Consider the following database implementations for solving the given requirements:

- **SQL databases**

  - Relational (tabular) databases with a rigid structure and limited storage

- **NoSQL (Not only SQL) databases**

  - Very flexible non-relational (non-tabular) databases with unlimited storage

However, the table below explains the challenges associated with SQL databases and why NoSQL is chosen instead of SQL databases.

Requirement	**Challenges using SQL databases**	**Solution using NoSQL databases**
Architecture	Centralized - single node dependency	Distributed - set up multiple machines
Massive storage	Cannot store more than 100s of GB	Can store more than 1000s of TB
High reliability	Requests cannot be processed if the central server is down	Data is copied to multiple nodes overcoming single node failure
Data model	Fixed schema	Flexible schema - can store images, audio, video and text
Scalability	Scale up vertically - hardware upgraded as load increases	Scale out horizontally - new machines are added as load increases
Overall cost	Expensive high end servers - mostly proprietary software with additional licensing cost	Open source software on low cost commodity machines reduce licensing cost by 50% - 80%
Performance	Performance degrades as concurrent users increase	Multiple nodes to process multiple requests
Latency	On-disk processing slows down query performance	In-memory (caching) processing delivers sub-millisecond response

SQL databases are not suitable for scenarios such as eCommerce. But, NoSQL databases are a better fit.

NoSQL are databases that store data using non-tabular structures. It is suitable for massive data storage and faster than RDBMS.

Some of the popular NoSQL databases are listed below:

- Redis
- Riak
- DynamoDB
- MemcacheD
- Apache HBase
- Apache Cassandra
- MongoDB
- CouchDB
- Couchbase
- MarkLogic
- Neo4j
- OrientDB
- InfiniteGraph
- Project Voldemort

Business scenario

An eCommerce system allows customers to shop products online. There are millions of customers accessing the site every day and thousands of vendors selling their products in the marketplace.

Owing to the above requirement for massive storage and efficiency in managing their operations, NoSQL has been chosen as the database implementation.

The eCommerce system under consideration has the following four main components:

- Shopping Cart - Contains a list of products that customers are interested in purchasing
- User Activity Logs - Track user interaction with the eCommerce site
- Product Catalog - Maintain details of each product

- Recommendations - Recommends purchase of related items based on the currently selected product

As part of this case study, you will go through various steps involved in identifying the best suitable database for each of the four components mentioned above.

The two stages mentioned below help you design the database infrastructure of the eCommerce system under consideration.

Stage 1: For each component, identify the most relevant data model, consistency level and the read/write load required

Stage 2: Map the outcome of Stage 1 to the characteristics of few popular NoSQL databases to choose the most suitable one

As part of Stage 1, for each eCommerce component, you will do the following.

1.1 Identify data model

- Is your data text-based / key-value / flexible / structured / highly related to one another?

1.2 Choose consistency level

- Does it require stronger consistency or higher availability?

1.3 Analyze whether operations are read/write intensive

- Does it mostly involve read or write operations?

The data model is chosen based on the type of data the component would be dealing with.

This step begins by identifying the most suitable NoSQL category below for each component of the eCommerce system:

a) Key-value stores

b) Column-family stores

c) Document-oriented databases

d) Graph-based databases

At the end of this step, you will be able to identify the data model for each eCommerce component as required.

Key-value database:

- Every single item in the database is stored as an attribute name(key) together with its value.
- Data is stored as a collection of key/value pairs.
- The value can be integer, string, or complex data types such as sets of data.
- The key in a key-value pair must be unique.
- Data is retrieved via an exact match on the key.
- New types of data can easily be added to the database as new key-value pairs.

Operations :

There are 3 operations performed in key-value databases.

1. Put(key,value)
2. Get(key)
3. Delete(key).

Database schema:

conversion of SQL structure to Key-value database.

Standard SQL table would be like this:

Student table:

Sid	Name	Grade	Phone
101	John	A	9883345678
102	Marry	C	7788992233

The key-value database schema would be like this:

Table_name: primary_key_value: attribute_name	=	value
Key		**value**

Conversion of SQL schema to key-value schema :

Student table:

Sid	Name	Grade	Phone
101	John	A	9883345678
102	Marry	C	7788992233

Key value database schema:

Table_name: primary_key_value: attribute_name = value

Key **value**

Student:101:name = “Jhon”

Student :101: grade = “A”

Student :101: phone= 9883345678

Student:102:name= “Marry”

Student:102: grade= “C”

Student:102: phone= 7788992233

Consider the **shopping cart** component in eCommerce:

Parameter	**Business need**	**Why not SQL?**	**Why use key-value stores?**
Uniqueness	Shopping cart with a list of items unique to each customer	In SQL, data is normalized and so does not allow multi-valued dependency. That is, for a given customer_id, multiple cart items cannot be stored in a single column	Can represent shopping cart using key-value stores
High Scalability	Manage shopping cart for millions of online customers	Limited storage due to single node model	Can scale out cart details onto multiple machines
Faster read/write	Quick access to cart while the customer is online	Slow performance accessing data on disk	Can store frequently accessed data in RAM(in-memory cache), hence faster than RDBMS

Observation

The above table shows that shopping cart in eCommerce is best represented using key-value stores.

Characteristics of key-value stores

- Each **key** is unique and holds a **value** that can be xml, json, binary, text, and so on
- Quick look-ups using the key (similar to dictionary search)
- Every key-value pair is unrelated to one another
- Best suited for storing user sessions, customer preferences, browsing patterns and so on
- Key-value pairs are distributed across multiple machines as shown below rather than storing all on a single machine

Limitations of key-value stores

Key-value stores do not allow retrieving by value or updating only part of the value. It can be retrieved only by customer_id (key), whereas look up using email_id (part of value) is not possible.

For shopping cart, key-value stores are apt as there is no need to search carts containing a Samsung mobile for instance. But for other scenarios, it is required to query by non-key fields or update only specific fields in which case column-family stores can be used.

Problem Statement:

Assume customer details such as name, location, purchase_history, contact_info are represented using key-value stores with customer_id as a key.

- To find customers making purchases at a given location

Are key-value stores appropriate for the given scenario?

Solution:

For the given requirement, you need to search by customers location.

Key-value stores allow retrieval only by the key (customer_id) and are therefore not suitable.

A better query support is required which is provided by column family stores.

Consider the eCommerce component for **user activity logs**.

Business Need

- *Requirement 1:* Require highly scalable databases to deal with continous logging of customer interaction with the system
- *Requirement 2:* Querying by fields other than the key. For example, fetching customer details using *email_id* which is a non-key attribute
- *Requirement 3:* Retrieving/updating only part of the value. For example, fetching or changing only the customer's *age* and *city* among other details of the customer

Why not SQL?

- Millions of customers interacting with the eCommerce system cause heavy logging activity
- As relational databases do not scale well, they fail to meet requirement 1
- Also, due to their single node model, these systems fail when overloaded with too many writes

Why not key-value stores?

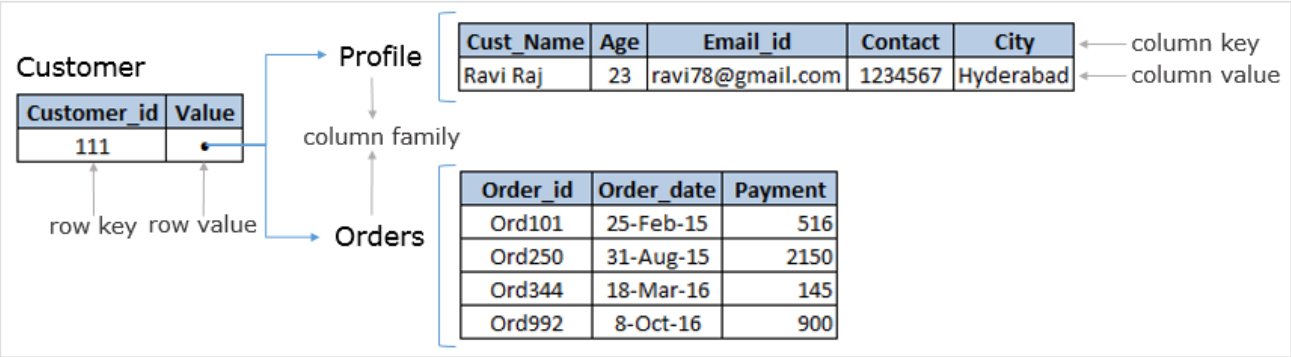
Key-value stores do not allow querying by value or updating only part of the value. Hence fail to meet requirements 2 and 3

Why use column-family stores?

- Column family stores are very efficient for heavy writes
- Unlike key-value stores, they allow index creation on non-key fields. For example, create an index on *email_id* to query by this field apart from querying by *customer_id* (key)
- Also, you can retrieve or update any fields as required. For example, retrieving or updating the *contact_number* of the customer

Hence, column-family stores meet all three requirements of the logging scenario above.

The below diagram is a representation of *Customer* details using column-family stores



Characteristics of column-family stores

These databases are structured into group of related columns called Column-Families

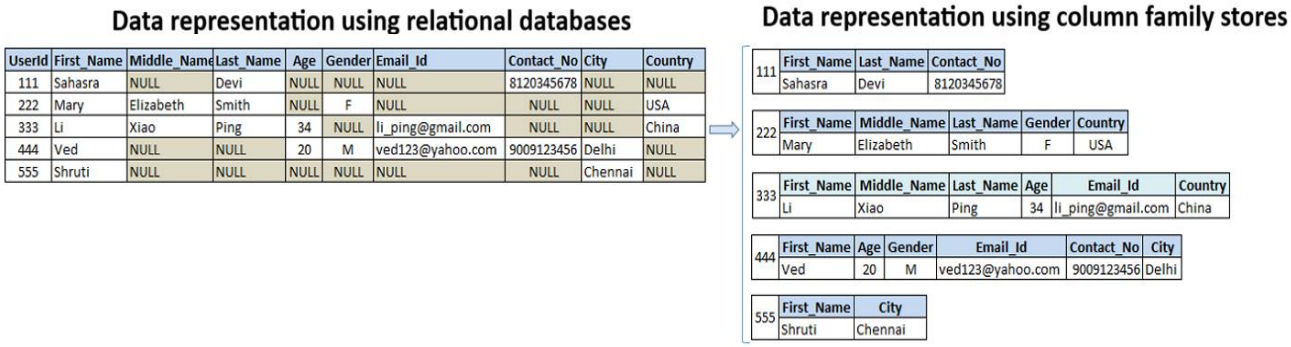
Each column family is a set of data that is often accessed together, for example, customer profile

These are highly scalable with the ability to take up loads to any extent

In column family stores, every row can have a different set of columns unlike relational databases

Very useful for sparsely populated tables (most of the columns having NULL values) and wide rows having too many columns

Column family stores optimize memory by ignoring columns with NULL values as illustrated below.



Limitations of column-family stores

It is not possible to query on non-indexed fields, for example, *age* (assume only *customer_id* and *email_id* are the indexed fields)

Support for read operations is limited compared to SQL. For example, group by operation is not available

Also, not suitable for queries involving multiple fields. For example, search for customers with *age>20* and *city='Chennai'*. Use document-oriented databases instead

Problem Statement:

Let us take a scenario where customers need to search the catalog for products based on various criteria involving multiple fields as specified below:

- *price* of products ranging between USD 100 to USD 1500
- *brand* of the product
- products sold by a particular *retailer*
- products pertaining to a specific *color, size, type* and so on

Do you think column-family store is suitable for the given scenario?

Solution:

- For key look ups, key-value stores are fastest.
- For frequently updating data and searching by primary key or indexed keys, column families work very well.
- But for queries involving multiple fields, creating indexes on many fields is not a good practice. It is also a very costly operation. Hence column family stores are not recommended for the given scenario.
- For scenarios where millions of customers are frequently searching for products based on various criteria (read heavy workloads), document-oriented databases are most suitable.

Consider **product catalog** in eCommerce. Each product has a varied set of attributes. For example,

Product	Attributes
book	ProductID, title, author, price, publication
mobile	ProductID, model, brand, screenSize, os, memory, price, camera
pen	ProductID, brand, type, color, price

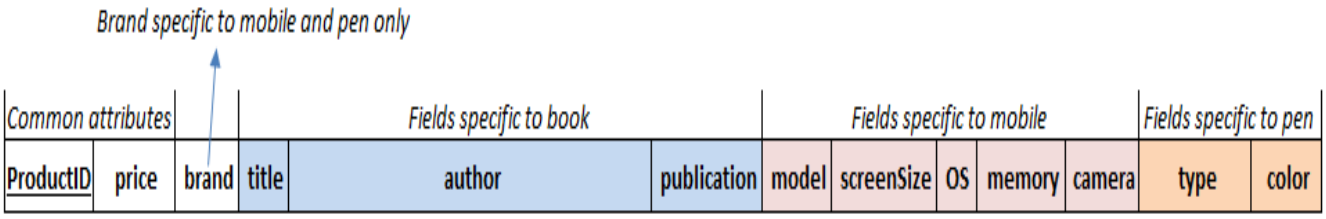
Business Need

Requirement 1: Require a flexible structure to store products with varied attributes

Requirement 2: Querying options such as sorting by price, filtering by brand and so on

Why not SQL?

The schema for ***ProductCatalog*** should represent unique attributes of all products as follows



This results in hundreds of columns and too many NULL values for columns not relevant to the product, e.g., for a *pen*, all fields except *ProductID, price, brand, type* and *color* would have NULL values

Also, a single change in attributes requires altering the entire schema

Therefore, RDBMS is inefficient for the given scenario.

Why not key-value stores?

Customers often search for products based on various criteria, e.g., *Samsung mobile with 5.5" screen size*

As *brand* or *screen size* cannot be used while looking up details using key-value stores (only *ProductID* can be used)

Why not column-family stores?

Column family stores do not allow querying on multiple fields, e.g., *searching for Parker pens which is less than Rs. 500*

Moreover, these databases have limited querying options, e.g., operation for *counting the number of books against each publication* is not available

Why use document-oriented databases?

These databases are schema-less - attributes of products in the catalog are varied and keep changing

Allows querying by non-indexed fields too - search by any given attribute of the product e.g., *model*, *price*

Supports very rich query language including storing, fetching, modifying, deleting, sorting, and grouping data

Support aggregations for e.g., *sum*, *count* and others string, math and set operations

Apt for heavy reads - product details are written to the database fewer times than the number of times they are read (searched) by customers

Hence document-oriented databases are most suitable for the product catalog component of eCommerce.

Characteristics of document-oriented databases

These databases are similar to key-value pairs but with richer querying options that SQL provide

Some of the operations supported by these databases are listed below:

CRUD - creation, retrieval, updation, and deletion of data

Relational operators - like <, >, <=, >=, not, in and so on

Rich data types – tweets, videos, podcasts, animated GIF and so on

Nested structure - customer has address

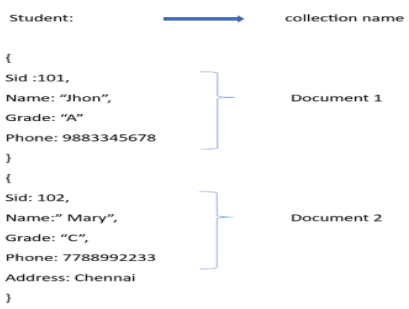
Limitation of document-oriented databases

It cannot represent complex relationships between data.

Document-oriented databases are just like key-value databases but here the tables were called collections and each and every row of a table will be called a document.

Example :

Conversion into the document oriented database schema:



The given scenario needs:

- *Flexible data model* -> SQL databases are not suitable
- *Rich queries* -> Key-value stores and column-family stores have limited query support, whereas document-oriented databases provide rich querying options

Hence document-oriented databases are best suited for the given scenario.

Problem Statement:

Assume, the online retailer wants to do the following:

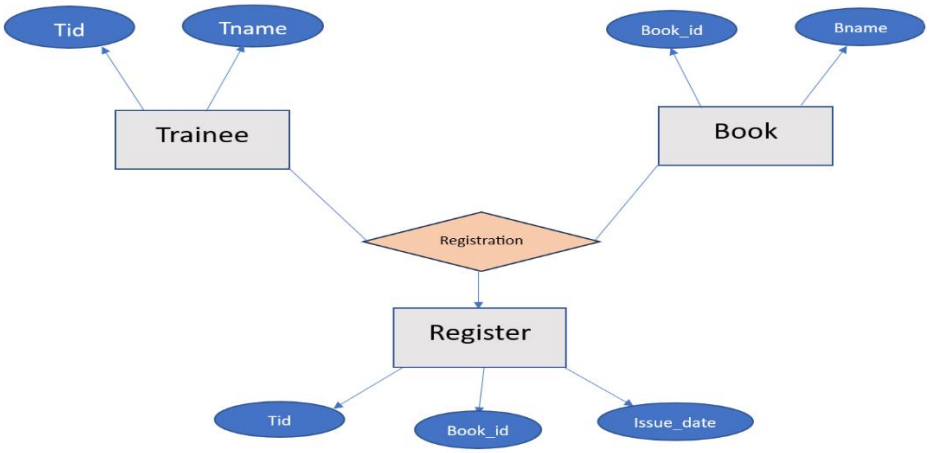
1. Keep track of the products you have purchased and suggest which of your friends have liked the same product
2. Recommend which other product can be purchased along with the product you have chosen

Do you think document-oriented databases is suitable for the given scenarios?

Solution:

- The first requirement needs to link your orders to the orders of other related customers.
- The second requirement needs to maintain information about related products.
- Both are highly inter-related data.
- Hence, for the given requirements, document-oriented databases are not suitable

Let us have a scenario where trainees take a book from the library:



Converting this ER diagram we'll get three tables:

1. Trainee(Tid,Tname)
2. Book(Book_id,Bname)
3. Register(Issue_date)

Normal table schema for the above tables:

Book:

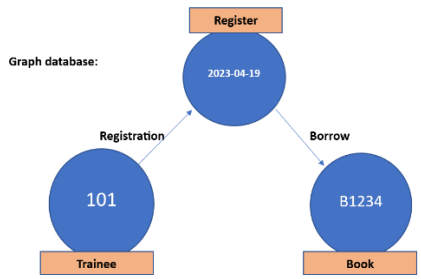
Book_id	Bname
B1234	Introduction to NoSQL
B2345	Introduction to python

Document representation of the above scenario:

Document representation:

{Tid:101,
Tname:" John",
Book: [{
Book_id:" B1234",
Bname:" Introduction to NoSQL",
Issue_date:" 2023-04-19"
},
{
Book_id:" B2345",
Bname:" Introduction to Python",
Issue_date:" 2023-07-23"
}]

Finally graph representation of the scenario:



Node: In a graph database, a node is a fundamental building block. It represents an individual entity or object within the database. Nodes are used to store and represent the data in a graph structure.

Edge(relationships): A relationship denotes a connection between a source node and a target node, where the source node serves as the origin, and the target node represents the destination in a network or graph context.

This connection is typically represented by an **edge**, indicating an association or interaction between the two nodes, which can be directed or undirected, and may carry additional weighted values.

Such relationships play a crucial role in understanding the interconnections and dynamics of entities within diverse systems, including social networks, transportation networks, and knowledge graphs.

Consider the **recommendations** component in an eCommerce system.

Business Need

- Track related products like smart phone and cover case
- Relate millions of customers and the products they purchase

Why not SQL?

- Join thousands of related products. For example, *recommending purchase of cover case while purchasing a smart phone*
- Joins are very time consuming
- Maintaining relationships between millions of customers and the products they purchase is very complex

Hence SQL is not the right solution for this scenario.

Why not key-value stores, column-families or document databases?

- Data in the recommendation scenario is highly related
- For example, recommending products to customers based on their past purchase history or browsing patterns
- None of these databases support complex relationships, hence not suitable

Why use graph-based databases?

- These databases use simple nodes and edges to store and retrieve highly interrelated data e.g., *customer* and *product*
- Graph traversals are much faster compared to SQL joins

- In the given scenario, use graph databases as described below
    - *Users* and *Products* are represented as nodes and
    - *Purchased* relationships are represented as edges
- Graph databases are also used for fraud detection wherein,
    - *Customers* are represented as nodes and
    - *Transactions* are represented as edges
    - *Transaction* paths that are not related to any *customer* are identified as frauds

Limitations of graph-based databases

- These databases are inappropriate for transactional data like financial accounting
- Do not scale out horizontally
- Difficulty in performing aggregations like sum and max efficiently
- Need to learn a new query language like CYPHER
- Have fewer vendors to choose from, so harder to get technical support

Take a look at the below code. You can get a detailed explanation of its key parts by clicking the below button:

key value

Observe the following changes of code from tabular(RDBMS) to Key-value databases.

```
Student:(SQL)
Sid Name Grade Phone
101 John A 9883345678
102 Marry C 7788992233

Student:(NoSQL)
Table_name: primary_key_value: attribute_name value
Student :101: name = "john"
Student :101:grade = "A"
Student :101:phone = 9883345678
```

Document oriented database

Observe the following changes of code from tabular(RDBMS) to Document-oriented databases.

```
Student: (SQL)
Sid Name Grade Phone Address
101 John A 9883345678 Null
102 Mary C 7788992233 Chennai

Student: (NoSQL)
{Sid :101,
Name: "Jhon", ==> Document 1
Grade: "A",
Phone: 9883345678
}

{Sid: 102,
Name: "Mary", ==> Document 2
Grade: "C",
Phone: 7788992233,
Address: Chennai
```

Column family store

Observe the following changes of code from tabula(RDBMS)r to Column family databases.

```
Users_details: (SQL)
Userid First_Name Age Gender Email_Id Address
111 Sky 23 NULL Sky@gmail.com NULL
222 Marry NULL F NULL NULL
333 Jack 44 M NULL Delhi

User_details: (NoSQL)
111 First_Name Age Email_id
Sky 23 Sky@gmail.com

222 First_Name Gender
Marry F

333 First_Name Age Gender Address
Jack 44 M Delhi
```

Graph Based Databases

Observe the following changes of code from tabular(RDBMS) to Graph-based databases.

```
Trainee: (SQL)
Tid Tname
101 John

Book: (SQL)
Book id Bname
B1234 Introduction to NoSQL
B2345 Introduction to python

Register: (SQL)
Tid Book_id Issue_date
101 B1234 2023-04-19
101 B2345 2023-07-23

(Trainee (label): 101(property) ) (node)---------------------------------->(Book(label) : B1234(property) )(node)
**************************************************Reads(relation)**********************************
```

Problem Statement:

Consider the eCommerce user activity logs scenario wherein

- Every customer interaction with the shopping site is tracked
- Logs keep growing in size

Is graph-based database suitable for the given scenario?

Solution:

- Logging involves heavy writes. Graph databases do not scale well horizontally.
- Moreover, each write creates a new node and relationship. This greatly impacts existing graph traversals.
- Hence, graph databases is not suitable for the given scenario.

Identifying appropriate data models – Exercise:

Given each of the following scenarios of the online platform for employees to create posts and add comments (refer to exercise 1), identify the appropriate data model based on the type of data each scenario deals with.

1. Manage employee profile (maintains employee details)
2. Manage session (track employee details from the time of login to the time of log out)
3. Post articles (employees create/edit/delete articles)
4. Comments (employees view comments on a given post or can post their own comments)
5. Search (employees search for specific posts)
6. Recommendations (recommend employees about other related posts)
7. Reporting (daily/weekly/monthly/yearly usage reports)

Solution:

Appropriate data models identified for the given workloads:

1. Manage employee profile: **Document-oriented** for flexible structure
2. Manage session: **Key-value** (maintain session details with quick access)
3. Post articles: **Document-oriented** for flexible structure
4. Comments: **Column-family** (text based)

5. Search: **Document-oriented** (text-based search on different criteria such as date_of_post, title_of_post, employee_based_search and so on)
6. Recommendations: **Graph** (highly inter-related data)
7. Reporting: **SQL** (structured)

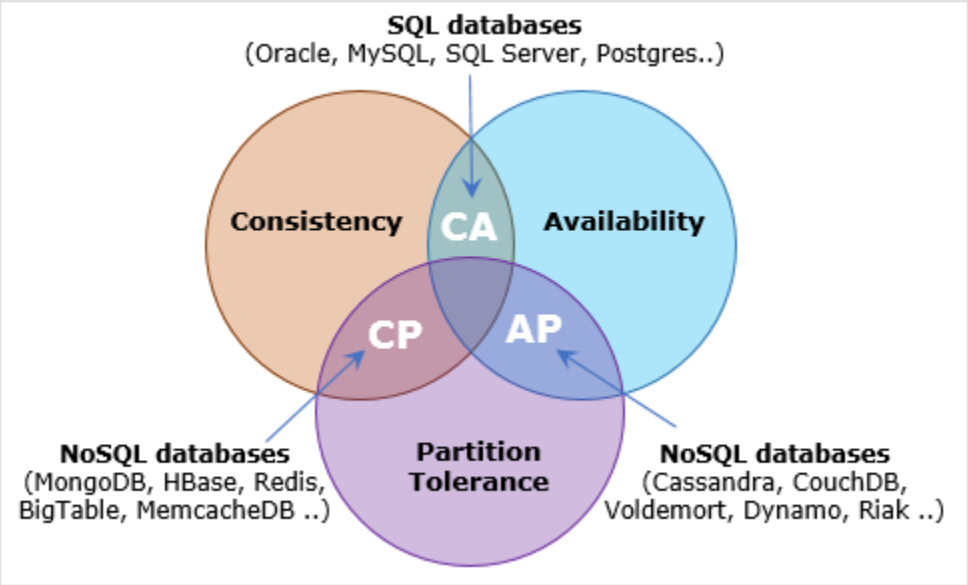
CAP theorem

This step uses CAP theorem (**C**onsistency, **A**vailability, **P**artition tolerance) to identify the level of consistency required. All databases should ideally provide the following features:

- *Consistency* – An end user should be able to view the latest data at all times
- *Availability* – Every database request must receive a response from the server while it is up and running
- *Partition tolerance* – When two systems cannot talk to each other in the network, it leads to partitions

CAP theorem states that

- Only two of the three properties can be guaranteed at the same
- Partitions occur only in distributed databases which SQL is not. Hence SQL databases support Consistency and Availability only
- NoSQL databases are always partition tolerant and hence you need to choose between Consistency and Availability



As seen in the figure above, NoSQL databases fall into one of the two categories:

1.2.a) CP databases (are Partition tolerant and support Consistency over Availability)

In an eCommerce system, when customers purchase products, corresponding stock quantities must be decremented at all other servers. If a network failure occurs, update to other servers is delayed until the link is restored. Would you allow the customer to purchase a product during a network failure?

The preferred choice is NO. Assume a user orders last available 10 T-Shirts of a particular brand and size. Other users would not be able to see that the product has gone out of stock. So, in this case, consistency is more important than availability.

1.2.b) AP databases (are Partition tolerant and support Availability over Consistency)

Consider the eCommerce review comments scenario. When a customer provides a product review, these comments would be displayed to all users. If a network failure occurs, latest comments are not visible to users at other servers. Do you think the customer should be allowed to review a product during the failure?

The answer is YES, as it is not mandatory that other users see all the reviews (some can appear later too). So, server availability is preferred over ensuring stronger consistency. This is the characteristic of AP databases.

At the end of this step, you can map the CAP scenario to each eCommerce component as shown below.

Component	**CAP Property**	**Reason for choosing**
Shopping Cart	Available	• There are generally two options provided to customers purchasing items "Add to cart" (shopping cart operation), and "Buy Now" (Payment operation) • "Add to cart" can be available to customers even during network failure, hence you choose Availability. Only when you reach the next stage, i.e, payment, you need consistency. Inventory is also reduced only on placement of order during payment, where consistency is required.

Component	CAP Property	Reason for choosing
User Activity Logs	Available	• Log data can be written to the current node only and later updated to other nodes once the network is restored
Product Catalog	Available	• Customers should be able to search for existing products in the catalog • It is fine if new products added to the catalog at other nodes are not immediately visible to all customers (They will be visible once the network is restored)
Recommendations	Available	• Relationships between existing products and customers can be used in recommendations. New relationships can appear once the network is restored

Note: The payment component in eCommerce would have chosen Consistency over Availability as it is important that the customer pays for the product only if it is in stock. So, during a network failure, payment request should not be processed by the system to get a consistent view of the product's availability.

Problem Statement:

Choose between consistency and availability for each of the following online employee platform (refer to exercise 1) scenarios:

1. Manage employee profile (maintains employee details)
2. Manage session (track employee details from the time of login to the time of log out)
3. Post articles (employees create/edit/delete articles)
4. Comments (employees view comments on a given post or can post their own comments)
5. Search (employees search for specific posts)
6. Recommendations (recommend employees about other related posts)
7. Reporting (daily/weekly/monthly/yearly usage reports)

Solution:

CAP property (Consistency/Availability) required for each given scenario is as follows:

1. Manage employee profile: Employee profile data needs to be **consistent**
2. Manage Session: Data tracked from the time of login to the time of log out for a given employee needs to be **consistent**
3. Post articles: Posting of articles should be **available**
4. Comments: **Availability** is chosen as comments may be updated to other machines at later point in time in case a network failure occurs
5. Search: This scenario needs to ensure higher **availability** as employees can search for the currently available articles wherein new articles can be displayed later when the network failure is repaired
6. Recommendations: The system can recommend related articles based on the currently **available** ones rather than making employees wait for the restoration of network for consistent information
7. Reporting: This component requires accuracy of reports and hence chooses **consistency** over availability

The last step in Stage 1 identifies whether your component requires to perform:

- too many reads compared to writes or
- too many writes compared to reads

Outcome of this step is as shown below.

Component	**Read or Write intensive**	**Reason for choosing read or write operations**
Shopping Cart	Write	• Customers keep adding new items, removing existing items or changing the quantities of some items in their carts rather than simply viewing them • Therefore, writes are more frequent than read operations on the cart

Component	Read or Write intensive	Reason for choosing read or write operations
User Activity Logs	Write	• Millions of customer interactions with the site are continuously being logged • Hence it is a write heavy operation
Product Catalog	Read	• Products are updated in the catalog fewer times than the number of times they are searched by millions of customers • Hence it is a read heavy operation
Recommendations	Read	• The eCommerce systems keeps recommending purchase of related products to its online customers • Hence it is a read heavy operation

Problem Statement:

Identify whether each given online employee platform (refer to exercise 1) scenario is read-intensive, write-intensive or both read/write intensive:

1. Manage employee profile (maintains employee details)
2. Manage session (track employee details from the time of login to the time of log out)
3. Post articles (employees create/edit/delete articles)
4. Comments (employees view comments on a given post or can post their own comments)
5. Search (employees search for specific posts)
6. Recommendations (recommend employees about other related posts)
7. Reporting (daily/weekly/monthly/yearly usage reports)

Solution:

1. Manage employee profile: **write-heavy** as thousands of employee details are being updated onto the database

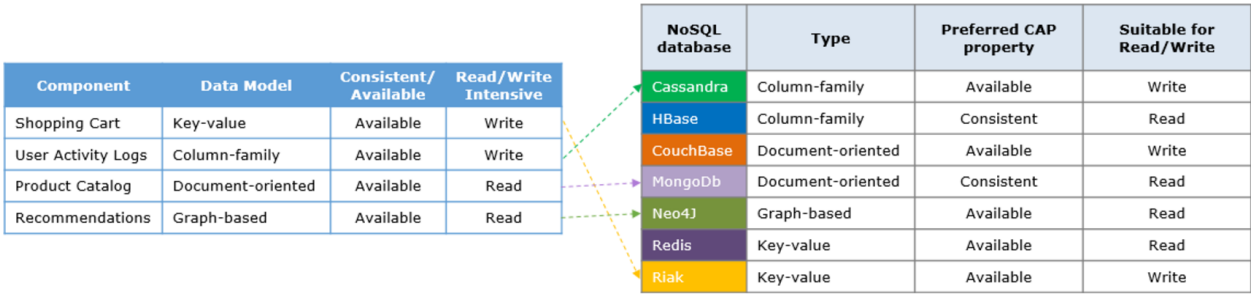
2. Manage session: involves both **read and write** loads for retrieving session details and tracking changes made during the session
3. Post articles: involves both **read and write** loads as existing articles are read by many employees and new articles are continuously posted by various employees
4. Comments: involves both **read and write** loads as employees read existing comments on articles and post their own comments
5. Search: involve heavy **reads** as employees keep searching for various articles
6. Recommendations: read-heavy as the system recommends related articles which would be useful to employees
7. Reporting: **read-heavy** as reports are being used for analysis of data

At the end of Stage 1, you can get together outcomes achieved at every step to the collective result as seen below.

Component	Data Model	Consistent/ Available	Read/ Write Intensive
Shopping Cart	Key-value	Available	Write
User Activity Logs	Column-family	Available	Write
Product Catalog	Document-oriented	Available	Read
Recommendations	Graph-based	Available	Read

Now you are ready to go ahead with Stage 2.

In Stage 2, you will be mapping the outcome of Stage 1 to the most suitable NoSQL database based on the required characteristics you are looking for.



Component	Data Model	Consistent/ Available	Read/Write Intensive
Shopping Cart	Key-value	Available	Write
User Activity Logs	Column-family	Available	Write
Product Catalog	Document-oriented	Available	Read
Recommendations	Graph-based	Available	Read

NoSQL database	Type	Preferred CAP property	Suitable for Read/Write
Cassandra	Column-family	Available	Write
HBase	Column-family	Consistent	Read
CouchBase	Document-oriented	Available	Write
MongoDb	Document-oriented	Consistent	Read
Neo4J	Graph-based	Available	Read
Redis	Key-value	Available	Read
Riak	Key-value	Available	Write

- **Shopping cart** needs a key-value store with high availability and write-intensive - **Riak** is most suitable
- Similarly, for **user activity logs** - **Cassandra** is the best fit
- For **product catalog** - **MongoDB** is most suitable (though consistent by default, it is tuned for availability as required)
- The **recommendations** component is best implemented using **Neo4J**

For each given online employee platform (refer to exercise 1) scenario, choose the most appropriate database.

1. Manage employee profile (maintains employee details)
2. Manage session (track employee details from the time of login to the time of log out)
3. Post articles (employees create/edit/delete articles)
4. Comments (employees view comments on a given post or can post their own comments)
5. Search (employees search for specific posts)
6. Recommendations (recommend employees about other related posts)
7. Reporting (daily/weekly/monthly/yearly usage reports)

Choosing the suitable database for each given scenario:

1. Manage employee profile: **MongoDB**
2. Manage session: **Redis**
3. Post articles: **MongoDB**
4. Comments: **Cassandra**
5. Search: **MongoDB**
6. Recommendations: **Neo4J**
7. Reporting: **SQL**

	Key-Value	Column-Family	Document-Oriented	Graph-Based
Databases	• Redis • Riak • ...	• Cassandra • HBase • ...	• MongoDB • Couchbase • ...	• Neo4j • OrientDB • ...
Features	• Simple design • Fast Read/Write • No indexes on non-key fields	• Storage not wasted on NULL values • Very useful for wide rows • Best suitable for Big Data Operations	• Flexible schema • Secondary Indexes • Rich query Langauge • Allows nested data structures	• Perfect for highly interrelated data • Data is stored in nodes and edges • Quick graph traversals
Use when	• You are just dealing with bytes/string • Your data is not highly related • All you need is basic CRUD	• You need very fast column operations including aggregation • Need compression or versioning	• You don't know much about the schema • The schema is likely to change often	• Your data looks more like a graph • Shortest path traversal

	Key-Value	Column-Family	Document-Oriented	Graph-Based
Suitable use cases	• Session information • User profiles, Preferences • Shopping cart	• Event logging • Content Management System (CMS), Blogging platforms • Expiring data	• Event logging • CMS, Blogging platforms • Web Analytics or real-time analytics • eCommerce (product catalog, orders)	• Routing, dispatch • Location-based services • Recommendation engines
Challenges	• Relationships among data • Query by value	• Need to know access patterns • Keys design is complex	• You need complex join-like queries • Circular dependencies	• Does not scale well horizontally
When not to use	• Not suitable for complex querying	• Column family design changes	• Data is highly related • If efficiency is preferred over consistency	• Updating all or subset of entities • Operations involving whole graph