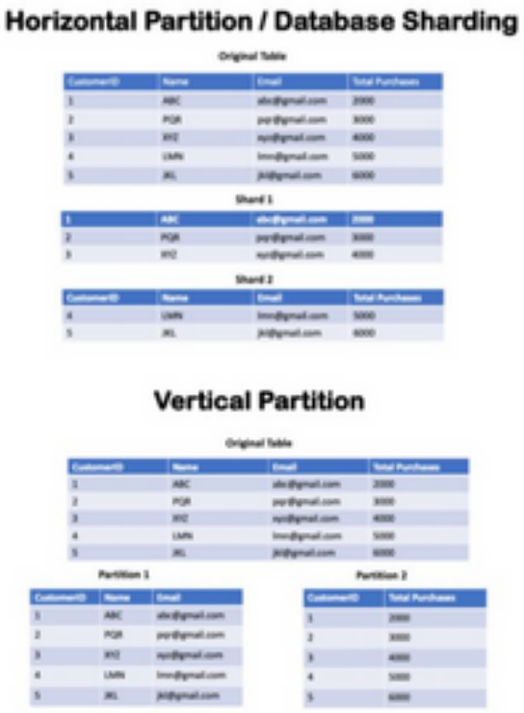
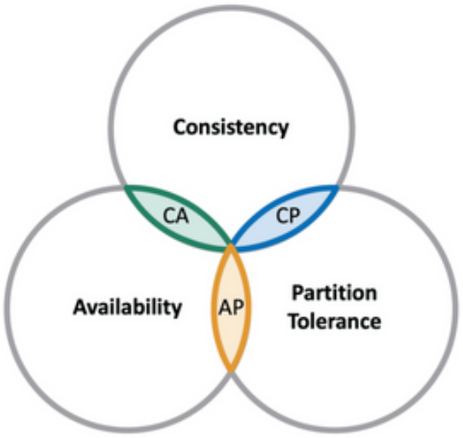
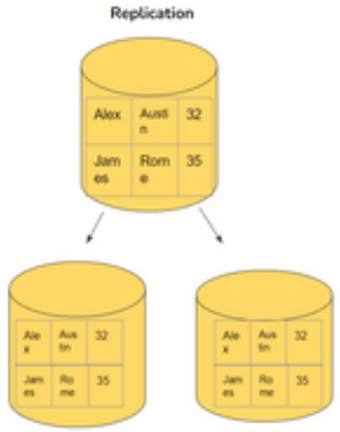






C.A.P Theorem & Database Sharding, Partitioning, Replication, Normalization, Mirroring, Leader Election (How Amazon elects a leader), Indexing, Consistency, Locks, ACID Vs Base

C.A.P Theorem

The CAP theorem, originally introduced as the CAP principle, can be used to explain some of the competing requirements in a distributed system with replication. It is a tool used to make system designers aware of the trade-offs while designing networked shared-data systems.

The three letters in CAP refer to three desirable properties of distributed systems with replicated data: consistency (among replicated copies), availability (of the system for read and write operations) and partition tolerance (in the face of the nodes in the system being partitioned by a network fault).

The CAP theorem states that it is not possible to guarantee all three of the desirable properties – consistency, availability, and partition tolerance at the same time in a distributed system with data replication.

The theorem states that networked shared-data systems can only strongly support two of the following three properties:

Consistency:

Consistency means that the nodes will have the same copies of a replicated data item visible for various transactions. A guarantee that every node in a distributed cluster returns the same, most recent and a successful write.

Consistency refers to every client having the same view of the data. There are various types of consistency models. Consistency in CAP refers to sequential consistency, a very strong form of consistency.

Availability:

Availability means that each read or write request for a data item will either be processed successfully or will receive a message that the operation cannot be completed. Every non-failing node returns a response for all the read and write requests in a reasonable amount of time. The key word here is “every”. In simple terms, every node (on either side of a network partition) must be able to respond in a reasonable amount of time.

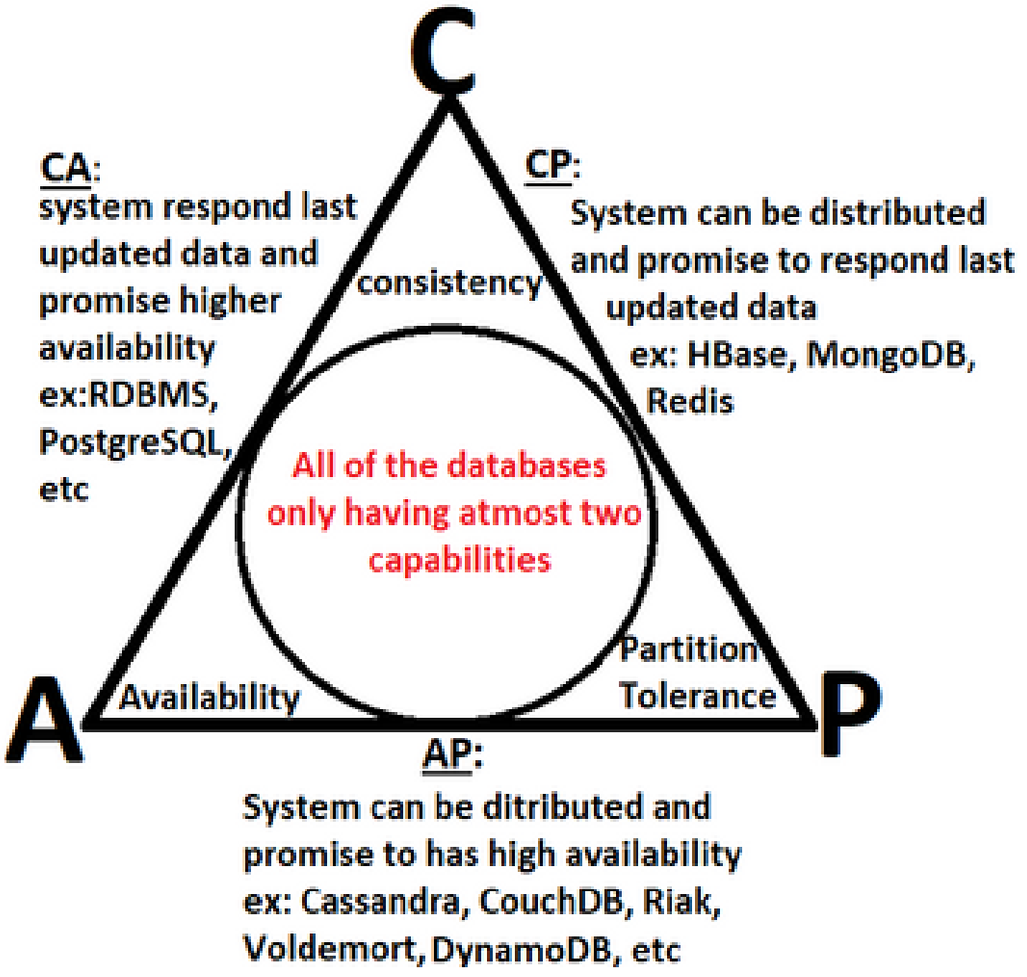
Partition Tolerance:

Partition tolerance means that the system can continue operating even if the network connecting the nodes has a fault that results in two or more partitions, where the nodes in each partition can only communicate among each other. That means, the system continues to function and upholds its consistency guarantees in spite of network partitions. Network partitions are a fact of life. Distributed systems guaranteeing partition tolerance can gracefully recover from partitions once the partition heals.

The use of the word consistency in CAP and its use in ACID do not refer to the same identical concept.

In CAP, the term consistency refers to the consistency of the values in different copies of the same data item in a replicated distributed system. In ACID, it refers to the fact that a transaction will not violate the integrity constraints specified on the database schema.

The CAP theorem states that distributed databases can have at most two of the three properties: consistency, availability, and partition tolerance. As a result, database systems prioritize only two properties at a time.



CA(Consistency and Availability):

The system prioritizes availability over consistency and can respond with possibly stale data.

Example databases: Cassandra, CouchDB, Riak, Voldemort, RDBMS, PostgreSQL

AP(Availability and Partition Tolerance):

The system prioritizes availability over consistency and can respond with possibly stale data. The system can be distributed across multiple nodes and is designed to operate reliably even in the face of network partitions.

Example databases: Amazon DynamoDB, Google Cloud Spanner.

CP(Consistency and Partition Tolerance):

The system prioritizes consistency over availability and responds with the latest updated data. The system can be distributed across multiple nodes and is designed to operate reliably even in the face of network partitions.

Example databases: Apache HBase, MongoDB, Redis.

It’s important to note that these database systems may have different configurations and settings that can change their behavior with respect to consistency, availability, and partition tolerance. Therefore, the exact behavior of a database system may depend on its configuration and usage.

for example, Neo4j, a graph database, the CAP theorem still applies. Neo4j prioritizes consistency and partition tolerance over availability, which means that in the event of a network partition or failure, Neo4j will sacrifice availability to maintain consistency.

More on the ‘CAP’ in the CAP theorem

Let’s take a detailed look at the three distributed system characteristics to which the CAP theorem refers.

Consistency

Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed ‘successful.’

Availability

Availability means that any client making a request for data gets a response, even if one or more nodes are down. Another way to state this—all working nodes in the distributed system return a valid response for any request, without exception.

Partition tolerance

A partition is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.

CAP theorem NoSQL database types

NoSQL databases are ideal for distributed network applications. Unlike their vertically scalable SQL (relational) counterparts, NoSQL databases are horizontally scalable and distributed by design—they can rapidly scale across a growing network consisting of multiple interconnected nodes. (See "SQL vs. NoSQL Databases: What's the Difference?" for more information.)

Today, NoSQL databases are classified based on the two CAP characteristics they support:

- **CP database:** A CP database delivers consistency and partition tolerance at the expense of availability. When a partition occurs between any two nodes, the system has to shut down the non-consistent node (i.e., make it unavailable) until the partition is resolved.

- **AP database:** An AP database delivers availability and partition tolerance at the expense of consistency. When a partition occurs, all nodes remain available but those at the wrong end of a partition might return an older version of data than others. (When the partition is resolved, the AP databases typically resync the nodes to repair all inconsistencies in the system.)

- **CA database:** A CA database delivers consistency and availability across all nodes. It can't do this if there is a partition between any two nodes in the system, however, and therefore can't deliver fault tolerance.

We listed the CA database type last for a reason—in a distributed system, partitions can't be avoided. So, while we can discuss a CA distributed database in theory, for all practical purposes a CA distributed database can't exist. This doesn't mean you can't have a CA database for your distributed application if you need one. Many relational databases, such as PostgreSQL, deliver consistency and availability and can be deployed to multiple nodes using replication.

MongoDB and the CAP theorem

MongoDB is a popular NoSQL database management system that stores data as BSON (binary JSON) documents. It's frequently used for big data and real-time applications running at multiple different locations. Relative to the CAP theorem, MongoDB is a CP data store—it resolves network partitions by maintaining consistency, while compromising on availability.

MongoDB is a *single-master* system—each replica set (link resides outside ibm.com) can have only one primary node that receives all the write operations. All other nodes in the same replica set are secondary nodes that replicate the primary node's operation log and apply it to their own data set. By default, clients also read from the primary node, but they can also specify a read preference (link resides outside ibm.com) that allows them to read from secondary nodes.

When the primary node becomes unavailable, the secondary node with the most recent operation log will be elected as the new primary node. Once all the other secondary nodes catch up with the new master, the cluster becomes available again. As clients can't make any write requests during this interval, the data remains consistent across the entire network.

Cassandra and the CAP theorem (AP)

Apache Cassandra is an open source NoSQL database maintained by the Apache Software Foundation. It's a wide-column database that lets you store data on a distributed network. However, unlike MongoDB, Cassandra has a masterless architecture, and as a result, it has multiple points of failure, rather than a single one.

Relative to the CAP theorem, Cassandra is an AP database—it delivers availability and partition tolerance but can't deliver consistency all the time. Because Cassandra doesn't have a master node, all the nodes must be available continuously. However, Cassandra provides *eventual consistency* by allowing clients to write to any nodes at any time and reconciling inconsistencies as quickly as possible.

As data only becomes inconsistent in the case of a network partition and inconsistencies are quickly resolved, Cassandra offers "repair" functionality to help nodes catch up with their peers. However, constant availability results in a highly performant system that might be worth the trade-off in many cases.

Microservices and the CAP theorem

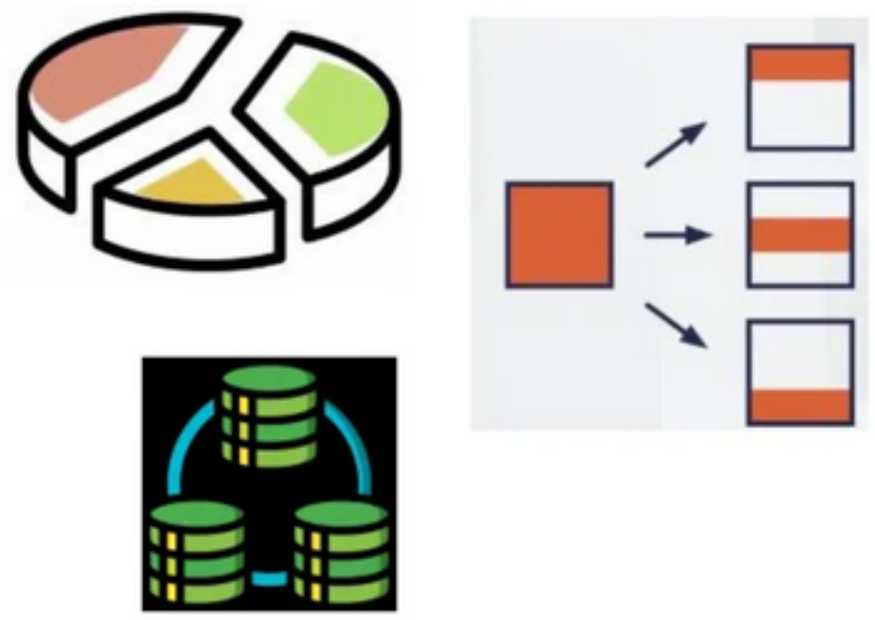
Microservices are loosely coupled, independently deployable application components that incorporate their own stack—including their own database and database model—and communicate with each other over a network. As you can run microservices on both cloud servers and on-premises data centers, they have become highly popular for hybrid and multicloud applications.

Understanding the CAP theorem can help you choose the best database when designing a microservices-based application running from multiple locations. For example, if the ability to quickly iterate the data model and scale horizontally is essential to your application, but you can tolerate eventual (as opposed to strict) consistency, an AP database like Cassandra or Apache CouchDB can meet your requirements and simplify your deployment. On the other hand, if your application depends heavily on data consistency—as in an eCommerce application or a payment service—you might opt for a relational database like PostgreSQL.

Distributed Data: Replication, Partitioning and Sharding

There are two common ways data is distributed across multiple nodes.

Replication and Partitioning (Sharding, when assigned to different nodes)



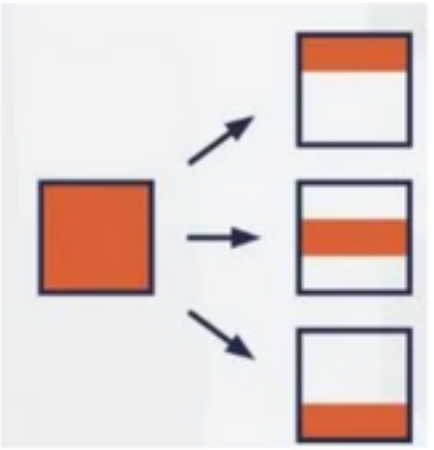
Patterns for Distribute Data

Replication

Replication (Copying data)— Keeping a copy of same data on multiple servers that are connected via a network.

Partitioning

Partitioning — Splitting up a large monolithic database into multiple smaller databases based on data cohesion. e.g. Horizontal (sharding) and Vertical (increase server size) partitioning.



Sharding

Sharding (Horizontal Partitioning)— A type of horizontal partitioning that splits large databases into smaller components, which are faster and easier to manage. In other words — Splitting up a large table of data horizontally i.e. row-wise.

Why to Shard a table?

A shard is an individual partition that exists on separate database server instance to spread load. It is needed when a dataset is too big to be stored in a single database.

As both the database size and number of transactions increase, so does the response time for querying the database. Costs associated with maintaining a huge database can also skyrocket due to the number and quality of computers you need to manage your workload. Data shards, on the other hand, have fewer hardware and software requirements and can be managed on less expensive servers.

Advantages of Sharding:

- Solve Scalability Issue: With a single database server architecture any application experiences performance degradation. At some point, you will be running out of disk space. Database sharding fixes all these issues by partitioning the data across multiple machines.
- High Availability: If an outage happens in sharded architecture, then only some specific shards will be down. All the other shards will continue the operation and the entire application won't be unavailable for the users.

- Speed Up Query Response Time: In a sharded database a query has to go through fewer rows, and you receive the response in less time.
- More Write Bandwidth: For many applications writings is a major bottleneck. With no master database serializing writes sharded architecture allows you to write in parallel and increase your write throughput.
- Scaling Out: Sharding a database facilitates ***horizontal scaling***, known as *scaling out*. In horizontal scaling, you **add more machines** in the network and distribute the load on these machines for faster processing and response.

Disadvantages of Sharding:

- Adds Complexity in the System: It's a complicated task and if it's not implemented properly then you may lose the data or get corrupt tables in your database. You also need to manage the data from multiple shard locations instead of managing and accessing it from a single-entry point. This may affect the workflow of your team which can be potentially disruptive to some teams.
- Rebalancing Data: In a sharded database architecture, sometimes shards become unbalanced (when a shard outgrows other shards) and may create database hotspot. To overcome this problem and to rebalance the data you need to do re-sharding for even data distribution. Moving data from one shard to another shard is not a good idea because it requires a lot of downtime.

- Joining Data from Multiple Shards is Expensive: In sharded architecture, you need to pull the data from different shards, and you need to perform joins across multiple networked servers. You can pull out the data and join the data across the network. This is going to be an expensive and time-consuming process. It adds latency to your system.
- No Native Support: Sharding is not natively supported by every database engine.

Why to do data Replication?

Data Replication is the process of storing data in more than one site or node.

Advantages of full Replication:

- Keep data geographically close to users (and thus reduce latency)
- Allow the system to continue working even if some of its parts have failed (and thus increase availability).
- Scale out the number of servers that can serve read queries (and thus increase read thruputs).
- Disadvantages of full Replication
- Concurrency is difficult to achieve in full replication.
- Slow update process as a single update must be performed at different databases to keep the copies consistent

Database Normalization:

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

- Making relations very large.
- It isn't easy to maintain and update data as it would involve searching many records in relation.
- Wastage and poor utilization of disk space and resources.
- The likelihood of errors and inconsistencies increases.

So to handle these problems, we should analyze and decompose the relations with redundant data into smaller, simpler, and well-structured relations that are satisfy desirable properties. Normalization is a process of decomposing the relations into relations with fewer attributes.

What is Normalization?

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships.
- The normal form is used to reduce redundancy from the database table.

Why do we need Normalization?

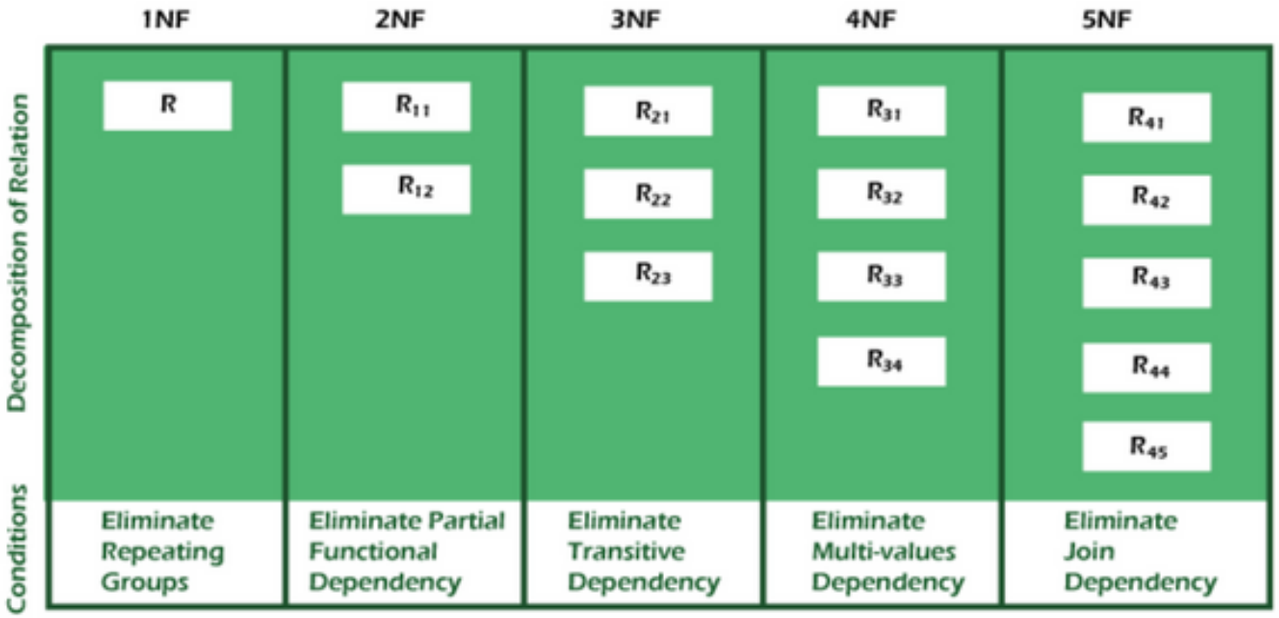
The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.

Data modification anomalies can be categorized into three types:

- Insertion Anomaly: Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
- Deletion Anomaly: The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
- Updatation Anomaly: The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

Types of Normal Forms:
Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.

Following are the various types of Normal forms:



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A stronger definition of 3NF is known as Boyce Codd's normal form.
4NF	A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.
5NF	A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless.

Advantages of Normalization:

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.
- Much more flexible database design.
- Enforces the concept of relational integrity.

Disadvantages of Normalization:

- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.
- Careless decomposition may lead to a bad database design, leading to serious problems.

Database Transaction Models: ACID Vs BASE

The CAP theorem states that it is impossible to achieve both consistency (ACID) and availability (BASE) in a partition tolerant distributed system (i.e., a system which continues to work in cases of temporary communication breakdowns).

ACID Model

The ACID database transaction model ensures that a performed transaction is always consistent. This makes it a good fit for businesses which deal with online transaction processing (e.g., finance institutions) or online analytical processing (e.g., data warehousing). These organizations need database systems which can handle many small simultaneous transactions. There must be zero tolerance for invalid states.

ACID acronym standing for Atomic, Consistent, Isolated, Durable.

ACID stands for:

- Atomic — Each transaction is either properly carried out or the process halts and the database reverts back to the state before the transaction started. This ensures that all data in the database is valid.
- Consistent — A processed transaction will never endanger the structural integrity of the database.
- Isolated — Transactions cannot compromise the integrity of other transactions by interacting with them while they are still in progress.

ACID Use Case Example

Financial institutions will almost exclusively use ACID databases. Money transfers depend on the atomic nature of ACID.

Which Databases are ACID compliant?

One safe way to make sure your database is ACID compliant is to choose a relational database management system. These include MySQL, PostgreSQL, Oracle, SQLite, and Microsoft SQL Server.

BASE acronym standing for Basically Available, Soft state, Eventually consistent

BASE stands for:

- Basically Available — Rather than enforcing immediate consistency, BASE-modelled NoSQL databases will ensure availability of data by spreading and replicating it across the nodes of the database cluster.
- Soft State — Due to the lack of immediate consistency, data values may change over time. The BASE model breaks off with the concept of a database which enforces its own consistency, delegating that responsibility to developers.
- Eventually Consistent — The fact that BASE does not enforce immediate consistency does not mean that it never achieves it. However, until it does, data reads are still possible (even though they might not reflect the reality).

BASE Use Case Example:

Marketing and customer service companies who deal with sentiment analysis will prefer the elasticity of BASE when conducting their social network research. Social network feeds are not well structured but contain huge amounts of data which a BASE-modeled database can easily store.

Which Databases are Using the BASE Model?

Just as SQL databases are almost uniformly ACID compliant, NoSQL databases tend to conform to BASE principles. MongoDB, Cassandra and Redis are among the most popular NoSQL solutions, together with Amazon DynamoDB and Couchbase.

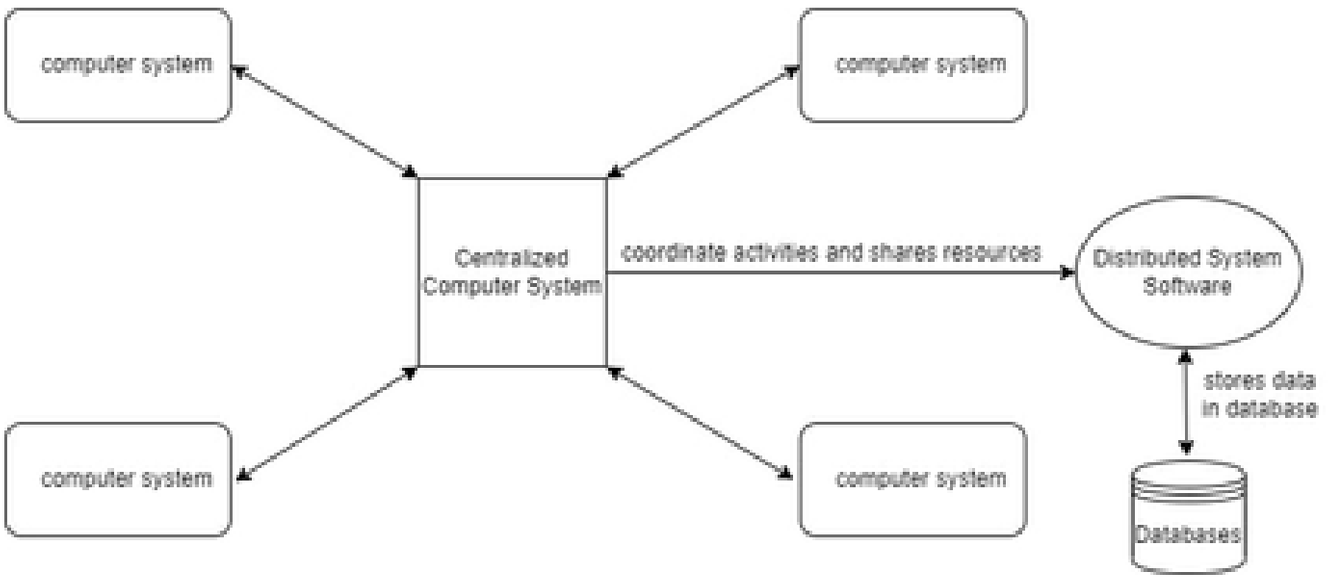
The difference between ACID and BASE database models is the way they deal with this limitation.

- The ACID model provides a consistent system.
- The BASE model provides high availability.

A fully ACID model database is perfectly fit for use cases where data reliability and consistency are the topmost priority like in banking, PayPal.

Consistency Model in Distributed System

A consistency model is a set of rules that govern the behavior of a distributed system. It establishes the circumstances in which the system's various parts can communicate with one another and decides how the system will react to modifications or errors. A distributed system's consistency model plays a key role in ensuring the system's consistency and dependability in the face of distributed computing difficulties including network delays and partial failures.



Consistency models in distributed systems refer to the guarantees provided by the system about the order in which operations appear to occur to clients. Specifically, it determines how data is accessed and updated across multiple nodes in a distributed system, and how these updates are made available to clients.

There are various types of consistency models available in distributed systems. Each consistency model has its own strengths and weaknesses, and the choice of model depends on the specific requirements of the system.

Types of Consistency Models:

1. Eventual Consistency : Eventual consistency is a consistency model that enables the data store to be highly available. It is also known as optimistic replication & is key to distributed systems. So, how exactly does it work? Let's Understand this with the help of a use case. Real World Use Case :

Think of a popular microblogging site deployed across the world in different geographical regions like Asia, America, and Europe. Moreover, each geographical region has multiple data center zones: North, East, West, and South.
Furthermore, each zone has multiple clusters which have multiple server nodes running. So, we have many datastore nodes spread across the world that micro-blogging site uses for persisting data. Since there are so many nodes running, there is no single point of failure.
The data store service is highly available. Even if a few nodes go down persistence service is still up. Let's say a celebrity makes a post on the website that everybody starts liking around the world.
At a point in time, a user in Japan likes a post which increases the "Like" count of the post from say 100 to 101. At the same point in time, a user in America, in a different geographical zone, clicks on the post, and he sees "Like" count as 100, not 101.

Reason for the above Use case :

Simply, because the new updated value of the Post “Like” counter needs some time to move from Japan to America and update server nodes running there. Though the value of the counter at that point in time was 101, the user in America sees old inconsistent values.
But when he refreshes his web page after a few seconds “Like” counter value shows as 101. So, data was initially inconsistent but eventually got consistent across server nodes deployed around the world. This is what eventual consistency is.

2. Strong Consistency: Strong Consistency simply means the data must be strongly consistent at all times. All the server nodes across the world should contain the same value as an entity at any point in time. And the only way to implement this behavior is by locking down the nodes when being updated. Real World Use Case :

Let’s continue the same Eventual Consistency example from the previous lesson. To ensure Strong Consistency in the system, when a user in Japan likes posts, all nodes across different geographical zones must be locked down to prevent any concurrent updates.
This means at one point in time, only one user can update the post “Like” counter value. So, once a user in Japan updates the “Like” counter from 100 to 101. The value gets replicated globally across all nodes. Once all nodes reach consensus, locks get lifted. Now, other users can Like posts.

If the nodes take a while to reach a consensus, they must wait until then. Well, this is surely not desired in the case of social applications. But think of a stock market application where the users are seeing different prices of the same stock at one point in time and updating it concurrently. This would create chaos. Therefore, to avoid this confusion we need our systems to be Strongly Consistent.
The nodes must be locked down for updates. Queuing all requests is one good way of making a system Strongly Consistent. The strong Consistency model hits the capability of the system to be Highly Available & perform concurrent updates. This is how strongly consistent ACID transactions are implemented.

In a strongly consistent system, all nodes in the system agree on the order in which operations occurred. Reads will always return the most recent version of the data, and writes will be visible to all nodes immediately after they occur. This model provides the highest level of consistency. There are some performance and availability issues.

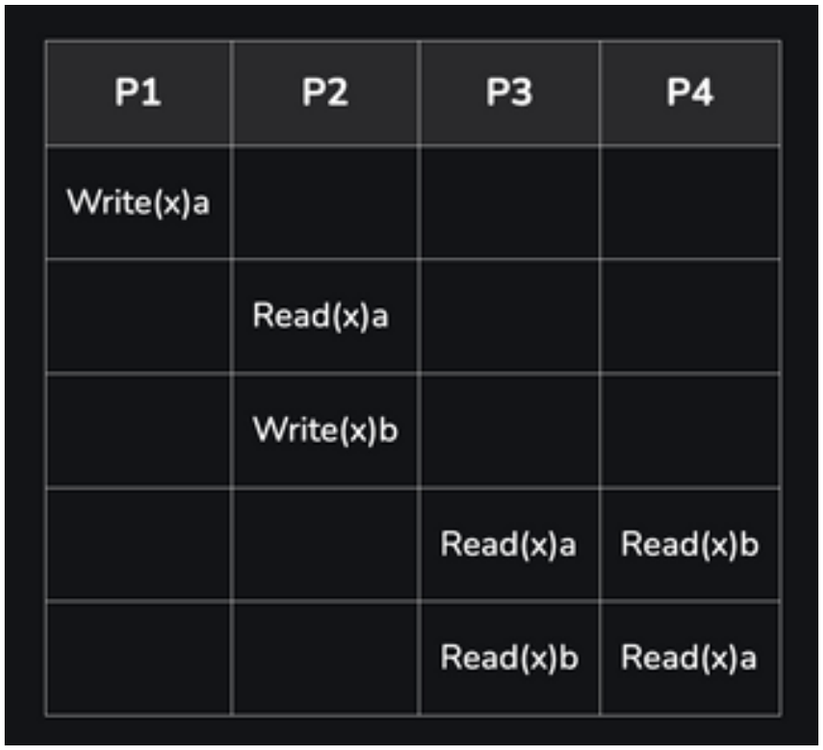
Process	Write Operation	Read Operation
P1	Write(x)a	
P2		Read(x)a

ACID Transaction Support :

Distributed systems like NoSQL databases which scale horizontally on the fly don’t support ACID transactions globally & this is due to their design. The whole reason for the development of NoSQL tech is the ability to be Highly Available and Scalable. If we must lock down nodes every time, it becomes just like SQL. So, NoSQL databases don’t support ACID transactions and those that claim to, have terms and conditions applied to them. Generally, transaction support is limited to a geographic zone or an entity hierarchy. Developers of tech make sure that all the Strongly consistent entity nodes reside in the same geographic zone to make ACID transactions possible.

Conclusion: For transactional things go for MySQL because it provides a lock-in feature and supports ACID transactions.

Pipelined Random Access Memory Or FIFO Consistency Model: PRAM is one of the weak consistency models. Here, all processes view the operations of a single process in the same order that they were issued by that process, while operations issued by different processes can be viewed in a different order from different processes.



P1	P2	P3	P4
Write(x)a			
	Read(x)a		
	Write(x)b		
		Read(x)a	Read(x)b
		Read(x)b	Read(x)a

Sequential Consistency Model: This model was proposed by Lamport. In this sequential consistency model, the result of any execution is the same as if the read and write operations by all processes were executed in some sequential order and the operations of each individual process appear in this sequence in the order specified by its program. The sequential consistency model is weak as compared to the strict consistency.

Process	Write Operation	Read Operation
P1	Write(x)a	
P2		Read(x)a
P3	Write(x)b	

Causal Consistency Model: The causal consistency model was introduced by Hutto and Ahamad in 1990. It makes sure that All processes see causally-related shared accesses in the same order. The causal consistency model is weaker as compared to strict or sequential consistency because the difference between causal consistency and sequential consistency is that causal consistency does not require a total order.

P1	P2	P3	P4
Write(x)a	Read(x)a	Read(x)a	Read(x)a
	Write(x)b		
Write(x)c		Read(x)c	Read(x)b
		Read(x)b	Read(x)c

Weak Consistency Model: A weakly consistent system provides no guarantees about the ordering of operations or the state of the data at any given time. Clients may see different versions of the data depending on which node they connect to. This model provides the highest availability and scalability but at the cost of consistency.

Processor Consistency Model: The processor consistency model was introduced by Goodman in 1989 and is analogous to the PRAM consistency model but there’s a small difference i.e there’s a restriction of memory coherence. Memory coherence refers to the fact that all processes should reflect in the same order for all write operations to any given memory address.

Database Mirroring

In database management systems (DBMS), there are two terms namely mirroring and replication which are related to copying of data. Mirroring is defined as the process of copying a database to another location, while replication is defined as the process of copying the database objects from one database to another.

Read this article to learn more about mirroring and replication and how they are different from each other.

What is Mirroring?

Mirroring refers to keeping a backup database server for a master database server. If for some reason, the master database is down, then the mirror database can be used as an alternative for the master database.

In principle, only one database server is active at a time and the requests for database is served from one server only which is active. Therefore, we can define mirroring as a process of creating multiple copies of a database which are located on different server machines. For this reason, mirroring is also called as shadowing. In case of failure of the primary server, the data can be accessed from the mirrored database.

What is Replication?

Replication refers to keeping multiple copies of databases spread across multiple geographic locations.

A classic example of replication is file servers which are replicated across continents so that the user can download the file from the nearest location to avoid network delays and any slow response. In other words, replication is defined as a process of distributing redundant data and other database objects across different databases for their enhanced availability to the users. Thus, replication increases parallel command execution.

The implementation of replication becomes easier in distributed databases. Note that replication can be done only on the data and database objects, and not on a complete database as a whole.

Key	Mirroring	Replication
Definition	Mirroring refers to keeping copies of database to a geographically different location.	Replication refers to creating multiple copies of data objects of a database for distribution efficiency.
Target	Mirroring is applicable on complete database as a whole.	Replication is done on database objects.
Cost	Mirroring is costlier than replication.	Replication is cheaper as compared to Mirroring.
Distributed Databases	Mirroring is not applicable in case of Distributed databases.	Replication can be easily implemented in case of distributed databases.
Location	Mirroring is done to create a copy of database on different hardware and on different location to serve as backup location.	Replication is done to create a copy of database objects and can be copied to a different database as well.

Both mirroring and replication are related to the copying of data in databases, but they are absolutely different from each other. The most significant difference that you should note here is that mirroring is applicable on a complete database as a whole, while replication is applicable on database objects only.

Leader Election in Distributed Systems

Leader election is the simple idea of giving one thing (a process, host, thread, object, or human) in a distributed system some special powers. Those special powers could include the ability to assign work, the ability to modify a piece of data, or even the responsibility of handling all requests in the system.

Leader election is a powerful tool for improving efficiency, reducing coordination, simplifying architectures, and reducing operations. On the other hand, leader election can introduce new failure modes and scaling bottlenecks. In addition, leader election may make it more difficult for you to evaluate the correctness of a system.

Because of these complications, we carefully consider other options before implementing leader election. For data processing and workflows, workflow services like AWS Step Functions can achieve many of the same benefits as leader election and avoid many of its risks. For other systems, we often implement idempotent APIs, optimistic locking, and other patterns that make a single leader unnecessary.

Advantages and disadvantages of leader election

Leader election is a common pattern in distributed systems because it has some significant advantages:

- A single leader makes systems easier for humans to think about. It puts all the concurrency in the system into a single place, reduces partial failure modes, and adds a single place to look for logs and metrics.
- A single leader can work more efficiently. It can often simply inform other systems about changes, rather than building consensus about the changes to be made.
- Single leaders can easily offer clients consistency because they can see and control all the changes made to the state of the system.
- A single leader can improve performance or reduce cost by providing a single consistent cache of data which can be used every time.
- Writing software for a single leader may be easier than other approaches like quorum. The single leader doesn't need to consider that other systems may be working on the same state at the same time.

Leader election also has some considerable downsides:

- A single leader is a single point of failure. If the system fails to detect or fix a bad leader, the whole system can be unavailable.
- A single leader means a single point of scaling, both in data size and request rate. When a leader-elected system needs to grow beyond a single leader, it requires a complete re-architecture.

• A leader is a single point of trust. If a leader is doing the wrong work with nobody checking it, it can quickly cause problems across the entire system. A bad leader has a high blast radius.
• Partial deployments may be hard to apply in leader-elected systems. Many software safety practices at Amazon depend on partial deployments, such as one-box, A-B testing, blue/green deployment, and incremental deployment with automatic rollback.

Many of these disadvantages can be mitigated by carefully choosing the leader's scope. How much of the system or data does the leader own? A common pattern here is sharding. Each item of data still belongs to a single leader, but the whole system contains many leaders. This is the fundamental design approach behind Amazon DynamoDB (DynamoDB), Amazon Elastic Block Store (Amazon EBS), Amazon Elastic File System (Amazon EFS), and many other systems at Amazon. Sharding has its own downsides, though. Specifically, more design complexity and the need to carefully think through how to shard data.

How Amazon elects a leader?

Locking

Deadlock in java is a programming situation where two or more threads are blocked forever. Java deadlock situation arises with at least two threads and two or more resources. Here I have written a simple program that will cause java deadlock scenario and then we will see how to analyze it.

Deadlock in Java Example

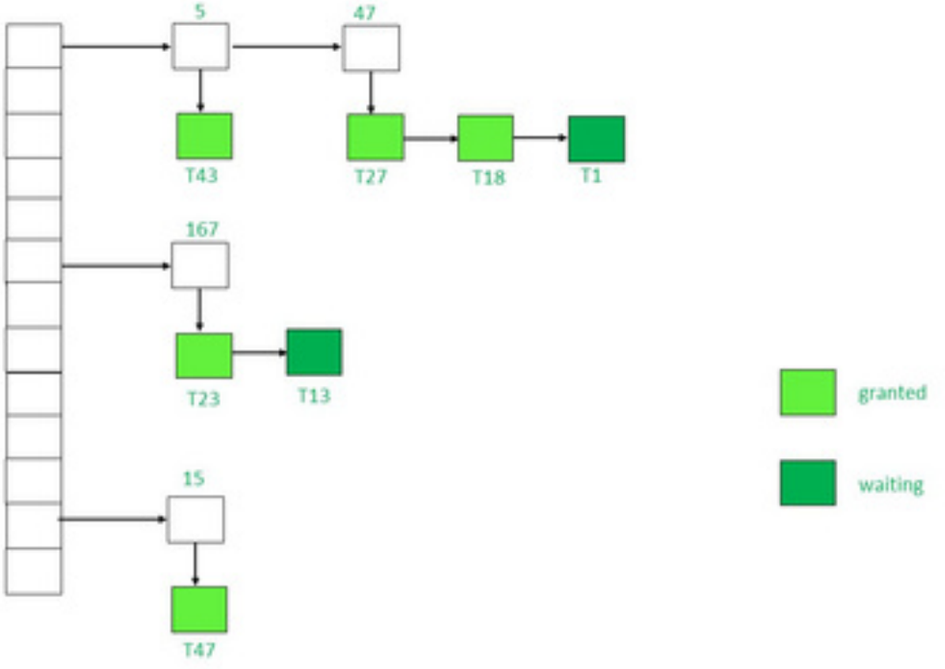
Technical tutorials, Q&A, events — This is an inclusive place where developers can find or lend support and discover new ways to contribute to the community.

Locking protocols are used in database management systems as a means of concurrency control. Multiple transactions may request a lock on a data item simultaneously. Hence, we require a mechanism to manage the locking requests made by transactions. Such a mechanism is called as Lock Manager. It relies on the process of message passing where transactions and lock manager exchange messages to handle the locking and unlocking of data items.

Data structure used in Lock Manager:

The data structure required for implementation of locking is called as Lock table.

- It is a hash table where name of data items are used as hashing index.
- Each locked data item has a linked list associated with it.
- Every node in the linked list represents the transaction which requested for lock, mode of lock requested (mutual/exclusive) and current status of the request (granted/waiting).
- Every new lock request for the data item will be added in the end of linked list as a new node.
- Collisions in hash table are handled by technique of separate chaining.



Explanation: In the above figure, the locked data items present in lock table are 5, 47, 167 and 15.

The transactions which have requested for lock have been represented by a linked list shown below them using a downward arrow.

Each node in linked list has the name of transaction which has requested the data item like T33, T1, T27 etc.

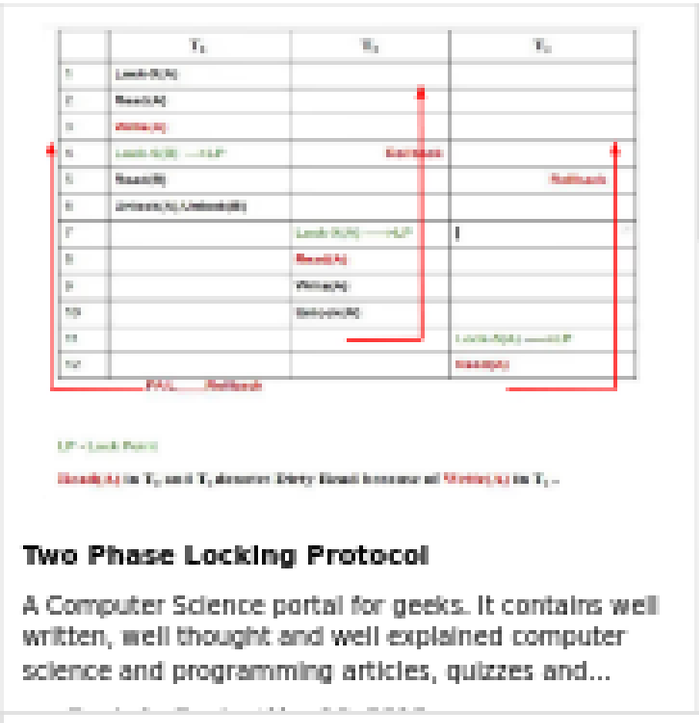
The colour of node represents the status i.e. whether lock has been granted or waiting.

Note that a collision has occurred for data item 5 and 47. It has been resolved by separate chaining where each data item belongs to a linked list. The data item is acting as header for linked list containing the locking request.

Working of Lock Manager –

- Initially the lock table is empty as no data item is locked.
- Whenever lock manager receives a lock request from a transaction Ti on a particular data item Qi following cases may arise:
- If Qi is not already locked, a linked list will be created and lock will be granted to the requesting transaction Ti.
- If the data item is already locked, a new node will be added at the end of its linked list containing the information about request made by Ti.

- If the lock mode requested by Ti is compatible with lock mode of transaction currently having the lock, Ti will acquire the lock too and status will be changed to ‘granted’. Else, status of Ti’s lock will be ‘waiting’.
- If a transaction Ti wants to unlock the data item it is currently holding, it will send an unlock request to the lock manager. The lock manager will delete Ti’s node from this linked list. Lock will be granted to the next transaction in the list.
- Sometimes transaction Ti may have to be aborted. In such a case all the waiting request made by Ti will be deleted from the linked lists present in lock table. Once abortion is complete, locks held by Ti will also be released.

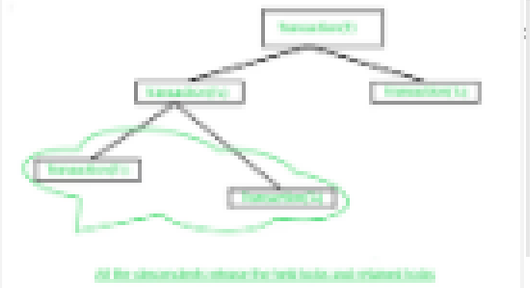


Two Phase Locking Protocol

A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and...

Predicate Locking

A Computer Science portal for geeks. It contains well written, well though...



MOSS Concurrency Control Protocol (Distributed Locking in Database)

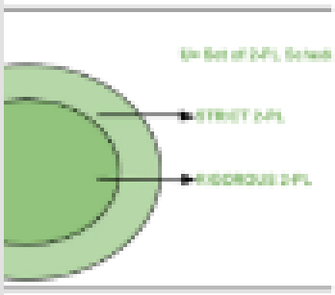
A Computer Science portal for geeks. It contains well written, well thought and well explained computer science and programming articles, quizzes and...

Distributed Locks w

A distributed lock patter

redis.io

MySQL has three lock levels: row-level locking, page-level locking and table-level locking.



Categories of Two Phase Locking (Strict, Rigorous & Conservative)

geeksforgeeks.org

Levels of Locking

A Computer Science

It contains well writt

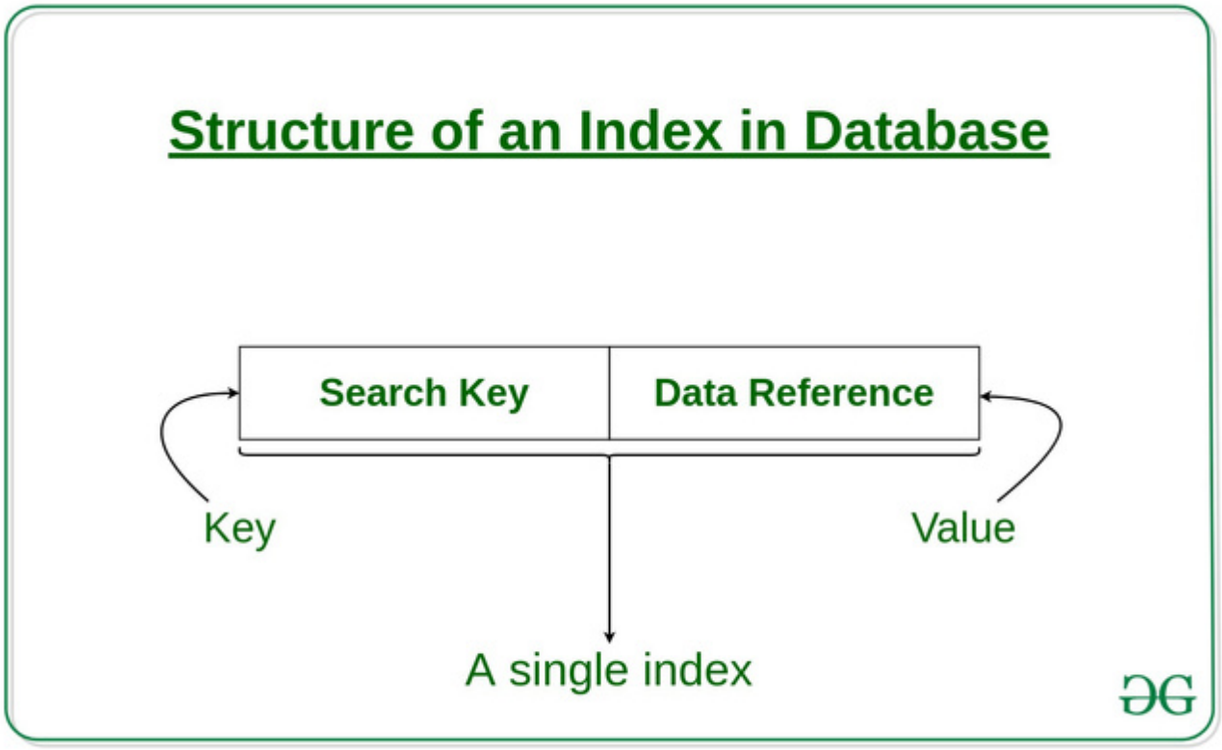
geeksforgeeks.org

Indexing in Databases

Indexing is a way to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed. It is a data structure technique which is used to quickly locate and access the data in a database.

Indexes are created using a few database columns.

The first column is the Search key that contains a copy of the primary key or candidate key of the table. These values are stored in sorted order so that the corresponding data can be accessed quickly.
Note: The data may or may not be stored in sorted order.
The second column is the Data Reference or Pointer which contains a set of pointers holding the address of the disk block where that particular key value can be found.



The indexing has various attributes:

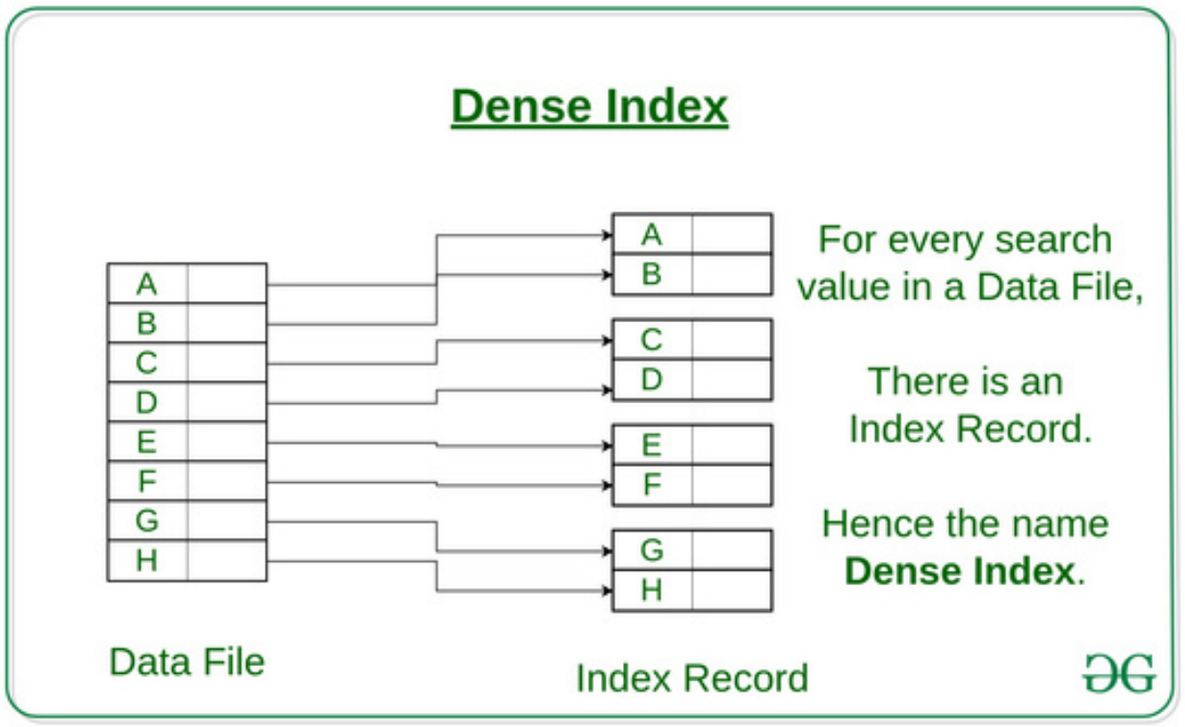
- Access Types: This refers to the type of access such as value based search, range access, etc.
- Access Time: It refers to the time needed to find particular data element or set of elements.
- Insertion Time: It refers to the time taken to find the appropriate space and insert a new data.
- Deletion Time: Time taken to find an item and delete it as well as update the index structure.
- Space Overhead: It refers to the additional space required by the index.

In general, there are two types of file organization mechanism which are followed by the indexing methods to store the data:

1. Sequential File Organization or Ordered Index File: In this, the indices are based on a sorted ordering of the values. These are generally fast and a more traditional type of storing mechanism. These Ordered or Sequential file organization might store the data in a dense or sparse format:

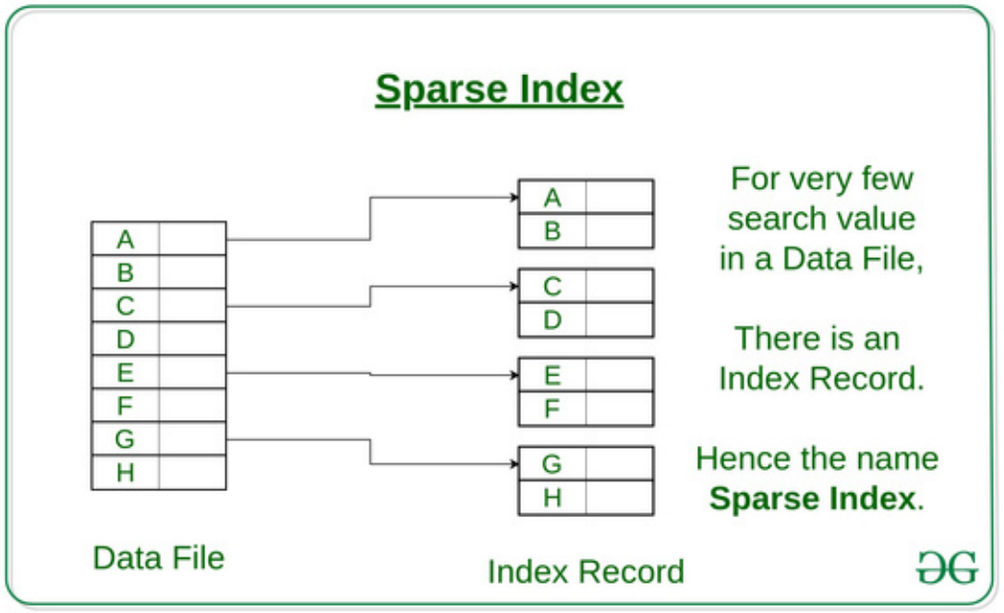
(i) Dense Index:

For every search key value in the data file, there is an index record. This record contains the search key and also a reference to the first data record with that search key value.



(ii) Sparse Index:

The index record appears only for a few items in the data file. Each item points to a block as shown.
To locate a record, we find the index record with the largest search key value less than or equal to the search key value we are looking for.
We start at that record pointed to by the index record, and proceed along with the pointers in the file (that is, sequentially) until we find the desired record.
Number of Accesses required=$\log_2(n)+1$, (here n=number of blocks acquired by index file)



2. Hash File organization: Indices are based on the values being distributed uniformly across a range of buckets. The buckets to which a value is assigned is determined by a function called a hash function.

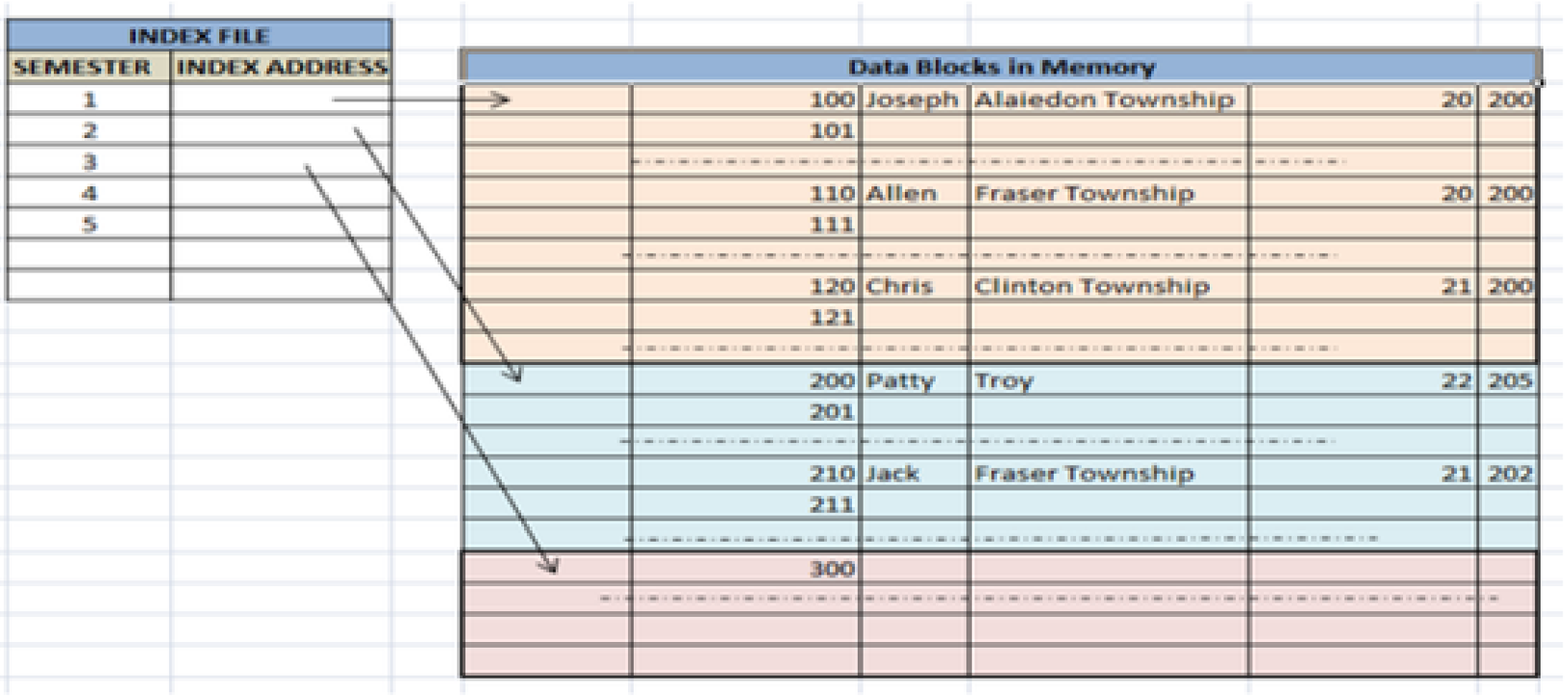
There are primarily three methods of indexing:

- Clustered Indexing
- Non-Clustered or Secondary Indexing
- Multilevel Indexing

1. Clustered Indexing
When more than two records are stored in the same file these types of storing known as cluster indexing. By using the cluster indexing we can reduce the cost of searching reason being multiple records related to the same thing are stored at one place and it also gives the frequent joining of more than two tables (records).

Clustering index is defined on an ordered data file. The data file is ordered on a non-key field. In some cases, the index is created on non-primary key columns which may not be unique for each record. In such cases, in order to identify the records faster, we will group two or more columns together to get the unique values and create index out of them. This method is known as the clustering index. Basically, records with similar characteristics are grouped together and indexes are created for these groups.

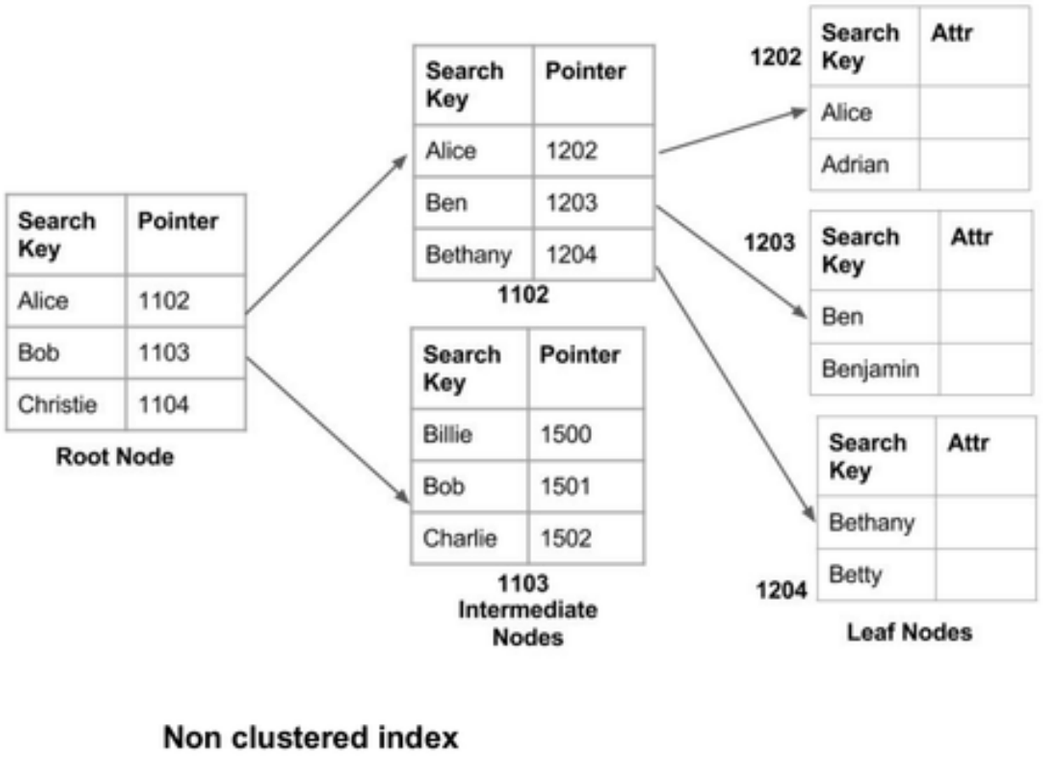
For example, students studying in each semester are grouped together. i.e. 1st Semester students, 2nd semester students, 3rd semester students etc. are grouped.



Clustered index sorted according to first name (Search key)

Primary Indexing:
This is a type of Clustered Indexing wherein the data is sorted according to the search key and the primary key of the database table is used to create the index. It is a default format of indexing where it induces sequential file organization. As primary keys are unique and are stored in a sorted manner, the performance of the searching operation is quite efficient.

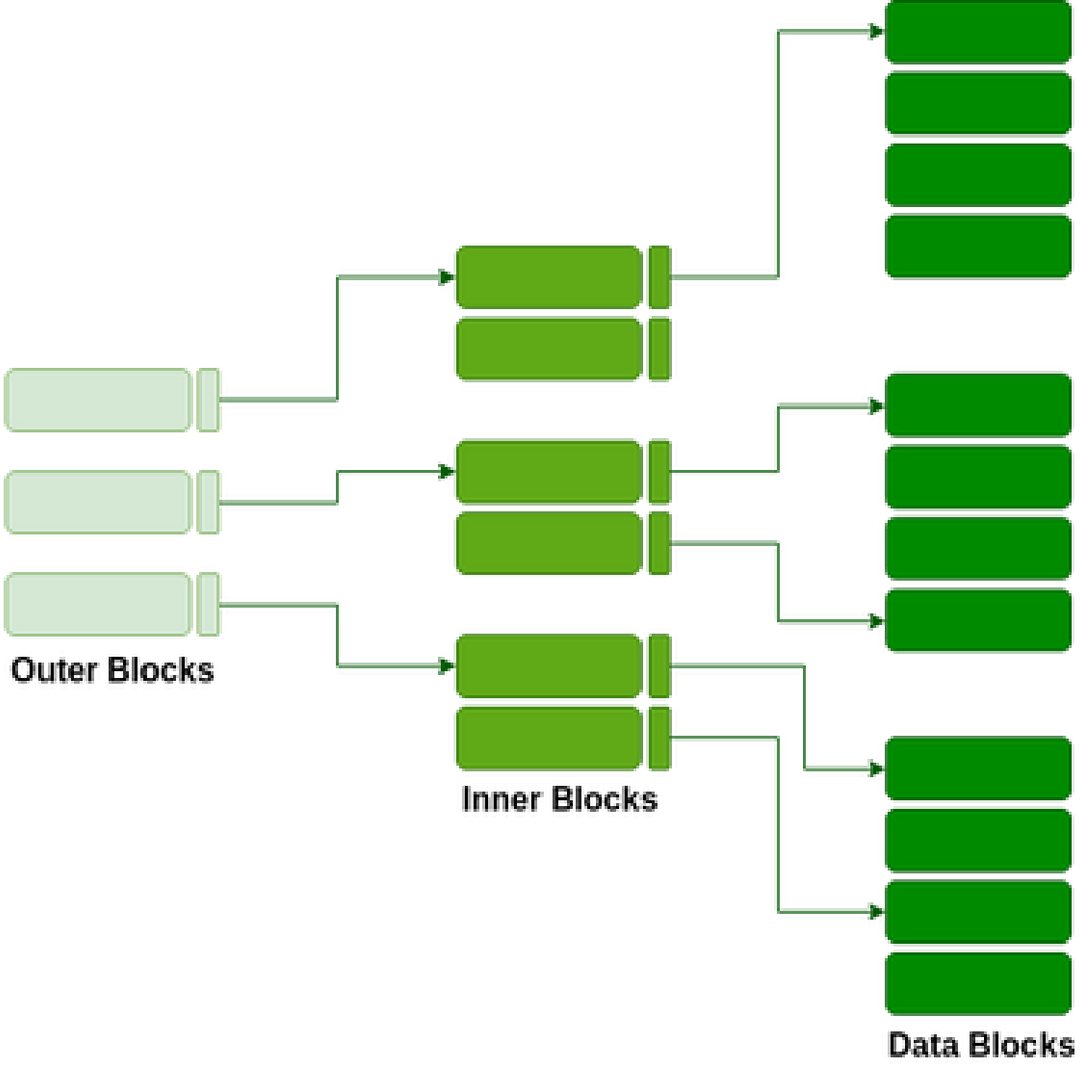
2. Non-clustered or Secondary Indexing
A non clustered index just tells us where the data lies, i.e. it gives us a list of virtual pointers or references to the location where the data is actually stored. Data is not physically stored in the order of the index. Instead, data is present in leaf nodes. For eg. the contents page of a book. Each entry gives us the page number or location of the information stored. The actual data here(information on each page of the book) is not organized but we have an ordered reference(contents page) to where the data points actually lie. We can have only dense ordering in the non-clustered index as sparse ordering is not possible because data is not physically organized accordingly.
It requires more time as compared to the clustered index because some amount of extra work is done in order to extract the data by further following the pointer. In the case of a clustered index, data is directly present in front of the index.



Non clustered index

3. Multilevel Indexing

With the growth of the size of the database, indices also grow. As the index is stored in the main memory, a single-level index might become too large a size to store with multiple disk accesses. The multilevel indexing segregates the main block into various smaller blocks so that the same can stored in a single block. The outer blocks are divided into inner blocks which in turn are pointed to the data blocks. This can be easily stored in the main memory with fewer overheads.



Indexing in databases provides several advantages, including: Improved Query Performance: Indexing allows for faster retrieval of data from the database. By creating an index on a column, the database can quickly locate the rows that match a particular value or set of values, reducing the amount of time it takes to execute a query.

Efficient Data Access: Indexing can improve the efficiency of data access by reducing the amount of disk I/O needed to retrieve data. By creating an index on frequently accessed columns, the database can keep the data pages for those columns in memory, reducing the need to read from disk.

Optimized Data Sorting: Indexing can also improve the performance of sorting operations. By creating an index on the columns used for sorting, the database can avoid sorting the entire table and instead sort only the relevant rows.

Consistent Data Performance: Indexing can help ensure consistent performance of the database, even as the amount of data in the database grows. Without indexing, queries may take longer to execute as the number of rows in the table increases, but with indexing, the performance remains relatively constant.

Enforced Data Integrity: Indexing can also help enforce data integrity by ensuring that only unique values are stored in columns that have been indexed as unique. This prevents duplicates from being stored in the database, which can cause problems when running queries or reports.

Overall, indexing in databases provides significant benefits for improving query performance, efficient data access, optimized data sorting, consistent data performance, and enforced data integrity