Databases and DBMS

Saturday, 17 February 2024 7:34 PM

What is database?

- organized collection of structured information, or data, typically stored electronically in a computer system.
- db is usually controlled by DBMS.
- together, the data and DBMS, along with applications that are associated with them, are referred as a database system, often shortened to database.

What is DBMS?

- database typically requires a comprehensive database software program known as a DBMS.
- A DBMS serves as an interface between the db and its end-users or programs, allowing users to retrieve, update and manage how the info is organized and optimized.
- DBMS also facilitates oversight and control of db, enabling a variety of administrative operations such as performance monitoring, tuning and backup and recovery.

Components

Schema
- role of schema is to define the shape of a data structure, and specify what kind of data can go where.
- schemas can be strictly enforced across the entire database, loosely enforced in part of db, or they might not exist at all.

Tables
- each table contains various columns just like in a spreadsheet.
- table can have a unique as too columns and upwards of a hundred or more columns, depending upon the kind of information being put in the table.

Columns
- contains a set of values of a particular type, one value for each row of the db.
- column may contain text values, numbers, enums, timestamps etc.

Row
- data in table is recorded in rows.
- there can be thousands or millions of rows in a table having any particular information.

Types

- SQL
- NoSQL — document, key-value, graph, timeseries, wide-column, multi-model

Challenges

: common challenges faced while running dbs at scale:
- Absorbing significant increase in data volume
  explosion of data coming in from sensors, connected machines and dozens of other sources.
- ensuring data security
  important than ever to ensure that data is secure but also easily accessible to users.
- keeping up with demand
  companies need real-time access to their data to support timely decision-making and to take advantage of new opportunities
- managing and maintaining the db and infrastructure
  as db becomes more complex and data volume grows, companies are faced with the expense of hiring additional talent to manage their databases.
- removing limits on scalability
  business ready to grow if they're going to survive, and his data urgent must grow along with it. But it's difficult to predict how much capacity the company will need, particularly with on-premises databases.
- ensuring data resiliency, data availability, or latency requirements
  some org have use cases that are better suited to run on-premises. In those cases, engineered systems that are pre-configured and pre-optimized for running the dbs are ideal.

SQL DB

- A SQL (or relational) db is a collection of data items with pre-defined relationships between them.
- these items are organized as a set of tables with columns and rows.
- tables are used to hold info about the objects to be represented in the db.
- each column in a table holds a certain kind of data and a field stores the actual value of an attribute.
- row in table represent a collection of related values of one object or entity.
- each row in table could be marked with a unique identifier called a primary key, and rows among multiple tables can be made related to foreign keys.
- data can be accessed in many different ways without requiring the db tables themselves.
- SQL db usually follows the ACID consistency model.

Materialized view
- pre-computed data set derived from a query specification and stored for later use.
- querying a materialized view is faster than executing a query against the base table of the view.
- performance difference can be significant when a query is run frequently or is sufficiently complex.
- also enables data caching and improves the performance of complex queries that run on large data set which reduces network loads.
- most used for performance and optimization.

N+1 query problem
- N+1 query problem happens when the data access logic executes N additional SQL statements to fetch the same data that could have been retrieved when executing the primary SQL query.
- The larger value of N, the more queries will be executed, the larger the performance impact.
- commonly seen in GraphQL and ORM tools and can be addressed by optimizing the SQL query or using a dataloader that batches consecutive requests and makes a single data request under the hood.

Advan	Disadv
simple and accurate	expensive to maintain
accessibility	difficult schema evolution
data consistency	performance hits (join, denormalization, etc)
flexibility	difficult to scale due to poor horizontal scalability.

Eg: PostgreSQL, MySQL, MariaDB, Amazon Aurora

NoSQL DB

- broad category that includes any db that doesn't use SQL as the primary data access lang.
- these type of dbs are referred as non-relational dbs.
- doesn't have to conform to a pre-defined schema, unlike relational db
- follows BASE consistency model.

Document
(also known as document-oriented db or document store) is a db that stores info in documents. general-purpose db that serves a variety of use cases for both transactional and analytical applications.

Advan	Disadv
flexible and flexible	schemaless
easy horizontal scaling	non-relational
schemaless	

Eg: MongoDB, Amazon DocumentDB, CouchDB

Key-value
Simplest type of NoSQL dbs, key-value db save data as a group of key-value pairs made up of two data items each.
- also referred to as a key-value store.

Advan	Disadv
simple and performant	Range query
highly suitable for high volumes of traffic	values can't be filtered
session management	lack indexing and sorting capabilities
optimized lookups	not optimized for complex queries

Eg: Redis, Memcached, Amazon DB, Aerospike

Graph
NoSQL db that uses graph structure for semantic queries with nodes, edges, and properties to represent and store data instead of tables or documents.
- graph relates the data items in the store to a collection of nodes and edges, the edges representing the relationships between the nodes.
- relationships allow data in the store to be linked together directly and, in many cases, retrieved with one operation.

Advan	Disadvan
easy query	complex
optimized flexible	no standardized query lang
optimal data representations	

Use cases
- fraud detection
- recommendation engines
- social networks
- network mapping

Eg: Neo4j, AnzenGDB, Amazon Neptune, TerminusDB

Time series
database optimized for time-stamped, or time series, data.

Advan
- fast insertion and retrieval
- efficient data storage

Use cases
- IOT data
- app monitoring
- market analysis
- underlying financial trends

Eg: InfluxDB, Apache Druid

Wide column
wide column db, also known as wide-column stores, are schema-agnostic. data is stored in column families, rather than in rows and columns.

Advan
- highly scalable, can handle petabytes of data
- ideal for real time big data applications

Disadv
- expensive
- increased write time

Use cases
- business analytics, Attribute-based data storage

Eg: Bigtable, Apache Cassandra, ScyllaDB

Multi-model
combines different db models (ie relational, graph, key-value, document etc) into a single, integrated backend.
- this means they can accommodate various data types, indexes, queries, and store data in more than one model.

Advan	Disadv
flexibility	complex
suitable for complex projects	less mature
data consistent	

Eg: — ArangoDB — Azure Cosmos DB — OrientDB — Couchbase

SQL vs NoSQL databases

- relational db are structured and have predefined schemas while non-relational db are unstructured, distributed, and have a dynamic schema.

High level differences

Storage
- SQL stores data in tables, where each row represents an entity and each column represents a data point about that entity.
- most db have different data storage models such as key-value, graph, document etc.

Schema
- In SQL, each record conforms to a fixed schema
- Schemas can be altered later, but it involves modifying the database using migrations
- whereas NoSQL schemas are dynamic. Fields can be added on the fly, and each record (or equivalent) doesn't have to contain data for each field.

Querying
- SQL dbs use SQL for defining and manipulating the data, which is very powerful.
- NoSQL db, queries are focused on a collection of documents.

Scalability
- SQL dbs are vertical in many situations, which can get very expensive.
- It is possible to scale a relational db across multiple servers, but this is a challenging and time-consuming process.
- NoSQL dbs are horizontally scalable, meaning we can add more servers easily to our NoSQL db infrastructure to handle large traffic.
- any cheap commodity hardware or cloud instance can host NoSQL db, thus making it alot more cost-effective than vertical scaling.
- alot of NoSQL tech also distribute data across servers automatically.

Reliability
- vast majority of relational dbs are ACID compliant.
- SQL dbs are better bet when it comes to data reliability and safe guarding of performing transactions
- most NoSQL solutions sacrifice ACID compliance for performance and scalability.

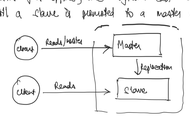
Reasons

For SQL	For NoSQL
structured data with strict schema	dynamic or flexible schema
relational data	non-relational data
need for complex joins	no need for complex joins
transactions	very data-intensive workload
lookups by index are very fast	very high throughput for IOPS

Database Replication

Replication is a process that involves sharing info to ensure consistency between redundant resources such as multiple databases, to improve reliability, fault-tolerance, or availability.

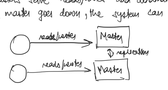
Master-Slave Replication
The master stores reads and writes, replicating writes to one or more slaves, which store only reads.
- slaves can also replicate additional slaves in a tree-like fashion.
- If the master goes offline, the system can continue to operate in read-only mode, until a slave is promoted to a master or a new master is produced



Advan
- backup of the entire db of relatively no impact on the master
- applications can read from the slaves without impacting the master
- slaves can be taken offline and synced back to master without any downtime

Disadv
- replication adds more hardware and additional complexity
- downtime and possibility loss of data when a master fails
- all writes also have to be made to the master in a master-slave arch.
- the more read slaves, the more we have to replicate, which will increase replication lag.

Master-Master Replication
Both masters serve reads/writes and coordinate with each other
- If either master goes down, the system can continue to operate with both reads and writes

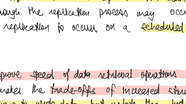


Advan
- app can read from both masters
- distribution write load across both master nodes
- simple, automatic, and quick failover

Disadvan
- not as simple as master-slave to configure and deploy.
- either loosely consistent or have increased write latency due to synchronization
- conflict resolution comes into play as more write nodes are added and as latency increases.

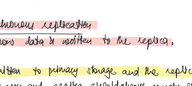
Synchronous vs Asynchronous replication
- primary diff is how data is written to the replica.
- In sync replication
  data is written to primary storage and the replica simultaneously. (on each primary copy and replica should always remain synchronized)
- In Async replication
  copies data to the replica after data is already written to primary storage. (although the replication process may occur in near-real-time, it is more common for replication to occur on a scheduled basis and it is more cost effective)

Indexes
- used to improve speed of data retrieval operations on the db store.
- an index makes the trade-offs of increased storage overhead and slower writes (since we not only have to write data but update the index as well) for benefits of faster reads
- used to quickly locate data without having to examine every row in a db table.
- can be created using one or more columns of a db table, providing the basis for both rapid random lookups and efficient access to ordered records.



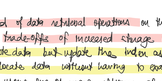
- Index is data structure that can be perceived as a table of contents that points to location where data is actually stored.
- also used to create different views on the same data.
- For large data sets, this is an excellent way to specify different filters or sorting criterias without resorting to creating multiple additional copies of data.
- Quality of index — they can be dense or sparse.

Dense index
- an index record is created for every row of the table. Records can be located directly as each record of the index holds the search key value and the pointer to the actual record.



- requires more maintenance than sparse indexes at write-time.
- since every row must have an entry, db must maintain the index on inserts, updates and deletes.
- having an entry for every row also means that dense index will require more memory
- lookup: value can be quickly found with fast a binary search.
- dense index do not impose any ordering requirement on data.

Sparse index
Index records are created only for some of the records



- requires less maintenance than dense indexes at write-time since they contain only a subset of values.
- lighter maintenance burden means that inserts, deletes, updates will be faster.
- having fewer entries mean that index will use less memory
- finding data is slower since a scan across the page typically follows binary search.
- optimal when working with ordered data.

Normalization and Denormalization

Terms
PK — column or group of columns that can be used to uniquely identify every row of the table.
composite key — primary key made up of multiple columns
super key — set of all keys that can uniquely identify all rows present in a table
candidate key — attribute that identify rows uniquely in a table.
foreign key — it is a reference to a primary key of another table.
alternate key — keys that are not PKs are known as alternate keys
surrogate key — system-generated values that uniquely identifies each entry in a table when no other columns was able to hold properties of a primary key.

Dependencies
Partial dependency — occurs when PK determines some other attribute
Functional dependency — relationship that exists between two attributes, typically between the PK and non-key attribute within a table.
Transitive functional dependency — occur when some non-key determines some other attribute.

Anomalies
Happens when there is a flaw in db due to improper planning or storing everything in a flat db. Generally addressed by process of normalization.
Three types of db anomalies:
- Insertion anomaly: occurs when we are not able to insert certain attributes in the db without the presence of other attributes
- Update anomaly: Occurs in case of data redundancy and partial update.
  — In other words, a correct update of db needs other actions such as addition, deletion or both.
- Delete anomaly: occurs when deletion of some data requires deletion of other data.

eg:

ID	Name	Role	Team
1	Peter	SE	A
2	Brian	Manager	B
3	Hailey	Product Manager	C
4	Hailey	Product Manager	C
5	Steve	Frontend Engineer	D

let's imagine we hired a person John but not yet assigned a team immediately
↳ causes insertion anomaly
let's say Hailey from Team C get promoted
↳ we need to update 2 records to maintain consistency which can cause update anomaly
let's say to remove Team D but to do that we also need to remove additional info
↳ delete anomaly

Normalization

- process of organizing data in a db.
- includes creating tables and establishing relationships between those tables according to rules designed both to protect the data and to make db more flexible by eliminating redundancy and inconsistent dependency.

Why do we need?
- goal is to eliminate redundant data and ensure data is consistent.
- fully normalized database allows its structure to be extended to accomodate new types of data without changing existing structure too much.
- as a result, apps interacting with db are minimally affected.

Normal forms
- series of guidelines to ensure that the db is normalized.

1NF
- repeating groups are not permitted
- identify each set of related data with a primary key
- set of related data should have a separate table.
- mixing data types in the same column is not permitted

2NF
- satisfies the 1NF
- should not have any partial dependency

3NF
- satisfies the 2NF
- transitive functional dependencies are not permitted

BCNF
Boyce-Codd normal form is slightly stronger version of the 3NF used to address certain types of anomalies not dealt with by 3NF as originally defined.
- satisfies 3NF
- for every functional dependency $X \to Y$, X should be the super key

In a relational db, a relation is often described as "normalized" if it meets 3NF. Most 3NF are free of insertion, update and deletion anomaly.

Advan	Disadv
reduces data redundancy	data design is complex
better data design	slower performance
increase data consistency	more time-consuming overhead
enforces referential integrity	require more joins

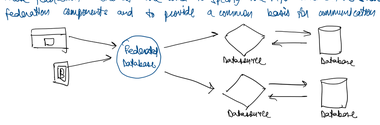
Denormalization
- db optimization technique in which we add redundant data to one or more tables.
- can help us avoid costly joins in a relational db.
- attempts to improve read performance at the expense of some write performance.
- redundant copies of data are written in multiple tables to avoid expensive joins.
- once data becomes distributed with techniques such as federation and sharding, managing joins across the network further increases complexity.
- denormalization might circumvent the need for such complex joins.

note: denormalization doesn't mean reversing normalization

Advan	Disadv
reducing data in faster	expensive inserts and updates
making queries is easier	increase complexity of db design
reduction in no. of tables	increase data redundancy
convenient to manage	more chances of data inconsistency

Last topic

Database Federation
Federation (or functional partitioning) splits up dbs by function. The federation architecture makes several distinct physical dbs appear as one logical db to end-users.
All of components in a federation are tied together by one or more federal schemas that express the commonality of data throughout the federation.
These federation schemas are used to specify the info that can be shared by the federation components and to provide a common basis for communication among them.



Federation also provides a cohesive, unified view of data derived from multiple sources.
- data sources for federated system can include dbs and systems other forms of structured and unstructured data.

Characteristics of federated db
- Transparency — federated db masks user differences and implementations of underlying data sources. Therefore, the users do not need to be aware of where the data is stored
- Heterogeneity — a federated db system can handle different hardware, network, protocols, data models etc.
- Extensibility — new sources may be needed to meet the changing needs of business. A good federated db sys needs to make it easy to add new sources.
- Autonomy — A fed db sys does not change existing data sources, interfaces should remain the same.
- Data Integration — A fed db can integrate data from different protocols, db mgmt sys etc.

Advan of federated database:	Disadv
flexible data sharing	add more hardware and complexity.
autonomy among database components	joining data from two dbs is complex.
access heterogeneous data in a unified way	dependence on autonomous data sources
no tight coupling of apps with legacy db.	query performance and scalability.

ACID and BASE consistency models

ACID
- used for maintaining data integrity during transaction processing.

In order to maintain consistency before and after a transaction relational db follows ACID properties:

Atomic
all operations in a transaction succeed or every operation is rolled back.

Consistent
on a completion of a transaction, the db is structurally sound.

Isolated
transactions do not contend with one another.
- contending access to data is mediated by the db so that transactions appear to run sequentially

Durable
- once transaction is complete and the writes and updates have been written to the disk, it will remain in the system even if system failure occurs.

BASE
- with increasing amount of data and high availability requirements, the approach to db design has also changed dramatically.
- to increase the ability to scale and at the same time be highly available, we move the usage from db to separate servers.
  In this way, db becomes more independent and focused on the actual process of storing data.
- In NoSQL, ACID transactions are less common as some db have loosened the requirements for immediate consistency, data freshness, and accuracy in order to gain other benefits, like scale and resilience.
- BASE properties are much looser than ACID guarantees, but there is not a direct mapping between the two consistency models.

Basic Availability
db appears to work most of the time.

Soft-state
stores don't have to be write-consistent, not do different replicas have to be mutually consistent all the time.

Eventual consistency
data might not be consistent immediately but eventually it becomes consistent.
- reads in the system are still possible even though they may not give the correct response due to inconsistency.

ACID vs BASE Trade-offs
- no right answer to whether our app needs ACID or BASE consistency model.
- both models are designed to satisfy different requirements
- A fully ACID db is the perfect fit for use cases where data reliability and consistency are essential.

CAP Theorem
CAP theorem states that a distributed system can deliver only two of three desired characteristics: consistency, availability and partition tolerance (CP).

Detail look at the three distributed system characteristics to which CAP theorem refers:

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

Availability
- means that any client making a request for data gets a response, even if one or more nodes are down

Partition tolerance
- means system continues to work despite message loss or partial failure.
- system that is partition-tolerant can sustain any amount of network failure that doesn't result in failure of entire network.
- data is sufficiently replicated across combinations of nodes and networks to keep the system up through intermittent outages.

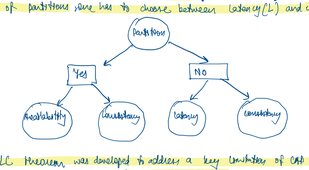
Consistency-Availability Trade-off
We can't guarantee the stability of a network, so distributed databases must choose P. This implies a trade-off between consistency (C) and availability (A).

CA db
- CA db delivers C and A across all nodes.
- It can't do this if there is a partition between any two nodes in the system, and therefore can't deliver fault-tolerance.
  eg: PostgreSQL, MariaDB

CP db
- CP delivers C and P tolerance at the expense of availability.
- when a partition occurs between any two nodes, the system has to shut down the non-consistent node until the partition is resolved.
  eg: MongoDB, Apache HBase

AP db
- AP delivers A and P tolerance - - - - - - - - consistency.
- when a partition occurs, all nodes remains available but those at wrong end of partition might return an older version of data than others.
- when partition is resolved, the AP db typically re-syncs the nodes to repair all inconsistencies in the system.
  eg: Apache Cassandra, CouchDB

PACELC Theorem (extension of CAP)
- PACELC extends CAP theorem by introducing latency (L) as an additional attribute of a distributed system.
  Theorem states that else (E), even when system is running normally in the absence of partitions, one has to choose between latency (L) and consistency (C).



PACELC theorem was developed to address a key limitation of CAP theorem as it makes no provision for performance or latency.
eg: according to CAP, a db can be considered available if a query returns a response after 30 days. Obviously, such latency would be unacceptable for any real-world application.

Transactions
- series of db operations that are considered to be a "single unit of work".
- operations in a transaction either all succeed, or they all fail.
- nature of transactions supports data integrity when part of system fails.

States

Active
transaction is being executed (initial state of every transaction)

Partially committed
when transaction executes the final operations, it is said to be in partially committed state.

Committed
If a transaction executes all its operations successfully, it is said to be committed.
- all effects are now permanently established on the database system.

Failed
If any of the checks made by db recovery system fails.
- failed transaction can no longer proceed further.

Aborted
- If any checks fail and transaction has reached a failed state, then recovery manager rolls back all the write operations on the db to bring the db back to its original state where it was prior to the execution of the transaction.
- db recovery module can select one of two operations after a transaction aborts:
  - restart the transaction
  - kill the transaction

Terminated
- If there isn't any roll-back or the transaction comes from the committed state, then the system is consistent and ready for a new transaction and the old transaction is terminated.

Distributed Transactions
- set of operations on data that is performed across two or more dbs.
- typically coordinated across separate nodes connected by a network, but may also span multiple dbs on a single server.

Need?
- Unlike ACID transaction on a single db, a distributed transaction involves altering data on multiple dbs.
- consequently, distributed transaction processing is more complicated, because the db must coordinate the committing or rollback of the changes in a transaction as a self-contained unit.
- In other words, all the other nodes must commit, or all must abort and entire transaction rolls back.
  This is why we need distributed transactions: (some popular solutions)

Two-Phase Commit (2PC)

Two-Phase commit

- 2PC protocol is a distributed algo that coordinates all the processes that participate in a distributed transaction on whether to commit or abort (roll back) the transaction.
- This protocol achieves the goal even in many cases of temporary system failure and is thus widely used.
- However, it is not resilient to all possible failure configurations, and in rare cases, manual intervention is needed to remedy an outcome.
- this protocol requires a coordinator node, which basically coordinates and oversees the transaction across different nodes.
  (coordinator tries to establish the consensus among set of processes in two phase, hence the name.)

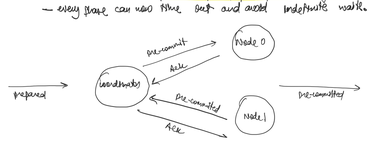
Phases in 2PC
Prepare phase: involves coordinator node collecting consensus from each of the participant nodes
- transaction will be aborted unless each of nodes respond that they're prepared
Commit phase: If all participants respond to the coordinator that they are prepared, the coordinator asks all the nodes to commit the transaction.
- If failure occurs, the transaction will rolled back.

Problems of 2PC
- what if one of the nodes crashes?
- " " coordinator itself crashes?
- It is a blocking protocol.

Three-Phase Commit
- extension of 2PC where commit phase is split into two phases.
- This helps with the blocking problem that occurs in 2PC protocol.

Phases
Prepare phase — same as 2PC
Pre-commit phase — coordinator issues the pre-commit message, and all the participating nodes must acknowledge it.
- If a participant fails to receive this message in time, the transaction is aborted
Commit phase — similar to 2PC protocol.

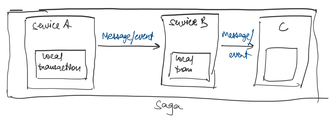
Why is pre-commit phase helpful?
pre-commit phase accomplishes the following:
- If the participant nodes are found in this phase, that means every participant has completed first phase.
  Completion of prepare phase is guaranteed.
- every phase can now time out and avoid indefinite wait.



3PC

Saga

- a saga is a sequence of local transactions.
- each local transaction updates the db and publishes a message or event to trigger the next local transaction in the saga.
- If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions



Saga

Coordination
Two common implementation approaches:
- choreography — each local transaction publishes domain events that trigger local transactions in other services
- orchestration — an orchestrator tells the participants what local transactions to execute

Problems
- saga pattern is hard to debug
- there's a risk of cyclic dependency between saga participants
- lack of participant data isolation imposes availability challenges
- testing is difficult because all services must be running to simulate a transaction.

Sharding

Before Sharding → lets look at data partitioning:

Data partitioning
- technique to break up a database into many smaller parts.
- process of splitting up a database, or a table, across multiple machines to improve the manageability, performance and availability of a database.

Methods
- many different ways one could use to decide how to break up an application database into multiple smaller DBs.
- Below are two of the most popular methods used by various large-scale applications:

Horizontal partitioning (or sharding)
- we split the table data horizontally based on range of values defined by the partition key. Also referred as db sharding

Vertical partitioning
- we partition the data vertically based on columns.
- we move tables into relatively smaller tables with less density, and each part is present in a separate partition.

What is sharding?
- Sharding is db architectural pattern related to horizontal partitioning, which is the practice of separating one table's rows into multiple different tables, known as partitions or shards.
- each partition has same schema and columns, but also a subset of the shared data.
- Justification for data sharding is that, after a certain point it is cheaper and more feasible to scale horizontally by adding more machines than to scale it vertically by adding powerful servers.
- Sharding can be implemented at both application or db level.

Partitioning Criteria
large number of criteria available for data partitioning:

Hash-based
This strategy divides rows into different partitions based on a hashing algorithm rather than grouping db rows based on continuous indexes.
Disadvantage is that dynamically adding/removing db servers becomes expensive.

List-Based
In a list-based partitioning, each partition is defined and selected based on the list of values on a column rather than a set of contiguous ranges of values.

Range Based
Range partitioning maps data to various partitions based on ranges of values of the partitioning key.
In other words, we partition the table in such a way that each partition contains rows within a given range defined by the partition key.
Ranges should be contiguous but not overlapping, wherein each range specifies a non-inclusive lower and upper bound for a partition.
Any partitioning key values equal to or higher than the upper bound of the range are added to next partition.

Composite
Composite partitioning partitions the data based on two or more partitioning techniques.
- Here, we first partition data using one technique, and then each partition is further sub-divided into sub-partitions using the same or some other method

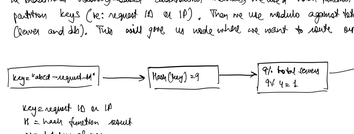
Advan of sharding
Availability — provides logical independence to the partitioned db, ensuring the high availability of our app. Here, individual partitions can be managed independently.
Scalability — proves to increase scalability by distributing the data across multiple partitions.
Security — helps improve the system's security by storing sensitive and non-sensitive data in different partitions. This could provide better manageability and security to sensitive data.
Query Performance — improves the performance of the system. Instead of querying the whole db, now the system has to query only a smaller partition.
Data Manageability — divides tables and indexes into smaller and more manageable units.

Disadv
- Complexity — sharding increases complexity of the system in general
- joins across shards — since a db is partitioned and spread across multiple machines it is often not feasible to perform joins that span multiple db shards.
  — such joins will not be performance efficient since data has to be retrieved from multiple servers.
- Rebalancing — If the data distribution is not uniform or there is a lot of load on a single shard, in such cases, we have to rebalance our shards so that the requests are as equally distributed among the shards as possible.

When to use sharding?
- Leveraging existing hardware instead of high-end machines.
- maintain data in distinct geographic regions
- quickly scale by adding new shards
- better performance as each machine is under less load
- when more concurrent connections are required

Consistent Hashing

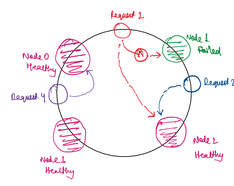
Need?
In traditional hashing-based distribution methods, we use a hash function to hash our partition keys (ie: request ID or IP). Then we use modulo against total no. of nodes (servers and dbs). This will give us node where we want to route our request.



Key = request ID or IP
H = hash function result
N = total no. of nodes
node = where request will be routed

Problem
- when we add or remove a node, N will change, meaning our mapping strategy will break as same requests will now map to a different server.
  — As a consequence, the majority of requests will need to be rebalanced which is very difficult.
- we want to uniformly distribute requests among different nodes such that we should be able to add or remove nodes with minimal effort.
  Hence, we need a distribution scheme that does not depend directly on the no. of nodes (or servers), so that, when adding or removing nodes, the no. of keys that need to be relocated is minimized.
- Consistent hashing solves this horizontal scalability problem by ensuring that every time we scale up or down, we do not have to rearrange all the keys or touch all the servers.

Working
- It is a distributed hashing scheme that operates independently of the no. of nodes in a distributed hash table by assigning them a position on an abstract circle, or hash ring.
  This allows servers and objects to scale without affecting the overall system.



using consistent hashing, only K/N data would require remediation.
R = K/N
where, R — data that would require redistribution
K — no. of partition keys
N — no. of nodes

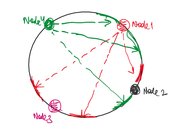
The output of the hash function is a range, let's say 0 — m, which can represent an arc, hash ring.
- we hash the request and distribute them on the ring depending on what the output was.
- similarly, we also hash the node and distribute them on the same ring as well.

$Hash(key_1) = P_1$
$Hash(key_n) = P_{m-1}$
where key: request ID or IP
P: position on hash ring
m: total range of hash ring

- now when request comes in we can simply move this to closest node in clockwise manner.
  This means that if a new node is added or removed, we can use the nearest node and only a fraction of the requests need to be re-mapped.
- In theory, consistent hashing should distribute the load evenly however it doesn't happen in practice.
  Usually, load distribution is uneven and one server may end up handling the majority of the request becoming a bottleneck, eventually a bottleneck for the system.
- we can fix this by adding extra nodes but can be expensive

Virtual Nodes

In order to ensure a more evenly distributed load, we can introduce the idea of virtual nodes (VNodes)
- Instead of assigning a single position to a node, the hash range is divided into multiple smaller ranges, and each physical node is assigned several of those smaller ranges.
- Each of these subranges is considered a VNode.
  Hence, virtual nodes are basically existing physical nodes mapped multiple times across the hash ring to minimize changes to a node's assigned range.



For this, we can use k no. of hash functions
$Hash_1(key) = P_1$
$Hash_k(key) = P_{m-1}$
where key — request ID or IP
k — no. of hash functions
P — position on hash ring
m — total range of hash ring

The VNodes help spread the load more evenly across the physical nodes on the cluster by dividing hash ranges into smaller subranges, this speed up the rebalancing process after adding or removing nodes.
This also helps reducing the probability of hotspots.

Data Replication
- To ensure high availability and durability, consistent hashing replicates each data item on multiple N nodes in the system where value of N is equivalent to replication factor.
  Replication factor is no. of nodes that will receive copy of same data.
  In eventually consistent systems, this is done asynchronously.

Advan of consistent hashing
makes rapid scaling up and down more predictable
facilitates partitioning and replication across nodes
enables scalability and availability
reduces hotspots

Disadv
increase complexity
cascading failures
load distribution can still be uneven
key mgmt can be expensive when nodes frequently fail.

Eg:
- Data partitioning in Apache Cassandra
- load distribution across multiple storage nodes in Amazon DynamoDB.