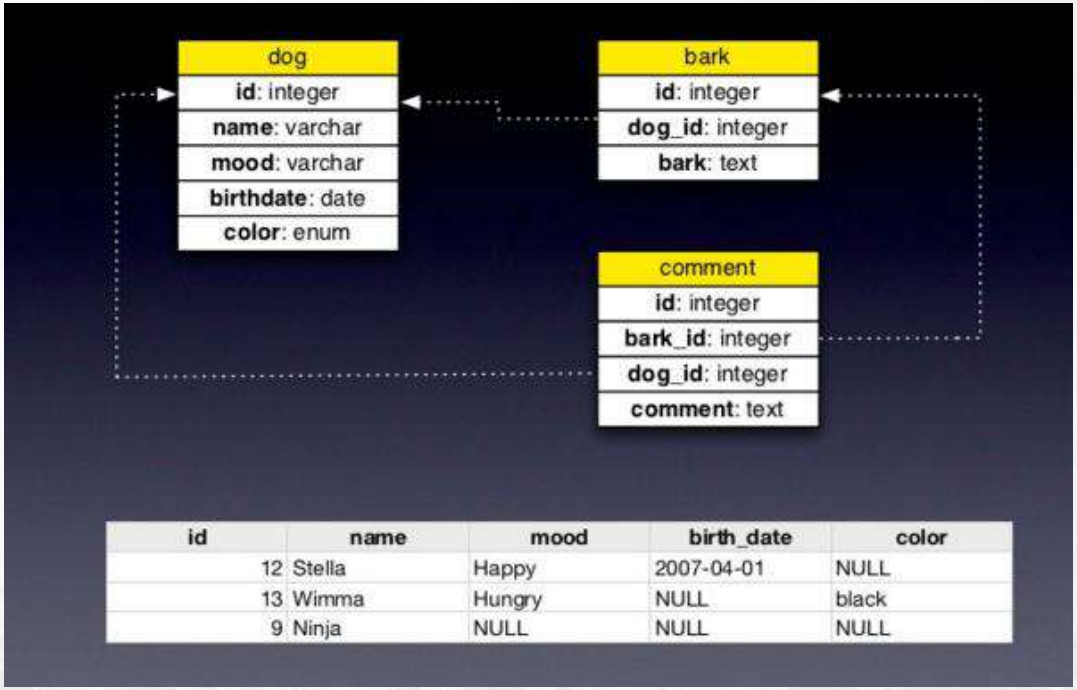
uses SQL for storing, manipulating, as well as maintaining the data.

E.F. Codd invented the database in 1970.

Each table carries a key that makes the data unique from others.

Examples : MySQL, Microsoft SQL Server, Oracle, etc.

Relational Database Representation



id	name	mood	birth_date	color
12	Stella	Happy	2007-04-01	NULL
13	Wimma	Hungry	NULL	black
9	Ninja	NULL	NULL	NULL

4. Cloud Database

data is stored in a virtual environment and executes over the cloud computing platform.

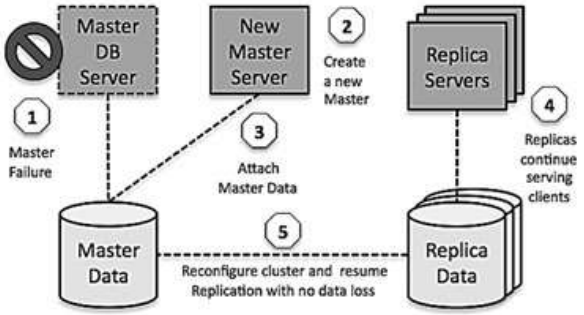
provides users with various cloud computing services (SaaS, PaaS, IaaS, etc.) for accessing the database.

[[Software as a Service (SaaS), Platform as a Service (PaaS), and Infrastructure as a Service (IaaS)]]

Cloud Database

- There are numerous cloud platforms, but the best options are:

- Amazon Web Services
- (AWS)
- Microsoft Azure
- Kamatera
- PhonixNAP
- ScienceSoft
- Google Cloud SQL, etc.



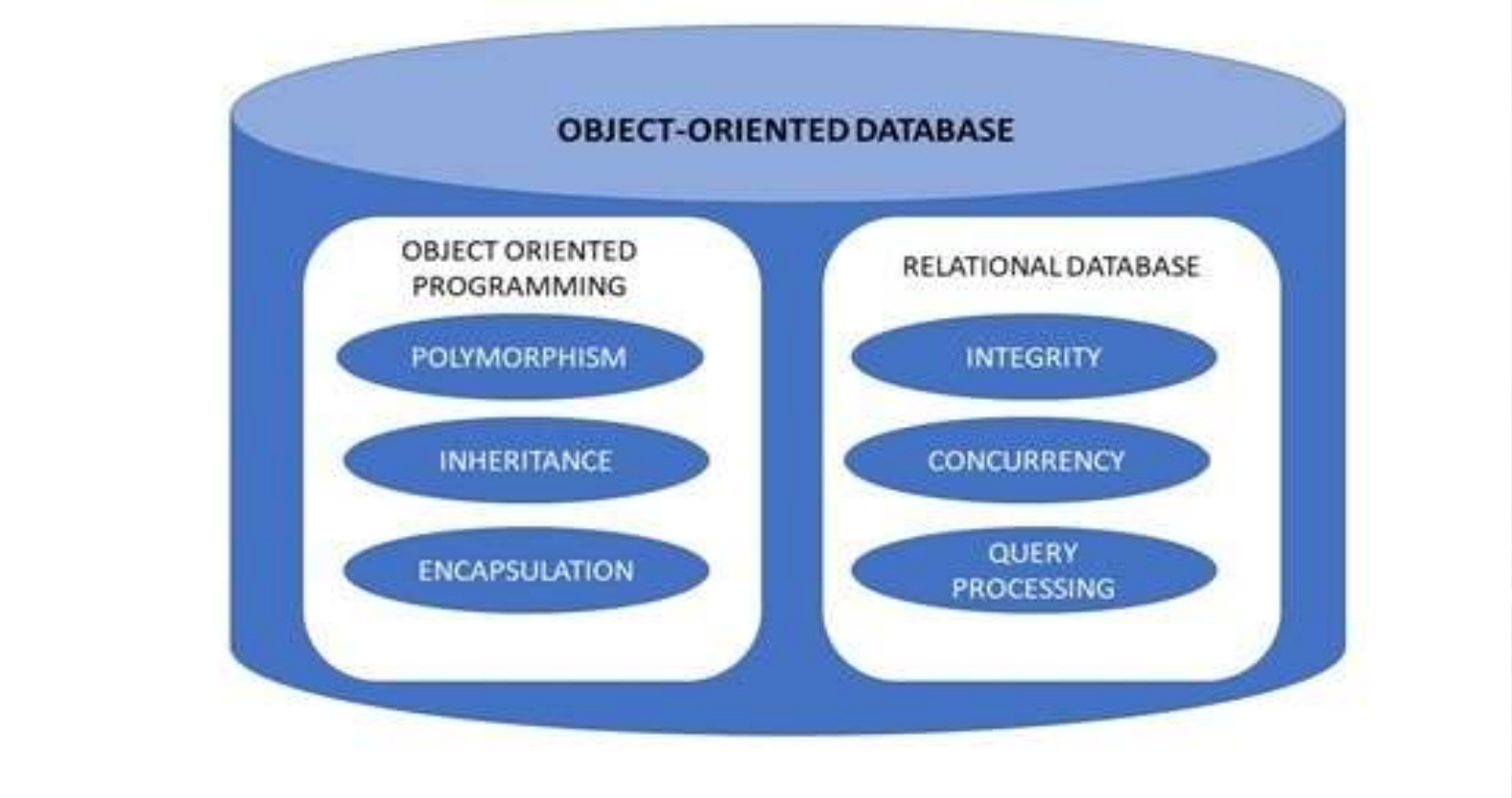
Object-oriented Database

Uses the object-based data model approach for storing data in the database system.

Data is represented and stored as objects which are similar to the objects used in the object-oriented programming language.

Examples : Smalltalk in Gemstone,LISP in Gbase.

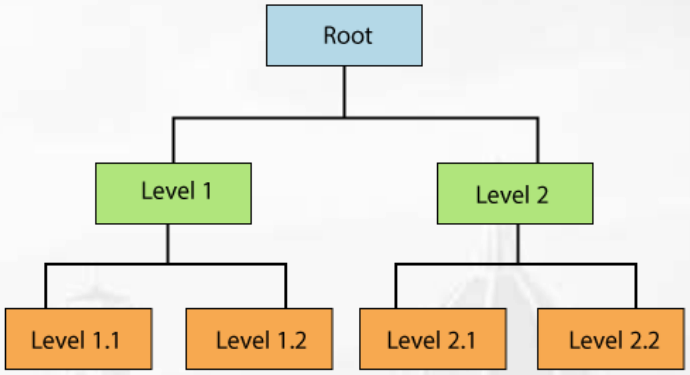
Object-oriented Database Representation



Hierarchical Database

Stores data in the form of parent-children relationship nodes.

It organizes data in a tree-like structure.



Hierarchical Database

Hierarchical Databases

Mandates that each child record has only one parent, whereas each parent record can have one or more child records.

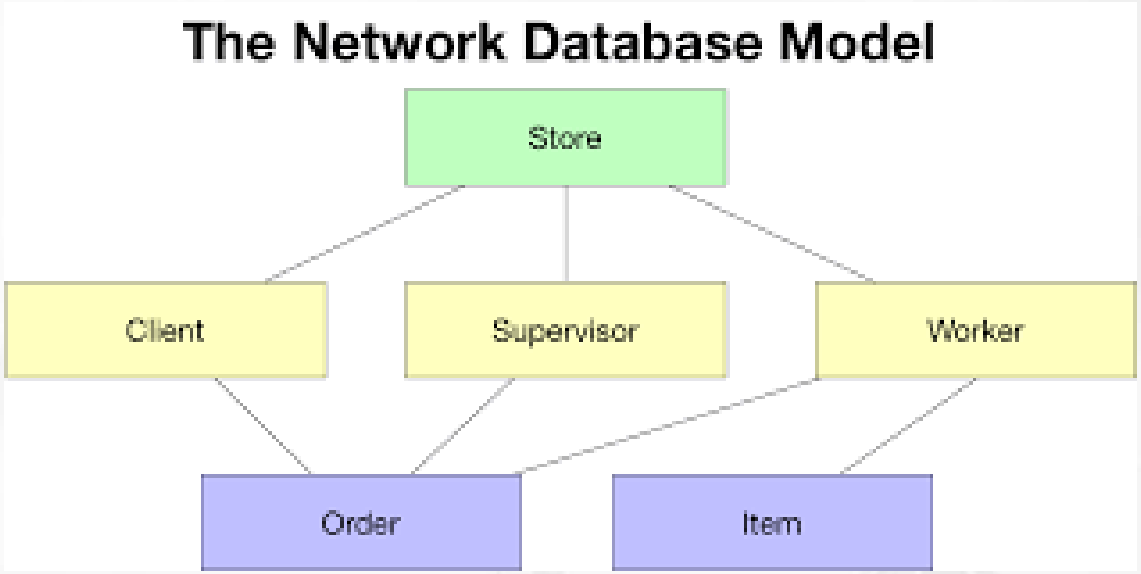
In order to retrieve data from a hierarchical database the whole tree needs to be traversed starting from the root node.

Examples : IBM Information Management System(IMS),RDM Mobile.

Network Database

- typically follows the network data model.
- the representation of data is in the form of nodes connected via links between them.
- Unlike the hierarchical database, it allows each record to have multiple children and parent nodes to form a generalized graph structure.

Network Database Continued



- Some well-known database systems that use the network model include:
- Raima Database Manager
- Integrated DMS (IDMS)
- TurboIMAGE

NoSQL Database

Not Only SQL :

a type of database that is used for storing a wide range of data sets.

non-relational

presents a wide variety of database technologies in response to the demands.

Schema-less

relaxes one or more of the ACID properties(Will be explained in CAP theorem)

Motivation for NoSQL Database

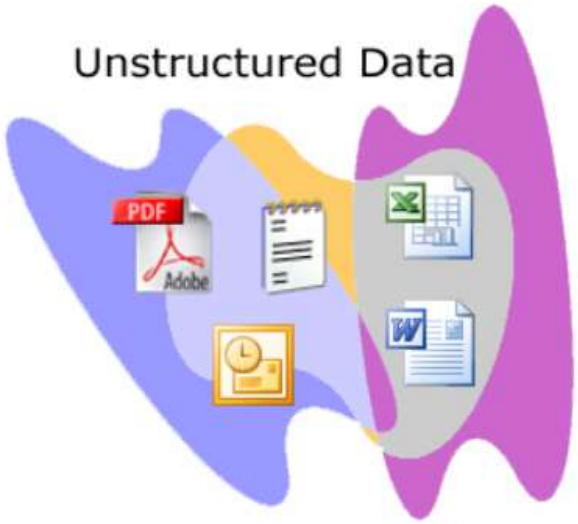
Limitations of relational databases

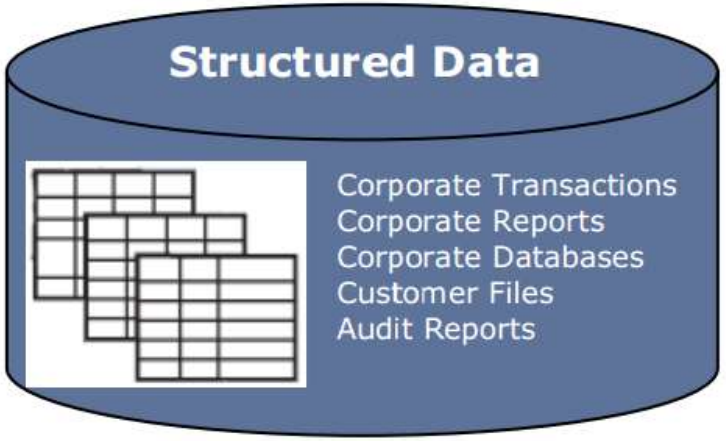
Not suitable for distributed applications because:

- Joins are expensive
- Hard to scale horizontally
- Can’t handle unstructured data
- Expensive : Product Cost ,Hardware maintenance.
- High availability not possible
- no partition tolerance
- speed is less

Structured vs Unstructured

- Structured systems are those where the activity of processing





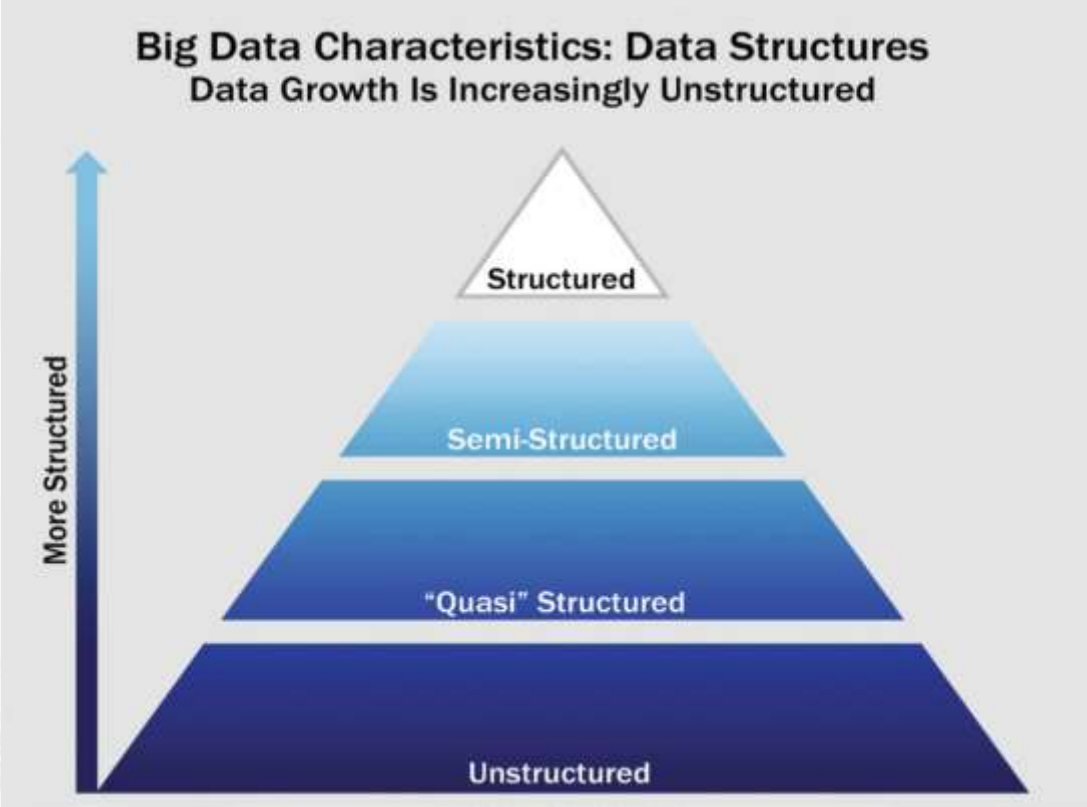
- The rules of unstructured system are fewer and less complex

Analyzing unstructured data requires processing of huge amounts of data. Which SQL databases are no good for, and were never designed for.

NoSQL databases have evolved to handle this huge data properly.

Data Structures

- Data comes in multiple forms - including structured, unstructured form
- 80–90% of future data growth coming from nonstructured data types



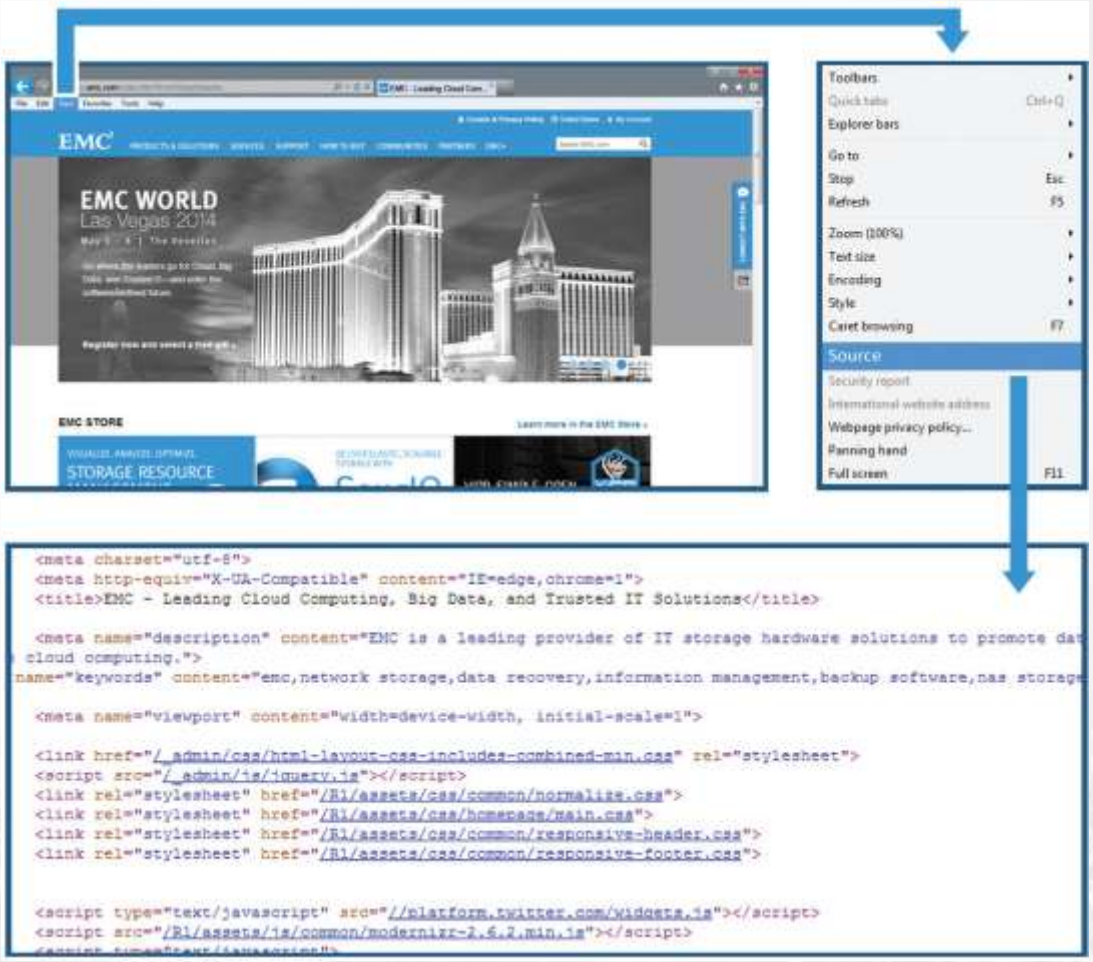
Data structures (contd..)

•Structured Data : Data containing a defined data type, format, and structure (that is, transaction data, online analytical processing [OLAP] data cubes, traditional RDBMS, CSV files, and even simple spreadsheets)

SUMMER FOOD SERVICE PROGRAM 1]				
(Data as of August 01, 2011)				
Fiscal Year	Number of Sites	Peak (July) Participation	Meals Served	Total Federal Expenditures 2]
	----------Thousands----------		--Mil.--	---Million $---
1969	1.2	99	2.2	0.3
1970	1.9	227	8.2	1.8
1971	3.2	569	29.0	8.2
1972	6.5	1,080	73.5	21.9
1973	11.2	1,437	65.4	26.6
1974	10.6	1,403	63.6	33.6
1975	12.0	1,785	84.3	50.3
1976	16.0	2,453	104.8	73.4
TQ 3]	22.4	3,455	198.0	88.9
1977	23.7	2,791	170.4	114.4
1978	22.4	2,333	120.3	100.3
1979	23.0	2,126	121.8	108.6
1980	21.6	1,922	108.2	110.1
1981	20.6	1,726	90.3	105.9
1982	14.4	1,397	68.2	87.1
1983	14.9	1,401	71.3	93.4
1984	15.1	1,422	73.8	96.2
1985	16.0	1,462	77.2	111.5
1986	16.1	1,509	77.1	114.7
1987	16.9	1,560	79.9	129.3
1988	17.2	1,577	80.3	133.3
1989	18.5	1,652	86.0	143.8
1990	19.2	1,692	91.2	163.3

Data Structures (contd..)

•Semi-structured data: Textual data files with a discernible pattern that enables parsing (such as Extensible Markup Language [XML] data files that are self-describing and defined by an XML schema)



Data Structures (contd..)

•Quasi-structured data: Textual data with erratic data formats that can be formatted with effort, tools, and time (for instance, web clickstream data that may contain inconsistencies in data values and formats).

Data Structures (contd..)

- Unstructured data: Data that has no inherent structure, which may include text documents, PDFs,images, and video.

Types of data in Big Data Scenario

- **Structured data :**
  - Is highly-organized and formatted in a way so it's easily searchable in relational databases.
  - Common relational database applications with structured data : airline reservation systems, inventory control, sales transactions, and ATM activity.

- **Unstructured data :**
  - Unstructured data has no pre-defined format or organization, making it much more difficult to collect, process, and analyze.
  - Unstructured data has internal structure but is not structured via pre-defined data models or schema.
  - It may be textual or non-textual, and human- or machine-generated.
  - It may also be stored within a non-relational database like NoSQL.

Types of data in Big Data Scenario

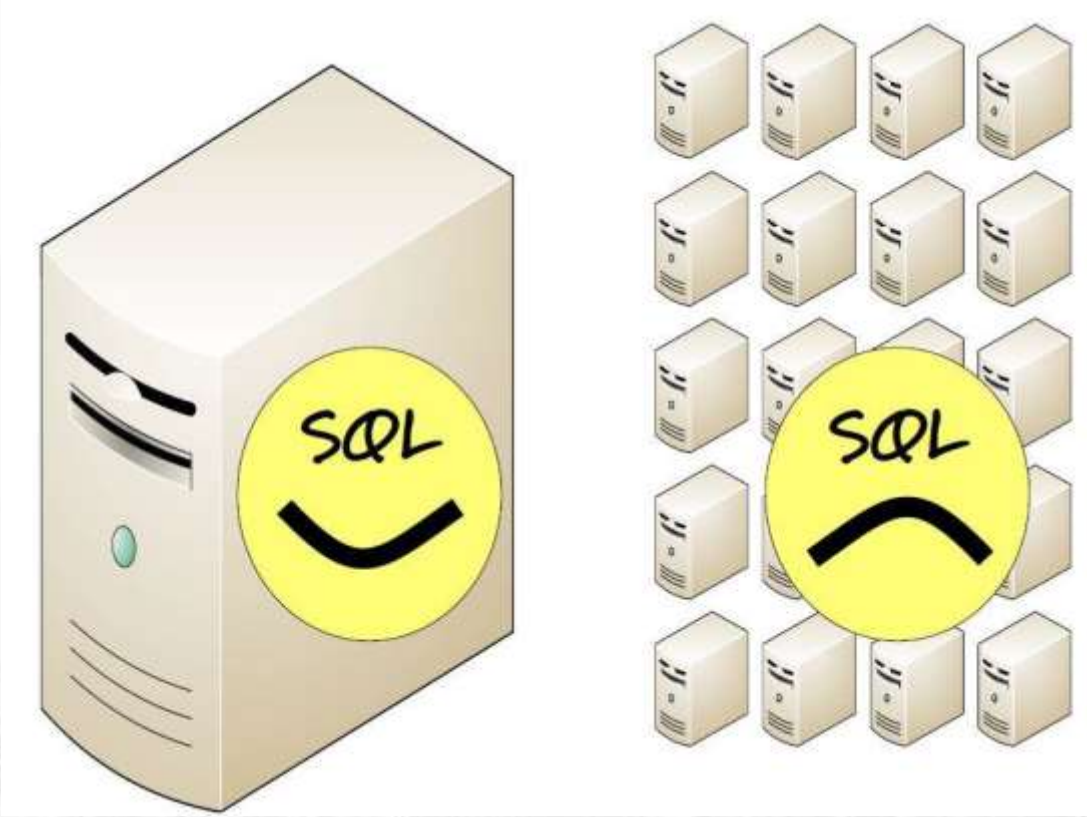
- **Semi-Structured data :**
  - maintains internal tags and markings that identify separate data elements, which enables information grouping and hierarchies.
  - Both documents and databases can be semi-structured.
  - This type of data only represents about 5-10% of the structured/semi-structured/unstructured data pie.
  - Typical use is in OO models
  - Examples : CSV,XML,JSON

Difference between Structured and Unstructured Data

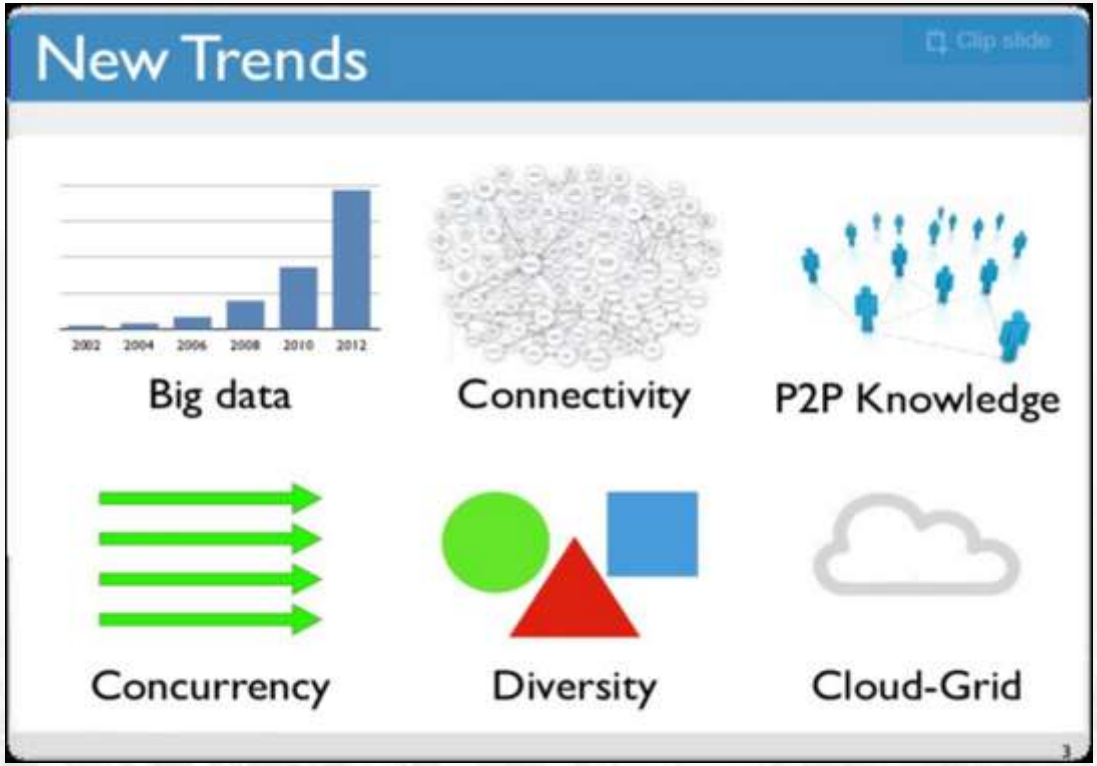
	Structured Data	Unstructured Data
Characteristics	• Pre-defined data models • Usually text only • Easy to search	• No pre-defined data model • May be text, images, sound, video or other formats • Difficult to search
Resides in	• Relational databases • Data warehouses	• Applications • NoSQL databases • Data warehouses • Data lakes
Generated by	Humans or machines	Humans or machines
Typical applications	• Airline reservation systems • Inventory control • CRM systems • ERP systems	• Word processing • Presentation software • Email clients • Tools for viewing or editing media
Examples	• Dates • Phone numbers • Social security numbers • Credit card numbers • Customer names • Addresses • Product names and numbers • Transaction information	• Text files • Reports • Email messages • Audio files • Video files • Images • Surveillance imagery

Motivation for NoSQL Database

- **Relational Databases are not suitable for distributed computing**

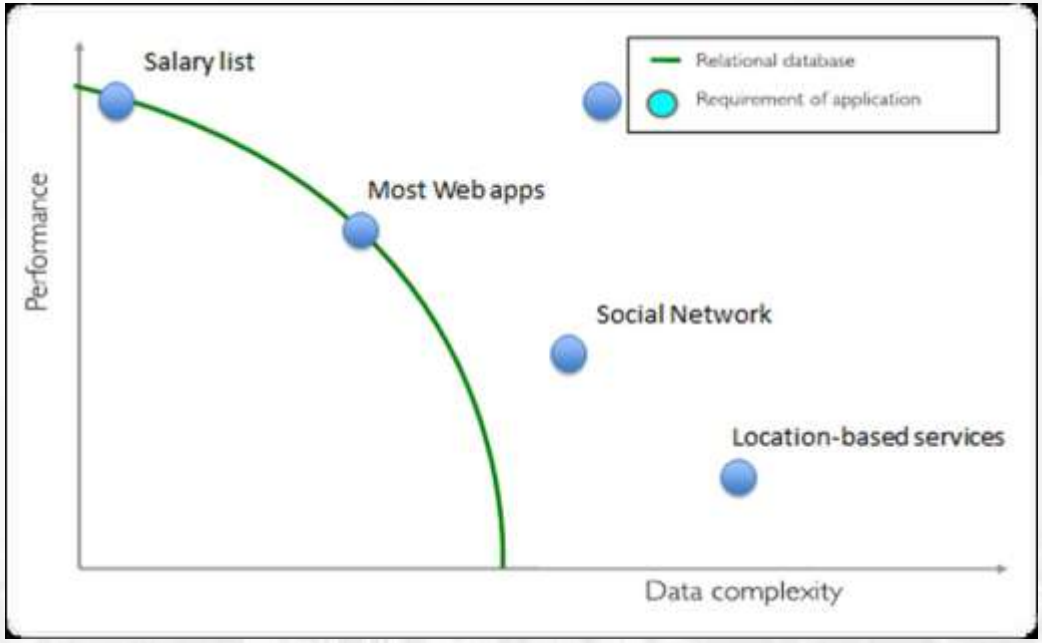


Motivation for NoSQL Database



Motivation for NoSQL Database

Performance of RDBMS for various applications



Addressing system growth

Every database has to be scaled to address the huge amount of data being generated each day.

This makes data available at all times for users.

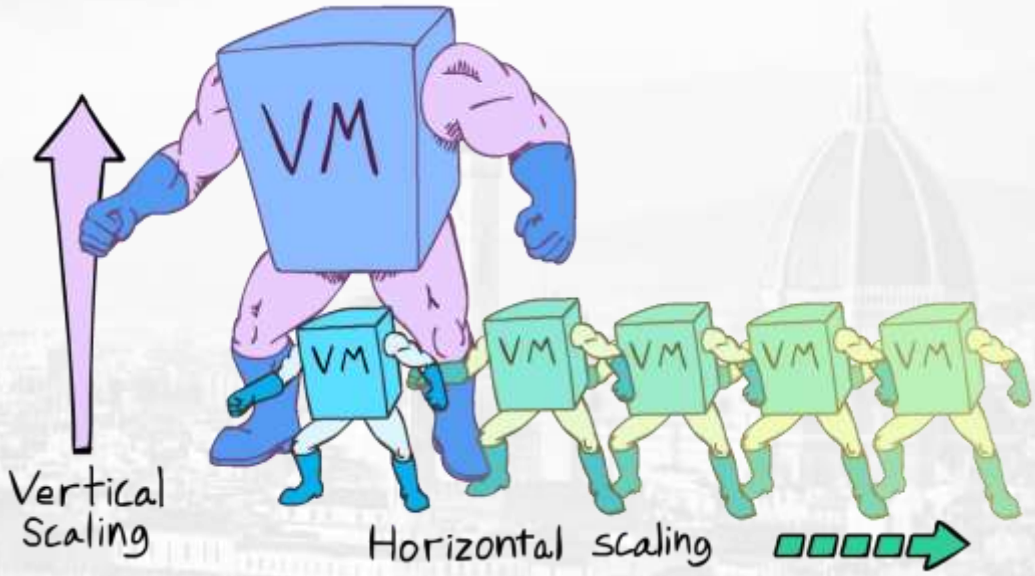
When the memory of the database is drained, or when it cannot handle multiple requests, it is not scalable.

Scalability : capability of a system, network, or process to handle a growing amount of work, or its potential to be enlarged to accommodate that growth.

Vertical Scaling and Horizontal Scaling

Scaling datbases

- *Elasticity : degree to which a system can adapt to workload changes by provisioning and de-provisioning resources in an on-demand manner, such that,*
- *At each point in time the available resources match the current demand as closely as possible*
- *Types of Scaling :*



Vertical Scaling

Vertical Scaling :adopted when the database couldn't handle the large amount of data.

- Example :Suppose you have a database server with 10GB memory and it has exhausted. Now, to handle more data, you buy an expensive server with memory of 2TB.
- it involves adding more power such as CPU and disk power to enhance your storage process.
- applicable to applications involving a limited range of users and minimal querying
- **Application** : **Relational databases** mostly use vertical scaling.

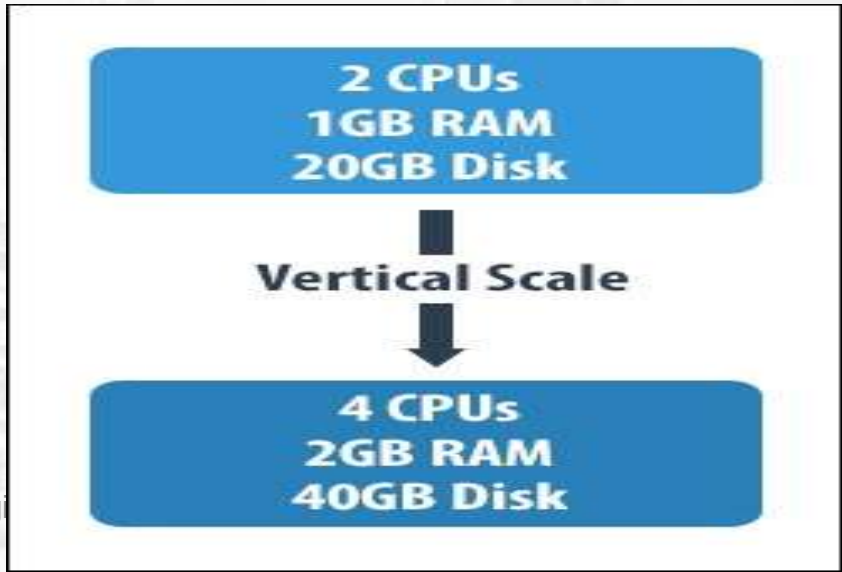
Vertical Scaling

Advantages :

- **Simple**, since everything exists in a single server. No need to manage multiple instances.
- **Performance Gain**, because you have faster RAM and memory power on each update.
- **Same Code. No change**—You need not change your implementation or your code at all.

Limitations :

- Difficult to perform multiple queries simultaneously.
- Chances of downtime are high, when the server exceeds maximum load.
- Expensive. Hardware resources are costly, after all.



Horizontal Scaling

Horizontal Scaling :scaling of the server horizontally by adding more machines.

- It divides the data set and distributes the data over multiple servers, or *shards.*
- Each shard is an independent database.
- Instead of buying a single 2 TB server, we are buying two hundred 10 GB servers.
  - If your application can allow redundancy and involves less joins, then you can use horizontal scaling.
- Applications : NoSQL databases mostly use horizontal scaling. It is less suitable for RDBMS as it relies on strict Consistency and Atomicity rules.

Horizontal Scaling

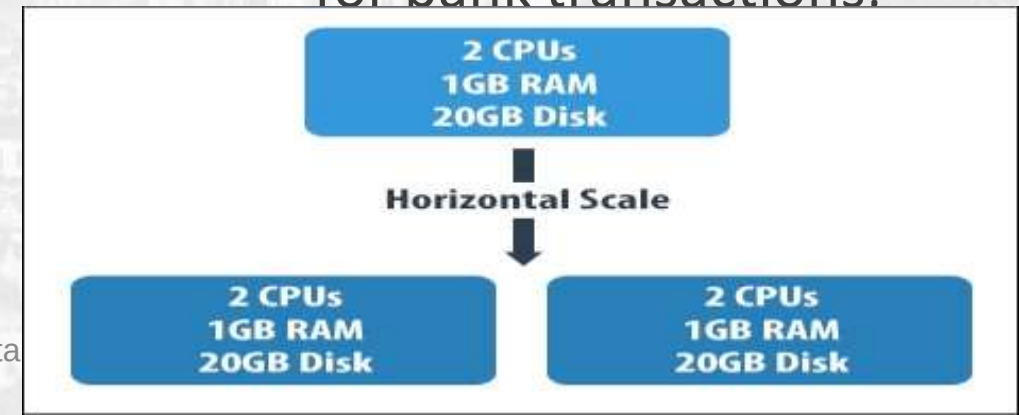
Advantages :

cheap compared to vertical scaling.

- Lesser Load, Better performance.
- Chances of downtime are less.
- Resilience and Fault Tolerance.
- Suitable for Distributed Databases

Limitations :

- Making joins is difficult, due to cross-server communication.
- Eventual consistency is only possible.
- It may not be best suited for bank transactions.



Horizontal and Vertical scaling

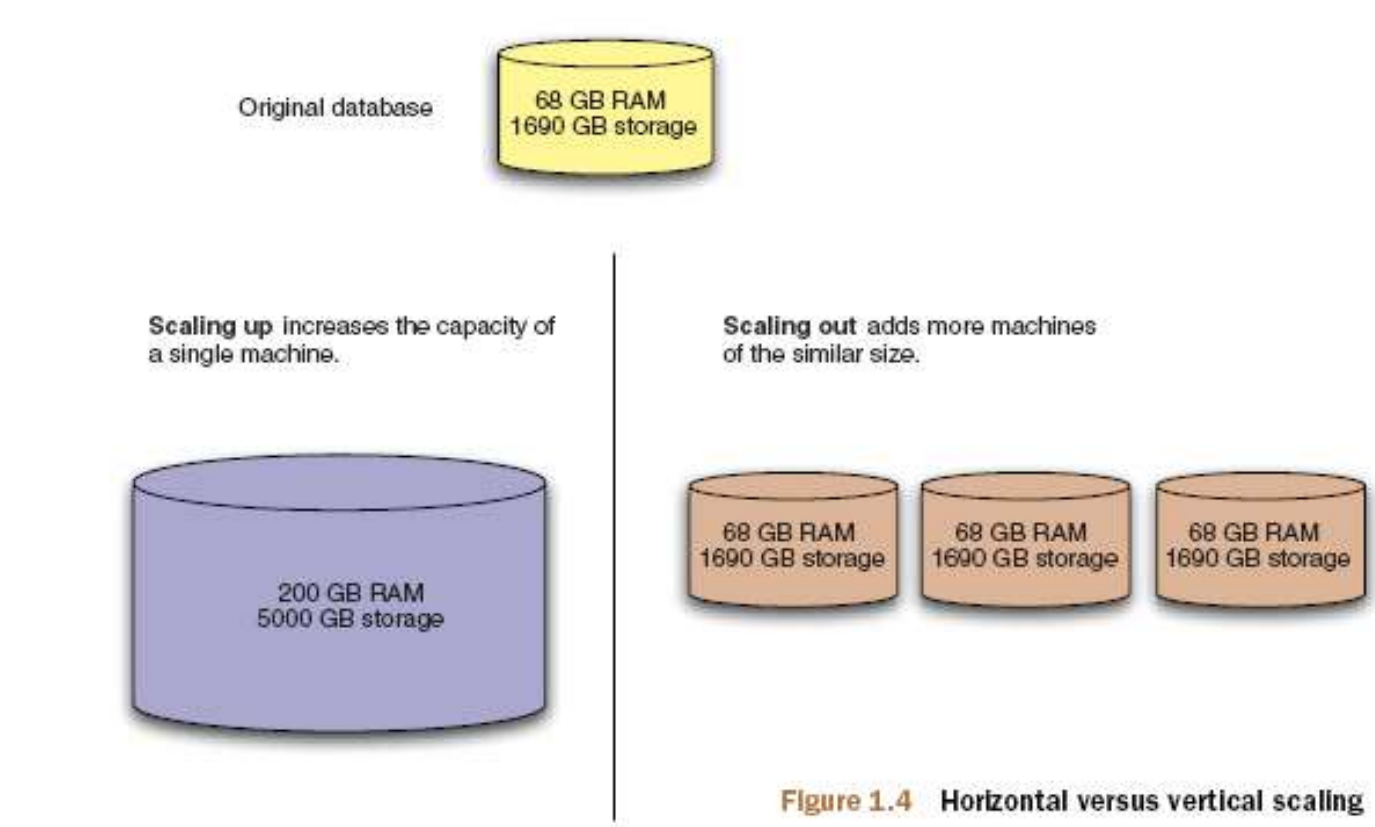
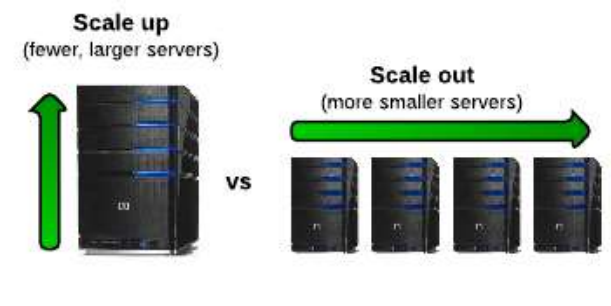


Figure 1.4 Horizontal versus vertical scaling

Horizontal and Vertical scaling

- Horizontal scaling means that you scale by adding more machines to your pool of resources.
- Often based on partitioning of the data i.e. splitting up the data so that each machine contains only part of the data.
- It is easier to scale dynamically by adding more machines into the existing pool.
- e.g. DynamoDB, Cassandra , MongoDB



- Vertical scaling means that you scale by adding more power (CPU, RAM, etc.) to your existing machine.
- Data resides on a single node and scaling is done through the use of multi-cores, etc. i.e. spreading the load between the CPU and RAM resources of that single machine.
- Often limited to the capacity of a single machine, scaling beyond that capacity often involves downtime and comes with an upper limit.
- e.g. MySQL

What is NoSQL?

NoSQL is a non-relational database management systems

Designed for distributed data stores where very large scale of data storing needs to be available

- e.g. Google or Facebook which collects terabits of data every day for their users

These data stores may not require fixed-table schemas, and usually avoid join operations and typically scale horizontally

"Non-relational" may be more accurate term than "NoSQL", as some NoSQL DBs do support a subset of SQL.

Characteristics of NoSQL Databases

- **NoSQL avoids :**
  - Overhead of ACID transactions
  - Complexity of SQL queries
  - Burden of schema-design
  - DBA presence
- **Provides :**
  - Easy and frequent changes to DB
  - Fast Development
  - Can handle large data volume (Big Data Applications)
  - Schema-less

When and when not to use NoSQL

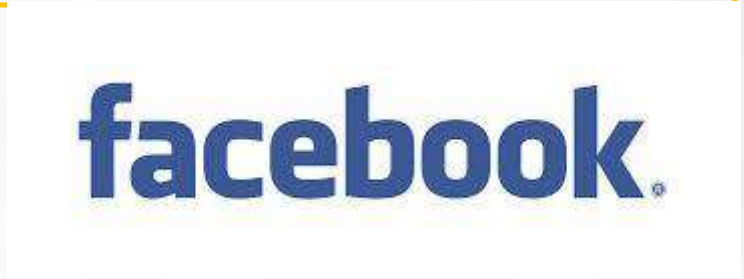
WHEN / WHY ?

- When traditional RDBMS model is too restrictive (flexible schema)
- When ACID support is not "really" needed
- Object-to-Relational (O/R) impedance
- Because RDBMS is neither distributed nor scalable by nature
- Logging data from distributed sources
- Storing Events / temporal data
- Temporary Data (Shopping Carts / Wish lists / Session Data)
- Data which requires flexible schema
- **Polyglot Persistence** i.e. best data store depending on nature of data.

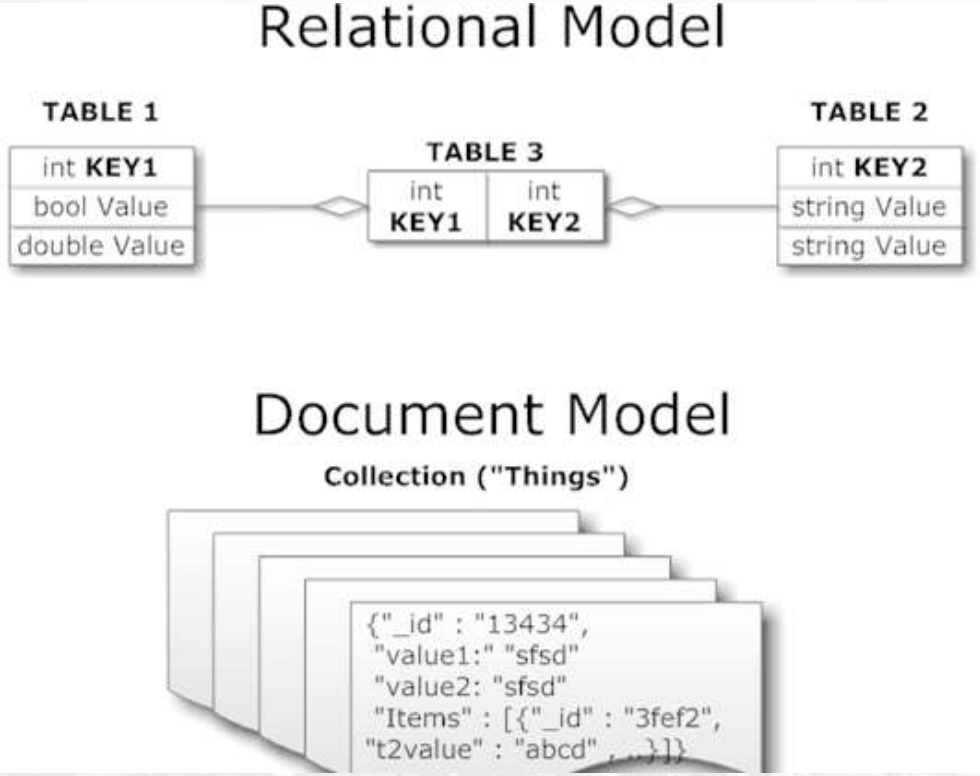
WHEN NOT ?

- Financial Data
- Data requiring strict ACID compliance
- Business Critical Data

NoSQL : Applications and Popularity



Schema-less Data Model



- No fixed schema to consider
- No implicit datatypes
- Most considerations done at application layer including transactions
- All aggregate data is gathered in documents.

SQL vs. NoSQL

	SQL Databases	NoSQL Databases
Data Manipulation	Specific language using Select, Insert, and Update statements, e.g. SELECT fields FROM table WHERE…	Through object-oriented APIs
Consistency	Can be configured for strong consistency	Depends on product. Some provide strong consistency (e.g., MongoDB) whereas others offer eventual consistency (e.g., Cassandra)
Scaling	Vertically, meaning a single server must be made increasingly powerful in order to deal with increased demand. It is possible to spread SQL databases over many servers, but significant additional engineering is generally required.	Horizontally, meaning that to add capacity, a database administrator can simply add more commodity servers or cloud instances. The database automatically spreads data across servers as necessary
Development Model	Mix of open-source (e.g., Postgres, MySQL) and closed source (e.g., Oracle Database)	Open-source
Supports Transactions	Yes, updates can be configured to complete entirely or not at all	In certain circumstances and at certain levels (e.g., document level vs. database level)

	SQL Databases	NoSQL Databases
Types	One type (SQL database) with minor variations	Many different types including key-value store, document databases, wide-column stores, and graph databases
Development History	Developed in 1970s to deal with first wave of data storage applications	Developed in 2000s to deal with limitations of SQL databases, particularly concerning scale, replication and unstructured data storage
Examples	MySQL, Postgres, Oracle Database	MongoDB, Cassandra, HBase, Neo4j
Data Storage Model	Relational model,ER Model, Network Model Individual records (e.g., "employees") are stored as rows in tables, with each column storing a specific piece of data about that record (e.g., "manager," "date hired," etc.), much like a spreadsheet. Separate data types are stored in separate tables, and then joined together when more complex queries are executed.	Key-Value, Column based, Document Based.. Varies based on database type. For example, key-value stores function similarly to SQL databases, but have only two columns ("key" and "value"), with more complex information sometimes stored within the "value" columns. Document databases do away with the table-and-row model altogether, storing all relevant data together in single "document" in JSON, XML, or another format, which can nest values hierarchically.

CAP – NoSQL Data models

Three basic requirements which exist in a special relation when designing applications for a distributed architecture.

- Consistency : the data in the database remains consistent after the execution of an operation.
- Availability : the system is always on (Service guarantee availability), no downtime
- Partition Tolerance : the system continues to function even if the communication among the servers is unreliable, i.e. the servers may be partitioned into multiple groups that cannot communicate with one another.

Generally, it may not be possible to fulfil all three requirements in a distributed system. CAP provides the basic requirements for a distributed system to follow two of the three requirements.

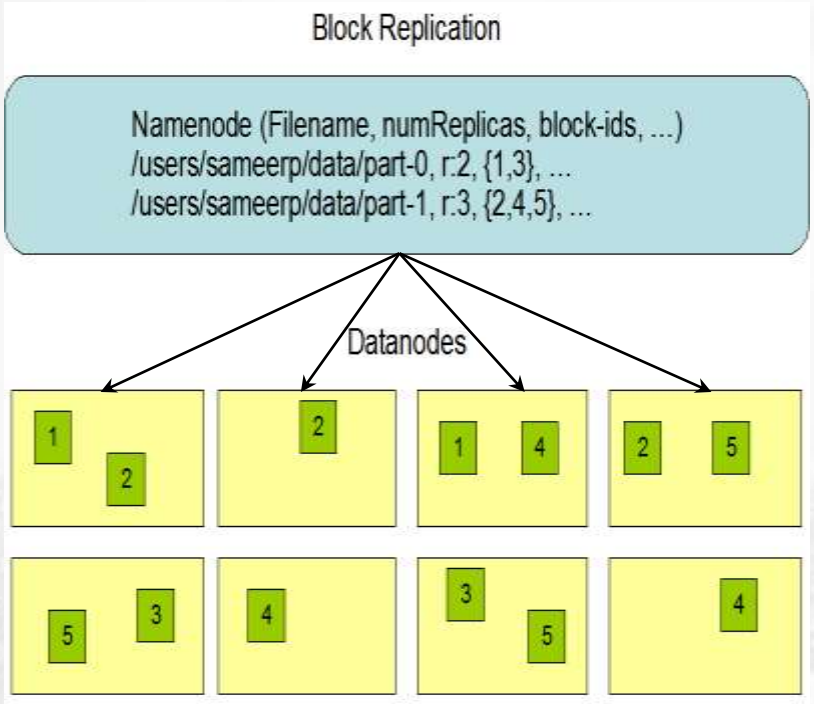
Distributed systems must be partition tolerant (P), so Current NoSQL databases follow the different combinations of C and A from the CAP theorem.

CAP Theorem

CAP Theorem

- **C**onsistency
  - All the servers in the system will have the same data so anyone using the system will get the same copy regardless of which server answers their request.
- **A**vailability
  - The system will always respond to a request (even if it's not the latest data or consistent across the system or just a message saying the system isn't working)
- **P**artition Tolerance
  - The system continues to operate as a whole even if individual servers fail or can't be reached..

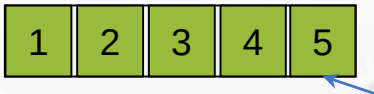
Hadoop Distributed File System (HDFS)



Centralized namenode

- Maintains metadata info about files

File *F*

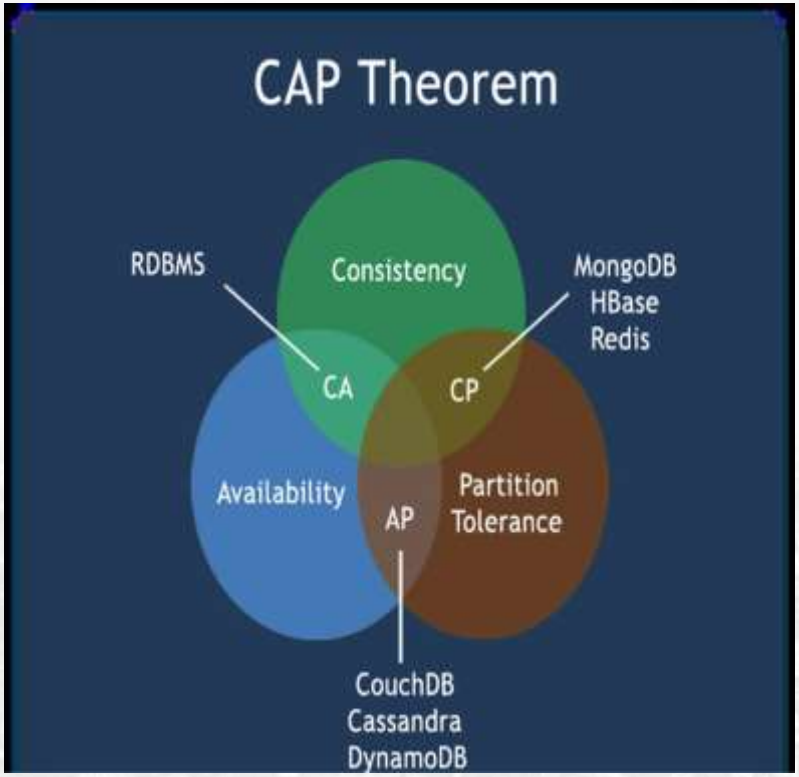


Blocks (64 MB)

Many datanode (1000s)

- Store the actual data
- Files are divided into blocks
- Each block is replicated *N* times (Default = 3)

CAP Theorem



- Acronym for Consistency, Availability and Partition Tolerance
  - ***Consistency*** : once data is written ,all future read requests will contain that data
  - ***Availability*** : database is always available and responsive.
  - ***Partition Tolerance*** : if part of database is unavailable ,other parts are not affected.

CAP Theorem

CA : Consistent and Available

Examples : standalone Mysql server/node which has no replication.

It provides consistency and availability till it goes down.

Applications : Bank Account Balance,Text messages which require higher consistency.RDBMS are CA systems.

AP : Available and Partition Tolerant

Example : Distributed NoSQL database where replication to nodes happens asynchronously.

system will always respond, but not all the nodes will have the latest version of the data when queried

Applications :E Commerce Sites which focus on high availability in case of partitions in distributed environment by trading off consistency..

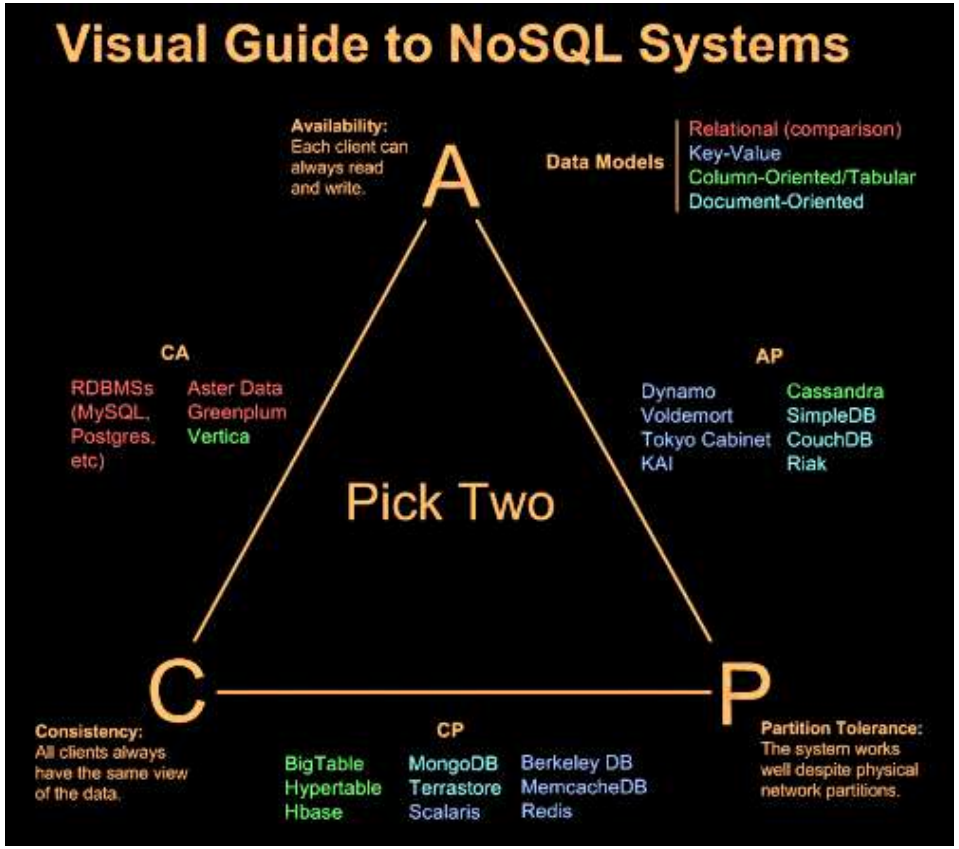
CP : Consistent and Partition Tolerant

Similar to CA systems but difference is its applicability to distributed environment.

In Mongodb the primary node is replicated into secondary nodes.If the primary node fails then system switches to secondary node.During this switch data is not made available to user.

CAP – NoSQL Datamodels

- CA - Single site cluster, therefore all nodes are always in contact. When a partition occurs, the system blocks.
- CP - Some data may not be accessible, but the rest is still consistent/accurate.
- AP - System is still available under partitioning, but some of the data returned may be inaccurate.



CAP Theorem

No distributed system is safe from network failures, thus network partitioning generally has to be tolerated.

In the presence of a partition, one is then left with two options: **consistency or availability.**

When choosing **consistency over availability:** the system will return an error or a time out if particular information cannot be guaranteed to be up to date due to network partitioning.

When choosing **availability over consistency:** the system will always process the query and try to return the most recent available version of the information, even if it cannot guarantee it is up to date due to network partitioning.

CAP Theorem

In the absence of network failure – that is, when the distributed system is running normally – both availability and consistency can be satisfied.

CAP is frequently misunderstood as if one has to choose to abandon one of the three guarantees at all times. In fact, the choice is really between **consistency and availability only when a network partition or failure happens**; at all other times, no trade-off has to be made.

CAP Theorem

A network partition refers to network decomposition into relatively independent subnets for their separate optimization as well as network split due to the failure of network devices.In both cases the partition-tolerant behavior of subnets is expected.

Partition Tolerance or robustness means that a given system continues to operate even with data loss or node failure.

A single node failure should not cause system failure.

CAP Theorem

- Database systems designed with traditional **ACID** guarantees in mind such as **RDBMS choose consistency over availability,**

  Systems designed around the **BASE** philosophy, common in the NoSQL movement for example, **choose availability over consistency.**

- [[**Network partition** refers to network decomposition into relatively independent subnets for their separate optimization as well as network split due to failure of network devices]]

Limitations of CAP theorem

- When companies such as Google and Amazon were designing large-scale databases, 24/7 Availability was a key
  - A few minutes of downtime means lost revenue
- When *horizontally* scaling databases to 1000s of machines, the likelihood of a node or a network failure increases tremendously
- Therefore, in order to have strong guarantees on Availability and Partition Tolerance, they had to sacrifice "strict" Consistency (*implied by the CAP theorem*)
- **The CAP theorem proves that it is impossible to guarantee strict Consistency and Availability while being able to tolerate network partitions**
- **This resulted in databases with relaxed ACID guarantees**

Trading-Off Consistency

- Maintaining consistency should balance between the strictness of consistency versus availability/scalability
  - Good-enough consistency *depends on your application*

Easier to implement, and is efficient

Generally hard to implement, and is inefficient

The BASE Properties

- To overcome the limitations of CAP theorem the BASE properties were introduced and implemented by such databases as follows :
- **Basically Available:** the system guarantees Availability
- **Soft-State**: the state of the system may change over time
- **Eventual Consistency: t**he system will *eventually* become consistent

The BASE Properties

- **Horizontally scalable Databases use BASE properties**

  Basically Available: the system guarantees Availability

- BASE databases spread data across many storage systems with a high degree of replication which guarantees
- In the unlikely event that a failure disrupts access to a segment of data, this does not necessarily result in a complete database outage.

  Soft-State: the state of the system may change over time

- Values stored in the system may change because of the eventual consistency model

  Eventual Consistency: the system will *eventually* become consistent

  As data is added to the system, the system's state is gradually replicated across all nodes, during the short period of time before all updated blocks are replicated, the state of the file system isn't consistent

  By sacrificing Permanent Consistency in favor of Eventual Consistency developers enable Horizontal Scalability.

BASE Transactions

Acronym contrived to be the opposite of ACID

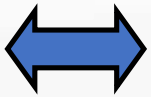
- **B**asically **A**vailable,
- **S**oft state,
- **E**ventually Consistent

Characteristics

- Weak consistency – stale data OK
- Availability first
- Best effort
- Simpler and faster

ACID - BASE

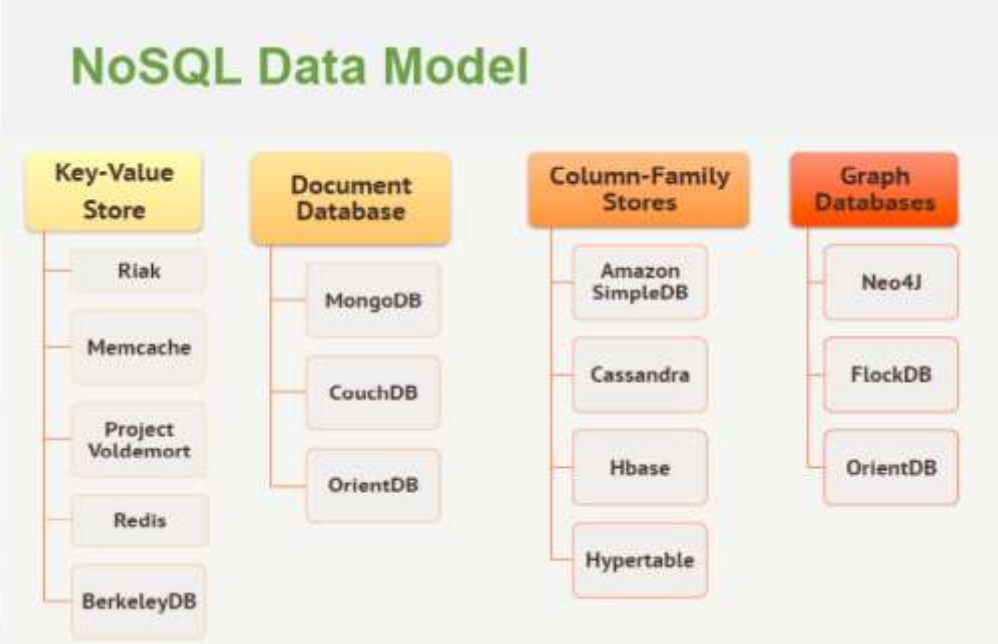
- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability



- **B**asically **A**vailable (CP)
- **S**oft-state
- **E**ventually consistent (AP)

Data Models and Types of NoSQL DBs

NoSQL Data Model



Types of NoSQL databases

Distributed Key-Value Systems - Lookup a single value for a key

- Amazon's Dynamo

Document-based Systems - Access data by key or by search of "document" data.

- CouchDB
- MongoDB

Column-based Systems

- Google's BigTable
- Facebook's Cassandra

Graph-based Systems - Use a graph structure

- Google's Pregel
- Neo4j

NoSQL
Database Types

Key-Value Pair (KVP) Stores

Access data (values) by strings called keys.

Data has no required format – data may have any format

Extremely simple interface

- Data model: (key, value) pairs
- Basic Operations: Insert(key,value), Fetch(key), Update(key), Delete(key)

Implementation: efficiency, scalability, fault-tolerance

- Records distributed to nodes based on key
- Replication
- Single-record transactions, “eventual consistency”

Example systems

- Amazon Dynamo

“Value” is stored as a “blob”

- **Without caring or knowing what is inside**
- **Application is responsible for understanding the data**

In simple terms, a NoSQL Key-Value store is a single table with two columns: one being the (Primary) Key, and the other being the Value.

Example of unstructured data for user records

Key: 1	ID: sj	First Name: Sam	
Key: 2	Email: jb@gmail.com	Location: London	Age: 37
Key: 3	Facebook ID: jkirk	Password: xxx	Name: James

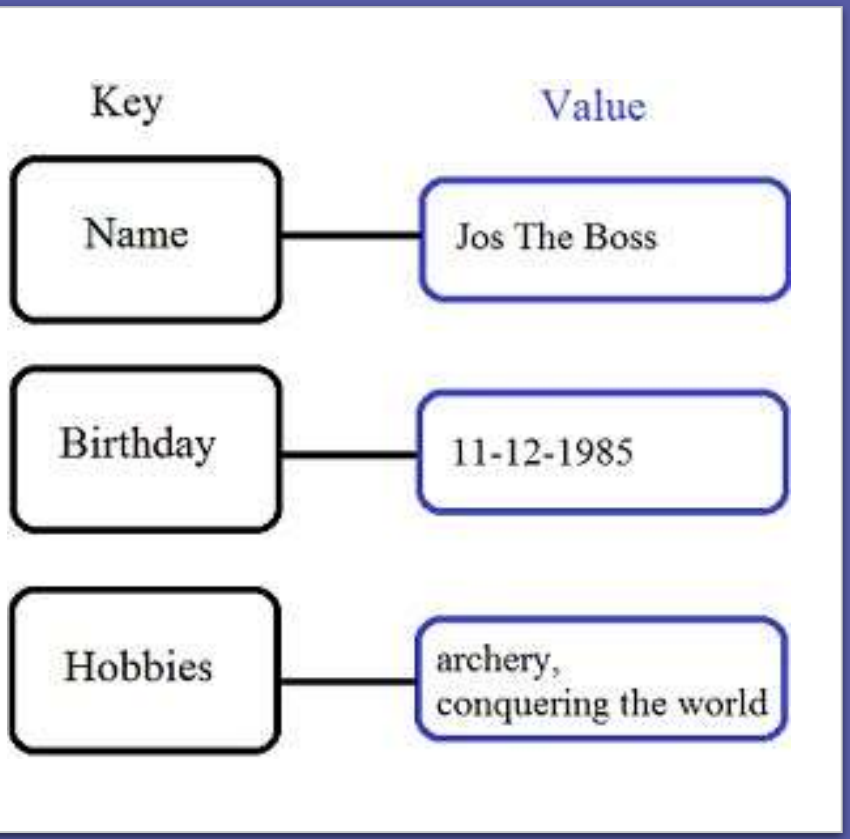
Each record may have a different schema

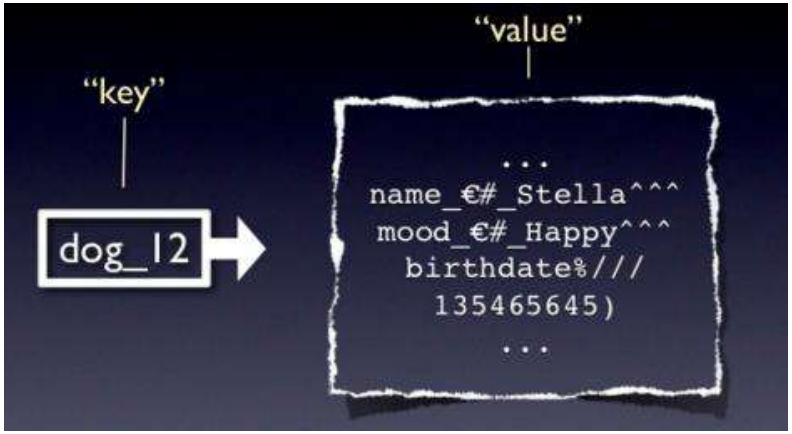
NoSQL Database Types:

Key value stores

- simplest type of database storage
- stores single item as a key (or attribute name) holding its value, together.
- Like a hash
- Does not require a specific data format.It can be any.
- Data Model : (Key + Value) pairs.
- Basic Operations : insert(key,value),Fetch(key),Update(key),Delete(key)
- **Examples** : Amazon DynamoDB,redis,riak.

Key-Value Store





Column-based Data Model

- **Column Family :**
- Column is the smallest instance of data.
- It is a row containing name,value and timestamp.
- Examples : Apache Cassandra used by Facebook

ColumnFamily: Authors	
Key	Value
"Eric Long"	Columns: Name = Value "email" = "eric (at) long.com" "country" = "United Kingdom" "registeredSince" = "01/01/2002"
"John Steward"	Columns: Name = Value "email" = "john.steward (at) somedomain.com" "country" = "Australia" "registeredSince" = "01/01/2009"
"Ronald Mathies"	Columns: Name = Value "email" = "ronald (at) sodeso.nl" "country" = "Netherlands, The" "registeredSince" = "01/01/2010"

Column-based Data Model

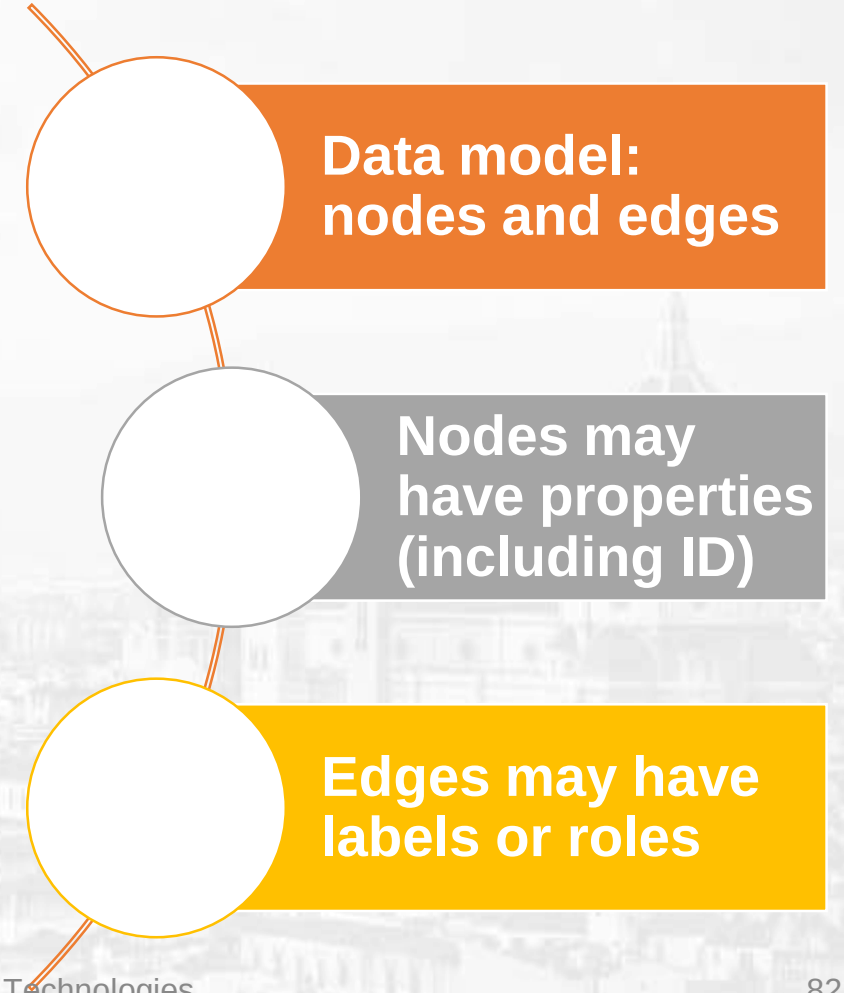
This type of data store is good for

(1) Distributed data storage, especially versioned data because of the time-stamps.

(2) Large-scale, batch-oriented data processing: sorting, parsing, conversion, algorithmic crunching, etc.

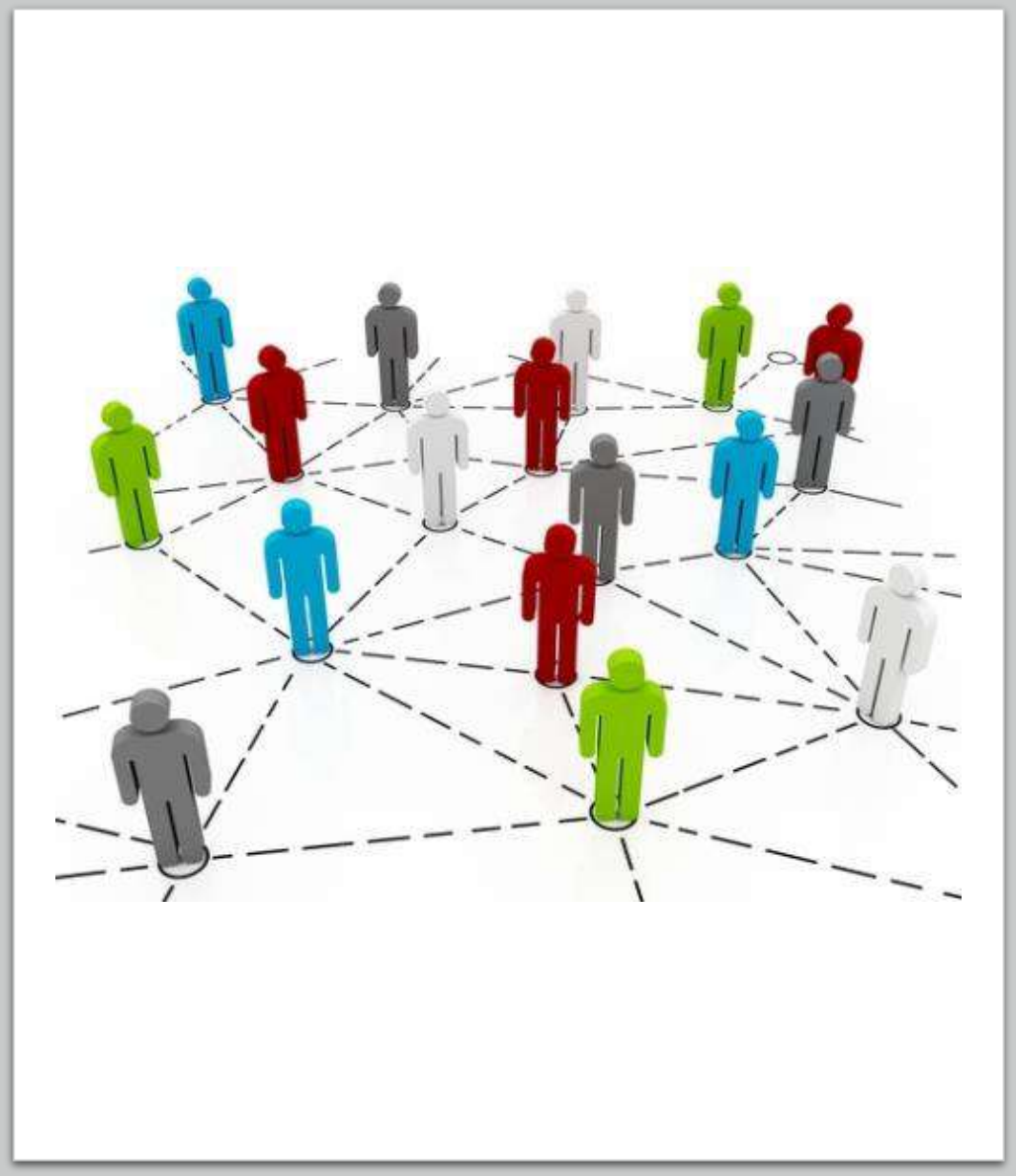
(3) Exploratory and predictive analytics – Business Intelligence.

Graph Database Systems

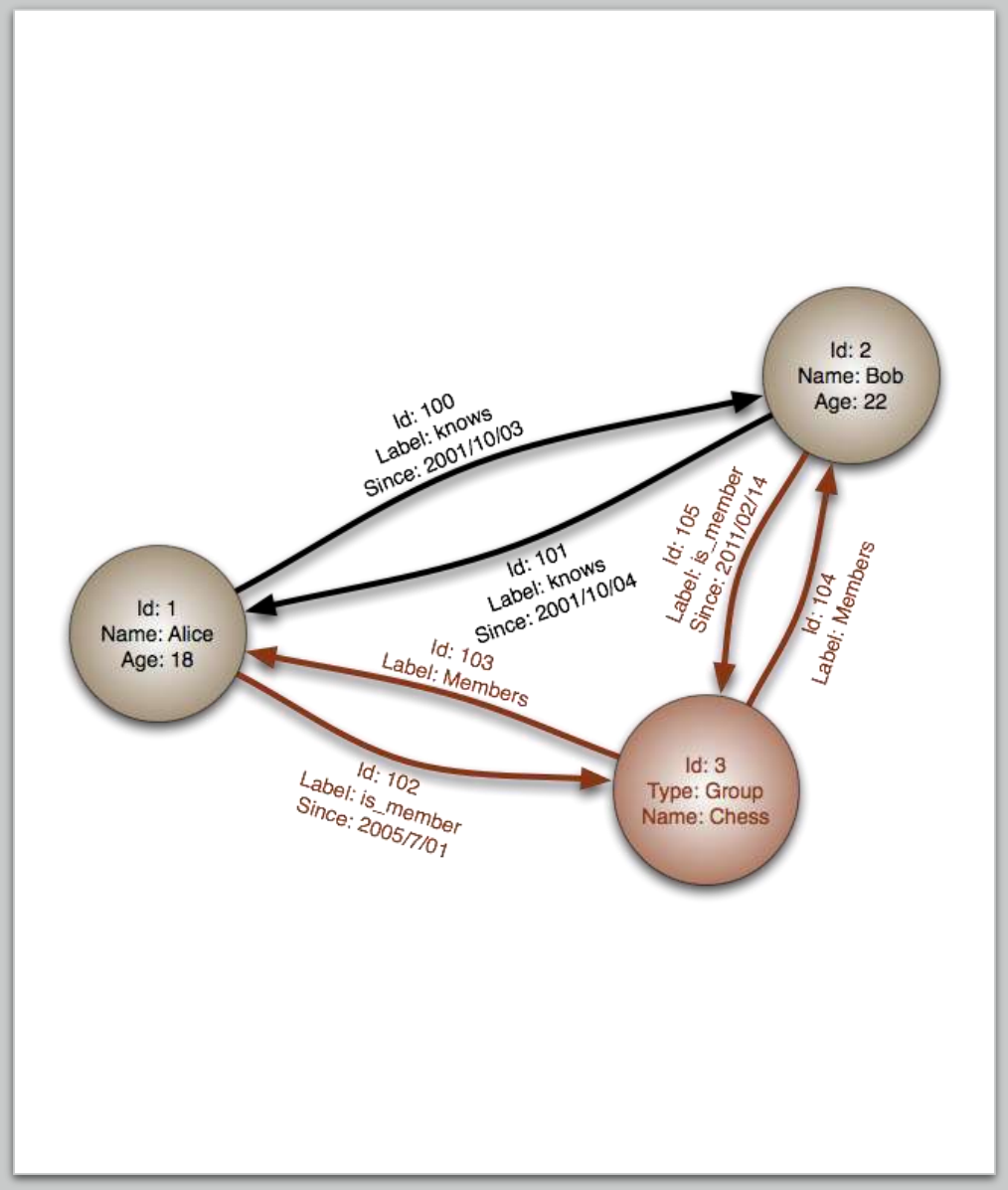


Graph Databases

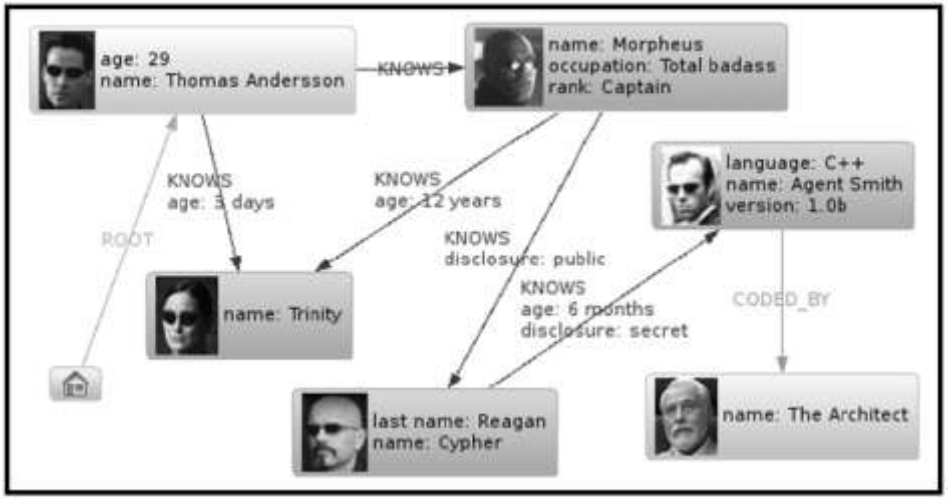
- **Based on graph theory**
- Vertical Scaling
- No clustering
- Transactions exist
- ACID followed
- Examples :Neo4j,Amazon Neptune, OrientDB, Dgraph.



Graph Databases



- **In general, graph databases are useful when you are more interested in relationships between data than in the data itself: for example, in representing and traversing social networks, generating recommendations, or conducting forensic investigations (e.g. pattern detection).**



Document-oriented Database:

used to store data as JSON-like document.

helps developers in storing data by using the same document-model format as used in the application code.

Examples : MongoDB,CouchDB

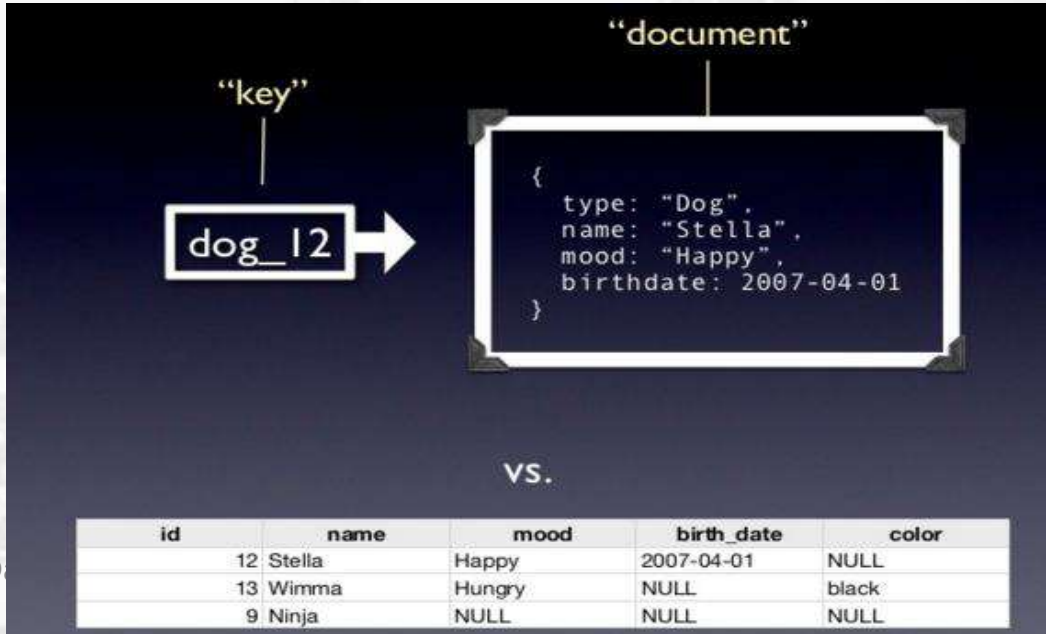
JSON : acronym for JavaScript Object Notation

an open-standard **file** format or data interchange format that uses human-readable text to transmit data objects consisting of attribute–value pairs and array data types.

Document-oriented Database:

- Pair each key with complex data structure.
- Indexing : using B-Trees
- Data stored in a format of documents which may contain multiple different key-value pairs or even nested documents.

```
{
  person: {
    first_name: "Peter",
    last_name: "Peterson",
    addresses: [
      {street: "123 Peter St"},
      {street: "504 Not Peter St"}
    ],
  }
}
```



Document databases are good for storing and managing Big Data-size collections of literal documents, like text documents, email messages, and XML documents, as well as conceptual "documents" like de-normalized (aggregate) representations of a database entity such as a product or customer.

They are also good for storing "sparse" data in general, that is to say irregular (semi-structured) data that would require an extensive use of "nulls" in an RDBMS.

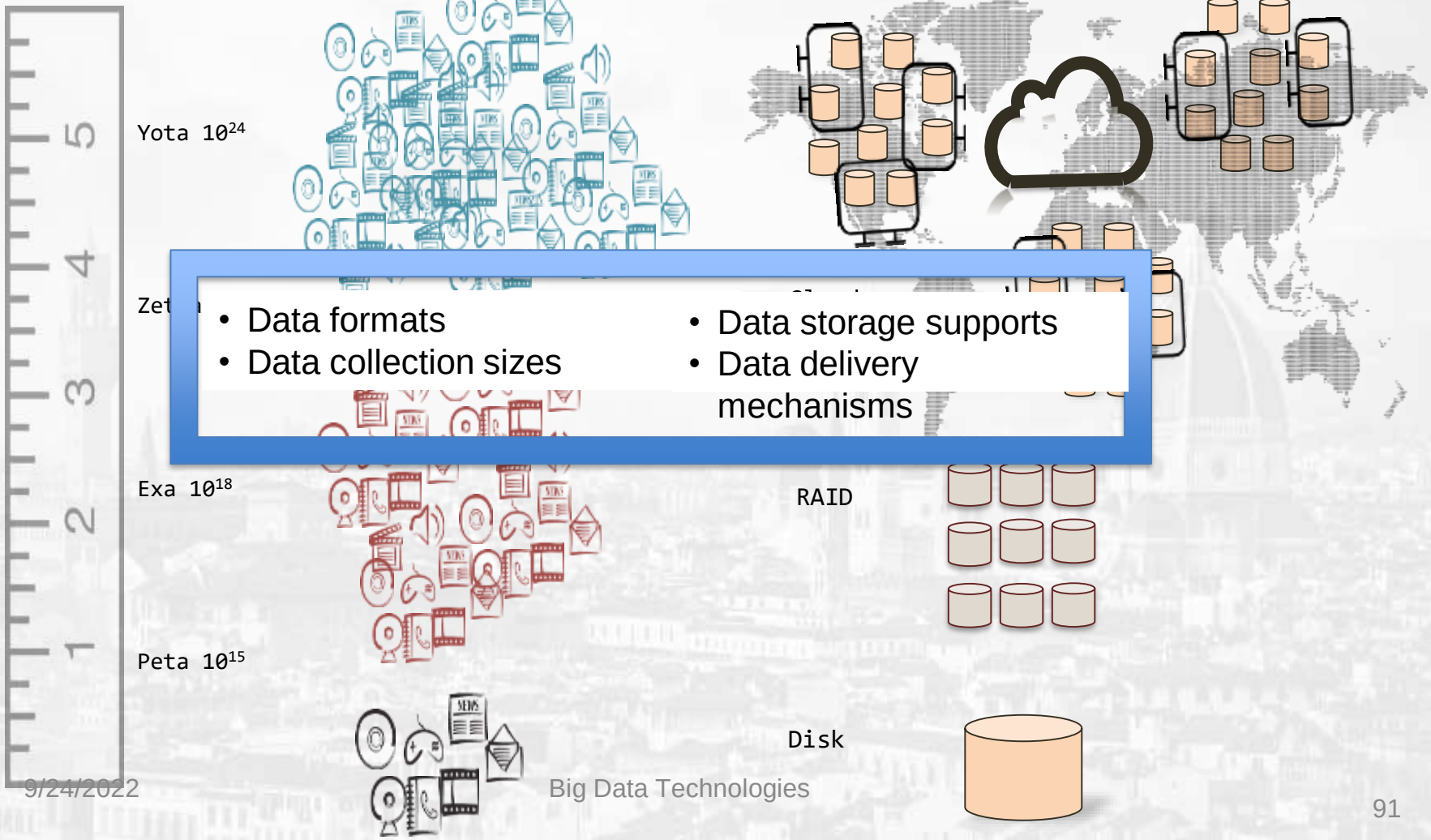
Choosing NoSQL database

- Key-value databases are generally useful for storing session information, user profiles, preferences, shopping cart data. We would avoid using Key-value databases when we need to query by data, have relationships between the data being stored or we need to operate on multiple keys at the same time.
- Document databases are generally useful for content management systems, blogging platforms, web analytics, real-time analytics, ecommerce-applications. We would avoid using document databases for systems that need complex transactions spanning multiple operations or queries against varying aggregate structures.
- Column family databases are generally useful for content management systems, blogging platforms, maintaining counters, expiring usage, heavy write volume such as log aggregation. We would avoid using column family databases for systems that are in early development, changing query patterns.
- Graph databases are very well suited to problem spaces where we have connected data, such as social networks, spatial data, routing information for goods and money, recommendation engines

Advantages of NoSQL Database

- enables good productivity in the application development as it is not required to store data in a structured format.
- is a better option for managing and handling large data sets.
- provides high scalability.
- Users can quickly access data from the database through key-value.
- designed for use with low-cost commodity hardware.
- Massive volumes of data (Big Data) are easily handled by NoSQL databases.
- **Economy**: can be easily installed in cheap commodity hardware clusters as transaction and data volumes increase. This means that you can process and store more data at much less cost.
- **Dynamic schemas**: NoSQL databases need no schemas to start working with data.

Storing and accessing huge amounts of data



Dealing with huge amounts of data

Yota 10^{24}

Zetta 10^{21}

Exa 10^{18}

Peta 10^{15}

Relational

Graph

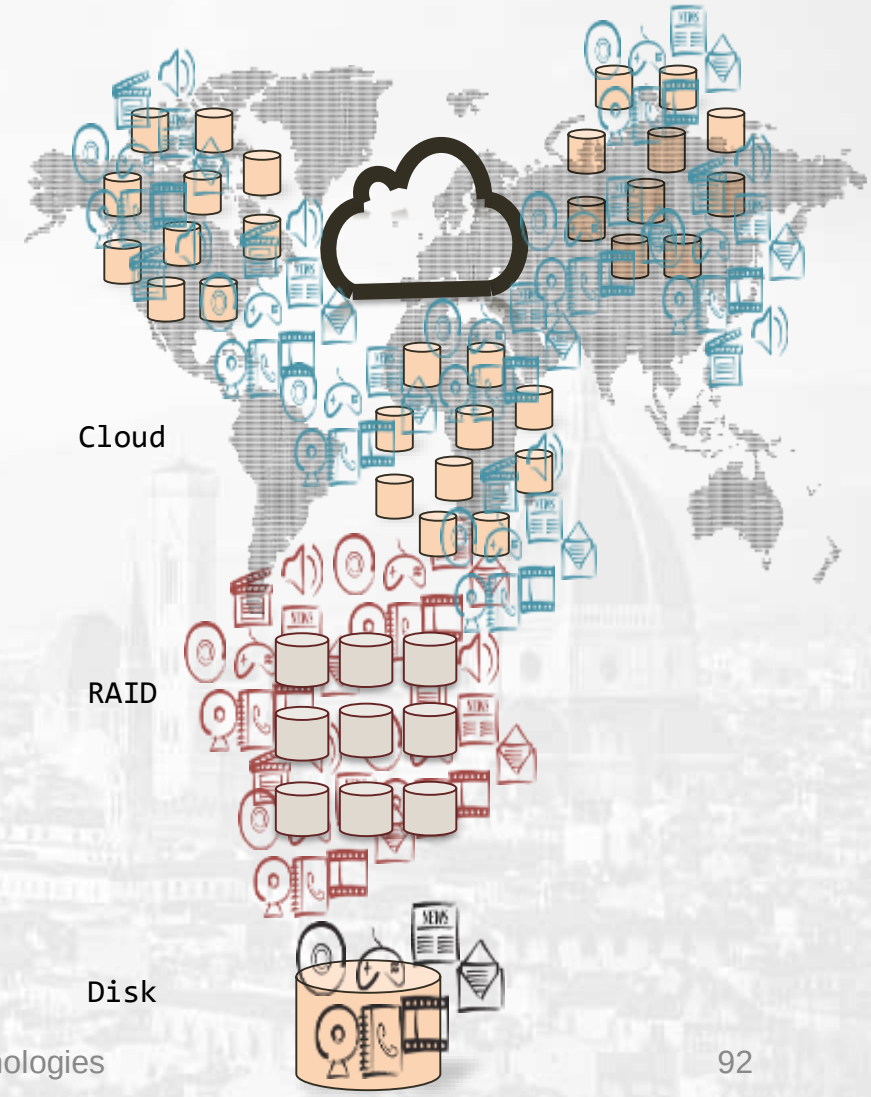
Key value

Columns

Concurrency

Consistency

Atomicity



Cloud

RAID

Disk

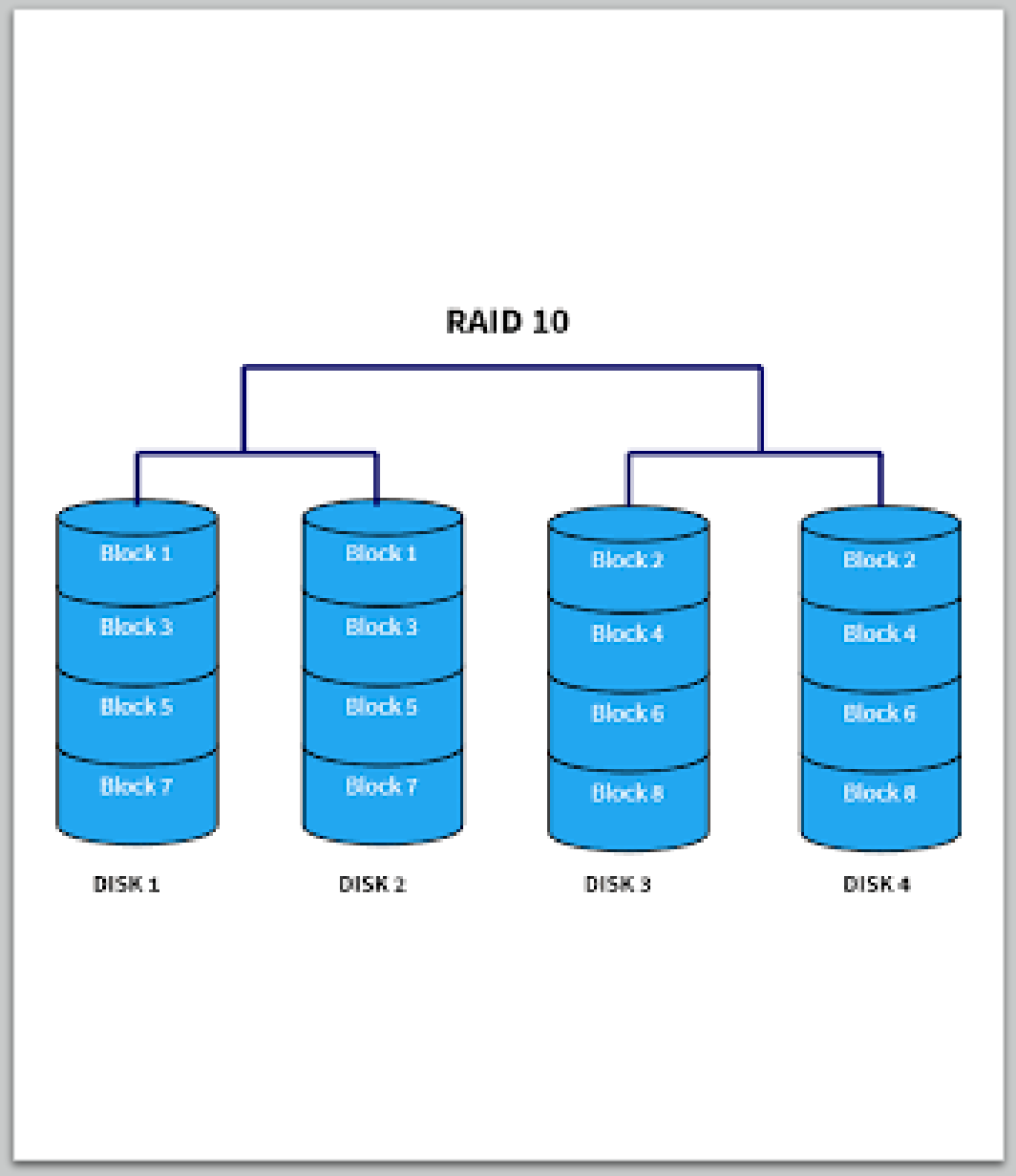
RAID

a data storage virtualization technology that combines multiple physical disk drive components into one or more logical units for the purposes of data redundancy, performance improvement, or both.

provides fault tolerance and optimizing performance.

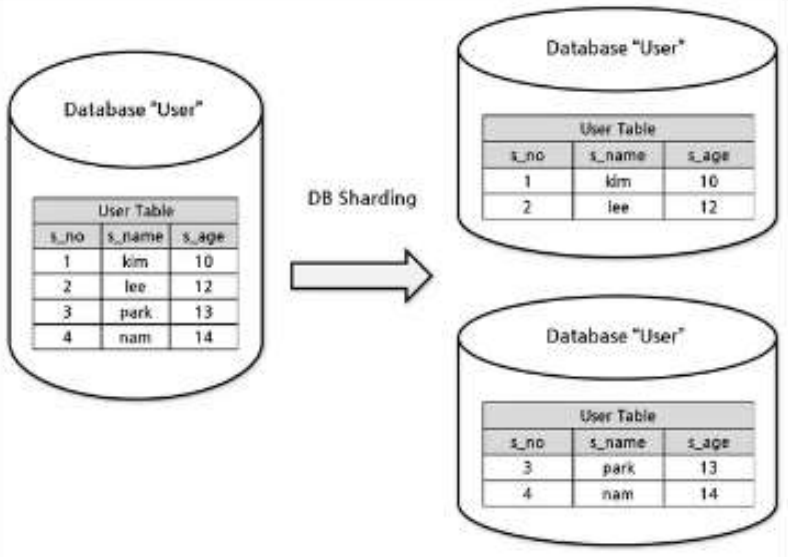
Synonym for : Redundant Array of Independent Disks

RAID


RAID 10
Block 1
Block 3
Block 5
Block 7
Block 1
Block 3
Block 5
Block 7
Block 2
Block 4
Block 6
Block 8
Block 2
Block 4
Block 6
Block 8
DISK 1
DISK 2
DISK 3
DISK 4

Database Sharding

- Database Sharding is the process where a huge Database is partitioned horizontally.
- This means that the attributes of the Database will remain the same but only the records will change.
- So the data in each partition is unique but the schema remains the same.



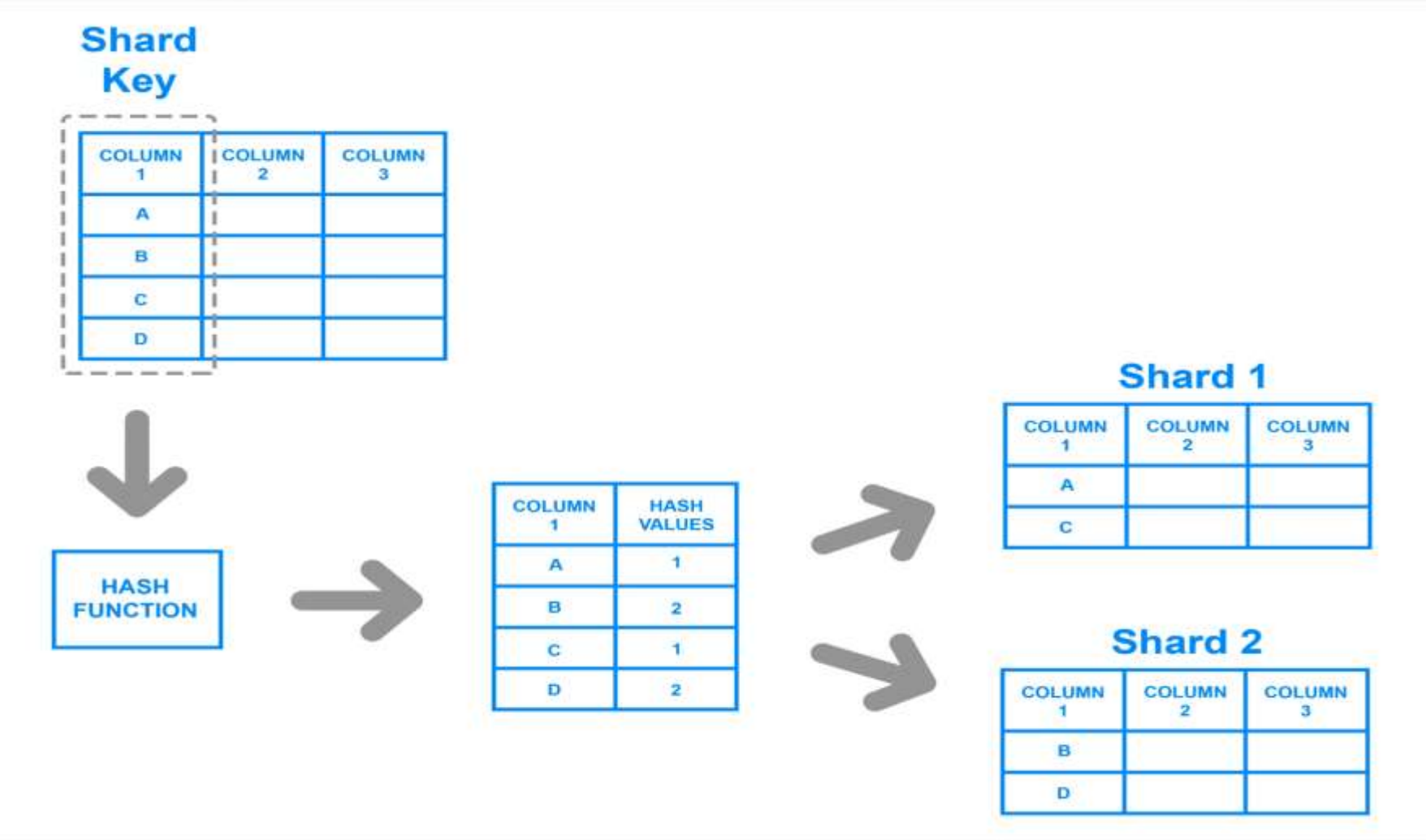
Why Database Sharding is Important?

1. Reduces Load on a Database
2. Improves Query Performance
3. Makes it Simple to Scale Horizontally
4. Easy Recovery during Crashes or Failures
5. Enables More Users to Use the Database
6. Saves Time

Types of Database Sharding Architectures are

1. Key-Based Sharding
2. Directory-Based Sharding
3. Range-Based Sharding

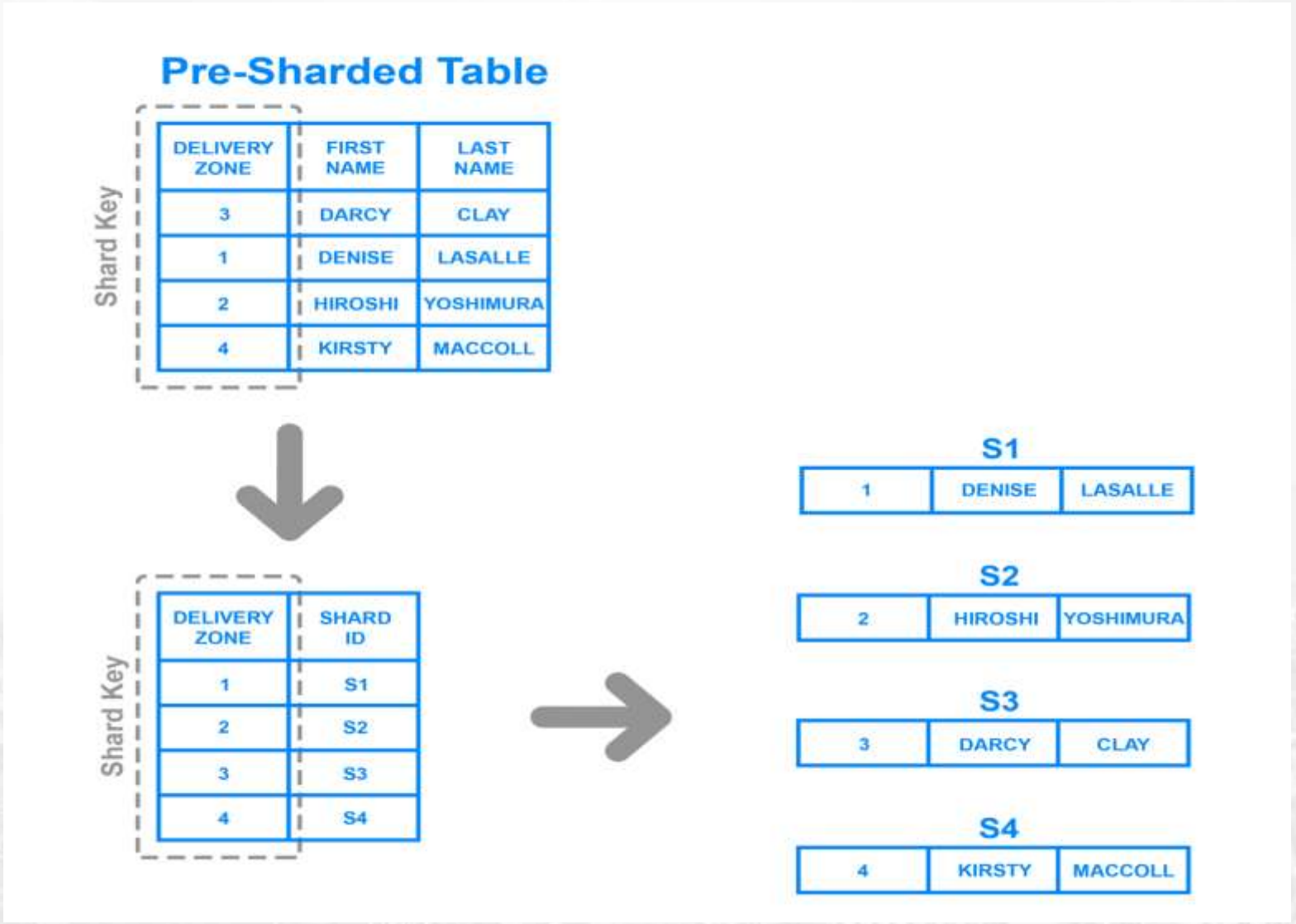
1. Key-Based Sharding



1. Key-Based Sharding (cont..)

- Hashing is popular to store key-value pairs. Each key has a unique value
- Hash function that maps each row to its Shard by taking in some data from the row and mapping it to the unique value which is the Shard in which the data should be stored.
- Suppose you have an Employee Database. You have designed your Hash function in such a way that it performs a function (Hash function(Employee id)=Emploee id modulo 6).
- This function returns remainders from 0 to 5. So, let's say there are 6 Shards now, the employee's data with employee id giving a remainder of 0 when divided with 6 goes into Shard 1.

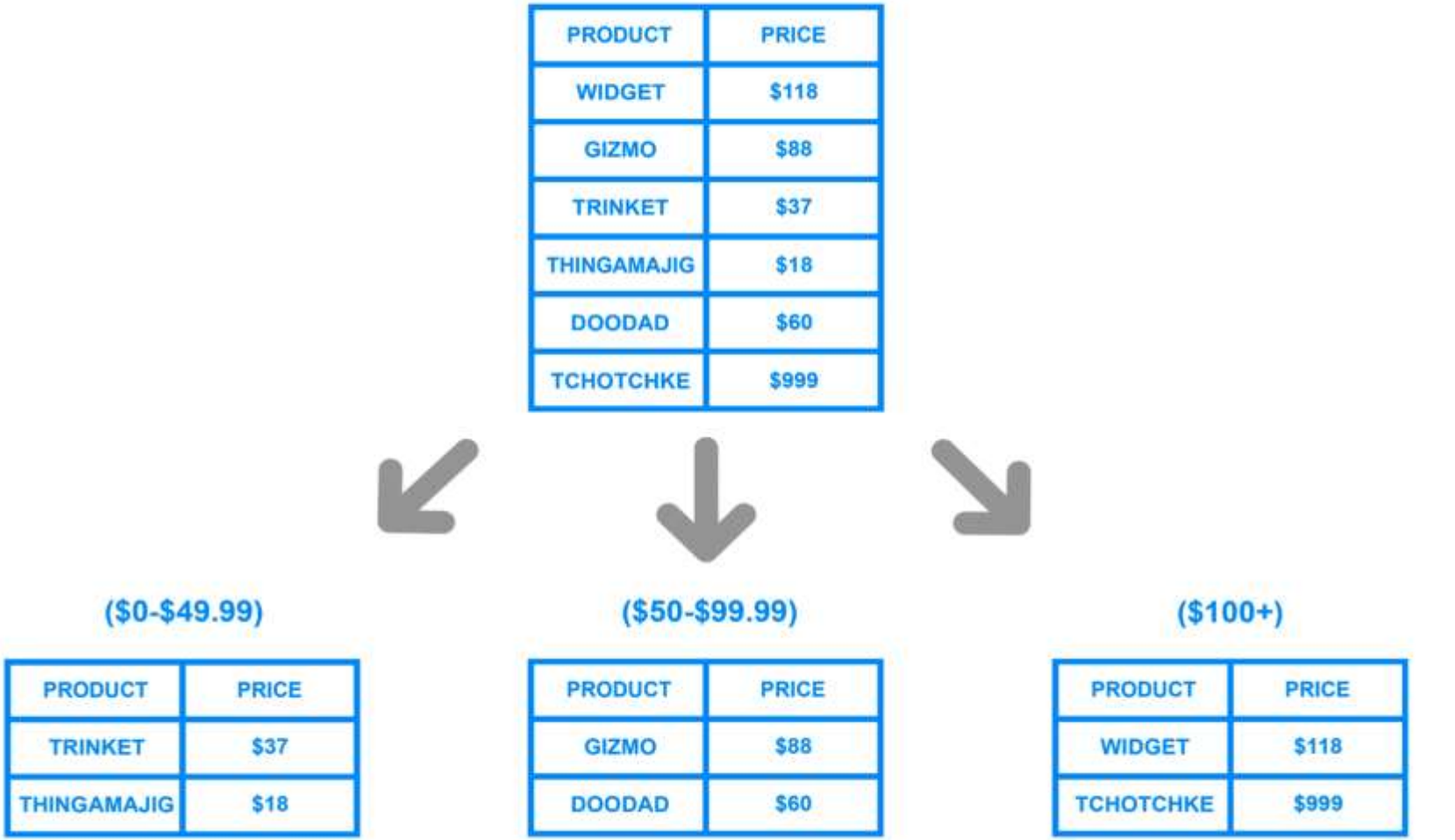
2. Directory-Based Sharding



2. Directory-Based Sharding (cont..)

- As mentioned earlier, to keep track of the data in a Database Shard this architecture uses lookup tables.
- The lookup table can give you information about where the data is stored.
- This Database Sharding architecture is more flexible as it allows you to have freedom over the range of values in the lookup table or create Shards based on algorithms and so on.
- The only drawback here is that every single time a query needs execution it needs to consult a lookup table to locate the concerned data.
- Also, the whole system will fail if the lookup table crashes because this architecture cannot function without it.

3. Range-Based Sharding



3. Range-Based Sharding (cont..)

- The lookup table will contain the Shard id and a range of values.
- If the data to be stored comes under a particular range of a Shard, it will be stored in that Shard.
- This will result in uneven distribution of data in some cases because some ranges of the Shards may have more frequency/occurrences than others.

Advantages of Sharding

1. Reads / Writes

- MongoDB distributes the read and write workload across the shards in the sharded cluster, allowing each shard to process a subset of cluster operations.

- Both read and write workloads can be scaled horizontally across the cluster by adding more shards.

- For queries that include the shard key or the prefix of a compound shard key, mongos can target the query at a specific shard or set of shards. These targeted operations are generally more efficient than broadcasting to every shard in the cluster.

Advantages of Sharding

2. Storage Capacity

- Sharding distributes data across the shards in the cluster, allowing each shard to contain a subset of the total cluster data.
- As the data set grows, additional shards increase the storage capacity of the cluster.

Advantages of Sharding

3. High Availability

- A sharded cluster can continue to perform partial read / write operations even if one or more shards are unavailable.

- While the subset of data on the unavailable shards cannot be accessed during the downtime, reads or writes directed at the available shards can still succeed.

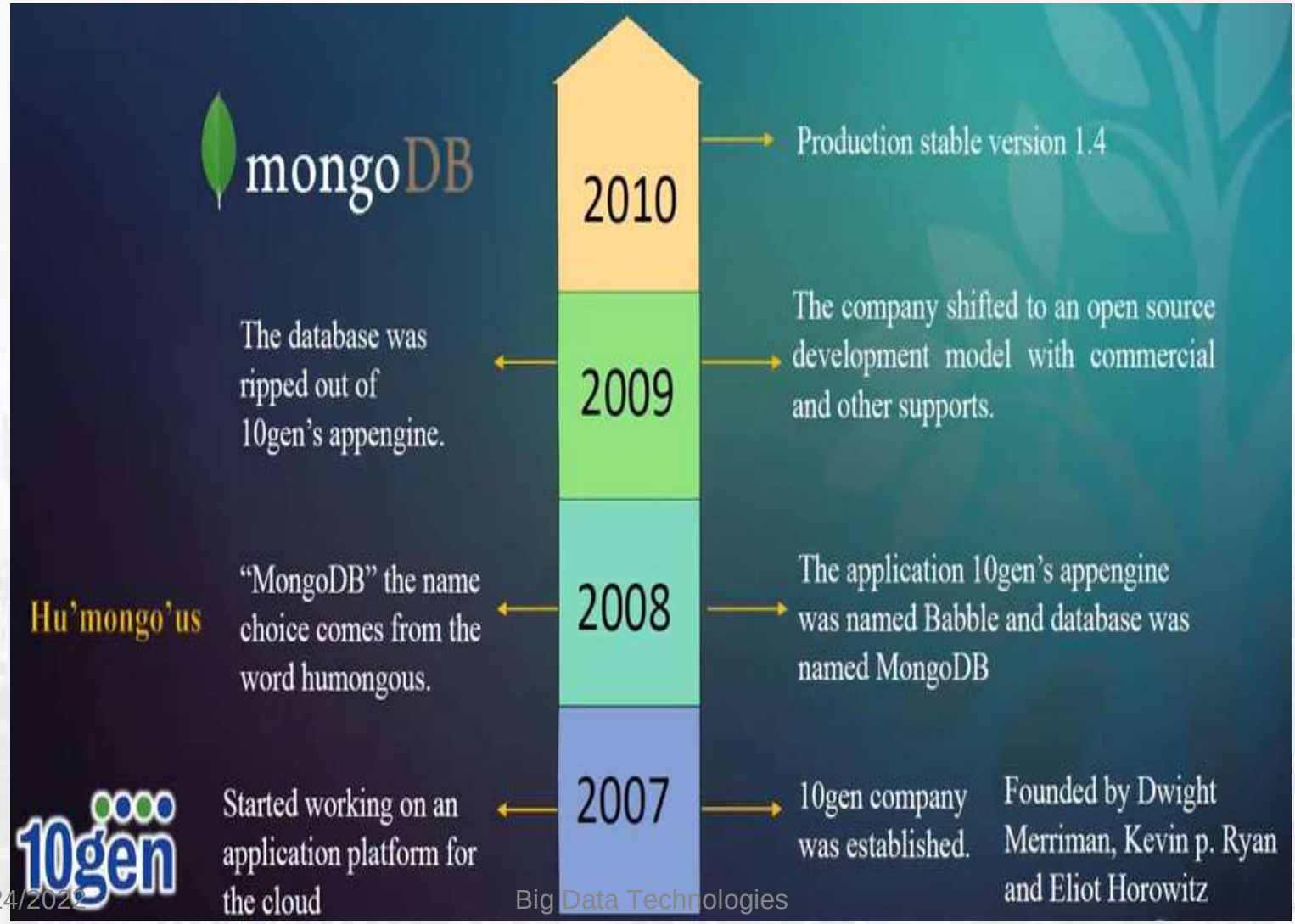
4. Useful for **worldwide distribution of applications** where communication links between the data centers would otherwise be a bottleneck.

Disadvantages of Sharding:

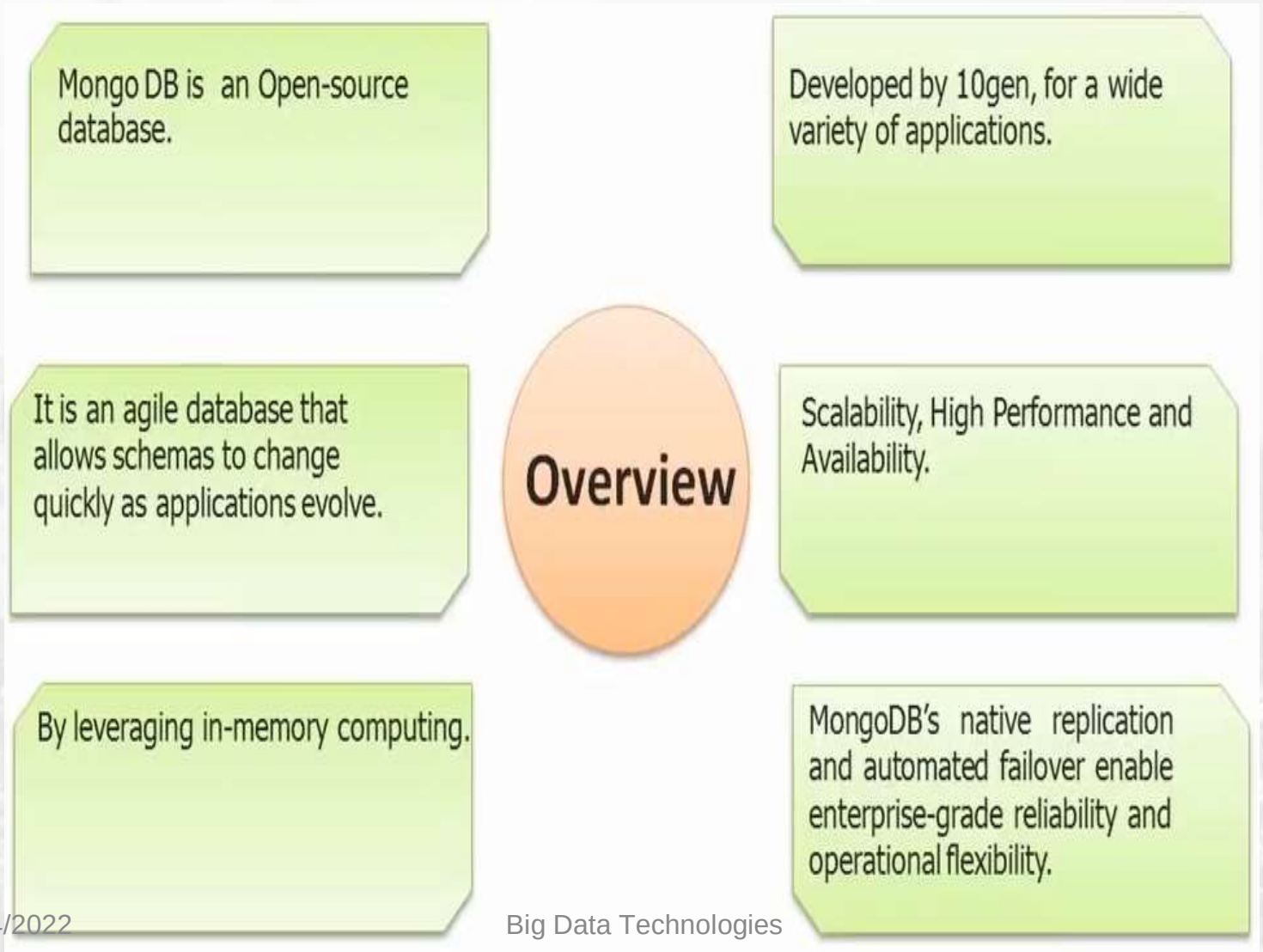
- Increased complexity of Query Language
- Increased complexity due to : partitioning, load balancing, co ordinating, ensuring integrity etc.
- Increased operational complexity :: adding / removing indexes, adding / deleting fields(columns of collection) etc.

NoSQL Document-Based Data Model
MongoDB Database

History of MongoDB



MongoDB Overview



What is MongoDB?

- a powerful, flexible and scalable general-purpose database. It is agile database that allows schemas to change quickly as application evolve. I
- a NoSQL Database.

JSON Format

JavaScript Object Notation

JSON Abbreviation

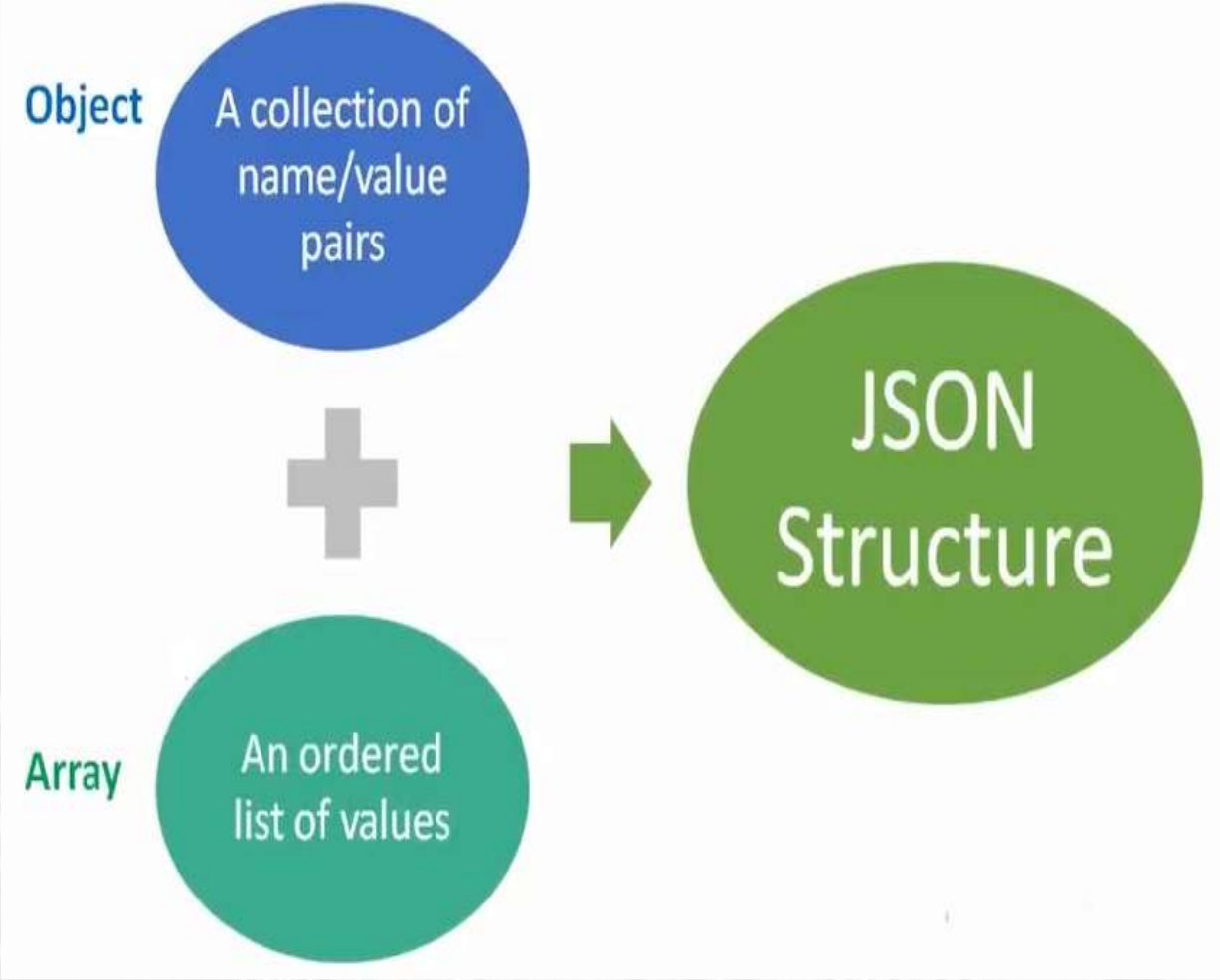
Lightweight data-interchange format

Easy for humans to read and write

Easy for machines to parse and generate

Text format that is completely language independent

JSON Cont…


Object
A collection of name/value pairs
Array
An ordered list of values
JSON Structure

JSON Features

- Light-weight text-based open standard for human-readable data interchange.
- Extended from JavaScript.
- aspects of data transfer are simplicity, extensibility,
- interoperability, openness and human readability
- Can be parsed by JavaScript Parser
- Can represent simple and complex data
- Support for Unicode
- Can be used in AJAX
- Use Key-Value Pairs
- Is a collection of Objects and Arrays

JSON Datatypes

- Strings
- Number
- Boolean,
- Objects and
- Arrays

JSON Format Example

```
{
"book": [
{
"id":"01",
"language": "Java",
"edition": "third",
"author": "Herbert Schildt"
},
{
"id":"07",
"language": "C++",
"edition": "second"
"author": "E.Balagurusamy"  }]
}
```

BSON

- “Binary JSON”
- Binary-encoded serialization of JSON-like docs
- a computer interchange format that is mainly used for data storage and as a network transfer format in the MongoDB database.
- a simple binary form which is used to represent data structures and associative arrays (often called documents or objects in MongoDB).

BSON Example

```
{
"_id" :          "37010"
"city" :         "ADAMS",
"pop" :          2660,
"state" :        "TN",
“councilman” : {
                    name: “John Smith”
                    address: “13 Scenic Way”
                  }
}
```

Why MongoDB ?

New Apps

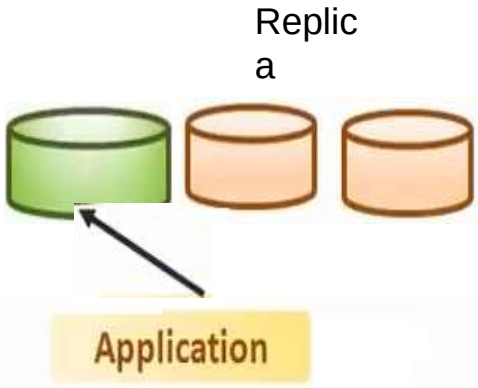
New Development Methods

New Data Volumes

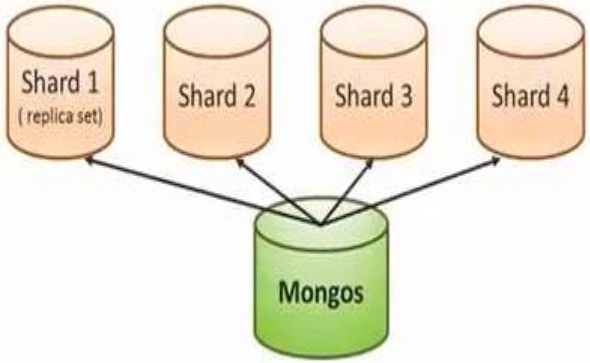
New Architectures

New Data Types

Why MongoDB ? Cont…

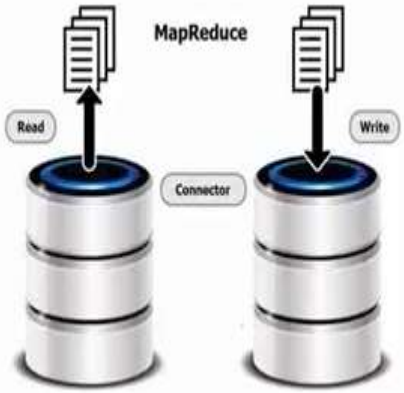


High Availability



Auto Sharding

Easy Query

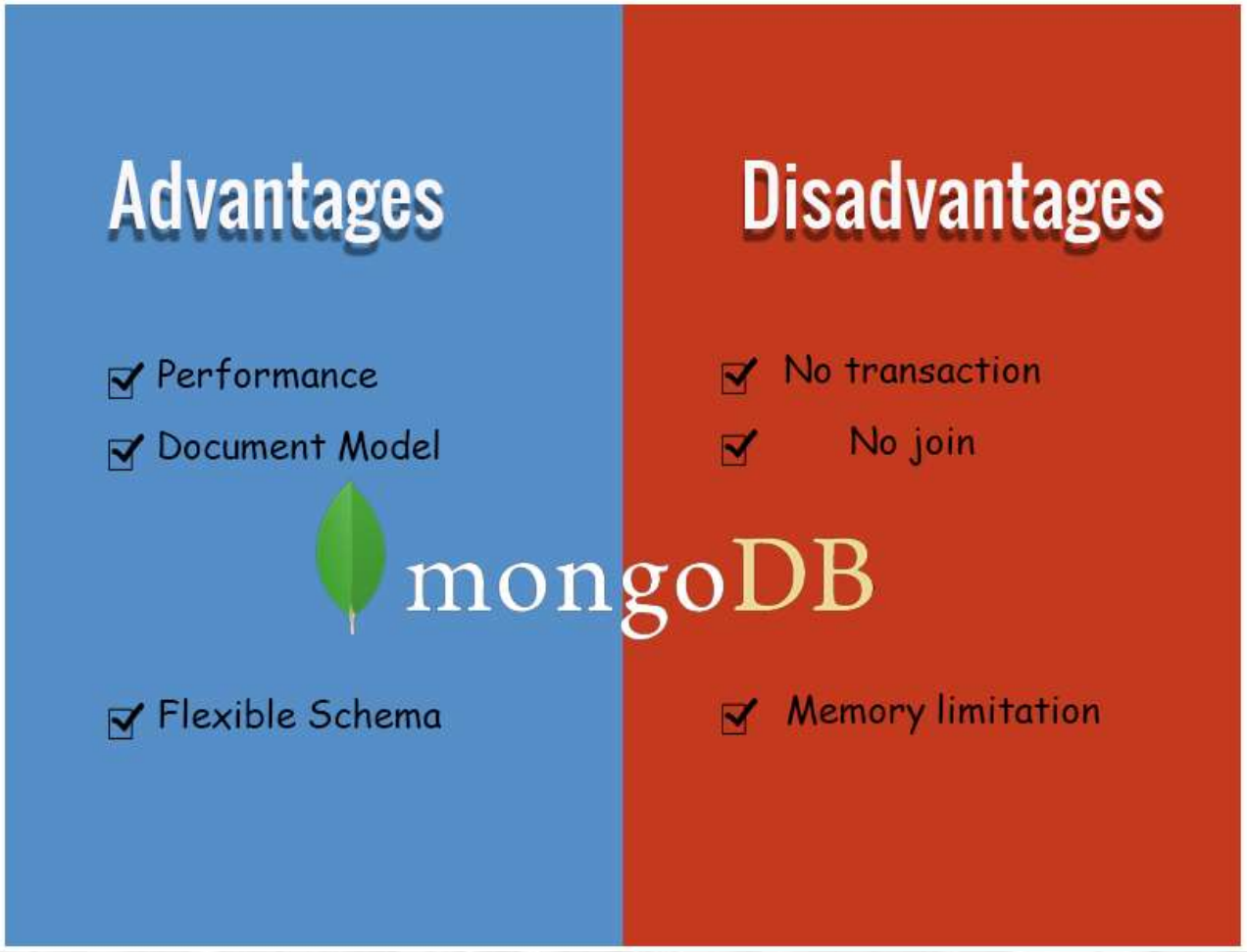


Map Reduce

Gird FS

Support from Expert

Pro's and Con's of MongoDB

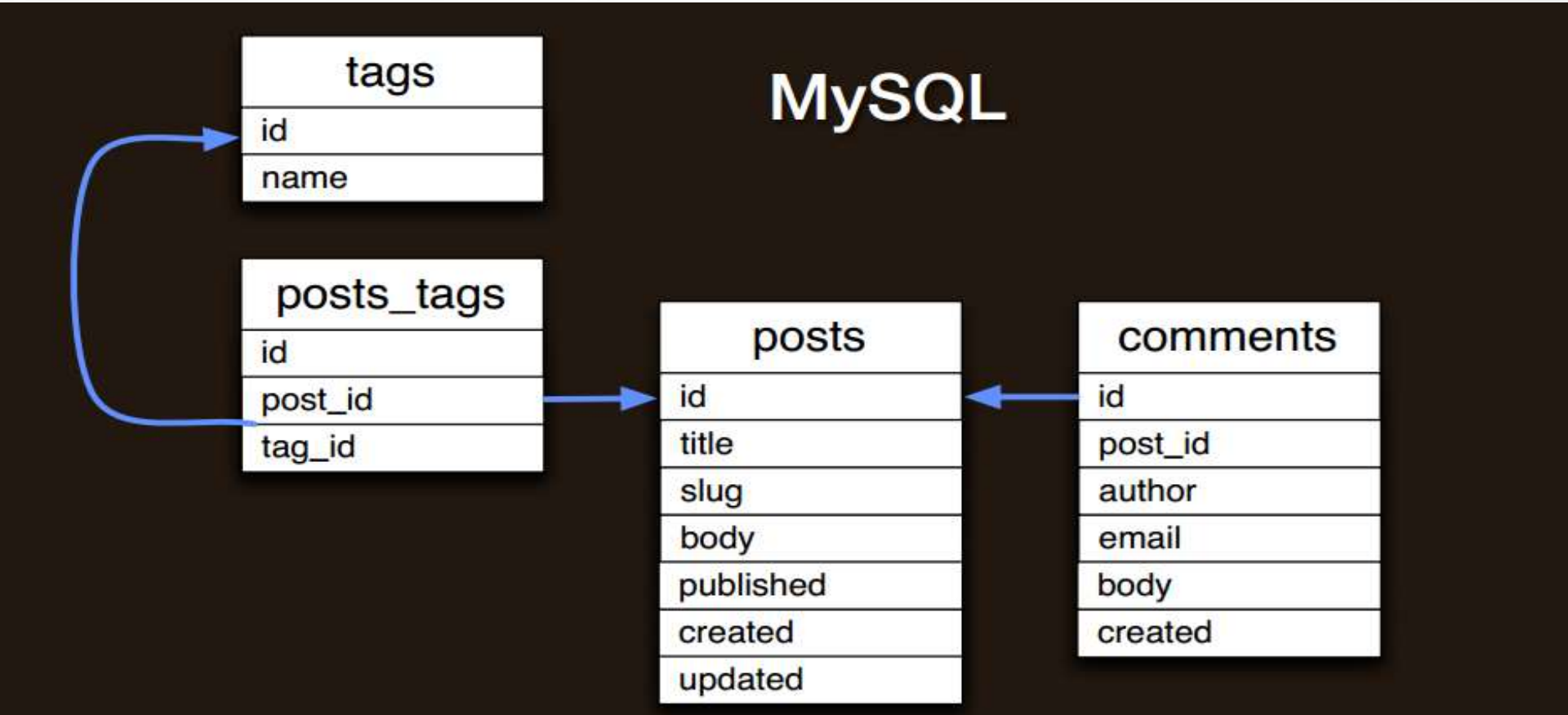


SQL Vs MongoDB

SQL Concepts	MongoDB Concepts
database	database
Table, View	Collection
Row	Document (BSON Document)
Column	Field
Index	Index
Table Join	Embedded documents & Linking
Primary key	Primary Key
Specify any unique column or column combination as primary key.	In MongoDB, the primary key is automatically set to the _id field.
aggregation (e.g. group by)	aggregation pipeline

Schema design

- RDBMS: join



Schema design

MongoDB

Schema design

MongoDB

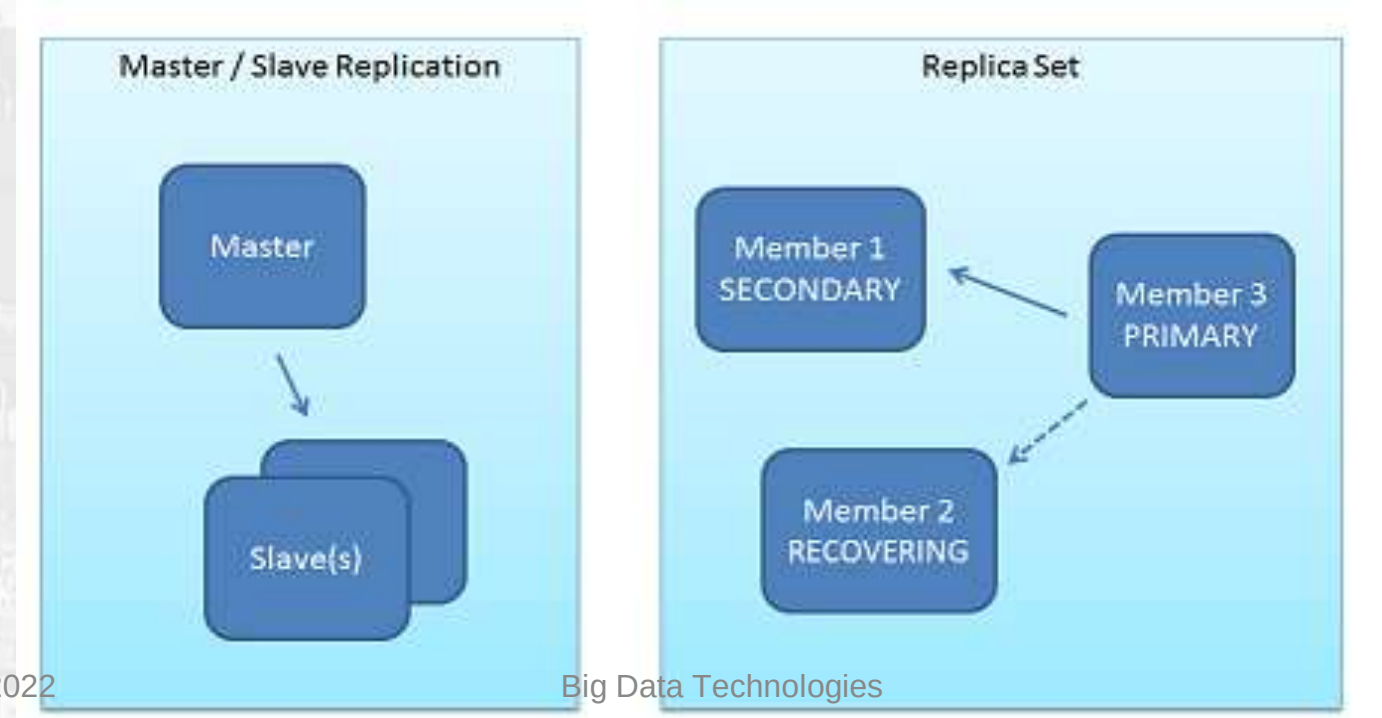
```
{
    "_id" : ObjectId("4c03e856e258c2701930c091"),
    "title" : "Welcome to MongoDB",
    "slug" : "welcome-to-mongodb",
    "body" : "Today, we're gonna totally rock your world...",
    "published" : true,
    "created" : "Mon May 31 2010 12:48:22 GMT-0400 (EDT)",
    "updated" : "Mon May 31 2010 12:48:22 GMT-0400 (EDT)",
    "comments" : [
        {
            "author" : "Bob",
            "email" : "bob@example.com",
            "body" : "My mind has been totally blown!",
            "created" : "Mon May 31 2010 12:48:22 GMT-0400 (EDT)"
        }
    ],
    "tags" : [
        "databases", "MongoDB", "awesome"
    ]
}
```

MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more ‘databases’
- A database may have zero or more ‘collections’.
- A collection may have zero or more ‘documents’.
- A document may have one or more ‘fields’.
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.

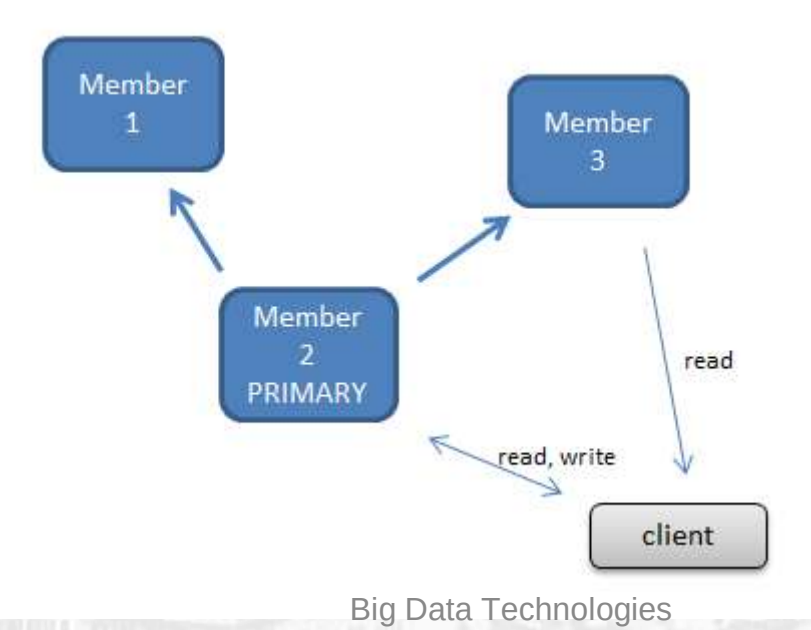
Replication

- Replica Sets and Master-Slave
- replica sets are a functional superset of master/slave and are handled by much newer, more robust code.



Replication

- Only one server is active for writes (the primary, or master) at a given time – this is to allow strong consistent (atomic) operations. One can optionally send read operations to the secondaries when eventual consistency semantics are acceptable.



Why Replica Sets

- Data Redundancy
- Automated Failover
- Read Scaling
- Maintenance
- Disaster Recovery(delayed secondary)

MongoDB Processes and configuration

MongoDB Processes and configuration

- mongod – database instance

- mongos –
  - Sharding processes
  - Analogous to a database router.
  - Processes all requests
  - Decides how many and which *mongods should receive the query*
  - *Mongos collates the results, and sends it back to the client.*

- mongo – an interactive shell (a client)

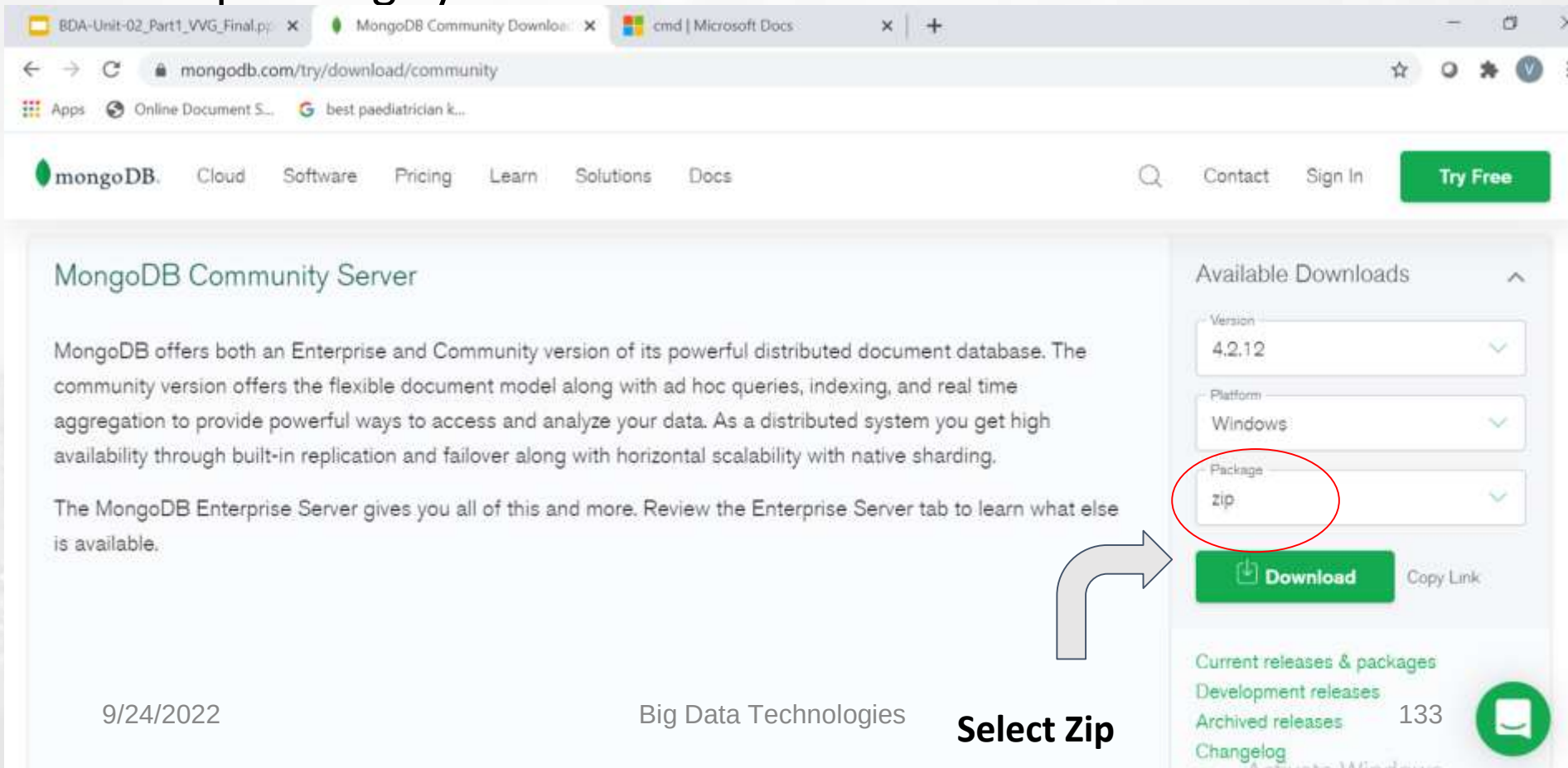
Fully functional JavaScript environment for use with a MongoDB

Terminology and Concepts

SQL Terms\ Concepts	Mongo DB Terms\ Concepts
Database	*Database*
Table	Collection
Row	Documents
Column	Field
Index	Index
Primary key	Primary Key

How To Install MongoDB?

- To install the MongoDB first download the mongodb community server version 4.2.12(zip file) as per your Operating System from :



Connectivity Between Client and Server

For Mongodb 4.2.12

1. **Server Configuration :**

We need to start the mongodb server using mongod through Command Prompt

Create a blank folder for storing database on any drive.

Syntax for starting server :

path of mongodb bin\ **mongod.exe --dbpath path_of_ newly_created_folder**

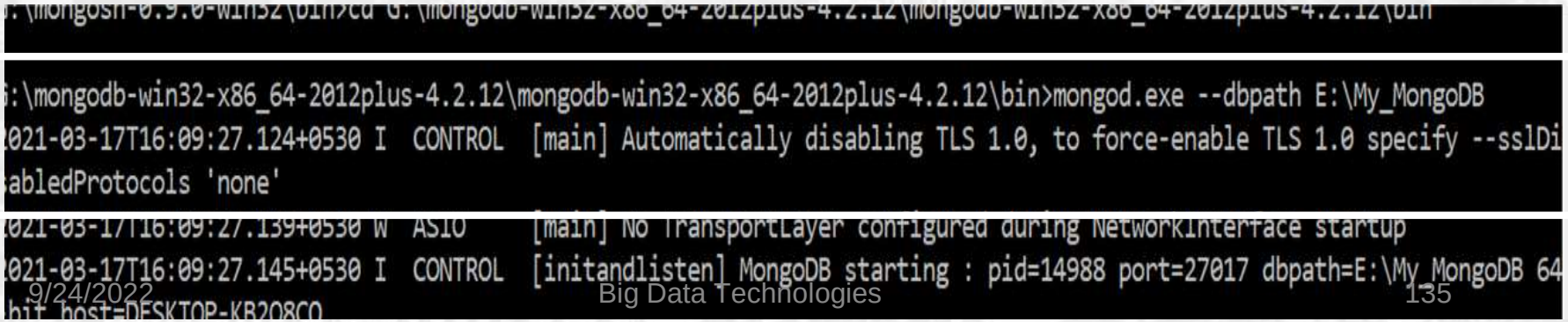
Client Configuration :

- Client : path of bin\mongo.exe

Connectivity Between Client and Server

Example : If MongoDB is extracted in G:\ drive,copy the path till bin folder and assume new folder (My_MongoDB) is created on E:\ drive then server can be started as follows through command prompt :

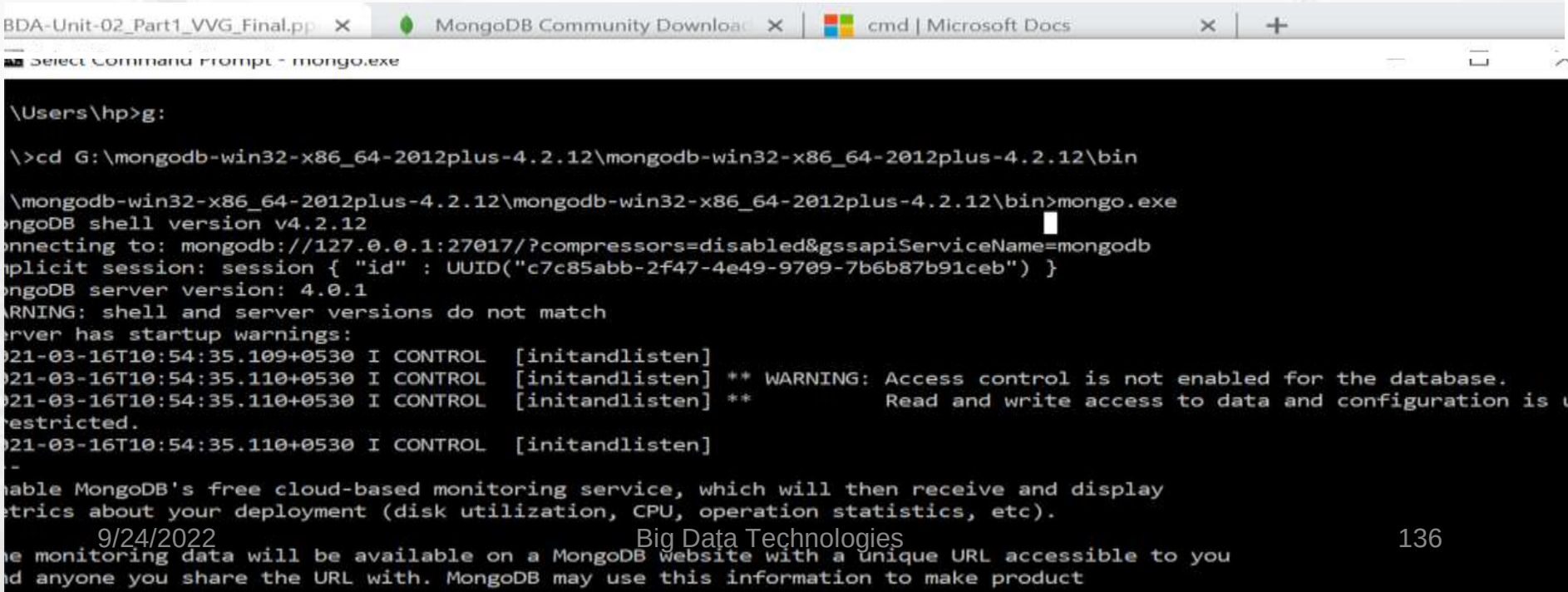
G:\mongodb-win32-x86_64-2012plus-4.2.12\mongodb-win32-x86_64-2012plus-4.2.12\bin>**mongod.exe --dbpath E:\My_MongoDB**



Connectivity Between Client and Server

Client Configuration :

G:\mongodb-win32-x86_64-2012plus-4.2.12\mongodb-win32-x86_64-2012plus-4.2.12\bin>**mongo.exe**



Executables

	Mongo DB	MySQL	Oracle	Informix	DB2
Database Server	mongod	mysqld	oracle	IDS	DB2 Server
Database Client	mongo	mysql	sqlplus	DB-Access	DB2 Client

MongoDB CRUD Operations

- Create Operations
- Read Operations
- Update Operations
- Delete Operations

Basic commands

show dbs

Print a list of all databases on the server.

```
> show dbs
abc      0.000GB
admin    0.000GB
config   0.000GB
db1      0.000GB
local    0.000GB
mr       0.000GB
vasu     0.000GB
>
```

Basic commands

> use myfirstdb

- Switch current database to < myfirstdb >. The mongo shell variable db is set to the current database.

```
> use my_database
switched to db my_database
>
```

Basic commands

> show collections

- Print a list of all collections for current database.

Creating a Collection

- Syntax :

  >db.createCollection **(“ collection_name”)**

  Consider ,we want to store student information so we can create a Student_info collection as follows :

  >>db.createCollection **(“Student_info”)**

Inserting Documents in a Collection

>db.collectionname.**insert({key1:value,key2:value,...})**

Inserts a document or documents into a collection.

>db. Student_info**.insert ({id:"151",**

name:"Vasundhara",city:"Pune"})

```
> use my_database
switched to db my_database
> db.createCollection("Student_info")
{ "ok" : 1 }
> show collections
Student_info
> db.Student_info.insert ({id:"151", name:"Vasundhara",city:"Pune"})
2021-03-17T17:08:13.714+0530 E  QUERY    [js] uncaught exception: SyntaxError: illegal character :
@(shell):1:28
> db.Student_info.insert ({id:"151", name:"Vasundhara",city:"Pune"})
WriteResult({ "nInserted" : 1 })
>
```

Display documents

>db.collectionname**.find()**

Returns all documents from a collection and returns all fields for the documents.

>db.Student_info.find()

{ "_id" : ObjectId("6051ea45f50ee3d1871c3398"), "id" : "151", "name" : "Vasundhara", "city" : "Pune" }

```
 db.Student_info.insert ({id:"151", name:"Vasundhara",city:"Pune"})
riteResult({ "nInserted" : 1 })
 db.Student_info.find()
 "_id" : ObjectId("6051ea45f50ee3d1871c3398"), "id" : "151", "name" : "Vasundhara", "city" : "Pune" }
```

Other operations

- *Insert one document :*
- **db.collection_name.insert({})**
- *Insert multiple documents*
- **db.collectionname.insert([{document1, },{document2, }, { document3}])**

- *Display all documents*
- **db.collection_name.find()**
- **db.collection_name.find.pretty()** //to display in structured format

Insert multiple documents

```
@(shell):1:46
>
> db.Student_info.insert([{id:"200",name:"Pankaj",age:"21",percentage:"67",},{id:"203",name:"Jyoti",age:"23"}])
BulkWriteResult({
        "writeErrors" : [ ],
        "writeConcernErrors" : [ ],
        "nInserted" : 2,
        "nUpserted" : 0,
        "nMatched" : 0,
        "nModified" : 0,
        "nRemoved" : 0,
        "upserted" : [ ]
})
> db.Student_info.find()
{ "_id" : ObjectId("6051ea45f50ee3d1871c3398"), "id" : "151", "name" : "Vasundhara", "city" : "Pune" }
{ "_id" : ObjectId("6052d772f50ee3d1871c3399"), "id" : "200", "name" : "Pankaj", "age" : "21", "percentage" : "67" }
{ "_id" : ObjectId("6052d772f50ee3d1871c339a"), "id" : "203", "name" : "Jyoti", "age" : "23" }
> db.Student_info.find().pretty()
{
        "_id" : ObjectId("6051ea45f50ee3d1871c3398"),
        "id" : "151",
        "name" : "Vasundhara",
        "city" : "Pune"
}
{
        "_id" : ObjectId("6052d772f50ee3d1871c3399"),
        "id" : "200",
        "name" : "Pankaj",
        "age" : "21",
        "percentage" : "67"
```

CRUD operations

- **Update**

db.collection.update(<query>, <update>, <options>)

syntax : db.collection.update(query, update, options)

Example :

db.Student_info.update

{ name:”Pankaj”},

{ $set: { age:24 } }

)

```
> db.Student_info.update({name:"Pankaj"},{$set:{age:23}})
WriteResult({ "nMatched" : 1, "nUpserted" : 0, "nModified" : 1 })
> db.Student_info.find().pretty()
{
        "_id" : ObjectId("6051ea45f50ee3d1871c3398"),
        "id" : "151",
        "name" : "Vasundhara",
        "city" : "Pune"
}
{
        "_id" : ObjectId("6052d772f50ee3d1871c3399"),
        "id" : "200",
        "name" : "Pankaj",
        "age" : 23,
        "percentage" : "67"
}
{
        "_id" : ObjectId("6052d772f50ee3d1871c339a"),
        "id" : "203",
        "name" : "Jyoti",
        "age" : "23"
}
>
```

Update Operation : Upsert

db.collection.update(<query>, <update>, { upsert: true })

Optional. If set to `true`, creates a new document when no document matches the query criteria. The default value is `false`, which does *not* insert a new document when no match is found.

Example

:>db.Student_info.update({name:"Ruhi"},{$set:{age:23}},{upsert

```
db.Student_info.update({name:"Ruhi"},{$set:{age:23}},{upsert:true})
riteResult({
        "nMatched" : 0,
        "nUpserted" : 1,
        "nModified" : 0,
        "_id" : ObjectId("6052db80e98826e59392fac3")
})
```

Update Operation : upsert

Select Command Prompt - mongo.exe

```
db.Student_info.find().pretty()

        "_id" : ObjectId("6051ea45f50ee3d1871c3398"),
        "id" : "151",
        "name" : "Vasundhara",
        "city" : "Pune"


        "_id" : ObjectId("6052d772f50ee3d1871c3399"),
        "id" : "200",
        "name" : "Pankaj",
        "age" : 23,
        "percentage" : "67"


        "_id" : ObjectId("6052d772f50ee3d1871c339a"),
        "id" : "203",
        "name" : "Jyoti",
        "age" : "23"


        "_id" : ObjectId("6052db80e98826e59392fac3"),
        "name" : "Ruhi",
        "age" : 23
```

Update Operation : multi

db.collection.update(<query>, <update>, { multi: true })

Optional. If set to `true`, updates multiple documents that meet the `query` criteria. If set to `false`, updates one document. The default value is `false`.

Example : Assume there are 2 documents with name as Jyoti.So to update city value for both we can use following command :

>db.Student_info.update({name:"Jyoti"},{$set:{city:"Bangalore"}},{multi:true})

Update Operation : multi

Select Command Prompt - mongo.exe

```
@(shell):1:50
> db.Student_info.insert({name:"Jyoti",age:"30",city:"Kolhapur"})
WriteResult({ "nInserted" : 1 })
> db.Student_info.update({name:"Jyoti"},{$set:{city:"Bangalore"}},{multi:true})
WriteResult({ "nMatched" : 2, "nUpserted" : 0, "nModified" : 2 })
```

```
db.Student_info.find()
"_id" : ObjectId("6051ea45f50ee3d1871c3398"), "id" : "151", "name" : "Vasundhara", "city" : "Pune" }
"_id" : ObjectId("6052d772f50ee3d1871c3399"), "id" : "200", "name" : "Pankaj", "age" : 23, "percentage" : "67" }
"_id" : ObjectId("6052d772f50ee3d1871c339a"), "id" : "203", "name" : "Jyoti", "age" : "23", "city" : "Bangalore" }
"_id" : ObjectId("6052db80e98826e59392fac3"), "name" : "Ruhi", "age" : 23 }
"_id" : ObjectId("6052ddcdf50ee3d1871c339b"), "name" : "Jyoti", "age" : "30", "city" : "Bangalore" }
```

}]

Delete operation

- **Delete?**
  db.collection.remove(<query>, <justOne>)
- Collection specifies the collection or the ‘table’ to store the document

Example :

> db.Student_info.remove({name:"Pankaj"})

```
> db.Student_info.remove({name:"Pankaj"})
WriteResult({ "nRemoved" : 1 })
```

```
> db.Student_info.find()
{ "_id" : ObjectId("6051ea45f50ee3d1871c3398"), "id" : "151", "name" : "Vasundhara", "city" : "Pune" }
{ "_id" : ObjectId("6052d772f50ee3d1871c339a"), "id" : "203", "name" : "Jyoti", "age" : "23", "city" : "Bangalore" }
{ "_id" : ObjectId("6052db80e98826e59392fac3"), "name" : "Ruhi", "age" : 23 }
{ "_id" : ObjectId("6052ddcdf50ee3d1871c339b"), "name" : "Jyoti", "age" : "30", "city" : "Bangalore" }
>
```

List of Other Commands

- **Use of Regular Expressions :**

- `{ <field>: { $regex: /pattern/, $options: '<options>' } }`

- **Pattern Matching :**

db.Student_info.find({ name: { $regex: “^V.*” } }) //starting with ‘V’

> db.Student_info.find({name:{$regex:"a"}}) // haing substring ‘a’

db.Student_info.find({ name: { $regex: /i$/ } }) //ending with i

db.Student_info..find({name:{$regex:”^s.*m$/}}) //displays the docs starting with R and ending with i

> db.Student_info.find({name:{$regex:new RegExp("^J.*i$","i")}}) //displays the docs starting with J and ending with i. case insensitive match

List of Other Commands

Querying Arrays:

insert array :

>

db.Student_info.insert({name:"Pooja",marks:[56,69,75],},{name:"A jay",age:20,marks:[80,90,65]})

WriteResult({ "nInserted" : 1 })

> db.v.find({ marks: {$all: [40] }})