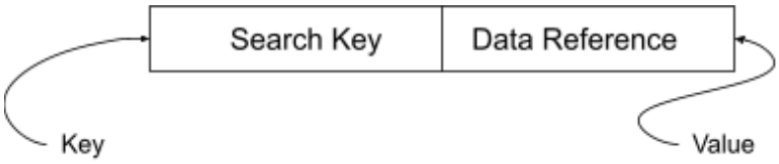
LEC-13: How to implement Atomicity and Durability in Transactions

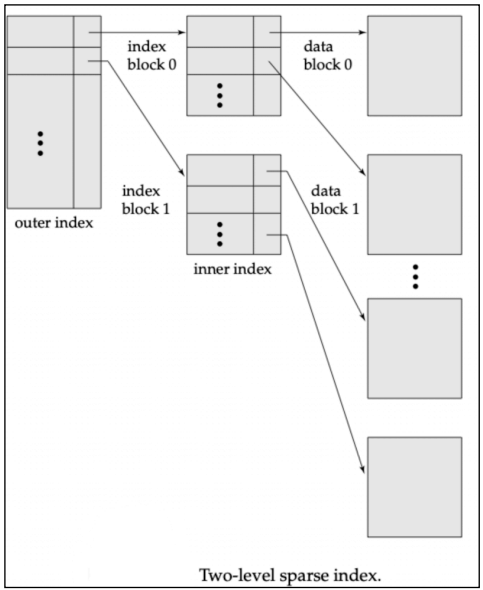
1. **Recovery Mechanism Component** of DBMS supports **atomicity and durability.**
2. **Shadow-copy scheme**
    1. Based on making copies of DB (aka, **shadow copies)**.
    2. Assumption only one Transaction (T) is active at a time.
    3. A pointer called **db-pointer** is maintained on the **disk**; which at any instant points to current copy of DB.
    4. T, that wants to update DB first creates a complete copy of DB.
    5. All further updates are done on new DB copy leaving the original copy (shadow copy) untouched.
    6. If at any point the **T has to be aborted** the system deletes the new copy. And the old copy is not affected.
    7. If T success, it is committed as,
        1. OS makes sure all the pages of the new copy of DB written on the disk.
        2. DB system updates the db-pointer to point to the new copy of DB.
        3. New copy is now the current copy of DB.
        4. The old copy is deleted.
        5. The T is said to have been **COMMITTED** at the point where the updated db-pointer is written to disk.
    8. **Atomicity**
        1. If T fails at any time before db-pointer is updated, the old content of DB are not affected.
        2. T abort can be done by just deleting the new copy of DB.
        3. Hence, either all updates are reflected or none.
    9. **Durability**
        1. Suppose, system fails are any time before the updated db-pointer is written to disk.
        2. When the system restarts, it will read db-pointer & will thus, see the original content of DB and none of the effects of T will be visible.
        3. T is assumed to be successful only when db-pointer is updated.
        4. If **system fails after** db-pointer has been updated. Before that all the pages of the new copy were written to disk. Hence, when system restarts, it will read new DB copy.
    10. The implementation is dependent on write to the db-pointer being atomic. Luckily, disk system provide atomic updates to entire block or at least a disk sector. So, we make sure db-pointer lies entirely in a single sector. By storing db-pointer at the beginning of a block.
    11. **Inefficient**, as entire DB is copied for every Transaction.
3. **Log-based recovery methods**
    1. The log is a sequence of records. Log of each transaction is maintained in some **stable storage** so that if any failure occurs, then it can be recovered from there.
    2. If any operation is performed on the database, then it will be recorded in the log.
    3. But the process of storing the logs should be done **before** the actual transaction is applied in the database.
    4. **Stable storage** is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures.
    5. **Deferred DB Modifications**
        1. Ensuring **atomicity** by recording all the DB modifications in the log but deferring the execution of all the write operations until the final action of the T has been executed.
        2. Log information is used to execute deferred writes when T is completed.
        3. If system crashed before the T completes, or if T is aborted, the information in the logs are ignored.
        4. If T completes, the records associated to it in the log file are used in executing the deferred writes.
        5. If failure occur while this updating is taking place, we preform redo.
    6. **Immediate DB Modifications**
        1. DB modifications to be output to the DB while the T is still in active state.
        2. DB modifications written by active T are called uncommitted modifications.
        3. In the event of crash or T failure, system uses old value field of the log records to restore modified values.
        4. Update takes place only after log records in a stable storage.
        5. Failure handling
            1. System failure before T completes, or if T aborted, then old value field is used to undo the T.
            2. If T completes and system crashes, then new value field is used to redo T having commit logs in the logs.

LEC-14: Indexing in DBMS

1. **Indexing** is used to **optimise the performance** of a database by minimising the number of disk accesses required when a query is processed.
2. The index is a type of **data structure.** It is used to locate and access the data in a database table quickly.
3. **Speeds up operation** with read operations like **SELECT** queries, **WHERE** clause etc.
4. **Search Key**: Contains copy of primary key or candidate key of the table or something else.
5. **Data Reference**: Pointer holding the address of disk block where the value of the corresponding key is stored.
6. Indexing is **optional**, but increases access speed. It is not the primary mean to access the tuple, it is the secondary mean.
7. **Index file is always sorted.**
8. **Indexing Methods**



   1. **Primary Index (Clustering Index)**
      1. A file may have several indices, on different search keys. If the data file containing the records is sequentially ordered, a Primary index is an index whose search key also defines the sequential order of the file.
      2. **NOTE**: The term primary index is sometimes used to mean an index on a primary key. However, such usage is **nonstandard** and **should be avoided.**
      3. All files are ordered sequentially on some search key. It could be Primary Key or non-primary key.
      4. **Dense And Sparse Indices**
         1. **Dense Index**
            1. The dense index contains an index record for every search key value in the data file.
            2. The index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search-key value would be stored sequentially after the first record.
            3. It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.
         2. **Sparse Index**
            1. An index record appears for only some of the search-key values.
            2. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.
      5. Primary Indexing can be based on Data file is sorted w.r.t Primary Key attribute or non-key attributes.
      6. **Based on Key attribute**
         1. Data file is sorted w.r.t primary key attribute.
         2. PK will be used as search-key in Index.
         3. Sparse Index will be formed i.e., no. of entries in the index file = no. of blocks in datafile.
      7. **Based on Non-Key attribute**
         1. Data file is sorted w.r.t non-key attribute.
         2. No. Of entries in the index = unique non-key attribute value in the data file.
         3. This is dense index as, all the unique values have an entry in the index file.
         4. E.g., Let's assume that a company recruited many employees in various departments. In this case, clustering indexing in DBMS should be created for all employees who belong to the same dept.
      8. **Multi-level Index**
         1. Index with two or more levels.
         2. If the single level index become enough large that the binary search it self would take much time, we can break down indexing into multiple levels.



   2. **Secondary Index (Non-Clustering Index)**
      1. Datafile is unsorted. Hence, Primary Indexing is not possible.
      2. Can be done on key or non-key attribute.
      3. Called secondary indexing because normally one indexing is already applied.
      4. No. Of entries in the index file = no. of records in the data file.
      5. It's an example of Dense index.

9. **Advantages of Indexing**
    1. Faster access and retrieval of data.
    2. IO is less.
10. **Limitations of Indexing**
    1. Additional space to store index table
    2. Indexing Decrease performance in INSERT, DELETE, and UPDATE query.

LEC-15: NoSQL

1. **NoSQL databases** (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide **flexible schemas** and **scale** easily with **large amounts of data** and **high user loads**.
    1. They are schema free.
    2. Data structures used are not tabular, they are more flexible, has the ability to adjust dynamically.
    3. Can handle huge amount of data **(big data).**
    4. Most of the NoSQL are open sources and has the capability of horizontal scaling.
    5. **It just stores data in some format other than relational.**
2. **History behind NoSQL**
    1. NoSQL databases emerged in the late 2000s as the cost of storage dramatically decreased. Gone were the days of needing to create a complex, difficult-to-manage data model in order to avoid data duplication. Developers (rather than storage) were becoming the primary cost of software development, so NoSQL databases optimised for developer productivity.
    2. Data becoming unstructured more, hence structuring (defining schema in advance) them had becoming costly.
    3. NoSQL databases allow developers to store huge amounts of unstructured data, giving them a lot of flexibility.
    4. Recognising the need to rapidly adapt to changing requirements in a software system. Developers needed the ability to iterate quickly and make changes throughout their software stack — all the way down to the database. NoSQL databases gave them this flexibility.
    5. Cloud computing also rose in popularity, and developers began using public clouds to host their applications and data. They wanted the ability to distribute data across multiple servers and regions to make their applications resilient, to scale out instead of scale up, and to intelligently geo-place their data. Some NoSQL databases like MongoDB provide these capabilities.
3. **NoSQL Databases Advantages**
    A. **Flexible Schema**
        1. RDBMS has pre-defined schema, which become an issue when we do not have all the data with us or we need to change the schema. It's a huge task to change schema on the go.

    B. **Horizontal Scaling**
        1. Horizontal scaling, also known as scale-out, refers to bringing on additional nodes to share the load. This is difficult with relational databases due to the difficulty in spreading out related data across nodes. With non-relational databases, this is made simpler since collections are self-contained and not coupled relationally. This allows them to be distributed across nodes more simply, as queries do not have to "join" them together across nodes.
        2. Scaling horizontally is achieved through **Sharding** OR **Replica-sets**.

    C. **High Availability**
        1. NoSQL databases are highly available due to its auto replication feature i.e. whenever any kind of failure happens data replicates itself to the preceding consistent state.
        2. If a server fails, we can access that data from another server as well, as in NoSQL database data is stored at multiple servers.

    D. **Easy insert and read operations.**
        1. Queries in NoSQL databases can be faster than SQL databases. Why? Data in SQL databases is typically normalised, so queries for a single object or entity require you to join data from multiple tables. As your tables grow in size, the joins can become expensive. However, data in NoSQL databases is typically stored in a way that is optimised for queries. The rule of thumb when you use MongoDB is data that is accessed together should be stored together. Queries typically do not require joins, so the queries are very fast.
        2. But difficult delete or update operations.

    E. **Caching** mechanism.

    F. **NoSQL** use case is more for **Cloud** applications.
4. **When to use NoSQL?**
    1. Fast-paced Agile development
    2. Storage of structured and semi-structured data
    3. Huge volumes of data
    4. Requirements for scale-out architecture
    5. Modern application paradigms like micro-services and real-time streaming.
5. **NoSQL DB Misconceptions**
    1. Relationship data is best suited for relational databases.
        1. A common misconception is that NoSQL databases or non-relational databases don't store relationship data well. NoSQL databases can store relationship data — they just store it differently than relational databases do. In fact, when compared with relational databases, many find modelling relationship data in NoSQL databases to be easier than in relational databases, because related data doesn't have to be split between tables. NoSQL data models allow related data to be nested within a single data structure.
    2. NoSQL databases don't support ACID transactions.

1. Another common misconception is that NoSQL databases don't support ACID transactions. Some NoSQL databases like MongoDB do, in fact, support ACID transactions.

6. Types of NoSQL Data Models

1. Key-Value Stores

1. The simplest type of NoSQL database is a key-value store. Every data element in the database is stored as a key value pair consisting of an attribute name (or "key") and a value. In a sense, a key-value store is like a relational database with only two columns: the key or attribute name (such as "state") and the value (such as "Alaska").
2. **Use cases** include shopping carts, user preferences, and user profiles.
3. e.g., Oracle NoSQL, Amazon DynamoDB, MongoDB also supports Key-Value store, Redis.
4. A key-value database associates a value (which can be anything from a number or simple string to a complex object) with a key, which is used to keep track of the object. In its simplest form, a key-value store is like a dictionary/array/map object as it exists in most programming paradigms, but which is stored in a persistent way and managed by a Database Management System (DBMS).
5. Key-value databases use compact, efficient index structures to be able to quickly and reliably locate a value by its key, making them ideal for systems that need to be able to find and retrieve data in constant time.
6. There are several use-cases where choosing a key value store approach is an optimal solution:
   a) Real time random data access, e.g., user session attributes in an online application such as gaming or finance.
   b) Caching mechanism for frequently accessed data or configuration based on keys.
   c) Application is designed on simple key-based queries.

2. Column-Oriented / Columnar / C-Store / Wide-Column

1. The data is stored such that each row of a column will be next to other rows from that same column.
2. While a relational database stores data in rows and reads data row by row, a column store is organised as a set of columns. This means that when you want to run analytics on a small number of columns, you can read those columns directly without consuming memory with the unwanted data. Columns are often of the same type and benefit from more efficient compression, making reads even faster. Columnar databases can quickly aggregate the value of a given column (adding up the total sales for the year, for example). **Use cases** include analytics.
3. e.g., Cassandra, RedShift, Snowflake.

3. Document Based Stores

1. This DB store data in documents similar to JSON (JavaScript Object Notation) objects. Each document contains pairs of fields and values. The values can typically be a variety of types including things like strings, numbers, booleans, arrays, or objects.
2. **Use cases** include e-commerce platforms, trading platforms, and mobile app development across industries.
3. Supports ACID properties hence, suitable for Transactions.
4. e.g., MongoDB, CouchDB.

4. Graph Based Stores

1. A graph database focuses on the relationship between data elements. Each element is stored as a node (such as a person in a social media graph). The connections between elements are called links or relationships. In a graph database, connections are first-class elements of the database, stored directly. In relational databases, links are implied, using data to express the relationships.
2. A graph database is optimised to capture and search the connections between data elements, overcoming the overhead associated with JOINing multiple tables in SQL.
3. Very few real-world business systems can survive solely on graph queries. As a result graph databases are usually run alongside other more traditional databases.
4. **Use cases** include fraud detection, social networks, and knowledge graphs.

7. NoSQL Databases Dis-advantages

1. Data Redundancy

1. Since data models in NoSQL databases are typically optimised for queries and not for reducing data duplication, NoSQL databases can be larger than SQL databases. Storage is currently so cheap that most consider this a minor drawback, and some NoSQL databases also support compression to reduce the storage footprint.

2. Update & Delete operations are **costly**.
3. All type of NoSQL Data model doesn't fulfil all of your application needs
   1. Depending on the NoSQL database type you select, you may not be able to achieve all of your use cases in a single database. For example, graph databases are excellent for analysing relationships in your data but may not provide what you need for everyday retrieval of the data such as range queries. When selecting a NoSQL database, consider what your use cases will be and if a general purpose database like MongoDB would be a better option.
4. Doesn't support ACID properties in general.
5. Doesn't support data entry with consistency constraints.

8. SQL vs NoSQL

	SQL Databases	NoSQL Databases
Data Storage Model	Tables with fixed rows and columns	Document: JSON documents, Key-value: key-value pairs, Wide-column: tables with rows and dynamic columns, Graph: nodes and edges
Development History	Developed in the 1970s with a focus on reducing data duplication	Developed in the late 2000s with a focus on scaling and allowing for rapid application change driven by agile and DevOps practices.
Examples	Oracle, MySQL, Microsoft SQL Server, and PostgreSQL	Document: MongoDB and CouchDB, Key-value: Redis and DynamoDB, Wide-column: Cassandra and HBase, Graph: Neo4j and Amazon Neptune
Primary Purpose	General Purpose	Document: general purpose, Key-value: large amounts of data with simple lookup queries, Wide-column: large amounts of data with predictable query patterns, Graph: analyzing and traversing relationships between connected data
Schemas	Fixed	Flexible
Scaling	Vertical (Scale-up)	Horizontal (scale-out across commodity servers)
ACID Properties	Supported	Not Supported, except in DB like MongoDB etc.
JOINS	Typically Required	Typically not required
Data to object mapping	Required object-relational mapping	Many do not require ORMs. MongoDB documents map directly to data structures in most popular programming languages.

LEC-16: Types of Databases

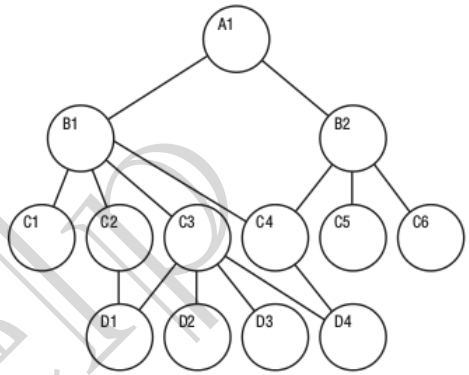
1. **Relational Databases**
    1. Based on Relational Model.
    2. Relational databases are quite **popular**, even though it was a system designed in the 1970s. Also known as relational database management systems (RDBMS), relational databases commonly use **Structured Query Language (SQL)** for operations such as **creating**, **reading**, **updating**, and **deleting** data. Relational databases store information in **discrete tables**, which can be **JOIN**ed together by fields known as **foreign** keys. For example, you might have a User table which contains information about all your users, and **join** it to a Purchases table, which contains information about all the purchases they've made. MySQL, Microsoft SQL Server, and Oracle are types of relational databases.
    3. they are ubiquitous, having acquired a steady user base since the 1970s
    4. they are highly optimised for working with structured data.
    5. they provide a stronger guarantee of data normalisation
    6. they use a well-known querying language through SQL
    7. **Scalability issues** (Horizontal Scaling).
    8. Data become huge, system become more complex.
2. **Object Oriented Databases**
    1. The object-oriented data model, is based on the **object-oriented-programming paradigm,** which is now in wide use. **Inheritance**, **object-identity**, and **encapsulation** (information hiding), with methods to provide an interface to objects, are among the key concepts of object-oriented programming that have found applications in data modelling. The object-oriented data model also supports a rich type system, including structured and collection types. While inheritance and, to some extent, complex types are also present in the E-R model, encapsulation and object-identity distinguish the object-oriented data model from the E-R model.
    2. Sometimes the database can be very complex, having multiple relations. So, maintaining a relationship between them can be tedious at times.
        1. In Object-oriented databases data is treated as an object.
        2. All bits of information come in one instantly available object package instead of multiple tables.
    3. **Advantages**
        1. Data storage and retrieval is easy and quick.
        2. Can handle complex data relations and more variety of data types that standard relational databases.
        3. Relatively friendly to model the advance real world problems
        4. Works with functionality of OOPs and Object Oriented languages.
    4. **Disadvantages**
        1. High complexity causes performance issues like read, write, update and delete operations are slowed down.
        2. Not much of a community support as isn't widely adopted as relational databases.
        3. Does not support views like relational databases.
    5. e.g., ObjectDB, GemStone etc.
3. **NoSQL Databases**
    1. NoSQL databases (aka "not only SQL") are non-tabular databases and store data differently than relational tables. NoSQL databases come in a variety of types based on their data model. The main types are document, key-value, wide-column, and graph. They provide flexible schemas and scale easily with large amounts of data and high user loads.
    2. They are schema free.
    3. Data structures used are not tabular, they are more flexible, has the ability to adjust dynamically.
    4. Can handle huge amount of data (big data).
    5. Most of the NoSQL are open sources and has the capability of horizontal scaling.
    6. It just stores data in some format other than relational.
    7. Refer **LEC-15** notes...
4. **Hierarchical Databases**
    1. As the name suggests, the hierarchical database model is most appropriate for use cases in which the main focus of information gathering is based on a **concrete hierarchy**, such as several individual employees reporting to a single department at a company.
    2. The schema for hierarchical databases is defined by its **tree-like** organisation, in which there is typically a **root** "parent" directory of data stored as records that links to various other subdirectory branches, and each subdirectory branch, or child record, may link to various other subdirectory branches.
    3. The hierarchical database structure dictates that, while a parent record can have several child records, each child record can only have **one parent** record. Data within records is stored in the form of fields, and each field can only contain one value. Retrieving hierarchical data from a hierarchical database architecture requires traversing the entire tree, starting at the root node.
    4. Since the **disk storage system** is also inherently a hierarchical structure, these models can also be used as physical models.
    5. The key **advantage** of a hierarchical database is its ease of use. The one-to-many organisation of data makes traversing the database simple and fast, which is ideal for use cases such as website drop-down menus or computer folders in systems like

Microsoft Windows OS. Due to the separation of the tables from physical storage structures, information can easily be added or deleted without affecting the entirety of the database. And most major programming languages offer functionality for reading tree structure databases.

6. The major **disadvantage** of hierarchical databases is their inflexible nature. The one-to-many structure is not ideal for complex structures as it cannot describe relationships in which each child node has multiple parents nodes. Also the tree-like organisation of data requires top-to-bottom sequential searching, which is time consuming, and requires repetitive storage of data in multiple different entities, which can be redundant.
7. e.g., IBM IMS.

5. Network Databases

1. **Extension** of Hierarchical databases
2. The child records are given the freedom to associate with multiple parent records.
3. Organised in a **Graph** structure.
4. Can handle complex relations.
5. Maintenance is tedious.
6. **M:N links** may cause slow retrieval.
7. Not much web community support.
8. e.g., Integrated Data Store (IDS), IDMS (Integrated Database Management System), Raima Database Manager, TurboIMAGE etc.



LEC-17: Clustering in DBMS

1. **Database Clustering** (making **Replica-sets**) is the process of combining more than one servers or instances connecting a single database. Sometimes one server may not be adequate to manage the amount of data or the number of requests, that is when a Data Cluster is needed. Database clustering, SQL server clustering, and SQL clustering are closely associated with SQL is the language used to manage the database information.
2. Replicate the same dataset on different servers.
3. **Advantages**
    1. **Data Redundancy:** Clustering of databases helps with data redundancy, as we store the same data at multiple servers. Don't confuse this data redundancy as repetition of the same data that might lead to some anomalies. The redundancy that clustering offers is required and is quite certain due to the synchronisation. In case any of the servers had to face a failure due to any possible reason, the data is available at other servers to access.
    2. **Load balancing:** or scalability doesn't come by default with the database. It has to be brought by clustering regularly. It also depends on the setup. Basically, what load balancing does is allocating the workload among the different servers that are part of the cluster. This indicates that more users can be supported and if for some reasons if a huge spike in the traffic appears, there is a higher assurance that it will be able to support the new traffic. One machine is not going to get all of the hits. This can provide **scaling** seamlessly as required. This **links directly to high availability.** Without load balancing, a particular machine could get overworked and traffic would slow down, leading to decrement of the traffic to zero.
    3. **High availability:** When you can access a database, it implies that it is available. High availability refers the amount of time a database is considered available. The amount of availability you need greatly depends on the number of transactions you are running on your database and how often you are running any kind of analytics on your data. With database clustering, we can reach extremely high levels of availability due to load balancing and have extra machines. In case a server got shut down the database will, however, be available.
4. **How does Clustering Work?**
    1. In cluster architecture, all requests are split with many computers so that an individual user request is executed and produced by a number of computer systems. The clustering is serviceable definitely by the ability of load balancing and high-availability. If one node collapses, the request is handled by another node. Consequently, there are few or no possibilities of absolute system failures.

LEC-18: Partitioning & Sharding in DBMS (DB Optimisation)

1. **A big problem** can be solved easily when it is chopped into several smaller sub-problems. That is what the partitioning technique does. It divides a big database containing data metrics and indexes into smaller and handy slices of data called partitions. The partitioned tables are directly used by SQL queries without any alteration. Once the database is partitioned, the data definition language can easily work on the smaller partitioned slices, instead of handling the giant database altogether. This is how partitioning cuts down the problems in managing large database tables.
2. **Partitioning** is the technique used to divide stored database objects into separate servers. Due to this, there is an increase in performance, controllability of the data. We can manage huge chunks of data optimally. When we horizontally scale our machines/servers, we know that it gives us a challenging time dealing with relational databases as it's quite tough to maintain the relations. But if we apply partitioning to the database that is already scaled out i.e. equipped with multiple servers, we can partition our database among those servers and handle the big data easily.

3. Vertical Partitioning

1. Slicing relation vertically / column-wise.
2. Need to access different servers to get complete tuples.

4. Horizontal Partitioning

1. Slicing relation horizontally / row-wise.
2. Independent chunks of data tuples are stored in different servers.

5. When Partitioning is Applied?

1. Dataset become much huge that managing and dealing with it become a tedious task.
2. The number of requests are enough larger that the single DB server access is taking huge time and hence the system's response time become high.

6. Advantages of Partitioning

1. Parallelism
2. Availability
3. Performance
4. Manageability
5. Reduce Cost, as scaling-up or vertical scaling might be costly.

7. Distributed Database

1. A single logical database that is, spread across multiple locations (servers) and logically interconnected by network.
2. This is the product of applying DB optimisation techniques like **Clustering, Partitioning and Sharding.**
3. Why this is needed? READ Point **5**.

8. Sharding

1. Technique to implement Horizontal Partitioning.
2. The **fundamental idea** of Sharding is the idea that instead of having all the data sit on one DB instance, we split it up and introduce a Routing layer so that we can forward the request to the right instances that actually contain the data.

3. Pros

1. Scalability
2. Availability

4. Cons

1. Complexity, making partition mapping, Routing layer to be implemented in the system, Non-uniformity that creates the necessity of Re-Sharding
2. Not well suited for Analytical type of queries, as the data is spread across different DB instances. (Scatter-Gather problem)

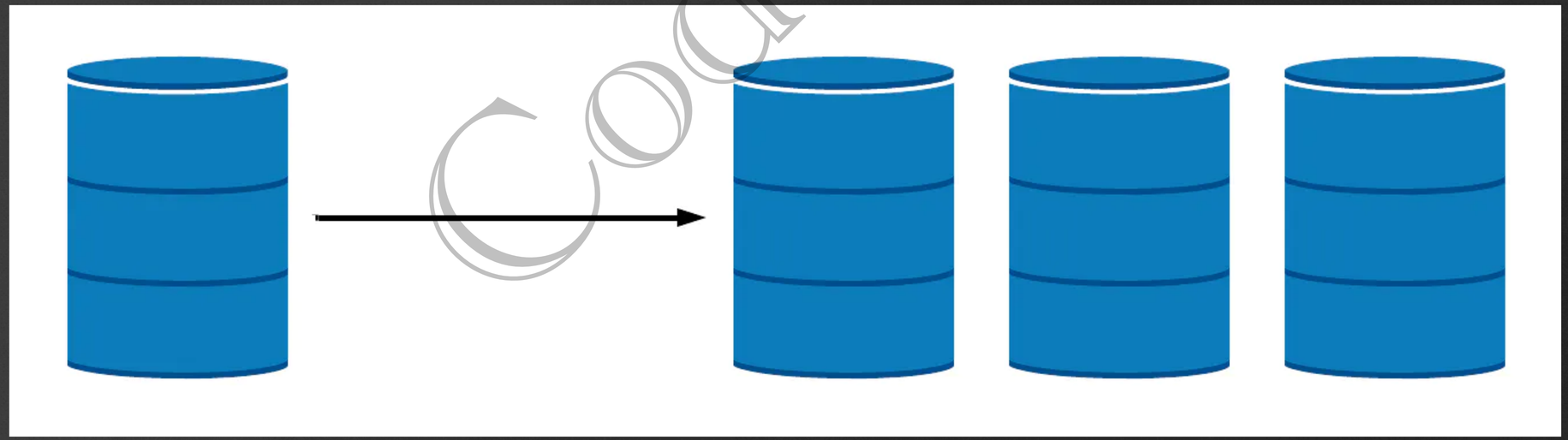
Database Scaling Patterns

Step by Step Scaling

- Lakshay

What will you learn?

- Step by Step manner, when to choose which Scaling option.
- Which Scaling option is feasible practically at the moment.



A Case Study

Cab Booking APP

- Tiny startup.
- ~10 customers onboard.
- A single small machine DB stores all customers, trips, locations, booking data, and customer trip history.
- ~1 trip booking in 5 mins.

Your App becoming famous, but…

The PROBLEM begins

- Requests scales upto 30 bookings per minute.
- Your tiny DB system has started performing poorly.
- API latency has increased a lot.
- Transactions facing Deadlock, Starvation, and frequent failure.
- Sluggish App experience.
- Customer dis-satisfaction.

Is there any solution?

- We have to apply some kind of performance optimisation measures.
- We might have to scale our system going forward.

Pattern 1

Query Optimisation & Connection Pool Implementation

- Cache frequently used non-dynamic data like, booking history, payment history, user profiles etc.
- Introduce Database Redundancy. (Or may be use NoSQL)
- Use connection pool libraries to **Cache DB connections**.
- Multiple application threads can use same DB connection.
- Good optimisations as of now.
- Scaled the business to one more city, and now getting ~100 booking per minute.

Pattern 2

Vertical Scaling or Scale-up

- Upgrading our initial tiny machine.
- RAM by 2x and SSD by 3x etc.
- Scale up is pocket friendly till a point only.
- More you scale up, cost increases exponentially.
- Good Optimisation as of now.
- Business is growing, you decided to scale it to 3 more cities and now getting 300 booking per minute.

Pattern 3

Command Query Responsibility Segregation (CQRS)

- The scaled up big machine is not able to handle all read/write requests.
- Separate read/write operations physical machine wise.
- 2 more machines as replica to the primary machine.
- All read queries to replicas.
- All write queries to primary.
- Business is growing, you decided to scale it to 2 more cities.
- Primary is not able to handle all write requests.
- Lag between primary and replica is impacting user experience.

Pattern 4

Multi Primary Replication

- Why not distribute write request to replica also?
- All machines can work as primary & replica.
- Multi primary configuration is a logical circular ring.
- Write data to any node.
- Read data from any node that replies to the broadcast first.
- You scale to 5 more cities & your system is in pain again. (~50 req/s)

Pattern 5

Partitioning of Data by Functionality

- What about separating the location tables in separate DB schema?
- What about putting that DB in separate machines with primary-replica or multi-primary configuration?
- Different DB can host data categorised by different functionality.
- Backend or application layer has to take responsibility to join the results.
- Planning to expand your business to other country.

Pattern 6

Horizontal Scaling or Scale-out

- Sharding - multiple shards.
- Allocate 50 machines - all having same DB schema - each machine just hold a part of data.
- Locality of data should be there.
- Each machine can have their own replicas, may be used in failure recovery.
- Sharding is generally hard to apply. But “No Pain, No Gain”
- Scaling the business across continents.

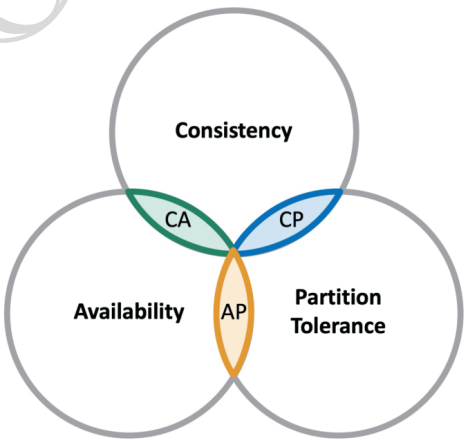
Pattern 7

Data Centre Wise Partition

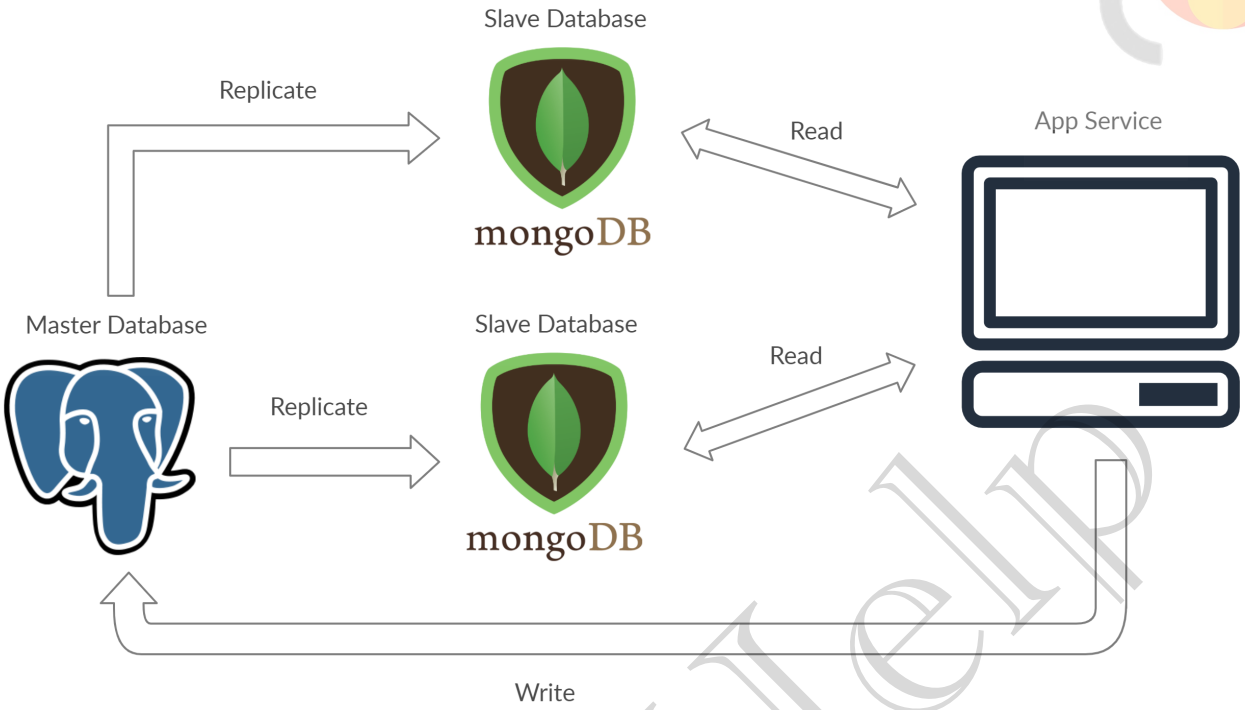
- Requests travelling across continents are having high latency.
- What about distributing traffic across data centres?
- Data centres across continents.
- Enable cross data centre replication which helps disaster recovery.
- This always maintain Availability of your system.
- Now, Plan for an IPO :p

LEC-20: CAP Theorem

1. Basic and one of the most important concept in **Distributed Databases.**
2. **Useful** to know this to design efficient distributed system for your given business logic.
3. Let's first breakdown CAP
    1. **Consistency**: In a consistent system, all nodes see the same data simultaneously. If we perform a read operation on a consistent system, it should return the value of the most recent write operation. The read should cause all nodes to return the same data. All users see the same data at the same time, regardless of the node they connect to. When data is written to a single node, it is then replicated across the other nodes in the system.
    2. **Availability**: When availability is present in a distributed system, it means that the system remains operational all of the time. Every request will get a response regardless of the individual state of the nodes. This means that the system will operate even if there are multiple nodes down. Unlike a consistent system, there's no guarantee that the response will be the most recent write operation.
    3. **Partition Tolerance**: When a distributed system encounters a partition, it means that there's a break in communication between nodes. If a system is partition-tolerant, the system does not fail, regardless of whether messages are dropped or delayed between nodes within the system. To have partition tolerance, the system must replicate records across combinations of nodes and networks.
4. What does the **CAP Theorem** says,
    1. The CAP theorem states that a distributed system can only provide **two of three properties** simultaneously: consistency, availability, and partition tolerance. The theorem formalises the **tradeoff between consistency and availability when there's a partition.**
5. **CAP Theorem NoSQL Databases:** NoSQL databases are great for distributed networks. They allow for horizontal scaling, and they can quickly scale across multiple nodes. When deciding which NoSQL database to use, it's important to keep the CAP theorem in mind.
    1. **CA Databases:** CA databases enable consistency and availability across all nodes. Unfortunately, CA databases can't deliver fault tolerance. In any distributed system, partitions are bound to happen, which means this type of database isn't a very practical choice. That being said, you still can find a CA database if you need one. Some relational databases, such as MySQL or PostgreSQL, allow for consistency and availability. You can deploy them to nodes using replication.
    2. **CP Databases:** CP databases enable consistency and partition tolerance, but not availability. When a partition occurs, the system has to turn off inconsistent nodes until the partition can be fixed. MongoDB is an example of a CP database. It's a NoSQL database management system (DBMS) that uses documents for data storage. It's considered schema-less, which means that it doesn't require a defined database schema. It's commonly used in big data and applications running in different locations. The CP system is structured so that there's **only one primary node that receives all of the write requests in a given replica set.** Secondary nodes replicate the data in the primary nodes, so if the primary node fails, a secondary node can stand-in. In banking system Availability is not as important as consistency, so we can opt it (MongoDB).
    3. **AP Databases:** AP databases enable availability and partition tolerance, but not consistency. In the event of a partition, all nodes are available, but they're not all updated. For example, if a user tries to access data from a bad node, they won't receive the most up-to-date version of the data. When the partition is eventually resolved, most AP databases will sync the nodes to ensure consistency across them. Apache Cassandra is an example of an AP database. It's a NoSQL database with no primary node, meaning that all of the nodes remain available. Cassandra allows for eventual consistency because users can re-sync their data right after a partition is resolved. For apps like Facebook, we value availability more than consistency, we'd opt for AP Databases like Cassandra or Amazon DynamoDB.



LEC-21: The Master-Slave Database Concept



1. Master-Slave is a general way to optimise IO in a system where number of requests goes way high that a single DB server is not able to handle it efficiently.
2. Its a Pattern 3 in LEC-19 (Database Scaling Pattern). ***(Command Query Responsibility Segregation)***
3. The true or latest data is kept in the Master DB thus write operations are directed there. Reading ops are done only from slaves. This architecture serves the purpose of safeguarding site ***eliability, availability, reduce latency etc .*** If a site receives a lot of traffic and the only available database is one master, it will be overloaded with reading and writing requests. Making the entire system slow for everyone on the site.
4. DB ***replication*** will take care of distributing data from Master machine to Slaves machines. This can be ***synchronous or asynchronous*** depending upon the system's need.