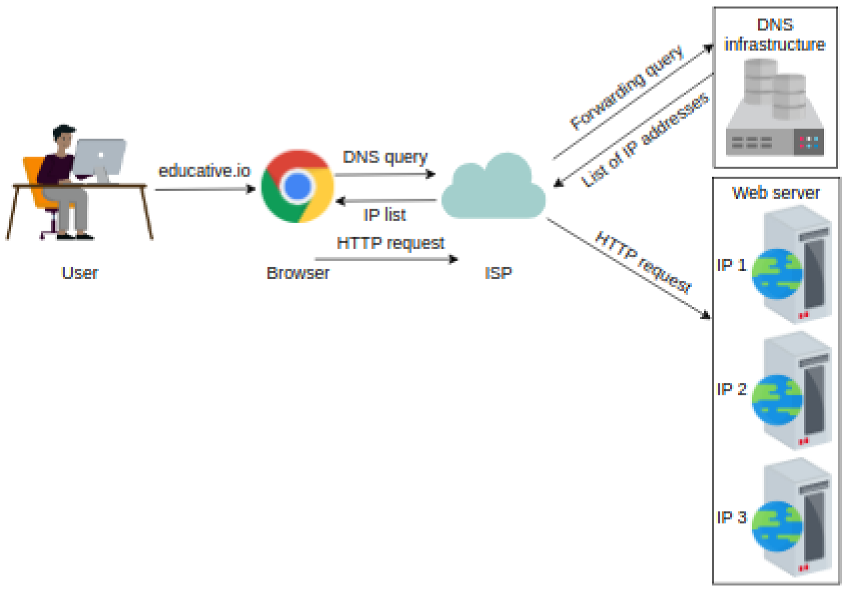
1. Domain Name System

DNS is the Internet's naming service that maps human-friendly domain names to machine-readable IP addresses. Typically, DNS service is transparent to users. The browser translates the domain name to IP address via DNS Infra.



The ISP forwards the HTTP request to the web server

Important Details

- Name servers: DNS isn't a single server. It's a complete infrastructure with numerous servers. DNS servers that respond to the users' queries are called name servers.
- Resource records: The DNS database stores domain name to IP address mappings in the form of record names (RR). The RR is the smallest unit of information that users request from the name servers. The 3 important pieces of information are type, name, and value.

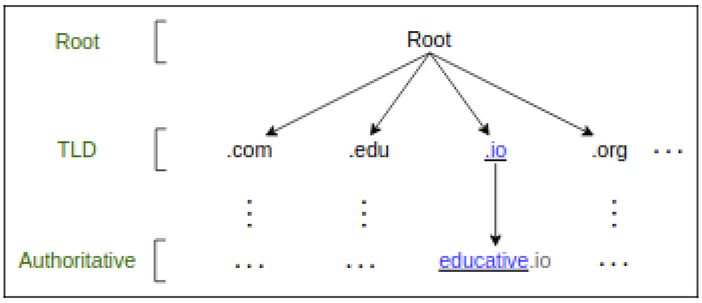
Type	Description	Name	Value	Example(Type, Name, Value)
A	Provides the hostname to IP address mapping	Hostname	IP address	(A, relay1.main.educative.io,104,18.2.119)
NS	Provides the hostname that is the authoritative DNS for a domain name	Domainname	Hostname	(NS,educative.io, dns.educative.io)
CNAME	Provides the mapping from alias to canonical hostname	Hostname	Canonical name	(CNAME, educative.io, server1.primary.educative.io)
MX	Provides the mapping of mail server from alias to canonoical hostname	Hostname	Canonical name	(MX, mail.educative.io, mailserver1.backup.educative.io)

- Caching: DNS uses caching at different layers to reduce request latency for the user.

- Hierarchy: DNS name servers are hierarchical form. The hierarchical structure allows DNS to be highly scalable because of its increasing size and query load.

DNS hierarchy

- DNS isn't a single server that accepts requests and responds to queries. It's a complete infrastructure with "Name Servers" at different hierarchies. Below are the different types of servers in the DNS hierarchy.
    - DNS resolver: Resolvers initiate the querying sequence and forward requests to the other DNS name servers. Typically, DNS resolvers lie within the premis of the user's network. These servers can also be called as local or defaults servers. Caching technique is used fo cater DNS queries.
    - Root-level name servers: These servers receives requests from the local servers. Root name servers maintain name servers based on top-level domain names, such as .com, .edu, .us, and so on. When a user requests the IP address of "docker.io", root-level name servers will return a list of top-level domain (TLD) servers that hold the IP addresses of the .io domain.
    - Top-level domain(TLD) name servers: These servers hold the IP addresses of authoritative name servers. The querying party will get a list of IP addresses that belong to the authoritative servers of the organization.
    - Authoritative name servers: These are the organization's DNS name servers that provide the IP addresses of the web or application servers.

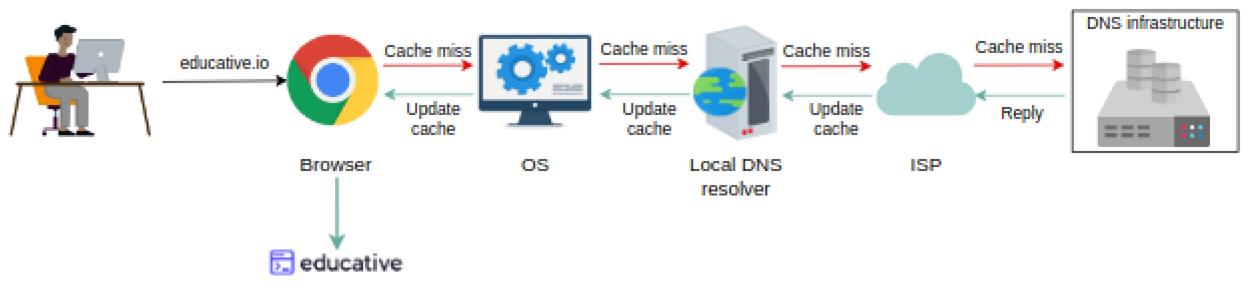


DNS hierarchy for resolution of domain/host names

- Typically, an iterative query is preferred to reduce query load on DNS infrastructure.

DNS caching

- A record is a data unit within the DNS database that shows a name-to-value binding. Caching reduces response time to the user and decreases network traffic.



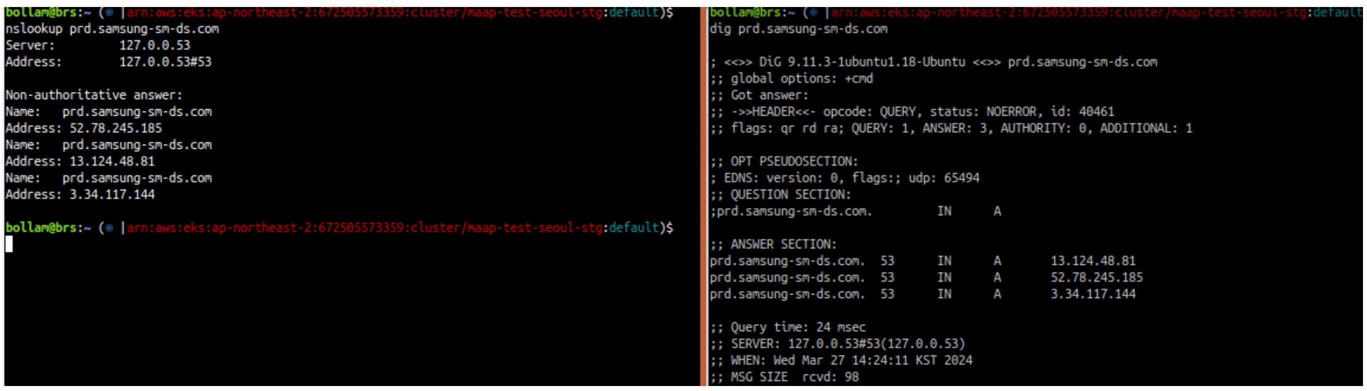
The cache will be updated at each hierarchy

DNS as a distributed system

- Highly Scalable There are 13 logical root name servers (named letter A to M) with over 1,000 replicated instances spread throughout the world strategically to handle user queries and managed by 12 different organizations.
- Reliable
  - Caching is done in the browser, the OS, and the local name server, and the ISP DNS resolvers also maintain a rich cache of freqently visited services. Even if the servers are down, cached records can be served.
  - Server replications DNS has replicated copies of each logical server spread systematiclly across the globe to entertain users requests at low latency.
  - Protocol Many clients rely on the unrealiable UDP to request and recevie DNS response, but is fast and improves DNS performance. DNS queries are usually retransmitted at the transport layer if there's no response for the previous one. Therefore, request-response might need additional round trips, which provides a shorter delay as compared to TCP, which needs a three-way handshake every time before data exchange.
- Consistent DNS provides eventual consistency and updates records on replicated servers lazily. Typically, it can take from few seconds up to three days on the DNS servers across the Internet. Consistency can suffer because of caching too. Since authoritativce servers are located within the ogranization, it may be possible that certain resource records are updated on the authoritative servers in case of server failures at the organization. To mitigate this issue, each cached record comes with an expiration time called (TTL)

Test

```
nsloop => Non-Authoritative answer --> Refererence to cached response.
dig => 53 refers to TTL and "Query time: 24 msec" refers to DNS response
time
```



```
bollam@brs:~ (⎈ |arn:aws:eks:ap-northeast-2:672505573359:cluster/maap-test-seoul-stg:default)$
nslookup prd.samsung-sm-ds.com
Server:         127.0.0.53
Address:        127.0.0.53#53

Non-authoritative answer:
Name:   prd.samsung-sm-ds.com
Address: 52.78.245.185
Name:   prd.samsung-sm-ds.com
Address: 13.124.48.81
Name:   prd.samsung-sm-ds.com
Address: 3.34.117.144

bollam@brs:~ (⎈ |arn:aws:eks:ap-northeast-2:672505573359:cluster/maap-test-seoul-stg:default)$
```

```
bollam@brs:~ (⎈ |arn:aws:eks:ap-northeast-2:672505573359:cluster/maap-test-seoul-stg:default
dig prd.samsung-sm-ds.com

; <<>> DiG 9.11.3-1ubuntu1.18-Ubuntu <<>> prd.samsung-sm-ds.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 40461
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 65494
;; QUESTION SECTION:
;prd.samsung-sm-ds.com.         IN      A

;; ANSWER SECTION:
prd.samsung-sm-ds.com.  53      IN      A       13.124.48.81
prd.samsung-sm-ds.com.  53      IN      A       52.78.245.185
prd.samsung-sm-ds.com.  53      IN      A       3.34.117.144

;; Query time: 24 msec
;; SERVER: 127.0.0.53#53(127.0.0.53)
;; WHEN: Wed Mar 27 14:24:11 KST 2024
;; MSG SIZE  rcvd: 98
```

2. Load balancers

Introduction

The job of the load balancer is to faily divide all client's requests among the pool of available servers. Load balancers perform this job to avoid overloading or crashing servers.

The load balancing layer is the first point of contact within a data center after the firewall. They provide following capabilities:-

- Scalability - Load balancers make upscaling or downscaling of servers transparent to the end users.
- Availability - One of the jobs of the load balancers is to hide faults and failuers of servers. Even if some servers go down or suffer a fault, the system still remains available.
- Performance - Load balancers can forward requests to servers with a lesser load so the user can get a quicker response time. This not only improves performance but also improves resource utilization.

Services offered by Load Balancers

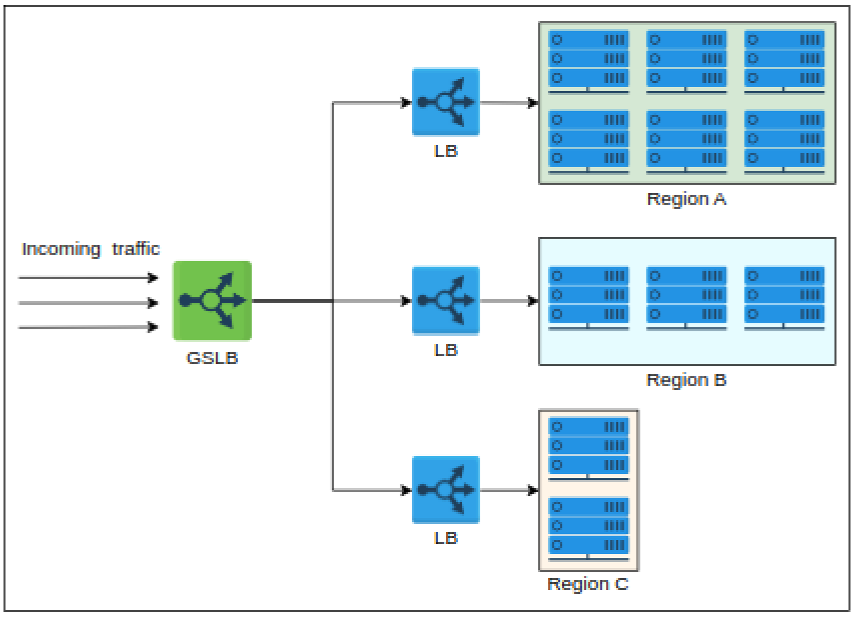
- Health checking: LBs use the heartbeat protocol to monitor the health and, therefore, reliability of end-servers.
- TLS termination: LBs reduce the burden on end-servers by handling TLS termination with the client.
- Predictive analytics: LBs can predict traffic patterns through analytics performed over traffic passing through them or using statistics of traffic obtained over time.
- Service discovery: Client's requests are forwarded to appropriate hosting servers by inquiring about the server registry.
- Security: LBs improve security by mitigating attacks like DoS at different layers of the OSI model (3,4, 7).

Global server load balancers (GSLB)

GSLB involves the distribution of traffic load across multiple geographical regions. GSLB ensures that globally arriving traffic load is intelligently forwarded to a data center.

GSLB takes forwarding decisions based on the users' geographic locations, the number of hosting servers in different locations, the health of data centers and so on. GSLB offers automatic zonal failover. Example: power or network failure in a data center requires that all the traffice be routed to another data center.

GSLB service can be installed on-premises or obtained through "Load balancing as a Service(LBaaS)".



Usage of global load balancing to send user requests to different regions

Local Load balancers

Load balancing achieved within a data center. This type of load balancing focuses on improving efficieny and better resource utliziation of the hosting servers in a data center. They behave like a "reverse proxy" and make their best effort to divide incoming requests among the pool of available servers. Incoming clients' requests seamlessly connect to the LB that uses a virtual IP address (VIP).

Advanced Load balancers

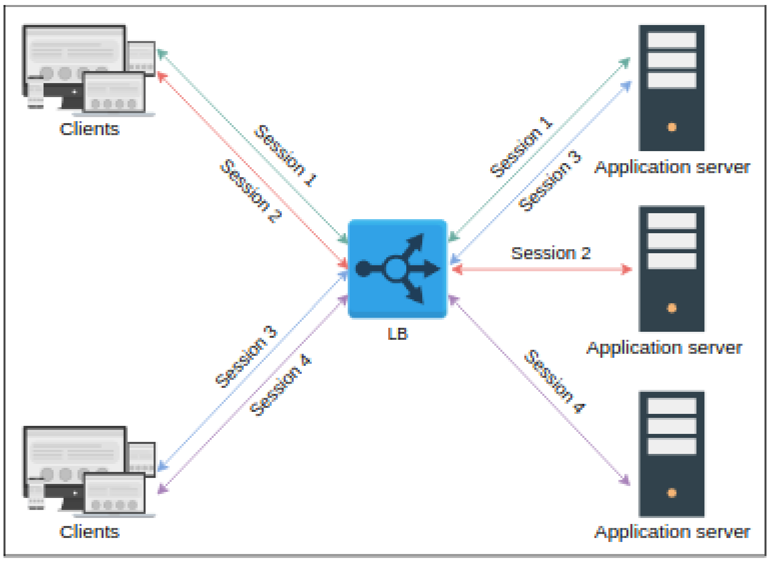
- **Algorithms of load balancers**
  - **Round-robing scheduling** - each request is forwarded to a server in the pool in a repeating sequential manner.
  - **Weighted round-robin** - If some servers have a higher capability of serving clients' requests, then it's preferred to use a weighted round-robin algorithm. Each node is assigned a weight. LBs forward requests according to the weight of the node. The higher the weight, higher the number of assignments.
  - **Least connections** - If all the servers have the same capacity to serve clients, uneven load on certain servers is still a possibility. Newer requests are assigned to servers with fewer existing connections. LBs keep a state of the number and mapping of existing connections in such a scenario.
  - **Least response time** - In performance-sensitive services, algorithms such as least response time are required. This algorithm ensures that the server with the least response time is requested to serve the clients.
  - **IP hash** - Some applications provide a different level of service to users based on their IP addresses. In that case, hashing the IP address is performed to assign users' requests to servers.

- **URL hash** - It may be possible that some services within the application are provided by specific servers only. In that case, a client requesting service from a URL is assigned to a certain cluster or set of servers. The URL hasing algorithm is used in those scenarios.
- **Static vs dynamic algorithms**
    - Static algorithms don't consider the changing state of the servers. Therefore, task assignment is carried out based on existing knowledge about the server's configuration.
    - Dynamic algorithms consider the current or recent state of the servers. Dynamic algorithms maintain state by communicating with the server, which adds a communication overhead. State maintenance makes the design of the algorithm much more complicated.

  In practice, dynamic algorithms provide far better results because they maintain a state of the serving hosts and are, therefore, worth the effort and complexity.

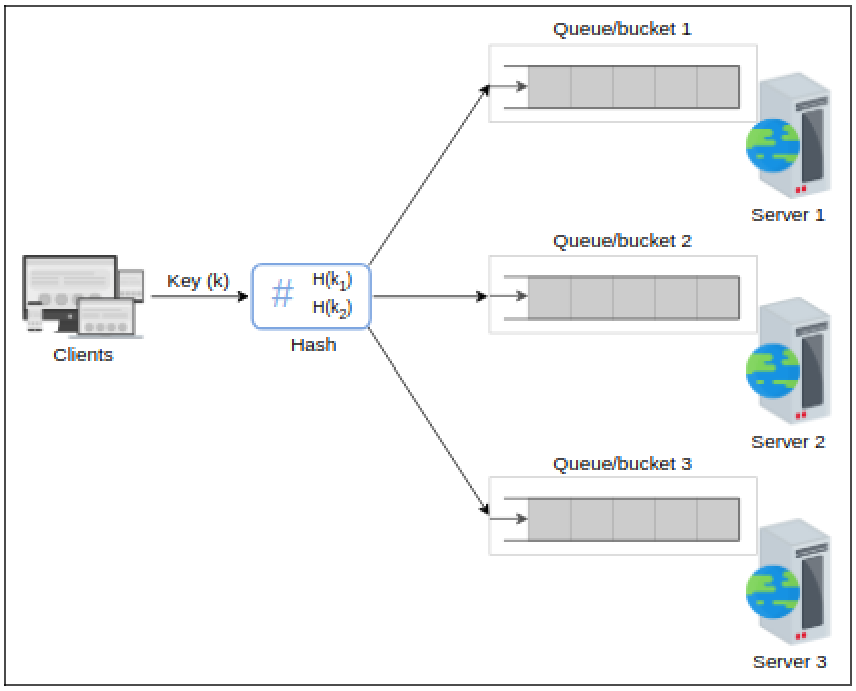
Stateful load balancers versus stateless LBs

- Stateful load balancing involved maintaining a state of the sessions established between clients and hosting servers. The stateful LB incorporates state information in its algorithm to perform load balancing. Stateful LBs increase complexity and limit scalability because session information of all the clients is maintained across all the load balancers.That is, load balancers share their state information with each other to make forwarding decisions.



Stateful load balancing

- Stateless load balancing maintains no state and is, therefore, faster and light weight. Stateless LBs use consistent hashing to make forwarding decisions. However, if infrastructure changes (a new application server is added), stateless LBs may not be resilient as stateful LBs because consistent hashing alone isn't enough to route a request to the correct application server. Therefore, local state may still be required along with consistent hashing.

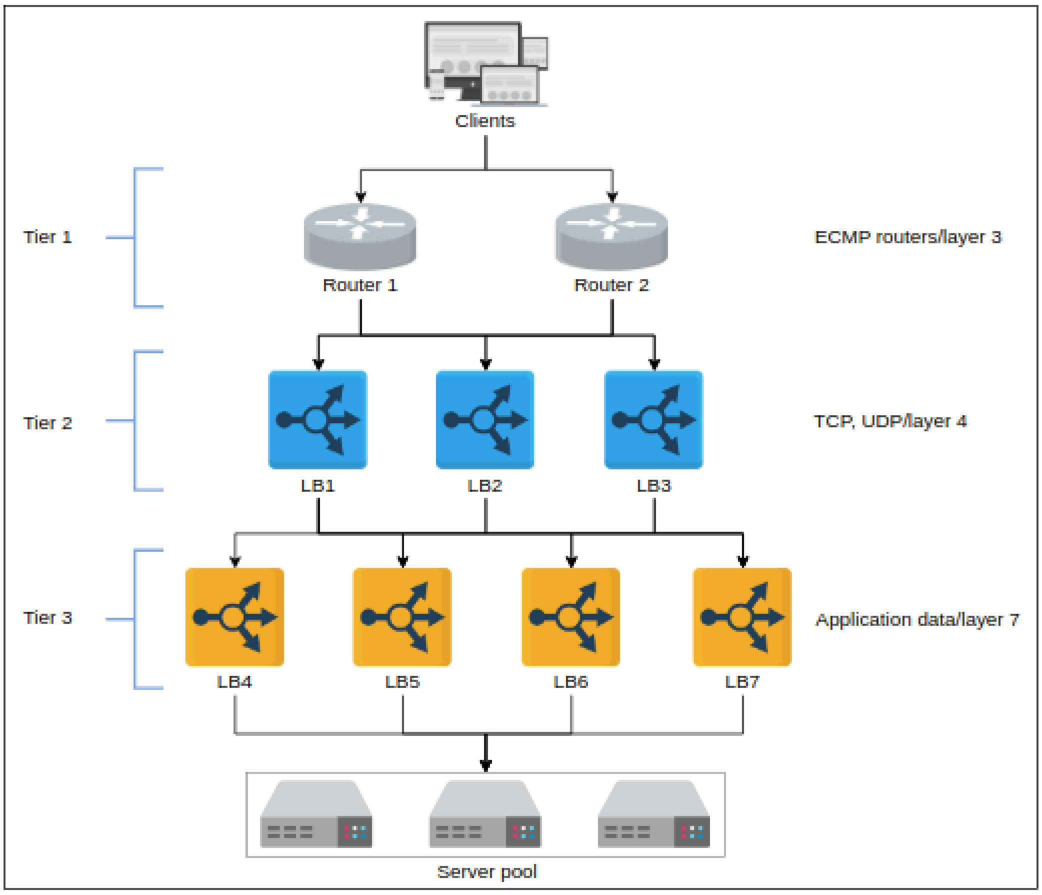


Stateless load balancers using hash buckets to map requests to end servers

- **Types of Load balancers**
    - **Layer 4 load balancer**: Layer 4 refers to load balancing performed on the basis of transport protocol like tcp/udp. These types of LBs maintain conneciton/session with the clients and ensure that the same TCP/UDP communication ends up being forwarded to the same back-end server. Even though TLS termination is performed at layer 7 LBs, some layer 4 LBs also support it.
    - **Layer 7 load balancer**: Layer 7 load balancers are based on the data of application layer protocols. It's possible to make application-aware forwarding decisions based on HTTP headers, URLs, cookies, and other application-specific data- for example, user ID. Apart from TLS termination, these LBs can take responsibility like rate limiting users, HTTP routing, and header rewriting.

  Layer 7 load balancers are smart in terms of inspection. However layer 4 load balancers are faster in terms of processing.

- **Load balancer deployments** In practice, a single layer LB isn't enough for a large data center. In fact, multiple layers of load balancers coordinate to take informed forwarding decisions as shown below:-



Three-tier load balancer in a typical data center

- **Tier-0 and Tier-1 LBs**: If DNS can be considered as the tier-0 load balancer, equal cost multipath (ECMP) routers are the tier-1 LBs. ECMP routers divides incoming traffic on the basis of IP or some other algo like round-robin or weighted round-robin. Tier-1 LBs will balance the load across different paths to higher tiers of load balancers. ECMP routers play a vital role in the horizontal scalability of the higher-tier LBs.

- **Tier-2 LBs**: The Tier-2 LBs include layer 4 load balancers. They make sure that for any connection, all incoming packets are forwarded to the same tier-3 LBs. Consistent Hashing can be used for this purpose. Since hashing has limiation with infra changes, local or global state needs to be maintained. Tier-2 LBs glues Tier-1 and Tier-3 LBs.

- **Tier-3 LBs**: These LBs are in direct contact with the back-end servers, they perform health monitoring of servers at HTTP level. This tier enables scalability by evenly distributing requests amoung the set of healthy back-end servers. This tier also reduces the burden on end-servers by handling low-level details like TCP-congestion control protocols, the discover of Path MTU (maximum transmission unit), the difference in application protocol between client and back-end servers, and so on.

Tier-1 balances the load among the load balancers themselves. Tier 2 enables a smooth transition from tier-1 and tier-3 in case of failures Tier-3 does the actual load balancing between the back-end servers.

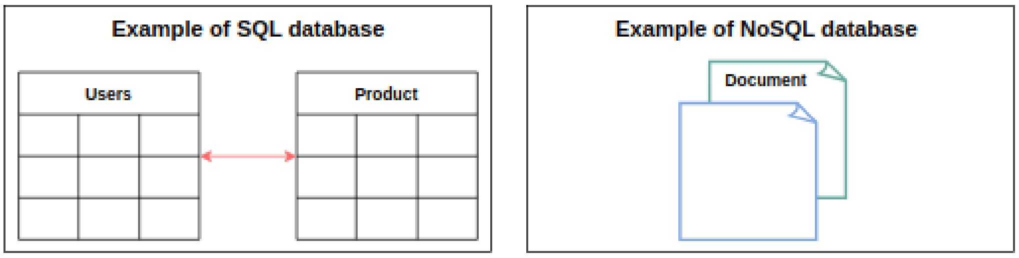
3. Databases

Introduction

- For an application like WhatsApp, why can't we store information in a File?
    1. We can't offer concurrent managment to seperate users accessing the storage files fromd different loc.
    2. We can't grant different access rights to different users.
    3. How will the system scale and be available when adding thousands of entires?
    4. How will we search content for different users in a short time?

A database is an organized collection of data that can be managed and accessed easily. Databases are created to make it easier to store, retrieve, modify, and delete data in connection with different data-processing procedures.

- Two basic types of databases:
    - SQL (relational database) Relational databases, like phone books that record contact numbers and addresses, are organized and have predetermined schemas.
    - NoSQL (non-relational database) Non-Relational databases, like file directories that store anything from a person's constant information to shopping preferences, are unstructured, scattered, and feature a dynamic schema.



Relational database has a well defined structure such as attributes (columns of the table). NoSQL databases such as document databases often have application-defined structure of data.

- Reasons why a database is important

    - Managing large data: A large amount of data can be easily handled with a database, which wouldn't be possible using other tools.

    - Retrieving accurate data (data consistency): Due to different constraints in databases, we can retrieve accurate data whenever we want.

    - Easy Updation: It is quite easy to update data in databases using data manipulation language (DML).

    - Security: Databases ensure the security of the data. A database only allows authorized users to access data.

    - Data Integrity: Databases ensure data integrity by using different constraints of data.

    - Availability: Databases can be replicated on different servers, which can be concurrently updated. These replicas ensure availability.

    - Scalability: Databases are divided (using data paritioning) to manage the load on a single node. This increases scalability.

Types of Database

1. Relational Database

```
Relational databases adhere to particular schemas before stroing the data. 
The data stored in relational databases
has prior structure. Mostly, this model organizes data into one or more 
relations called tables, with a unique key
for each tuple
Since each tuple has a unique key, a tuple in one table can be linked to a 
tuple in other tables by storing the
primary keys in other tables, generally known as foreign keys.

A SQL (Structural Querying Language) is used for manipulating the database. 
This includes insertion, deletion, and
retrieval of data.

Relational databases provide Atomicity, consistency, isolation and 
durablity (ACID) properties to maintain the
integrity of the database.
    - Atomicity: A transasction is considered an atomic unit. If a 
statement fails within a transaction, it should
        be aborted and rolled back.
    - Consistency: At any given time, the database should be in a 
consistent stated, and it should remain consistent
        after every transaction.
    - Isolation: In the case of multiple transactions running concurrently, 
they shouldn't be affected by each
        other. The final state of the database should be the same as the 
transactions were executed sequentially.
    - Durability: The system should guarantee that completed transactions 
will survive permanently in the database
        even in system failures.

    Examples - MySQL, Oracle Database, Microsoft SQL Server, IBM DB2, 
Postgres, SQLite

One of the greatest powers of the relational database is its abstractions 
of ACID transactions and related
programming semantics.
```

Flexibility

```
In the context of SQL, DDL (Data Definition Language) provides us the 
flexibility to modify the database, including
tables, columns, renaming the tables, and other changes. DDL even allows us 
to modify the schema while other
queries are happening and the database server is running.
```

Reduced redundancy

```
One of the biggest advantages of the relational database is that it eliminates data redundancy. The information
related to a specific entity appears in one table while the relevant data to that specific entity apprears in the
other tables linked through foriegn keys. This process is called normalization and has the additional benefit of
removing an inconsistent dependency.
```

Concurrency

```
A transaction is considered an atomic operation, so it also works in error handling to either roll back or commit
a transaction on successful execution.
```

Integration

```
The process of aggregating data from multiple sources is a common practice in enterprise applications. A common
way to perform this aggregation is to integrate a shared database where multiple applications store their data.
```

Backup and disaster recovery

```
Relational databases gurantee the state of data is consistent at any time. Most cloud-based relational databases
perform continous mirroing to avoid loss of data and make the restoration process easier and quicker.
```

Drawbacks of Relational databases

Impedance mismatch Impedance mismatch is the difference between the relational model and the in-memory data structures. One is a structured data and other is a complex in-memory data structure. so, the impedence mismatch requires translation between two representations.

2. Non-Relational Database

A NoSQL database is designed for a variety of data models to access and manage data. These databases are used in applications that require a large volume of semi-structured and unstructured data, low latency, and flexible data models.

Simple design

Unlike relational databases, NoSQL doesn't require dealing with the impedance mismatch - for ex storing all the employees' data in one document instead of multiple tables that require join operations. This strategy makes it simple and easier to write less code, debug, and maintain.

Horizontal scaling

Primarily, NoSQL is preferred due to its ability to run databases on a large scale. As the data is stored in one document. NoSQL databases often spread data across multiple nodes and balance data and queries across nodes automatically. In case of a node failure, it can be transparently replaced without any application disruption.

Availability

To enhace the availability of data, node replacement can be performed without application downtime. Data replication is supported to ensure high availability and disaster recovery.

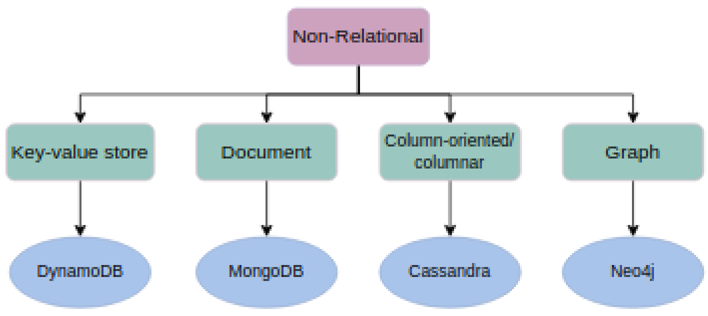
Support for unstructured and semi-structured data

Many NoSQL databases work with data that doesn't have schema at the time of database configuration or data writes. For ex: document databases are structureless; they allow documents (JSON, XML, BSON and so on) to have different fields.

Cost

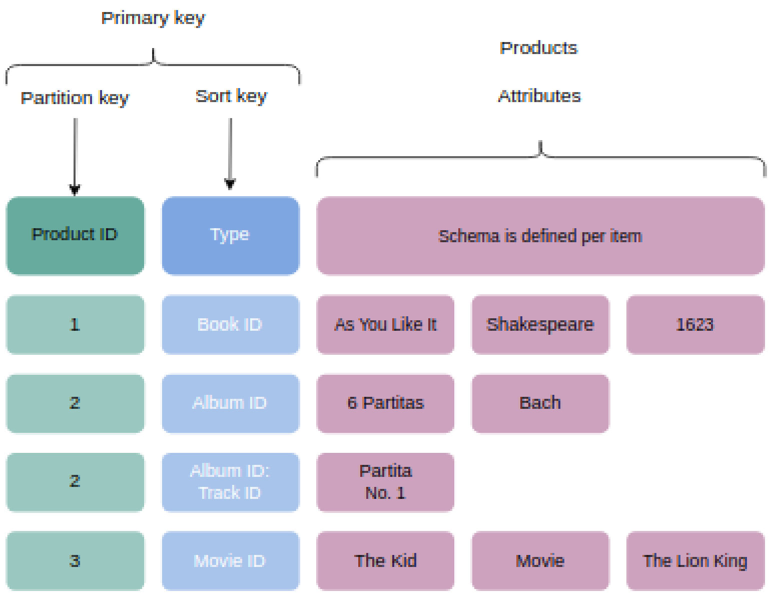
Licenses for many RDBMSs are pretty expensive, while many NoSQL databases are opensource and freely availble

Types of Databases



The types of NoSQL databases

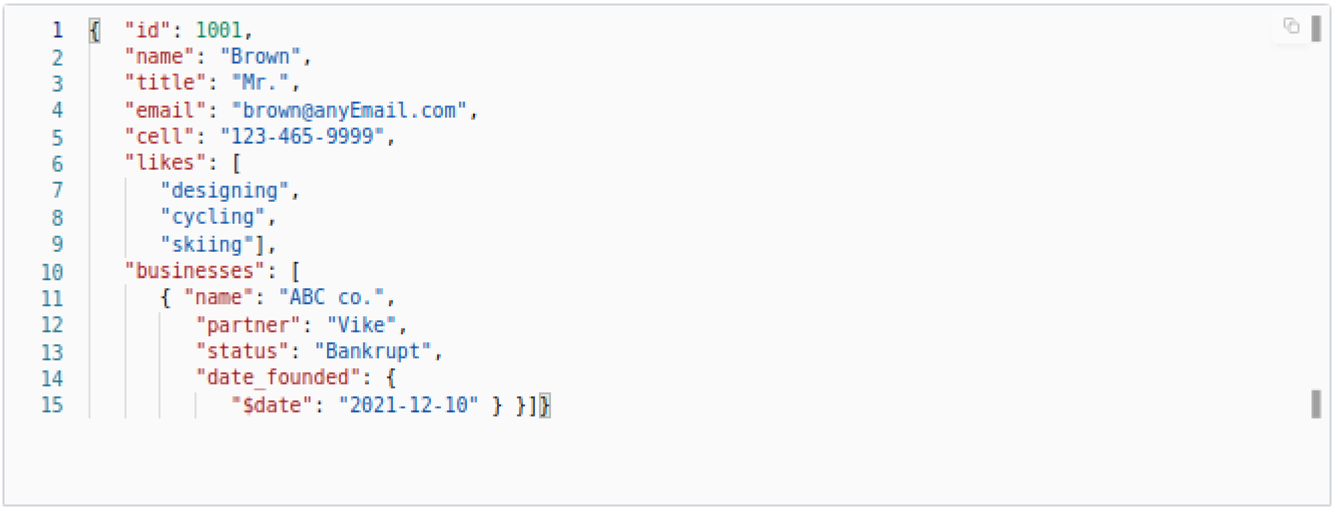
Key-Value database: use key-value methods like hash tables to store data in key-value pairs. Key servers as a unique or primary key, and the values can be anything ranging from simple scalar values to complex objects. Allow easy partitioning and horizontal scaling of the data. Examples: Amazon DynamoDB, Redis, and Memcached DB.



Data stored in the form of key-value pair in DynamoDB, where the key is the combination of two attributes (Product ID and Type)

Use Case: Key-value databases are efficient for session-oriented applications. Session oriented-applications, such as web applications, store users' data in the main memory or in a database during a session.

Document database: is designed to store and retrieve documens in formats like XMl, JSON, BSON, and so on. These documents are composed of hierarchical tree data structure that can include maps, collections, and scalar values. Documents in this type of database may have varying structures and data. Examples : MongoDB, Google Cloud Firestore.

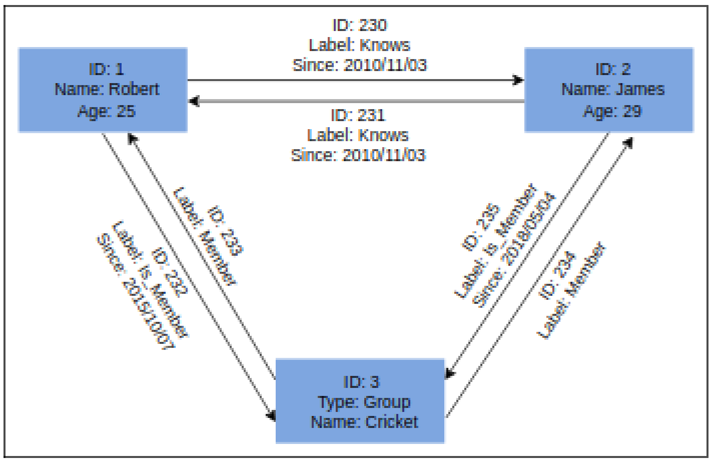


```
{  "id": 1001,
   "name": "Brown",
   "title": "Mr.",
   "email": "brown@anyEmail.com",
   "cell": "123-465-9999",
   "likes": [
      "designing",
      "cycling",
      "skiing"],
   "businesses": [
      { "name": "ABC co.",
        "partner": "Vike",
        "status": "Bankrupt",
        "date_founded": {
           "$date": "2021-12-10" } }]}
```

A JSON file containing data of a businessman

Use Case: Suitable for unstructured catalog data, like JSON files or other complex structured hierarchical data. for example in e-commerce applications, a product has thousands of attributes, which is unfeasible to store in a relational database due to its impact on the reading performance.

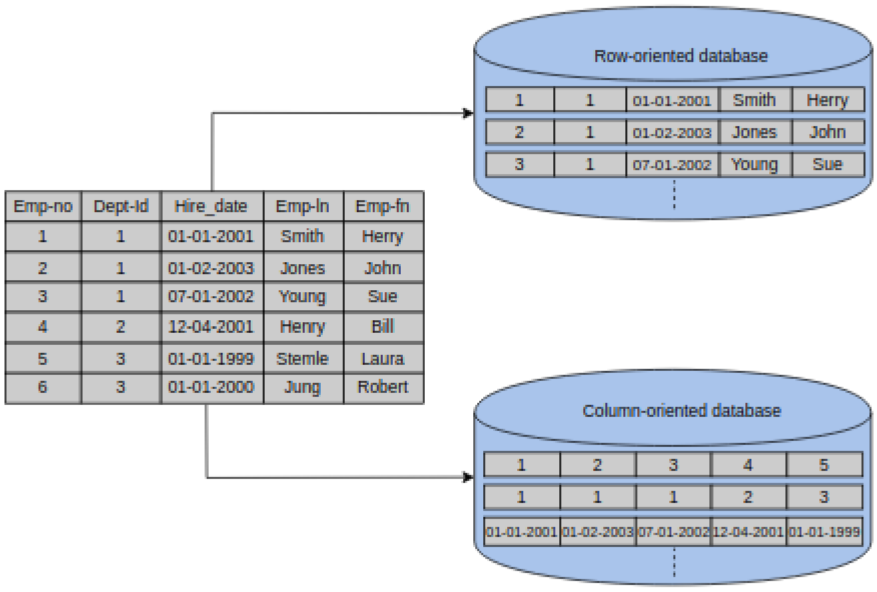
Graph database: uses graph data structure to store data, where nodes represent entities, and edges show relationships between entities. The organizaton of nodes based on relationships leads to interesting patterns between the nodes. This database allows us to store the data once and then interpret it differently based on relationships. Graph data is kept in store files for persistent storage. Examples: Neo4J, OrientDB, InfiniteGraph.



A graph consists of nodes and links. This graph captures entities and their relationships with each other

UseCase: can be used in social applications and provide interesting facts and figures among different kinds of users and their activities. The focus of graph database is to store data and pave the way to drive analyses and decisions based on relationships between entities. The nature of graph databases makes them suitable for various applications such as data regulation and privacy, machine learning research, financial services-based applications and many more.

Columnar database: store data in columns instead of rows. They enable across to all entries in the database column quickly and efficiently. Popular columnar databases include Cassandra, HBase, Hypertable, and Amazon redshift.



Emp-no	Dept-Id	Hire_date	Emp-ln	Emp-fn
1	1	01-01-2001	Smith	Herry
2	1	01-02-2003	Jones	John
3	1	07-01-2002	Young	Sue
4	2	12-04-2001	Henry	Bill
5	3	01-01-1999	Stemle	Laura
6	3	01-01-2000	Jung	Robert

1	1	01-01-2001	Smith	Herry
2	1	01-02-2003	Jones	John
3	1	07-01-2002	Young	Sue

1	2	3	4	5
1	1	1	2	3
01-01-2001	01-02-2003	07-01-2002	12-04-2001	01-01-1999

Column-oriented and row-oriented database

UseCase: Columnar database are efficient for a large number of aggregation and data analytics queries. It drastically reduces the disk I/O requirements and the amount of data required to load from the disk. For example, in applications related to financial institutions, there's a need to sum the financial transaction over a period of time. Columnar databases make this operation quicker by just reading the column for the amount of money, ignoring other attributes of customers.

Choosing the right database

Relational database	Non-relational database
If the data to be stored is structured	If the data to be stored is unstructured
If ACID properties are required	If there's a need to serialize and deserialize data
If the size of the data is relatively small and can fit on a node	If the size of the data to be stored is large

Drawbacks of NoSQL databases

Lack of standardization - NoSQL doesn't follow any specific standard, like how relational database follow relational algebra. Porting applications from one type of NoSQL database to another might be a challenge.

Consistency - NoSQL provides different products based on the specific trade-offs between consistency and availability when failures can happen. We won't have strong data integrity, like primary and referential integrities in a relational database.

Data might not be strongly consistent but slowly converging using a weak model like eventual consistency.

Data Replication

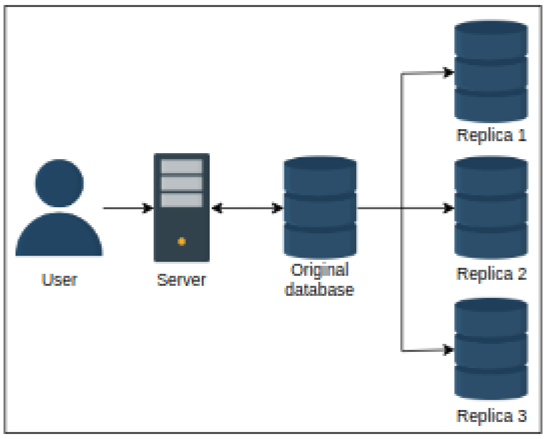
Following characteristics are required from a data store :-

- Availability under faults (failure of some disk, nodes, and network and power outages).
- Scalability (with increasing reads, writes, and other operations).
- Performance (low latency and high throughput for the clients).

It's challenging, or even impossible, to achieve the above characteristics on a single node.

Replication refers to keeping multiple copies of the data at various nodes (preferably geographically distributed) to achieve availability, and performance.

Replication is relatively simple if the replicated data doesn't frequent changes. The main problem in replication arises when we have to maintain changes in the replicated data over time.



Replication in action

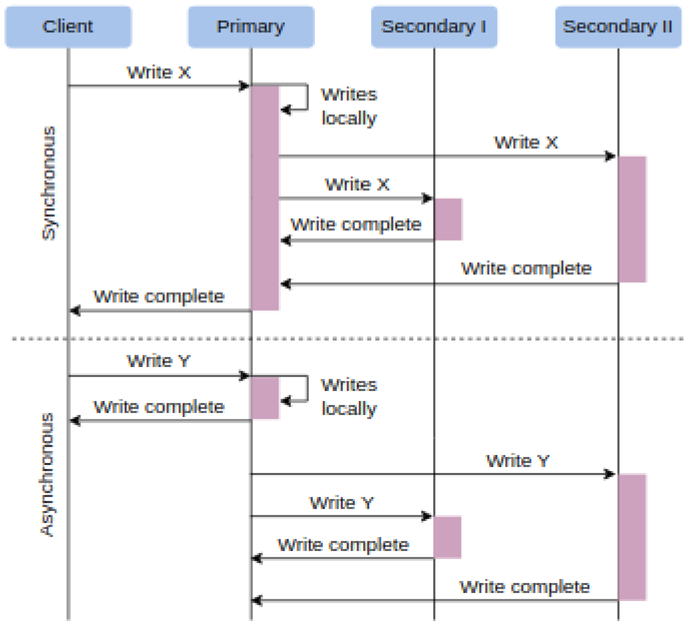
Additional complexities :-

1. How do we keep multiple copies of data consistent with each other?
2. How do we deal with failed replica nodes?
3. Should we replicate synchronously or asynchronously?
    - How to deal with replication lag in case of asynchronous replication?
4. How do we handle concurrent writes?
5. What consistency model needs to be exposed to the end programmers?

Synchronous Vs Asynchronous

There are two ways to disseminate changes to the replica node:

- Synchronous replication
    - Primary node waits for acknowledgment from secondary nodes about updating the data. After receiving all the acknowledgments, primary node reports success to the client.
- Asynchronous replication
    - Primary node doesn't wait for the acknowledgment from the secondary nodes and reports success to the client after updating itself.



Synchronous versus asynchronous replication

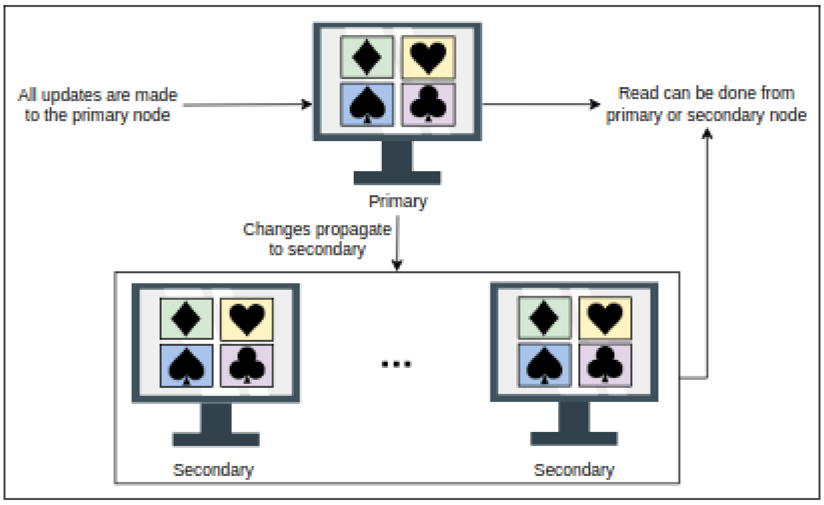
Data replication models

1. Single leader/primary-secondary replication

Data is replicated across multiple nodes. One node is designated as the primary. It's responsible for processing any write to data stored on the cluster. It also sends all the writes to the secondary nodes and keep them in sync.

Appropriate for read-heavy workloads. However, replicating data to many followers can make a primary bottleneck. In appropriate if workload is write-heavy.

It is read resilient. Secondary nodes can still handle read requests in case of primary node failure.


All updates are made to the primary node
Read can be done from primary or secondary node
Primary
Changes propagate to secondary
...
Secondary
Secondary

Primary-secondary data replication model where data is replicated from primary to secondary

There are many replication methods in primary-secondary replication-

- Statement-based replication
- Write-ahead log (WAL) shipping
- Logical (row-based) replication

Statement-based replication (SBR) is an approach used in MySQL databases. In this approach, the primary node executes the SQL statements such as INSERT, UPDATE, DELETE, etc., and then the statements are written into a log file.

In the next step, the log file is sent to the secondary nodes for execution. This type of replication was used in MySQL before version 5.1.

While this type of replication seems good, it also has some disadvantages. For example, any nondeterministic functions such as NOW() might result in distinct writes on the primary and secondary nodes.

Write-ahead log (WAL) shipping is a data replication technique used in both PostgreSQL and Oracle. In this technique, when a transaction occurs, it's initially recorded in a transactional log file, and the log file is written to disk.

Subsequently, the recorded operations are executed on the primary database before being transmitted to secondary nodes for execution. Unlike SBR, WAL maintains transactional logs instead of SQL statements into a log file, ensuring consistency when dealing with nondeterministic functions.

Writing to disk also aids in recovery in case of crash failures

For example, when an operation like an UPDATE is executed in PostgreSQL, it's first written to the transactional log file and disk before being applied to the database. This entry in the transactional log can include details such as the transaction ID, operation type, affected table, and new values, after which the changes are replicated to the secondary nodes. However, the drawback of WAL is its tight coupling with the inner structure of the database engine, making software upgrades on the leader and followers complicated.

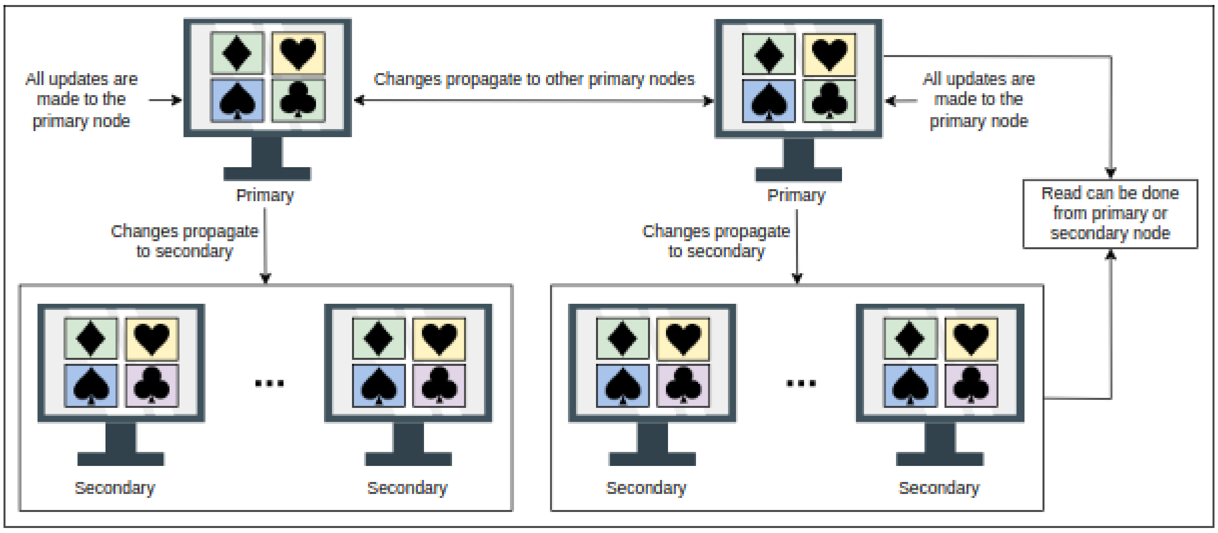
Logical (row-based) replication is utilized in various relational databases, including PostgreSQL and MySQL. In this approach, changes made to the database are captured at the level of individual rows and then replicated to the secondary nodes. Instead of replicating the actual physical changes made to the database, this approach captures the operations in a logical format and then executes them on secondary nodes.

For example, when operations like INSERT or UPDATE are performed, the entire affected row is captured on the primary node, containing all the column values of the specified row. This captured change is then executed on secondary nodes, where they ensure that the data remains consistent with the data on the primary node. It offers advantages in terms of flexibility and compatibility with different types of schemas.

2. Multi-leader replication

Multi-leader replication is an alternative to single leader replication. There are multiple primary nodes that process the writes and send them to all other primary and secondary nodes to replicate. This type of replication is used in databases along with external tools like the Tungsten Replicator for MySQL.

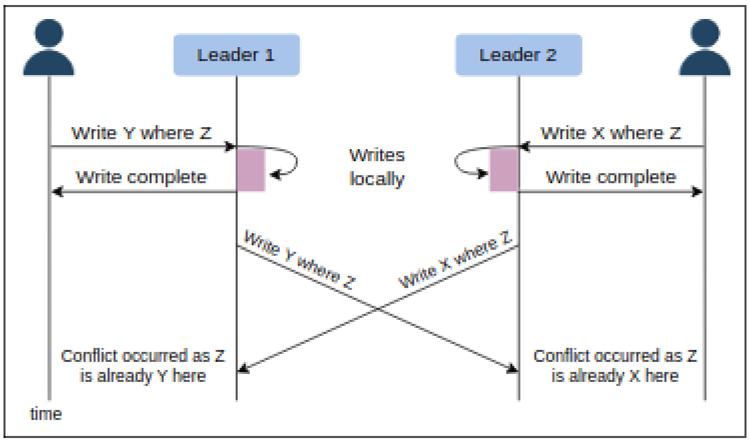
This kind of replication is quite useful in applications in which we can continue work even if we're offline—for example, a calendar application in which we can set our meetings even if we don't have access to the internet. Once we're online, it replicates its changes from our local database (our mobile phone or laptop acts as a primary node) to other nodes.


All updates are made to the primary node
Changes propagate to other primary nodes
All updates are made to the primary node
Primary
Primary
Read can be done from primary or secondary node
Changes propagate to secondary
Changes propagate to secondary
...
...
Secondary
Secondary
Secondary
Secondary

Multi-leader data replication model (all-to-all topology)

Conflict Multi-leader replication gives better performance and scalability than single leader replication, but it also has a significant disadvantage. Since all the primary nodes concurrently deal with the write requests, they may modify the same data, which can create a conflict between them. For example, suppose the same data is edited by two clients simultaneously. In that case, their writes will be successful in their associated primary nodes, but when they reach the other primary nodes asynchronously, it creates a conflict.

Handle conflicts



Conflict of writes

Conflict Avoidance A simple strategy to deal with conflicts is to prevent them from happening in the first place. Conflicts can be avoided if the application can verify that all writes for a given record go via the same leader.

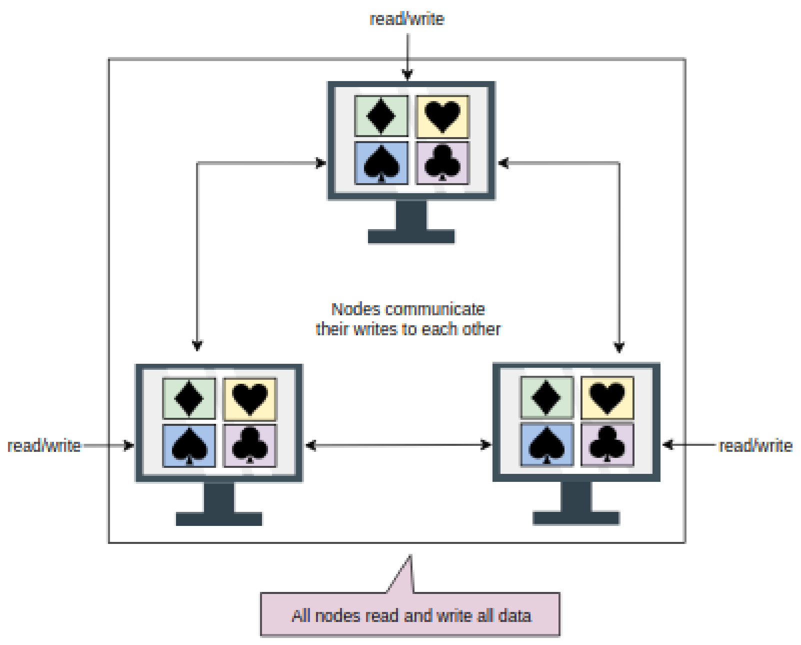
However, the conflict may still occur if a user moves to a different location and is now near a different data center. If that happens, we need to reroute the traffic. In such scenarios, the conflict avoidance approach fails and results in concurrent writes.

Last-write-wins Using their local clock, all nodes assign a timestamp to each update. When a conflict occurs, the update with the latest timestamp is selected. But clock synchronization across nodes is challenging in distributed systems.

Custom logic Write own logic to handle conflicts according to the needs of our application.

3. Peer to peer/leaderless replication

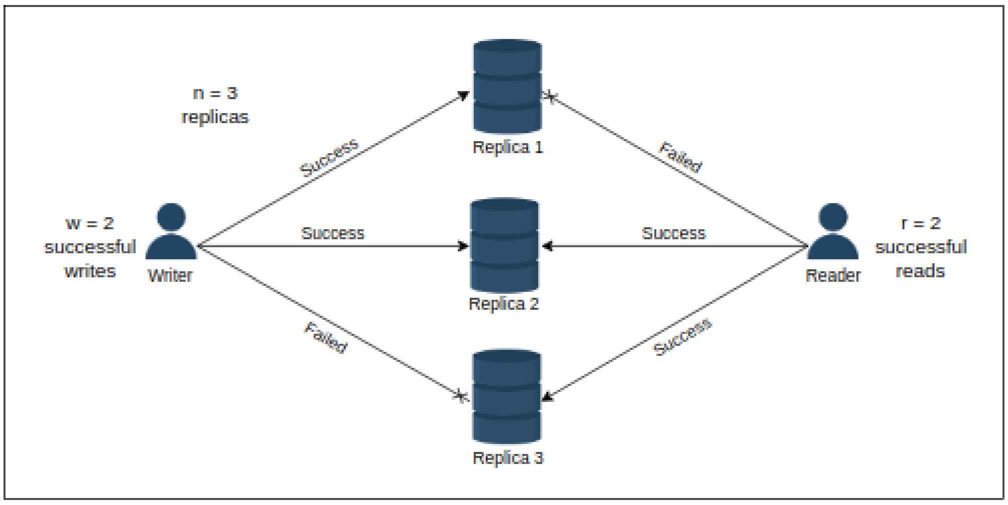
In primary-secondary replication, the primary node is a bottleneck and a single point of failure. Moreover, it helps to achieve read scalability but fails to provide write scalability. The peer-to-peer replication model resolves these problems by not having a single primary node. All the nodes have equal weightage and can accept read and write requests. This replication scheme can be found in the Cassandra database.


read/write
Nodes communicate their writes to each other
read/write
read/write
All nodes read and write all data

Peer-to-peer data replication model where all nodes apply reads and writes to all the data

Like primary-secondary replication, this replication can also yield inconsistency. This is because when several nodes accept write requests, it may lead to concurrent writes. A helpful approach used for solving write-write inconsistency is called quorums.

Quorums Let’s suppose we have three nodes. If at least two out of three nodes are guaranteed to return successful updates, it means only one node has failed. This means that if we read from two nodes, at least one of them will have the updated version, and our system can continue working.



Reader getting an updated value from replica 2

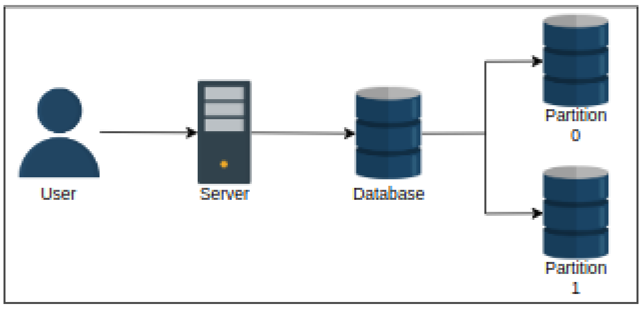
If we have n nodes, then every write must be updated in at least w nodes to be considered a success, and we must read from r nodes. We'll get an updated value from reading as long as w+r > n because at least one of the nodes must have an updated write from which we can read. Quorum reads and writes adhere to these r and w values. Then n, w, and r are configurable in Dynamo-style databases.

Data Partition

Data is an asset for any organization. Increasing data and concurrent read/write traffic to the data puts scalability pressure on traditional databases. As a result, the latency and throughput are affected. Traditional databases are attractive due to their properties such as range queries, secondary indices, and transactions with the ACID properties.

At some point, a single node-based database isn't enough to tackle the load. We might need to distribute the data over many nodes but still export all the nice properties of relational databases. In practice, it has proved challenging to provide single-node database-like properties over a distributed database.

Data partitioning (or sharding) enables us to use multiple nodes where each node manages some part of the whole data. To handle increasing query rates and data amounts, we strive for balanced partitions and balanced read/write load.



A database with two partitions to distribute the data and associated read/write load

Sharding

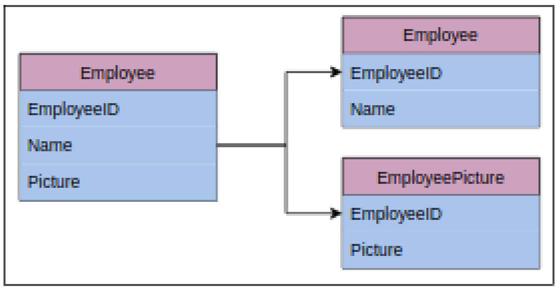
To divide load among multiple nodes, we need to partition the data by a phenomenon known as partitioning or sharding. In this approach, we split a large dataset into smaller chunks of data stored at different nodes on our network.

The partitioning must be balanced so that each partition receives about the same amount of data. If partitioning is unbalanced, the majority of queries will fall into a few partitions. Partitions that are heavily loaded will create a system bottleneck.

1. Vertical Sharding

We can put different tables in various database instances, which might be running on a different physical server. We might break a table into multiple tables so that some columns are in one table while the rest are in the other. We should be careful if there are joins between multiple tables. We may like to keep such tables together on one shard.

vertical sharding is used to increase the speed of data retrieval from a table consisting of columns with very wide text or a binary large object (blob). In this case, the column with large text or a blob is split into a different table.



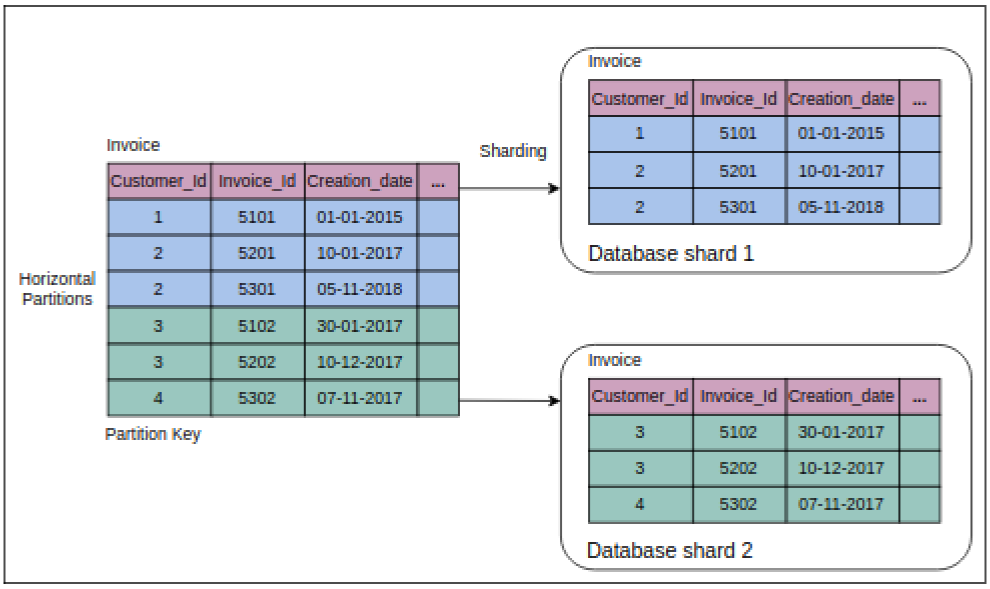
Vertical partitioning

2. Horizontal Sharding

Horizontal sharding or partitioning is used to divide a table into multiple tables by splitting data row-wise. Each partition of the original table distributed over database servers is called a shard.

Key-range based sharding

In the key-range based sharding, each partition is assigned a continuous range of keys.



Invoice

Customer_Id	Invoice_Id	Creation_date	...
1	5101	01-01-2015	
2	5201	10-01-2017	
2	5301	05-11-2018	
3	5102	30-01-2017	
3	5202	10-12-2017	
4	5302	07-11-2017	

Invoice

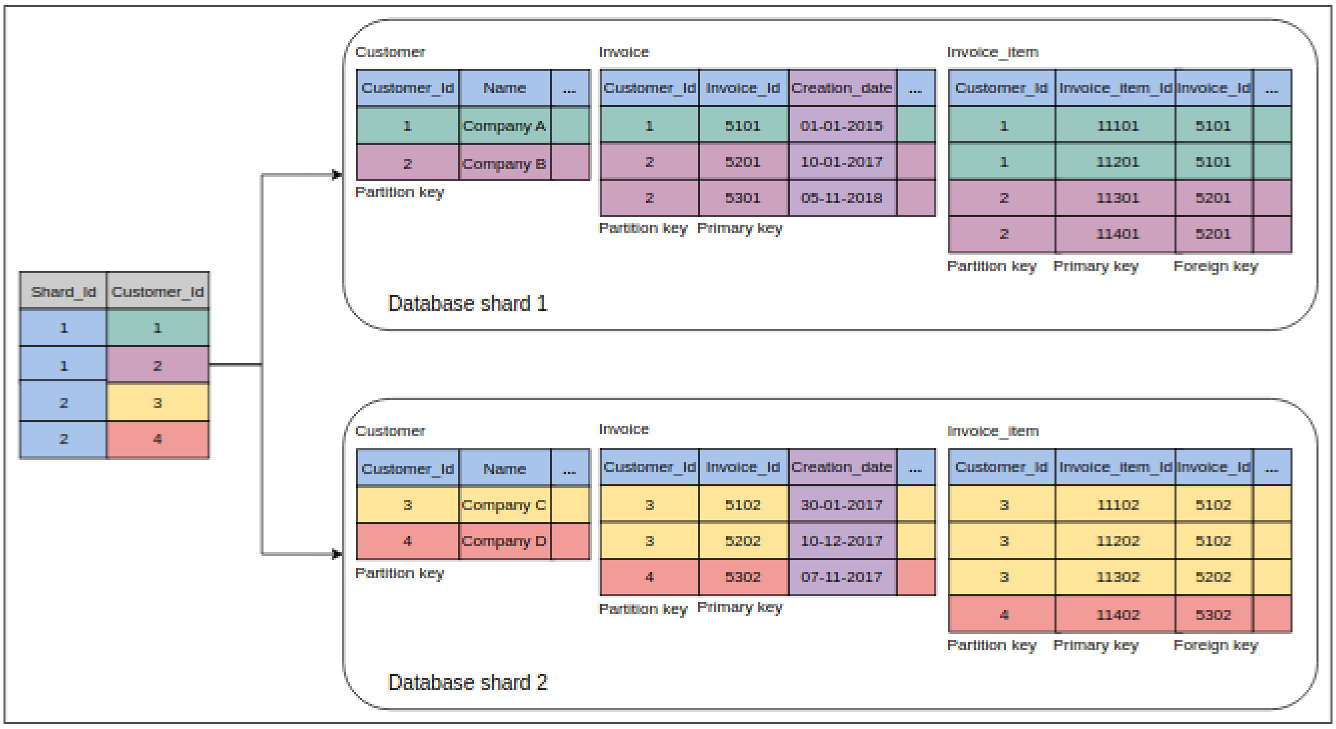
Customer_Id	Invoice_Id	Creation_date	...
1	5101	01-01-2015	
2	5201	10-01-2017	
2	5301	05-11-2018	

Invoice

Customer_Id	Invoice_Id	Creation_date	...
3	5102	30-01-2017	
3	5202	10-12-2017	
4	5302	07-11-2017	

Horizontal partitioning

Sometimes, a database consists of multiple tables bound by foreign key relationships. In such a case, the horizontal partition is performed using the same partition key on all tables in a relation. Tables (or sub-tables) that belong to the same partition key are distributed to one database shard. The following figure shows that several tables with the same partition key are placed in a single database shard:



Shard_Id	Customer_Id
1	1
1	2
2	3
2	4

Database shard 1

Customer

Customer_Id	Name	...
1	Company A	
2	Company B	

Invoice

Customer_Id	Invoice_Id	Creation_date	...
1	5101	01-01-2015	
2	5201	10-01-2017	
2	5301	05-11-2018	

Invoice_item

Customer_Id	Invoice_Item_Id	Invoice_Id	...
1	11101	5101	
1	11201	5101	
2	11301	5201	
2	11401	5201	

Database shard 2

Customer

Customer_Id	Name	...
3	Company C	
4	Company D	

Invoice

Customer_Id	Invoice_Id	Creation_date	...
3	5102	30-01-2017	
3	5202	10-12-2017	
4	5302	07-11-2017	

Invoice_item

Customer_Id	Invoice_Item_Id	Invoice_Id	...
3	11102	5102	
3	11202	5102	
3	11302	5202	
4	11402	5302	

Horizontal partitioning on a set of tables

The basic design techniques used in multi-table sharding are as follows:

- There's a partition key in the Customer mapping table. This table resides on each shard and stores the partition keys used in the shard. Applications create a mapping logic between the partition keys and database shards by reading this table from all shards to make the mapping efficient. Sometimes, applications use advanced algorithms to determine the location of a partition key belonging to a specific shard.

- The partition key column, Customer_Id, is replicated in all other tables as a data isolation point. It has a trade-off between an impact on increased storage and locating the desired shards efficiently. Apart from this, it's helpful for data and workload distribution to different database shards. The data routing logic uses the partition key at the application tier to map queries specified for a database shard.

- Primary keys are unique across all database shards to avoid key collision during data migration among shards and the merging of data in the online analytical processing (OLAP) environment.

- The column Creation_date serves as the data consistency point, with an assumption that the clocks of all nodes are synchronized. This column is used as a criterion for merging data from all database shards into the global view when essential.

Advantages

Using key-range-based sharding method, the range-query-based scheme is easy to implement. We precisely know where (which node, which shard) to look for a specific range of keys.

Range queries can be performed using the partitioning keys, and those can be kept in partitions in sorted order. How exactly such a sorting happens over time as new data comes in is implementation specific.

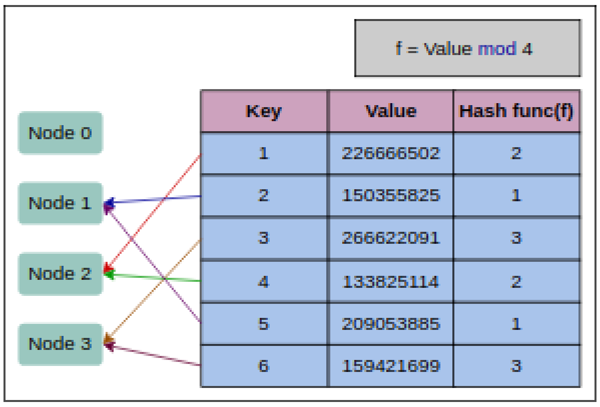
Disadvantages

Range queries can't be performed using keys other than the partitioning key. If keys aren't selected properly, some nodes may have to store more data due to an uneven distribution of the traffic.

Hash-based sharding

Hash-based sharding uses a hash function on an attribute. This hash function produces a hash value that is used to perform partitioning. The main concept is to use a hash function on the key to get a hash value and then mod by the number of partitions. Once we've found an appropriate hash function for keys, we may give each partition a range of hashes (rather than a range of keys). Any key whose hash occurs inside that range will be kept in that partition.

In the illustration below, we use a hash function of Value mode = n. The n is the number of nodes, which is four. We allocate keys to nodes by checking the mod for each key. Keys with a mod value of 2 are allocated to node 2. Keys with a mod value of 1 are allocated to node 1. Keys with a mod value of 3 are allocated to node 3. Because there's no key with a mod value of 0, node 0 is left vacant.



Key	Value	Hash func(f)
1	226666502	2
2	150355825	1
3	266622091	3
4	133825114	2
5	209053885	1
6	159421699	3

Hash-based sharding

Advantages

- Keys are uniformly distributed across the nodes.

Dis-advantages

- We can't perform range queries with this technique. Keys will be spread over all partitions.

Consistent hashing assigns each server or item in a distributed hash table a place on an abstract circle, called a ring, irrespective of the number of servers in the table. This permits servers and objects to scale without compromising the system's overall performance.

Advantages

It's easy to scale horizontally. It increases the throughput and improves the latency of the application.

Disadvantages

Randomly assigning nodes in the ring may cause non-uniform distribution.

3. Re-balance the partitions

Request Routing

1. ZooKeeper

To track changes in the cluster, many distributed data systems need a separate management server like ZooKeeper. Zookeeper keeps track of all the mappings in the network, and each node connects to ZooKeeper for the information. Whenever there’s a change in the partitioning, or a node is added or removed, ZooKeeper gets updated and notifies the routing tier about the change. HBase, Kafka and SolrCloud use ZooKeeper.

Trade-offs in Databases

Both horizontal and vertical sharding involve adding resources to our computing infrastructure. Our business stakeholders must decide which is suitable for our organization. We must scale our resources accordingly for our organization and business to grow, to prevent downtime, and to reduce latency. We can scale these resources through a combination of adjustments to CPU, physical memory requirements, hard disk adjustments, and network bandwidth.

Centralized database

Advantages

- Data maintenance, such as updating and taking backups of a centralized database, is easy.
- Centralized databases provide stronger consistency and ACID transactions than distributed databases.
- Centralized databases provide a much simpler programming model for the end programmers as compared to distributed databases.
- It’s more efficient for businesses that have a small amount of data to store that can reside on a single node.

Disadvantages

- A centralized database can slow down, causing high latency for end users, when the number of queries per second accessing the centralized database is approaching single-node limits.

- A centralized database has a single point of failure. Because of this, its probability of not being accessible is much higher.

Distributed database

Advantages

- It’s fast and easy to access data in a distributed database because data is retrieved from the nearest database shard or the one frequently used.
- Data with different levels of distribution transparency can be stored in separate places.
- Intensive transactions consisting of queries can be divided into multiple optimized subqueries, which can be processed in a parallel fashion.

Disadvantages

- Sometimes, data is required from multiple sites, which takes more time than expected.

- Relations are partitioned vertically or horizontally among different nodes. Therefore, operations such as joins need to reconstruct complete relations by carefully fetching data. These operations can become much more expensive and complex.
- It’s difficult to maintain consistency of data across sites in the distributed database, and it requires extra measures.
- Updates and backups in distributed databases take time to synchronize data.

Query optimization and processing speed in a distributed database

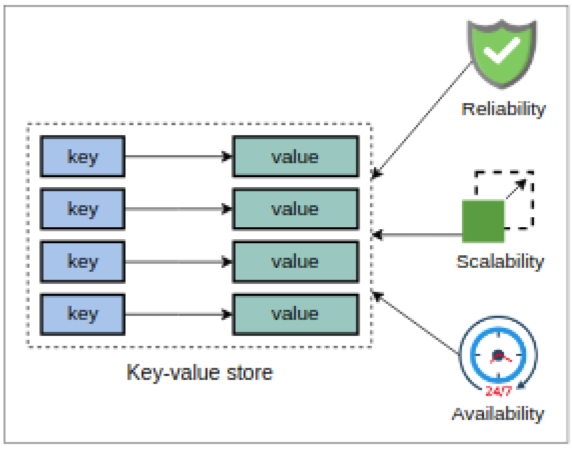
<<< TODO >>>

Parameters assumption

Possible approaches

4. Key-Value Store

Key-value stores are distributed hash tables (DHTs). A key is generated by the hash function and should be unique. In a key-value store, a key binds to a specific value and doesn’t assume anything about the structure of the value. A value can be a blob, image, server name, or anything the user wants to store against a unique key



Key-value store

It’s preferred to keep the size of value relatively smaller (KB to MB). We can put large data in the blob store and put links to that data in the value field. Key-value stores are useful in many situations, such as storing user sessions in a web application and building NoSQL databases.

Key-Value system design can be divided into the following:-

- ***Design of a key-value store***: Define the requirements of a key-value store and design the API.
- ***Ensure Scalability and Replication***: Achieve scalability using consistent hashing and replicate the partitioned data.
- ***Versioning Data and Achieving Configurability***: Resolve conflicts that occur due to changes made by more than one entity, and we'll make our system more configurable for different use cases.
- ***Enable Fault Tolerance and Failure Detection***: Make a key-value store fault-tolerant and to detect failures in the system.

Design of a key-value store

Requirements

Functional Requirements Typical key-value stores are expected to offer functions such as *get* and *put*. However, below are requirements for the system

- **Configurable service:** Some applications might have a tendency to trade strong consistency for higher availability. We need to provide a configurable service so that different applications could use a range of consistency models. We need tight control over the trade-offs between availability, consistency, cost-effectiveness, and performance.

  Note: Such configurations can only be performed when instantiating a new key-value store instance and cannot be changed dynamically when the system is operational.

- **Ability to always write** (when we picked "A" over "C" in the context of CAP): The applications should always have the ability to write into the key-value storage. If the user wants strong consistency, this requirement might not always be fulfilled due to the implications of the CAP theorem.

  Note: The context of the problem determines what will be classified as a functional requirement and what will be classified as non-functional. For example, the ability to always write (high availability) is a functional requirement for Amazon’s shopping cart application, while in other cases, high availability may be considered a non-functional requirement. Drawing inspiration from Amazon’s Dynamo key-value store, we can categorize the ability to always write as a functional requirement.

- **Hardware heterogeneity**: We want to add new servers with different and higher capacities, seamlessly, to our cluster without changing or upgrading existing servers. Our system should be able to accommodate and leverage different capacity servers, ensuring correct core functionality (get and put data) while balancing the workload distribution according to each server’s capacity. This calls for a peer-to-peer design with no distinguished nodes.

Non Functional Requirements The non-functional requirements are as follows:

- **Scalability**: Key-value stores should run on tens of thousands of servers distributed across the globe. Incremental scalability is highly desirable. We should add or remove the servers as needed with minimal to no disruption to the service availability. Moreover, our system should be able to handle an enormous number of users of the key-value store.

- **Fault tolerance**: The key-value store should operate uninterrupted despite failures in servers or their components.

Assumptions

We’ll assume the following to keep our design simple:

- The data centers hosting the service are trusted (non-hostile).
- All the required authentication and authorization are already completed.
- User requests and responses are relayed over HTTPS.

API design

The ***get*** function - We return the associated value on the basis of the parameter key. When data is replicated, it locates the object replica associated with a specific key that's hidden from the end user. It's done by the system if the store is configured with a weaker data consistency model. For example, in eventual consistency, there might be more than one value returned against a key.

Parameter	Description
Key	It's the **key** against which we want to get **value**

The ***put*** function - It stores the value associated with the key. The system automatically determines where data should be placed. Additionally, the system often keeps metadata about the stored object. Such metadata can include the version of the object.

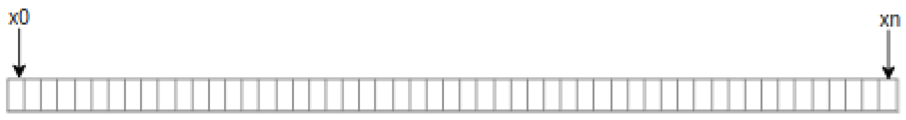
Parameter	Description
Key	It's the **key** against which we have to store **value**
value	It's the object to be stored against the **key**.

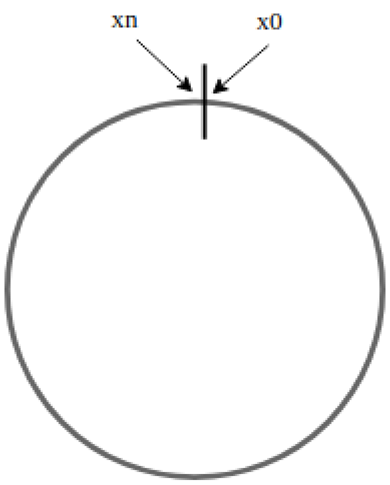
Data Type - The key is often a primary key in a key-value store, while the value can be any arbitrary binary data. **Note**: Dynamo uses MD5 hashes on the key to generate a 128-bit identifier. These identifiers help the system determine which server node will be responsible for this specific key.

Ensure Scalability

Consistent Hashing is a special kind of hashing such that when a hash table is re-sized and consistent hashing is used, only k/n keys need to be remapped on average, where k is the number of keys, and n is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped

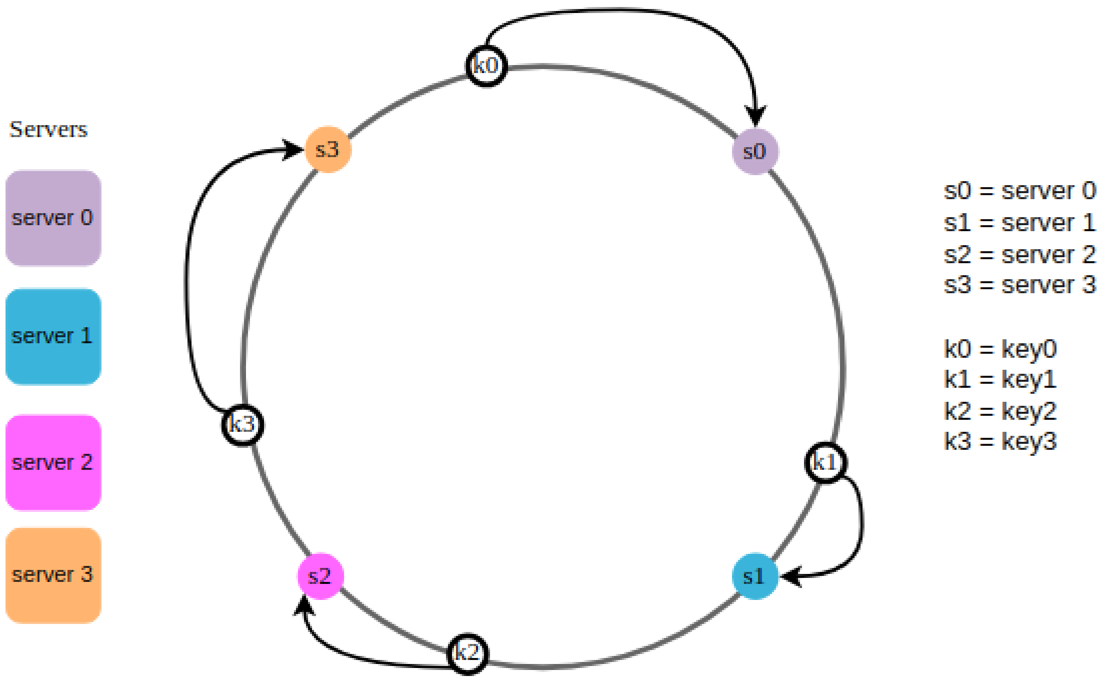
Hash space and hash ring Now we understand the definition of consistent hashing, let us find out how it works. Assume SHA-1 is used as the hash function f, and the output range of the hash function is: x0, x1, x2, x3, …, xn. In cryptography, SHA-1's hash space goes from 0 to 2^160 - 1. That means x0 corresponds to 0, xn corresponds to 2^160 – 1, and all the other hash values in the middle fall between 0 and 2^160 - 1. Figure 3 shows the hash space.

Hash space 

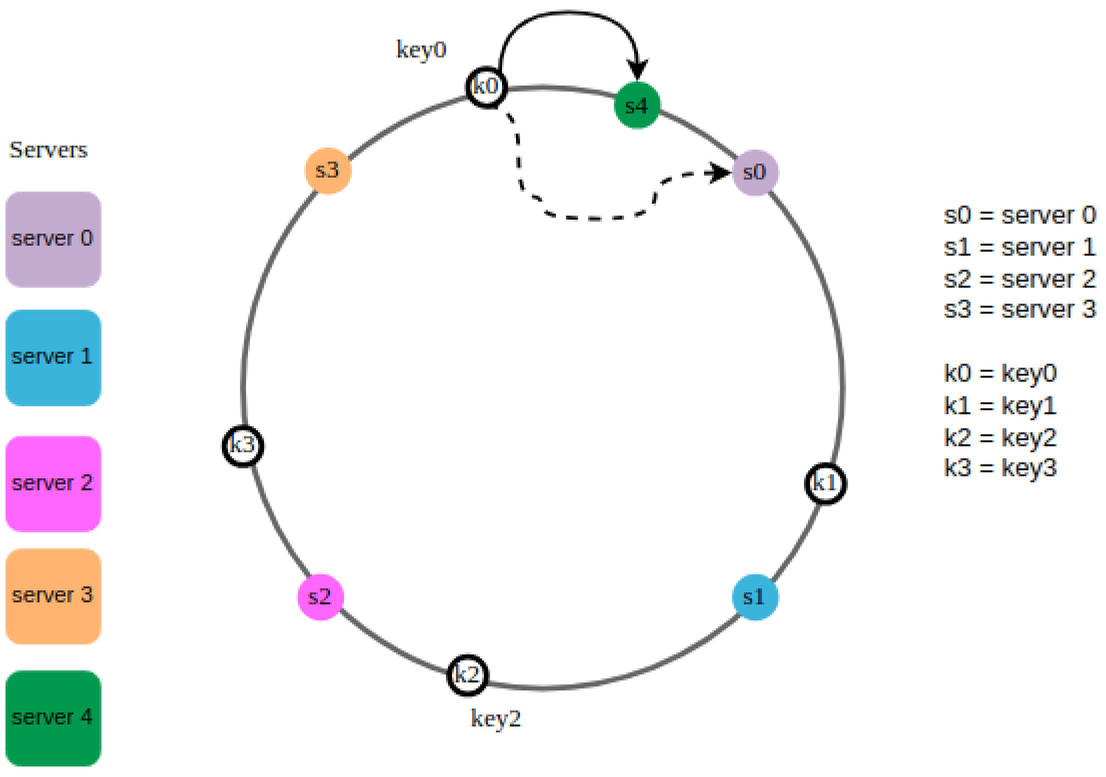


Hash Ring

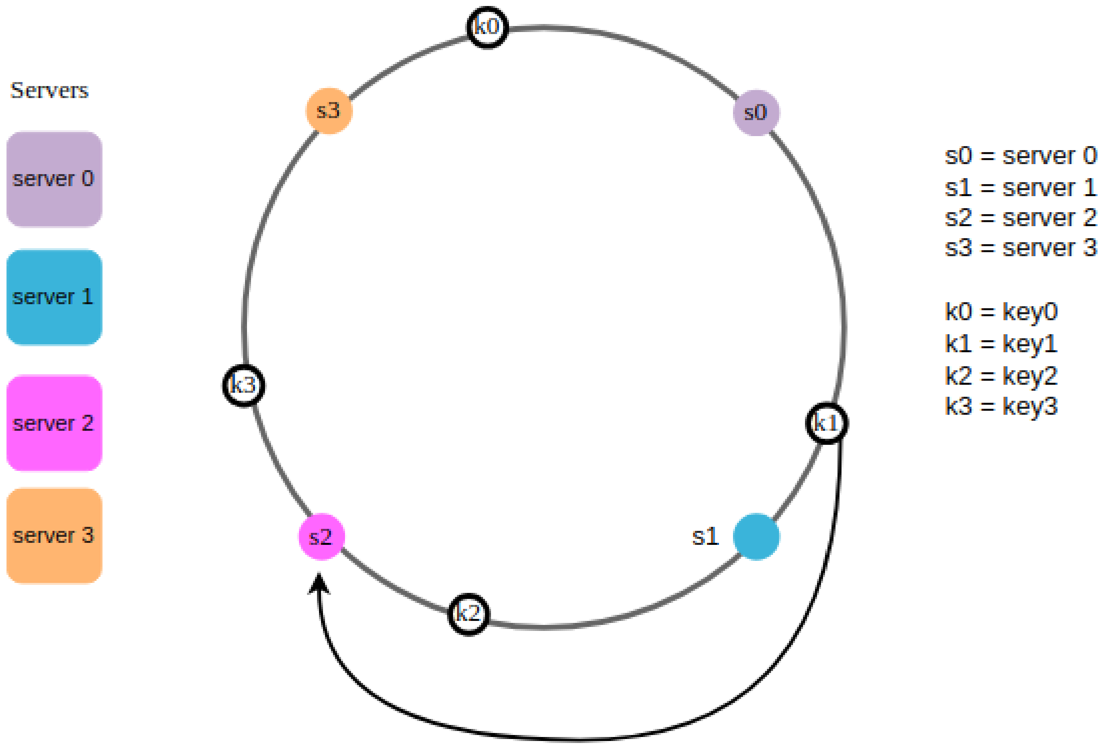
Server lookup - To determine which server a key is stored on, we go clockwise from the key position on the ring until a server is found. Below explains this process. Going clockwise, key0 is stored on server 0; key1 is stored on server 1; key2 is stored on server 2 and key3 is stored on server 3.



Add a server - Using the logic described above, adding a new server will only require redistribution of a fraction of keys. Below, after a new server 4 is added, only key0 needs to be redistributed. k1, k2, and k3 remain on the same servers. Let us take a close look at the logic. Before server 4 is added, key0 is stored on server 0. Now, key0 will be stored on server 4 because server 4 is the first server it encounters by going clockwise from key0's position on the ring. The other keys are not redistributed based on consistent hashing algorithm.



Remove a server - When a server is removed, only a small fraction of keys requires redistribution with consistent hashing. Below diagram, when server 1 is removed, only key1 must be remapped to server 2. The rest of the keys are unaffected.



Two issues in the basic approach

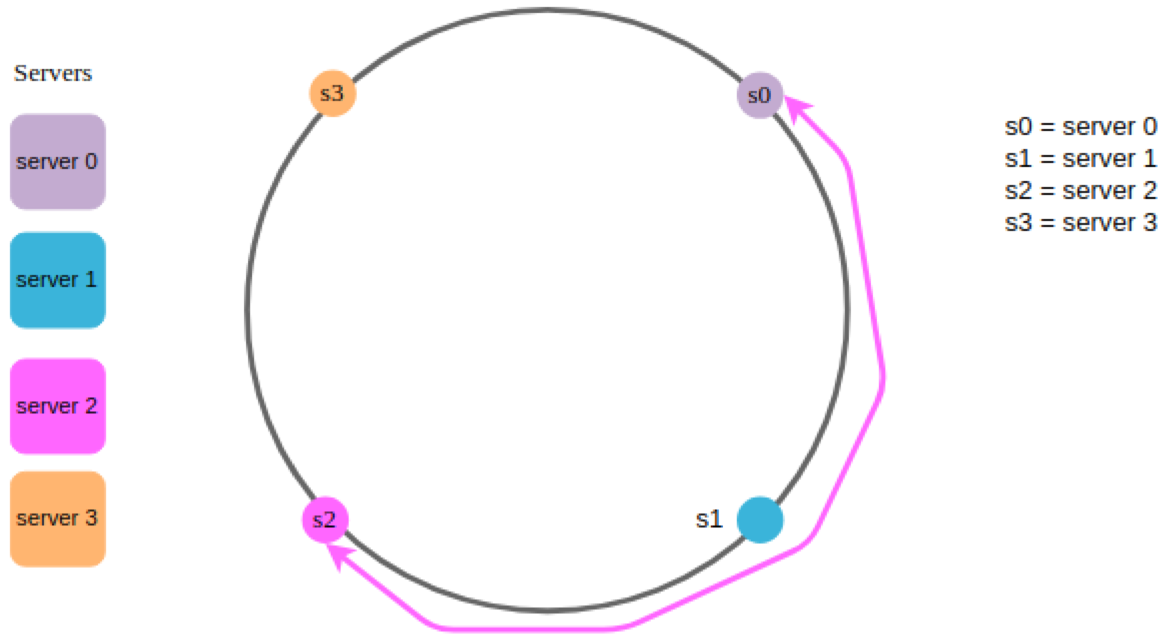
Basic steps :-

- Map servers and keys on to the ring using a uniformly distributed hash function.

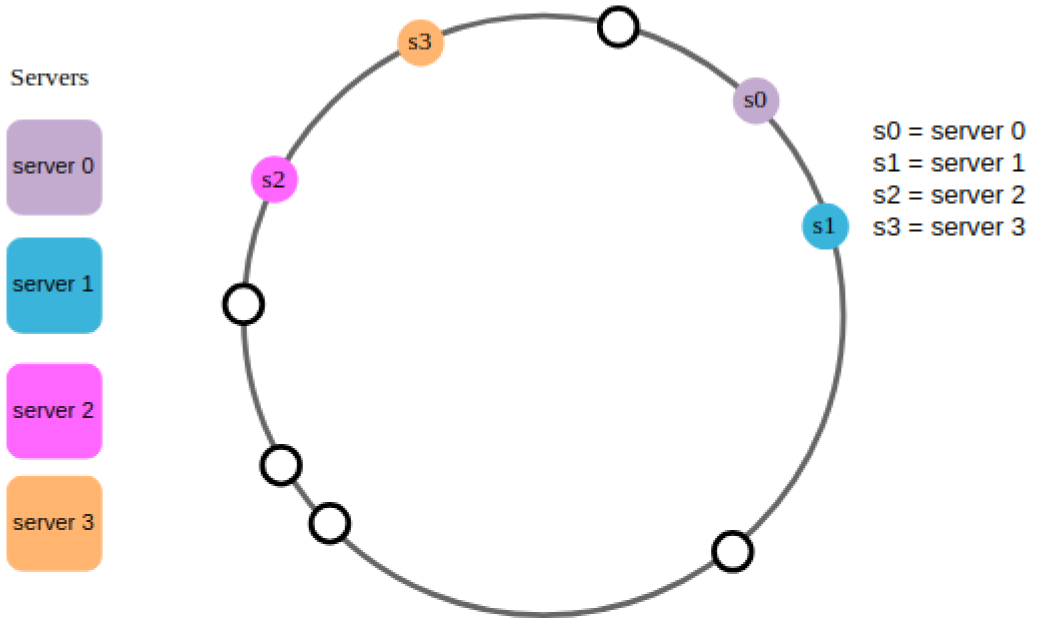
- To find out which server a key is mapped to, go clockwise from the key position until the first server on the ring is found.

Problems :-

1. It is impossible to keep the same size of partitions on the ring for all servers considering a server can be added or removed. A partition is the hash space between adjacent servers. It is possible that the size of the partitions on the ring assigned to each server is very small or fairly large. Below, if s1 is removed, s2's partition (highlighted with the bidirectional arrows) is twice as large as s0 and s3's partition.



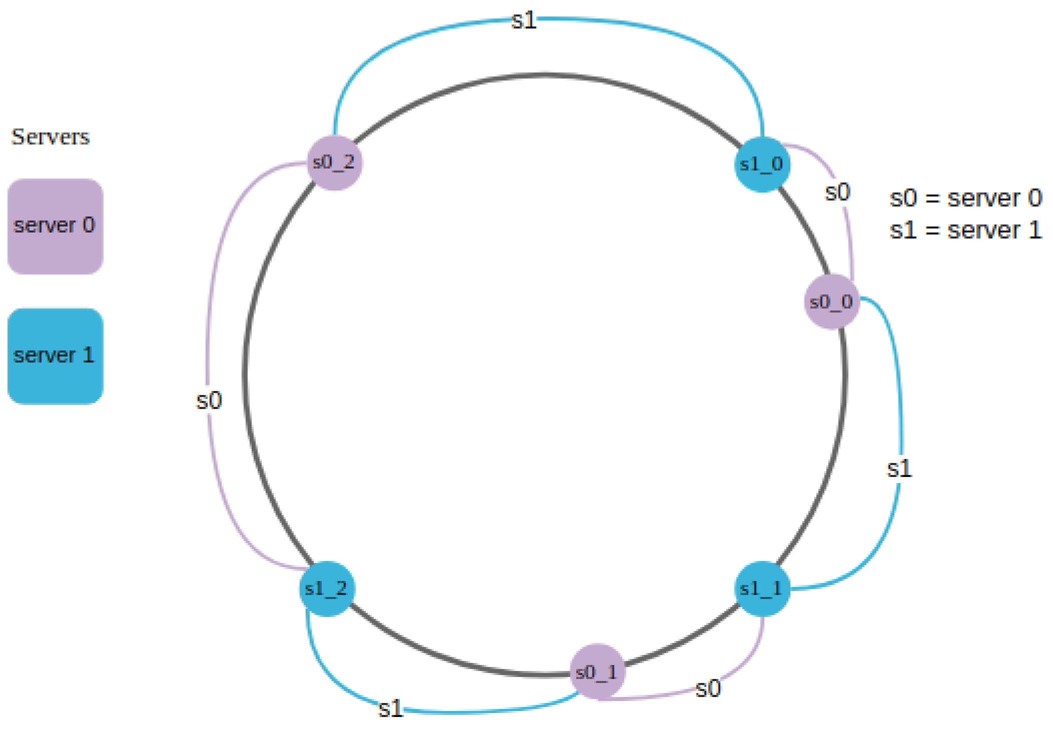
2. It is possible to have a non-uniform key distribution on the ring. For instance, if servers are mapped to positions listed below, most of the keys are stored on server 2. However, server 1 and server 3 have no data.



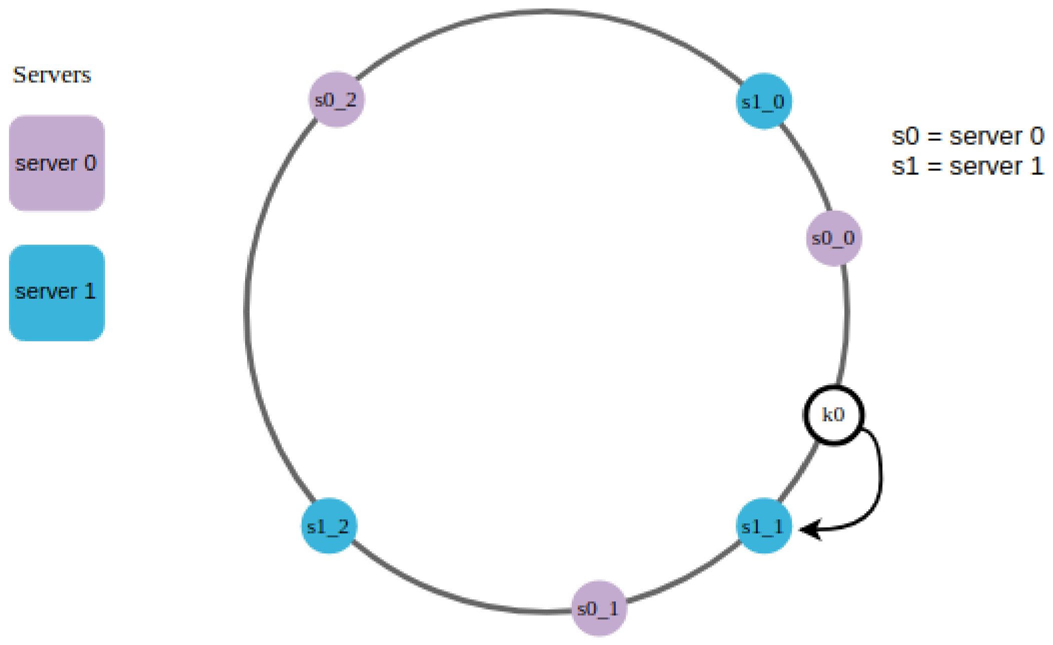
A technique called virtual nodes or replicas is used to solve these problems.

Virtual nodes

A virtual node refers to the real node, and each server is represented by multiple virtual nodes on the ring. Below, both server 0 and server 1 have 3 virtual nodes. The 3 is arbitrarily chosen; and in real-world systems, the number of virtual nodes is much larger. Instead of using s0, we have s0_0, s0_1, and s0_2 to represent _server 0 on the ring. Similarly, s1_0, s1_1, and s1_2 represent server 1 on the ring. With virtual nodes, each server is responsible for multiple partitions. Partitions (edges) with label s0 are managed by server 0. On the other hand, partitions with label s1 are managed by server 1.



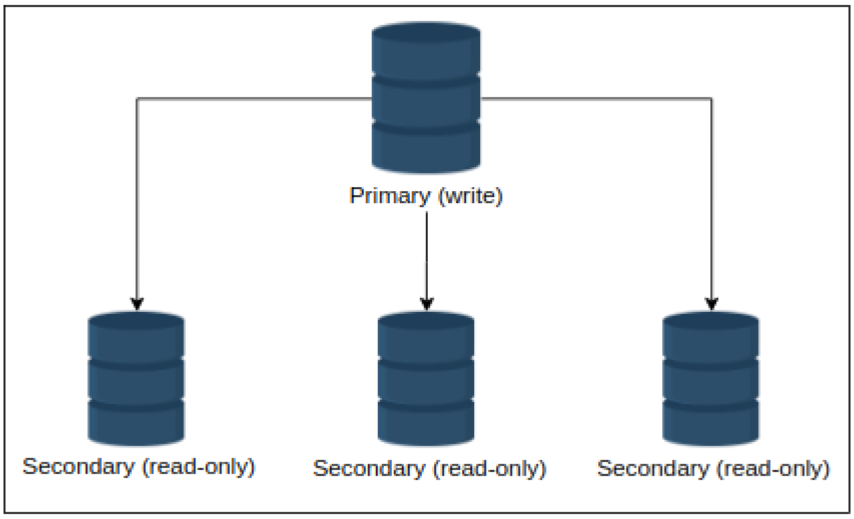
To find which server a key is stored on, we go clockwise from the key's location and find the first virtual node encountered on the ring. Below, to find out which server k0 is stored on, we go clockwise from k0's location and find virtual node s1_1, which refers to server 1.



As the number of virtual nodes increases, the distribution of keys becomes more balanced. This is because the standard deviation gets smaller with more virtual nodes, leading to balanced data distribution. Standard deviation measures how data are spread out. The standard deviation will be smaller when we increase the number of virtual nodes. However, more spaces are needed to store data about virtual nodes. This is a tradeoff, and we can tune the number of virtual nodes to fit our system requirements.

Ensure Replication

Primary-secondary approach - one of the storage areas is primary, and other storage areas are secondary. The secondary replicates its data from the primary. The primary serves the write requests while the secondary serves read requests. After writing, there's a lag for replication. Moreover, if the primary goes down, we can't write into the storage, and it becomes a single point of failure.



Primary-secondary approach

peer-to-peer - all involved storage areas are primary, and they replicate the data to stay updated. Both read and write are allowed on all nodes. Usually, it's inefficient and costly to replicate in all n nodes. Instead, three or five is a common choice for the number of storage nodes to be replicated. Refer to topic on ***Quorum*** for more details.

In the context of the CAP theorem, key-value stores can either be consistent or be available when there are network partitions. For key-value stores, we prefer availability over consistency. It means if the two storage nodes lost connection for replication, they would keep on handling the requests sent to them, and when the connection is restored, they'll sync up. In the disconnected phase, it's highly possible for the nodes to be inconsistent. So, we need to resolve such conflicts.

Versioning Data and Archiving

Data versioning - When network partitions and node failures occur during an update, an object's version history might become fragmented. As a result, it requires a reconciliation effort on the part of the system. It's necessary to build a way that explicitly accepts the potential of several copies of the same data so that we can avoid the loss of any updates. It's critical to realize that some failure scenarios can lead to multiple copies of the same data in the system. So, these copies might be the same or divergent. Resolving the conflicts among these divergent histories is essential and critical for consistency purposes.

Scenario -

```
1. Two nodes replicating their data while handling requests.
2. The network connection between two nodes breaks.
3. Both nodes continue handling the requests.
4. The connection is restored, but the data in both nodes isn't necessarily
the same.
```

To handle inconsistency, we need to maintain causality between the events. We can do this using the timestamps and update all conflicting values with the value of the latest request. But time isn't reliable in a distributed system, so we can't use it as a deciding factor.

Another approach to maintaining causality effectively is by using vector clocks. A vector clock is a list of (node, counter) pairs. There's a single vector clock for every version of an object. If two objects have different vector clocks, we're able to tell whether they're causally related or not (more on this in a bit). Unless one of the two changes is reconciled, the two are deemed at odds.

Enable Fault Tolerance and Failure Detection

Handle temporary failures - Typically, distributed systems use a quorum-based approach to handle failures. A quorum is the minimum number of votes required for a distributed transaction to proceed with an operation. If a server is part of the consensus and is down, then we can't perform the required operation. It affects the availability and durability of our system.

We'll use a sloppy quorum instead of strict quorum membership. Usually, a leader manages the communication among the participants of the consensus. The participants send an acknowledgment after committing a successful write. Upon receiving these acknowledgments, the leader responds to the client. However, the drawback is that the participants are easily affected by the network outage. If the leader is temporarily down and the participants can't reach it, they declare the leader dead. Now, a new leader has to be reelected. Such frequent elections have a negative impact on performance because the system spends more time picking a leader than accomplishing any actual work.

In the sloppy quorum, the first n healthy nodes from the preference list handle all read and write operations. The n healthy nodes may not always be the first n nodes discovered when moving clockwise in the consistent hash ring.

Handle permanent failure - In the event of permanent failures of nodes, we should keep our replicas synchronized to make our system more durable. We need to speed up the detection of inconsistencies between replicas and reduce the quantity of transferred data. We'll use ***Merkle trees*** for that.

Conclusion - A key-value store provides flexibility and allows us to scale the applications that have unstructured data. Web applications can use key-value stores to store information about a user's session and preferences. When using a user key, all the data is accessible, and key-value stores are ideal for rapid reads and write operations. Key-value stores can be used to power real-time recommendations and advertising because the stores can swiftly access and present fresh recommendations.

5. Content Delivery Network

System Design: The Content Delivery Network (CDN)

Introduction to CDN

In-depth Investigation of CDN:

Evaluation of CDN Design

6. Sequencer

System Design Sequencer

Design of a Unique ID Generator

Requirements for unique identifiers -

- **Uniqueness**: We need to assign unique identifiers to different events for identification purposes.
- **Scalability**: The ID generation system should generate at least a billion unique IDs per day.
- **Availability**: Since multiple events happen even at the level of nanoseconds, our system should generate IDs for all the events that occur. 64-bit numeric ID: We restrict the length to 64 bits because this bit size is enough for many years in the future. Let's calculate the number of years after which our ID range will wrap around.

Total numbers available = 2^{64} = 1.8446744 x 10^{19}

Estimated number of events per day = 1 billion = 10^9

Number of events per year = 365 billion = 365×10^9

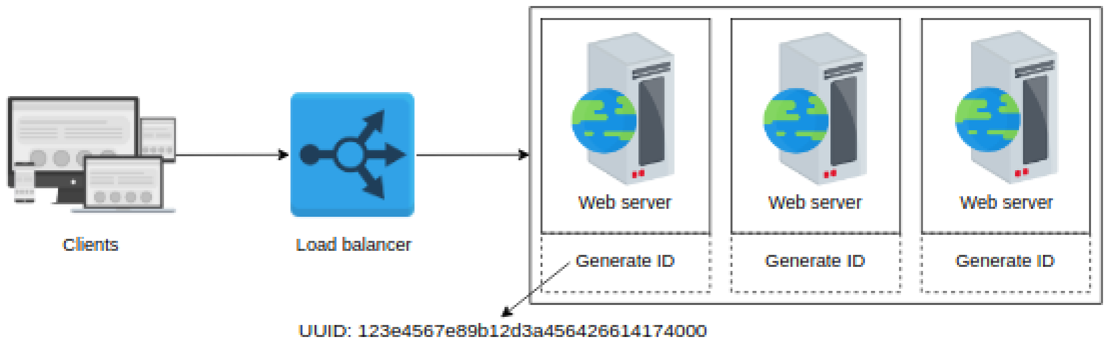
Number of years to deplete identifier range = $\frac{2^{64}}{365 \times 10^9}$ = 50,539,024.8595 years

64 bits should be enough for a unique ID length considering these calculations.

First solution: UUID

A straw man solution for our design uses UUIDs (universally unique IDs). This is a 128-bit number and it looks like ***123e4567e89b12d3a456426614174000*** in hexadecimal. It gives us about 10^38 numbers. UUIDs have different versions. We opt for version 4, which generates a pseudorandom number.

Each server can generate its own ID and assign the ID to its respective event. No coordination is needed for UUID since it's independent of the server. Scaling up and down is easy with UUID, and this system is also highly available. Furthermore, it has a low probability of collisions. The design for this approach is given below:



Generating a unique ID using the UUID approach

Cons - Using 128-bit numbers as primary keys makes the primary-key indexing slower, which results in slow inserts. A workaround might be to interpret an ID as a hex string instead of a number. However, non-numeric identifiers might not be suitable for many use cases. The ID isn't of 64-bit size. Moreover, there's a chance of duplication. Although this chance is minimal, we can't claim UUID to be deterministically unique. Additionally, UUIDs given to clients over time might not be monotonically increasing. The following table summarizes the requirements we have fulfilled using UUID:

	Unique	Scalable	Available	64-bit numeric ID
Using UUID	No	Yes	Yes	No

Second solution: using a database

Centralized Database: Consider a central database that provides a current ID and then increments the value by one. We can use the current ID as a unique identifier for our events.

Cons: Single point of failure.

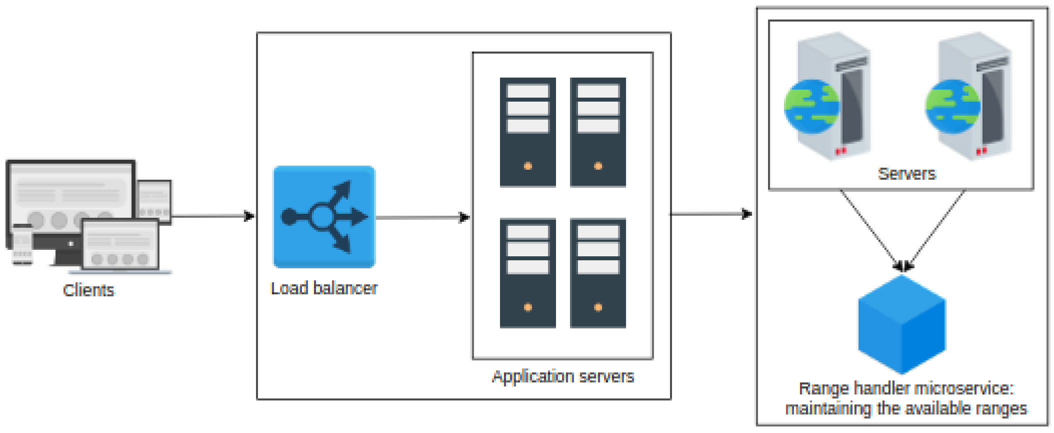
Third solution: using a range handler

Using a range handler: We can use ranges in a central server. Suppose we have multiple ranges for one to two billion, such as 1 to 1,000,000; 1,000,001 to 2,000,000; and so on. In such a case, a central microservice can provide a range to a server upon request.

Any server can claim a range when it needs it for the first time or if it runs out of the range. Suppose a server has a range, and now it keeps the start of the range in a local variable. Whenever a request for an ID is made, it provides the local variable value to the requestor and increments the value by one.

This resolves the problem of the duplication of user IDs. Each application server can respond to requests concurrently. We can add a load balancer over a set of servers to mitigate the load of requests.

We use a microservice called **range handler** that keeps a record of all the taken and available ranges. The status of each range can determine if a range is available or not. The state—that is, which server has what range assigned to it—can be saved on a replicated storage.


Clients
Load balancer
Application servers
Servers
Range handler microservice: maintaining the available ranges

Design of the range handler microservice

This microservice can become a single point of failure, but a failover server acts as the savior in that case. The failover server hands out ranges when the main server is down. We can recover the state of available and unavailable ranges from the latest checkpoint of the replicated store.

Pros - This system is scalable, available, and yields user IDs that have no duplicates. Moreover, we can maintain this range in 64 bits, which is numeric.

Cons - We lose a significant range when a server dies and can only provide a new range once it's live again. We can overcome this shortcoming by allocating shorter ranges to the servers, although ranges should be large enough to serve identifiers for a while.

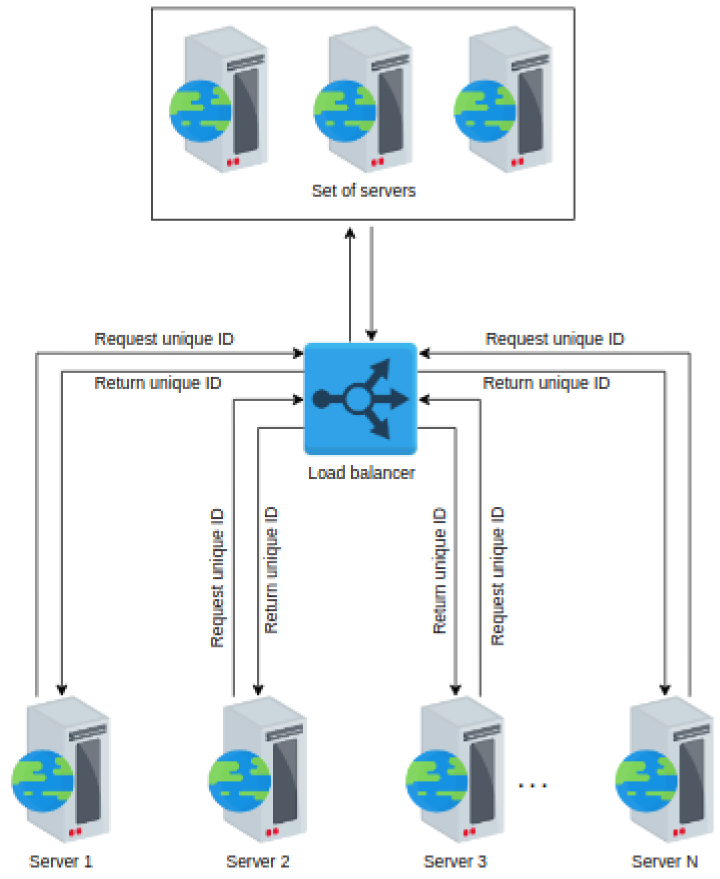
Unique IDs with Causality

UNIX time stamps

NIX time stamps are granular to the millisecond and can be used to distinguish different events. We have an ID-generating server that can generate one ID in a single millisecond. Any request to generate a unique ID is routed to that server, which returns a time stamp and then returns a unique ID. The ability to generate an ID in milliseconds allows us to generate a thousand identifiers per second

This means we can get 24 * 60 * 60 * 1000 = 86400000IDs in a day.

Our system works well with generating IDs, but it poses a crucial problem. The ID-generating server is a single point of failure (SPOF), and we need to handle it. To cater to SPOF, we can add more servers. Each server generates a unique ID for every millisecond. To make the overall identifier unique across the system, we attach the server ID with the UNIX time stamp. Then, we add a load balancer to distribute the traffic more efficiently. The design of a unique ID generator using a UNIX time stamps is given below:



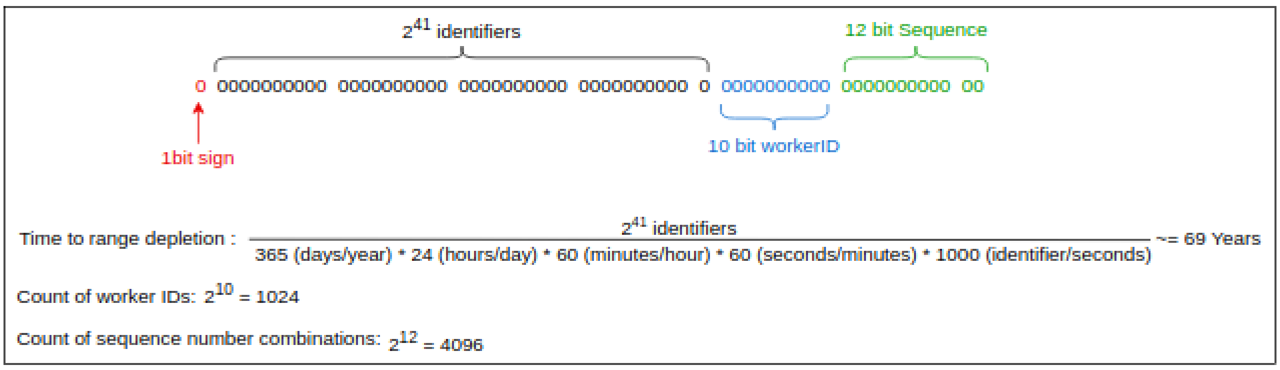
Using the time stamp as an ID

pros - This approach is simple, scalable, and easy to implement. It also enables multiple servers to handle concurrent requests.

cons - For two concurrent events, the same time stamp is returned and the same ID can be assigned to them. This way, the IDs are no longer unique.

Twitter Snowflake

We can use some bits out of our targetted 64 bits for storing time and the remaining for other information. An overview of division is below:



Time to range depletion : $\frac{2^{41} \text{ identifiers}}{365 \text{ (days/year)} * 24 \text{ (hours/day)} * 60 \text{ (minutes/hour)} * 60 \text{ (seconds/minutes)} * 1000 \text{ (identifier/seconds)}}$ ~= 69 Years

Count of worker IDs: $2^{10} = 1024$

Count of sequence number combinations: $2^{12} = 4096$

Overview of the division of bits in Twitter Snowflake

• **Sign bit**: A single bit is assigned as a sign bit, and its value will always be zero. It makes the overall number positive. Doing so helps to ensure that any programming environment using these identifiers interprets them as positive integers.

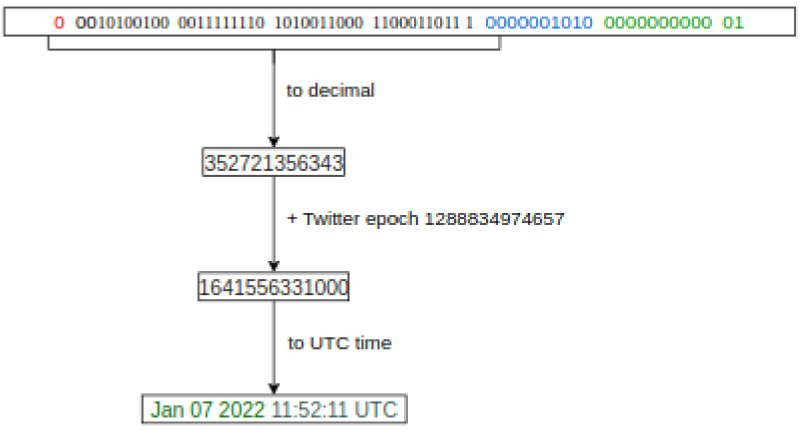
• **Time stamp**: 41 bits are assigned for milliseconds. The Twitter Snowflake default epoch will be used. Its value is 1288834974657, which is equivalent to November 4, 2010, 01:42:54 UTC. We can initiate our own epoch when our system will be deployed, say January 1, 2022, at 12 midnight can be the start of our epoch from zero. The maximum time to deplete this range is shown below:

Time to range depletion = 2^41/(365*2460*601000) ~= 60 years

The above calculations give us 69 years before we need a new algorithm to generate IDs. As we saw earlier, if we can generate 1,000 identifiers per second, we aren't able to get our target of a billion identifiers per day. Though now, in the Snowflake proposal, we have ample identifiers available when we utilize worker ID and machine local sequence numbers.

• **Worker number**: The worker number is 10 bits. It gives us 2^10 = 1,024 worker IDs. The server creating the unique ID for its events will attach its ID.

• **Sequence number**: The sequence number is 12 bits. For every ID generated on the server, the sequence number is incremented by one. It gives us **2^12 = 4,096 unique sequence numbers**. We'll reset it to zero when it reaches 4,096. This number adds a layer to avoid duplication.



Convert milliseconds to UTC

Pros - Twitter Snowflake uses the time stamp as the first component. Therefore, they're time sortable. The ID generator is highly available as well

Cons - IDs generated in a dead period are a problem. The dead period is when no request for generating an ID is made to the server. These IDs will be wasted since they take up identifier space. The unique range possible will deplete earlier than expected and create gaps in our global set of user IDs.

TrueTime API

Google's TrueTime API in Spanner is an interesting option. Instead of a particular time stamp, it reports an interval of time. When asking for the current time, we get back two values: the earliest and latest ones. These are the earliest possible and latest possible time stamps.

Based on its uncertainty calculations, the clock knows that the actual current time is somewhere within that interval. The width of the interval depends, among other things, on how long it has been since the local quartz clock was last synchronized with a more accurate clock source.

Google deploys a GPS receiver or atomic clock in each data center, and clocks are synchronized within about 7 ms. This allows Spanner to keep the clock uncertainty to a minimum. The uncertainty of the interval is represented as epsilon.

Summary

- We want to avoid duplicate identifiers. Consider what will happen if duplicate payment or purchase orders are generated.
- UUIDs provide probabilistic guarantees about the keys' non-collision. Deterministically getting non-collision guarantees might need consensus among different distributed entities or stores and read from the replicated store.
- As key length becomes large, it often causes slower tuple updates in a database. Therefore, identifiers should be big enough but not too big.
- Often, it's desirable that no one is able to guess the next ID. Otherwise, undesirable data leaks can happen, and the organization's competitors may learn how many orders were processed in a day by simply looking at order IDs. Adding a few random numbers to the bits of the identifier make it hard to guess, although this comes at a performance cost.
- We can use simple counters for generating unique IDs if we don't want to relate ID to time. Fetching time stamps is slower than simple counters.
- Fetching time stamps is slower than simple counters, though this requires that we store generated IDs persistently. The counter needs to be stored in the database. Storage comes with its own issues. These include multiple concurrent writes becoming overwhelming for the database and the database being the single point of failure.
- For some distributed databases, such as Spanner, it can hurt to generate monotonically increasing or decreasing IDs. Google reports the following: "In fact, using monotonically increasing (or decreasing) values as row keys does not follow best practices in Spanner because it creates hotspots in the database, leading to a reduction in performance."

7. Distributed Monitoring

Distributed Monitoring

A good monitoring system needs to clearly define what to measure and in what units (metrics). The monitoring system also needs to define threshold values of all metrics and the ability to inform appropriate stakeholders (alerts) when values are out of acceptable ranges.

Monitoring systems that collect measurements, show data, and send warnings when something appears wrong are helpful for the support team

Metrics - Metrics objectively define what we should measure and what units will be appropriate. Metric values provide an insight into the system at any point in time We need to collect values of metrics with minimal performance penalty.

Populating the Metrics - The metrics should be logically centralized for global monitoring and alerting purposes. Fetching metrics is crucial to the monitoring system. Metrics can either be pushed or pulled into a monitoring system, depending on the preference of the user.

In **pull strategy**, each monitored server merely needs to store the metrics in memory and send them to an exposed endpoint. The exposed endpoint allows the monitoring application to fetch the metrics itself. Servers sending too much data or sending data too frequently can't overload the monitoring system. The monitoring system will pull data as per its own schedule.

In other situations, though, **pushing strategy** may be beneficial, such as when a firewall prevents the monitoring system from accessing the server directly. The monitoring system has the ability to adjust a global configuration about the data to be collected and the interval at which servers and switches should push the data.

Persist the data - A centralized in-memory metrics repository may be all that's needed. However, for a large data center with millions of things to monitor, there will be an enormous amount of data to store, and a time-series database can help in this regard.

Time-series databases help maintain durability, which is an important factor. Without a historical view of events in a monitoring system, it isn't very useful. Samples having a value of time stamp are stored in chronological sequence. So, a whole metric's timeline can be shown in the form of a time series.

Application Metrics - We may need to add code or APIs to expose metrics we care about for other components, notably our own applications. We embed logging or monitoring code in our applications, called code instrumentation, to collect information of interest.

Alerting - Alerting is the part of a monitoring system that responds to changes in metric values and takes action. There are two components to an alert definition: a metrics-based condition or threshold, and an action to take when the values fall outside the permitted range.

Monitor Service-side errors

Requirements for -

- Monitor critical local processes on a server for crashes.
- Monitor any anomalies in the use of CPU/memory/disk/network bandwidth by a process on a server.
- Monitor overall server health, such as CPU, memory, disk, network bandwidth, average load, and so on.
- Monitor hardware component faults on a server, such as memory failures, failing or slowing disk, and so on.
- Monitor the server's ability to reach out-of-server critical services, such as network file systems and so on.
- Monitor all network switches, load balancers, and any other specialized hardware inside a data center.

- Monitor power consumption at the server, rack, and data center levels.
- Monitor any power events on the servers, racks, and data center.
- Monitor routing information and DNS for external clients.
- Monitor network links and paths' latency inside and across the data centers.
- Monitor network status at the peering points.
- Monitor overall service health that might span multiple data centers—for example, a CDN and its performance.

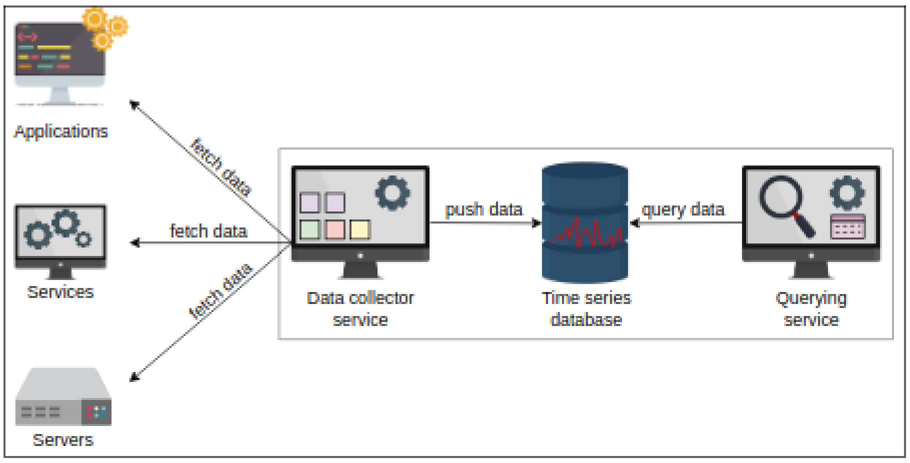
System Design

High-level design

Storage: A time-series database stores metrics data, such as the current CPU use or the number of exceptions in an application.

Data collector service: This fetches the relevant data from each service and saves it in the storage.

Querying service: This is an API that can query on the time-series database and return the relevant information.



High-level design of a monitoring system

Detailed Design of a Monitoring System

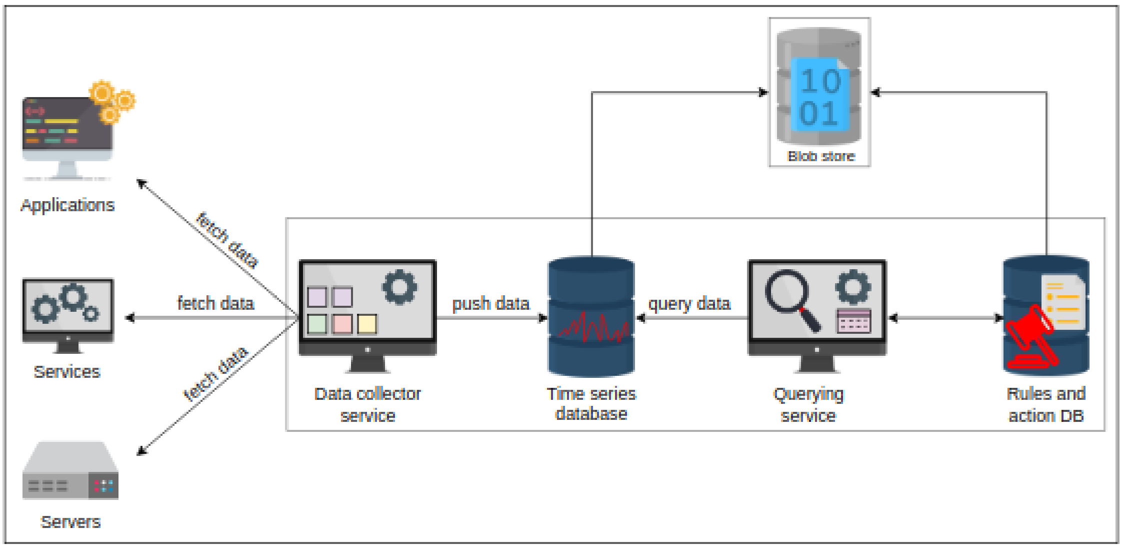
Let's discuss the core components of our monitoring system, identify the shortcomings of our design, and improve the design to fulfill our requirements.

Storage We'll use time-series databases to save the data locally on the server where our monitoring service is running. Then, we'll integrate it with a separate storage node. We'll use blob storage to store our metrics.

We need to store metrics and know which action to perform if a metric has reached a particular value. For example, if CPU usage exceeds 90%, we generate an alert to the end user so the alert receiver can do take the necessary steps, such as allocate more resources to scale. For this purpose, we need another storage area that will contain the rules and actions. Let's call it a rules database. Upon any violation of the rules, we can take appropriate action.

Add two more components in our design—that is,

1. a rules and action database
2. a storage node (a blob store).



Adding blob storage and a rules and action database

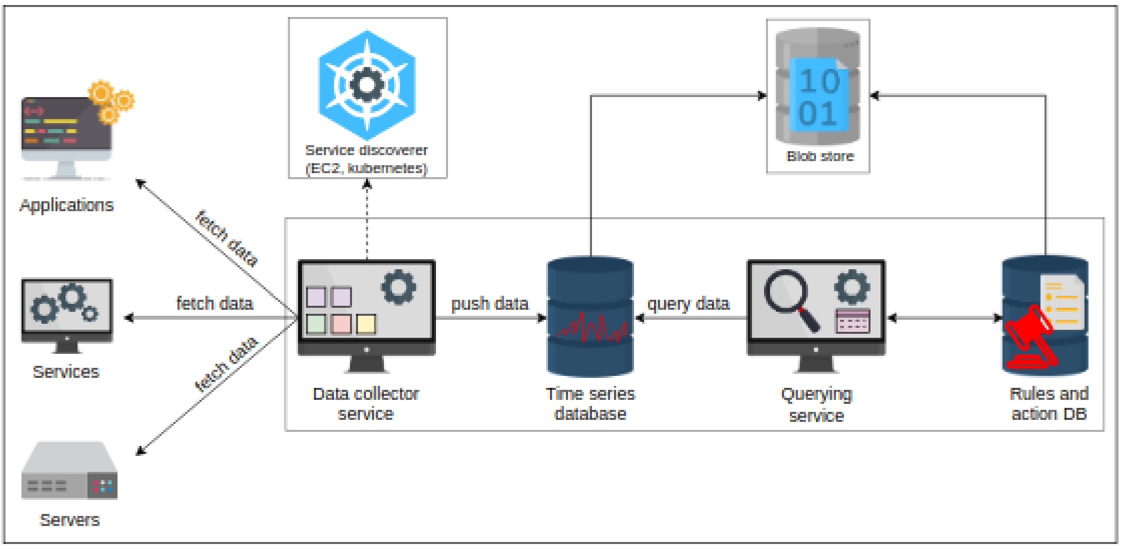
Data collector - We need a monitoring system to update us about our several data centers. We can stay updated if the information about our processes reaches us, which is possible through logging. We'll choose a pull strategy. Then, we'll extract our relevant metrics from the logs of the application.

We shall use a distributed messaging queue. The message in the queue has the service name, ID, and a short description of the log. This will help us identify the metric and its information for a specific service. Exposing the relevant metrics to the data collector is necessary for monitoring any service so that our data collector can get the metrics from the service and store them into the time-series database.

Service discoverer - The data collector is responsible for fetching metrics from the services it monitors. This way, the monitoring system doesn't need to keep a track of services. Instead, it can find them using discoverer service. We'll save the relative information of the services we have to monitor. We'll use a service discovery solution and integrate with several platforms and tools, including EC2, Kubernetes, and Consul. This will allow us to discover which services we have to monitor. Similar dynamic discovery can be used for newly commissioned hardware.

A new component is added to our existing design

- Service Discoverer



Adding the service discoverer

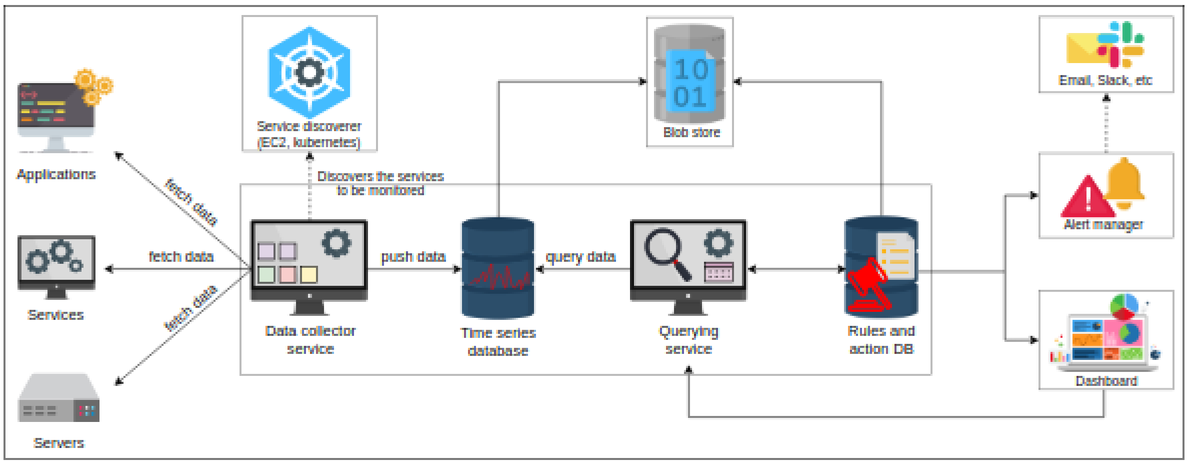
Querying service - We want a service to access the database and fetch the relevant query results. We need this because we want to view the errors like values of a particular node's memory usage, or send an alert if a metric exceeds the set limit. Let's add the two components we need along with querying.

Two more components are added

- Alert Manager
- Dashboard

Alert Manager - The alert manager is responsible for sending alerts upon violations of set rules. It can send alerts as an email, a Slack message, and so on.

Dashboard - We can set dashboards by using the collected metrics to display the required information—for example, the number of requests in the current week.



Detailed design of monitoring system

Our all-in-one monitoring service works for actively tracking systems and services. It collects and stores data, and it supports searches, graphs, and alerts.

Pros

- The design of our monitoring service ensures the smooth working of the operations and keeps an eye on signs of impending problems.
- Our design avoids overloading the network traffic by fetching the data itself.
- The monitoring service provides higher availability.

Cons

- The system seems scalable, but managing more servers to monitor can be a problem. For example, we have a dedicated server responsible for running the monitoring service. It can be a single point of failure (SPOF). To cater to SPOF, we can have a failover server for our monitoring system. Then, we also need to maintain consistency between actual and failover servers. However, such a design will also hit a scalability ceiling as the number of servers further increase.

- Monitoring collects an enormous amount of data 24/7, and keeping it forever might not be feasible. We need a policy and mechanisms to delete unwanted data periodically to efficiently utilize the resources.

Improving our design

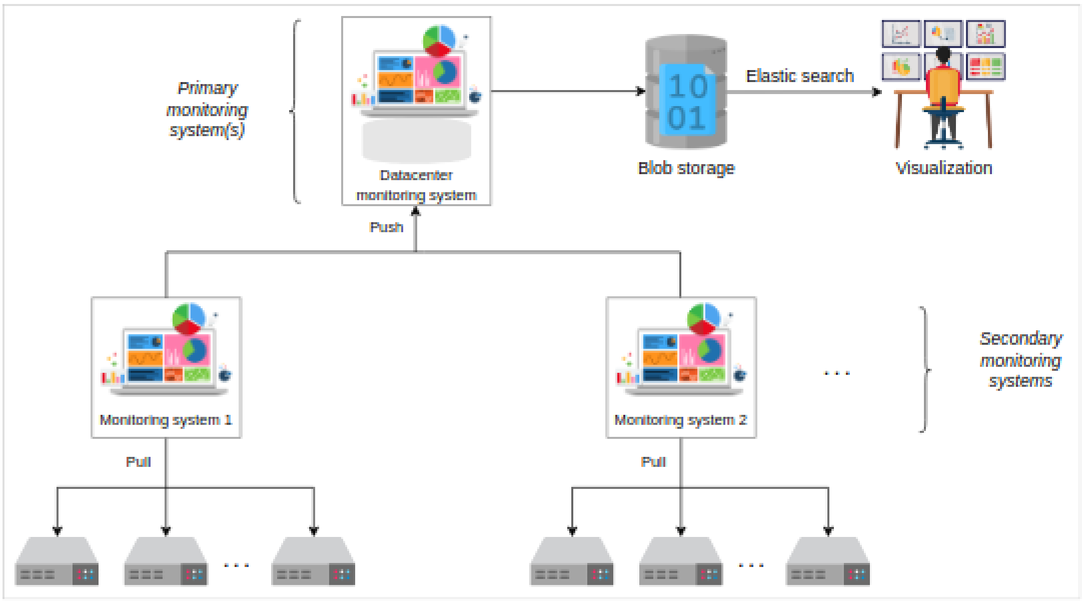
We want to improve our design so that our system can scale better and decide what data to keep and what to delete. Let’s see how the push-based approach works. In a push-based approach, the application pushes its data to the monitoring system.

1. We used a pull-based strategy to avoid network congestion. This also allows the applications to be free of the aspect that they have to send the relevant monitoring data of to the system. Instead, the monitoring system fetches or pulls the data itself. To cater to scaling needs, we need to apply a push-based approach too. We’ll use a hybrid approach by combining our pull-based strategy with the push-based strategy.

2. We’ll keep using a pull-based strategy for several servers within a data center. We’ll also assign several monitoring servers for hundreds or thousands of servers within a data center—let’s say one server monitoring 5,000 servers. We’ll call them secondary monitoring servers.

3. Now, we’ll apply the push-based strategy. The secondary monitoring systems will push their data to a primary data center server. The primary data center server will push its data to a global monitoring service responsible for checking all the data centers spread globally.

We’ll use blob storage to store our excessive data, apply elastic search, and view our relevant stats using a visualizer. As our servers or data centers increase, we’ll add more monitoring systems. The design for this is given below.



Monitoring systems pull the data from various servers and then push the data to the data center monitoring system

Visualize

Large data centers have millions of servers, and visualizing the health data for all of them is challenging. An important aspect of monitoring a fleet of servers is to know which ones are alive and which ones are offline. A modern data center can house many thousands of servers in a building. We can use a heat map to display information about thousands of servers compactly in a data center.

A heat map is a data visualization technique that shows the magnitude of a phenomenon in two dimensions by using colors.

Summary -

- Monitoring systems are critical in distributed systems because they help in analyzing the system and alerting the stakeholders if a problem occurs.
- We can make a monitoring system scalable using a hybrid of the push and pull methods.
- Heat maps are a powerful tool for visualization and help us learn about the health of thousands of servers in a compact space.

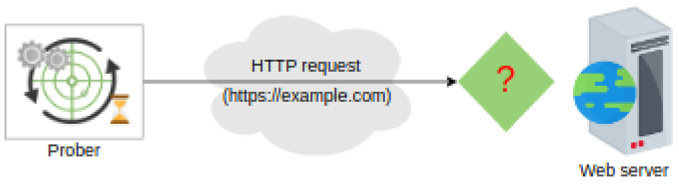
Monitor Client-side errors

Client-side errors In a distributed system, clients often access the service via an HTTP request. We can monitor our web and application servers' logs if a request fails to process. If multiple requests fail, we can observe a spike in internal errors (error 500).

System Design

A service has no visibility of the errors that don't occur at its infrastructure. Still, such failures are equally frustrating for the customers, and they might have to ask their friends, "Is the service X down for you as well?" or head to sites like Downdetector to see if anyone else is reporting the same issues. They might report the problem via a Tweet or some other communication channel. However, all such cases have a slow feedback loop. As a service provider, we want to detect such problems as quickly as possible to take remedial measures. Let's design such a system.

Initial Design - To ensure that the client's requests reach the server, we'll act as clients and perform reachability and health checks. We'll need various vantage points across the globe. We can run a service, let's call it prober, that periodically sends requests to the service to check availability. This way, we can monitor reachability to our service from many places.



Prober checking availability of website

Issues with probers -

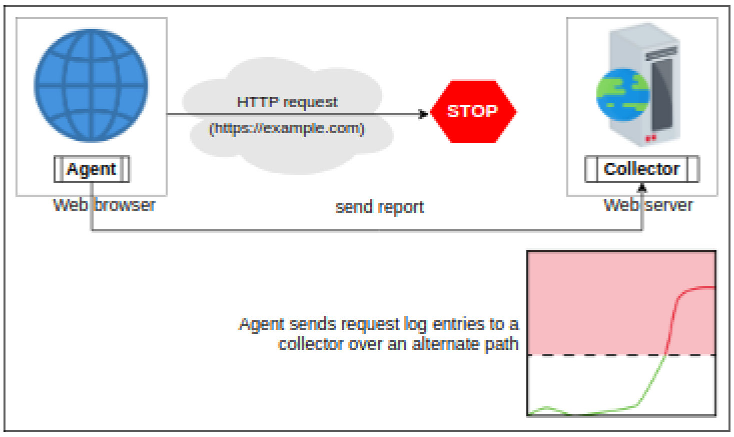
Incomplete coverage: We might not have good coverage across all autonomous systems. There are 100,000 unique autonomous systems on the Internet as of March 2021. It's not cost-effective or even possible to put those many probes across the globe. Country or ISP-specific regulations and the need for periodic maintenance are additional hurdles to implementing such a scheme.

Lack of user imitation: Such probes might not represent a typical user behavior to explain how a typical user will use the service.

Improve the design - Instead of using a prober on vantage points, we can embed the probers into the actual application instead.

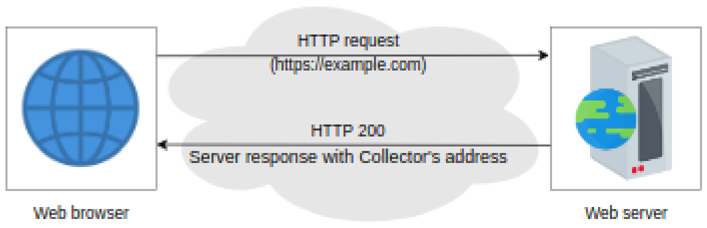
Below two components could be added -

- **Agent**: This is a prober embedded in the client application that sends the appropriate service reports about any failures.
- **Collector**: This is a report collector independent of the primary service. It's made independent to avoid the situations where client agents want to report an error to the failed service. We summarize errors reports from collectors and look for spikes in the errors graph to see client-side issues.



Evaluating the agent's report to see a spike in errors

Activate and deactivate reports -



Activate and deactivate reports

Reach collectors under faulty conditions - The collectors need to be in a different failure domain from the web service endpoint that we're trying to monitor. The client side can try various collectors in different failure domains until one works. We can see a similar pattern in the following examples. At times, we refer to such a phenomenon as being outside the blast radius of a fault.

Protect user privacy - The human user who uses the client-side software should be in full control to precisely know what data is collected and sent with each request. The user should also be able to reactivate the feature any time they wish.

Conclusion

- In a distributed system, it's difficult to detect and respond to errors on the client side. So, it's necessary to monitor such events to provide a good user experience.
- We can handle errors using an independent agent that sends service reports about any failures to a collector. Such collectors should be independent of the primary service in terms of infrastructure and deployment.

8. Distributed Caching

System Design

High-level Design

Detailed Design

Evaluation

Memcached vs Redis

9. Distributed Messaging Queue

Requirements

Considerations

System Design

Evaluation

10. Publish-Subscribe System

System Design

11. Rate Limiter

Requirements

System Design

Algorithms

12. Blob Store

Requirements

System Design

Evaluation

13. Distributed Search

Requirements

Indexing

System Design

Scaling

Evaluation

14. Distributed Logging

Logging - A **log file** records details of events occurring in a software application. The details may consist of microservices, transactions, service actions, or anything helpful to debug the flow of an event in the system. Logging is crucial to monitor the application's flow.

Need for logging - Logging is essential in understanding the flow of an event in a distributed system. It seems like a tedious task, but upon facing a failure or a security breach, logging helps pinpoint when and how the system failed or was compromised. It can also aid in finding out the root cause of the failure or breach. It decreases the meantime to repair a system.

Concurrent activity by a service running on many nodes might need causality information to stitch together a correct flow of events properly. We must be careful while dealing with causality in a distributed system. We use a logging service to appropriately manage the diagnostic and exploratory data of our distributed software.

Log analysis helps us with the following scenarios:

- To troubleshoot applications, nodes, or network issues.
- To adhere to internal security policies, external regulations, and compliance.
- To recognize and respond to data breaches and other security problems.

- To comprehend users' actions for input to a recommender system.

Use categorization Let's look into the logging support provided by various programming languages. For example, there's log4j and logging in Python. The following severity levels are commonly used in logging:

- DEBUG
- INFO
- WARNING
- ERROR
- FATAL/CRITICAL

Usually, the production logs are set to print messages with the severity of WARNING and above. But for more detailed flow, the severity levels can be set to DEBUG and INFO levels too.

Points to consider while logging

- Avoid logging personally identifiable information (PII), such as names, addresses, emails, and so on.
- Avoid logging sensitive information like credit card numbers, passwords, and so on.
- Avoid excessive information. Logging all information is unnecessary. It only takes up more space and affects performance. Logging, being an I/O-heavy operation, has its performance penalties.
- The logging mechanism should be secure and not vulnerable because logs contain the application's flow, and an insecure logging mechanism is vulnerable to hackers.

System Design

Requirements

- **Functional Requirements**

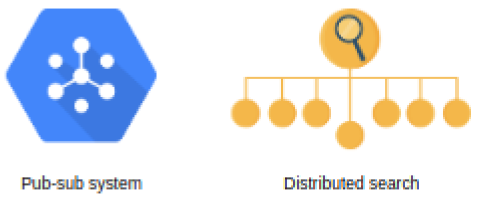
  - **Writing logs**: The services of the distributed system must be able to write into the logging system.
  - **Searchable logs**: It should be effortless for a system to find logs. Similarly, the application's flow from end-to-end should also be effortless.
  - **Storing logging**: The logs should reside in distributed storage for easy access.
  - **Centralized logging visualizer**: The system should provide a unified view of globally separated services.

- **Non-Functional Requirements**

  - **Low latency**: Logging is an I/O-intensive operation that is often much slower than CPU operations. We need to design the system so that logging is not on an application's critical path.
  - **Scalability**: We want our logging system to be scalable. It should be able to handle the increasing amounts of logs over time and a growing number of concurrent users.
  - **Availability**: The logging system should be highly available to log the data.

Building Blocks

- **Pub-sub System** - To handle the huge size of logs
- **Distributed search** - To query logs efficiently



API Design

Write a Message -

```
write(unique_ID, msg_to_be_logged)
```

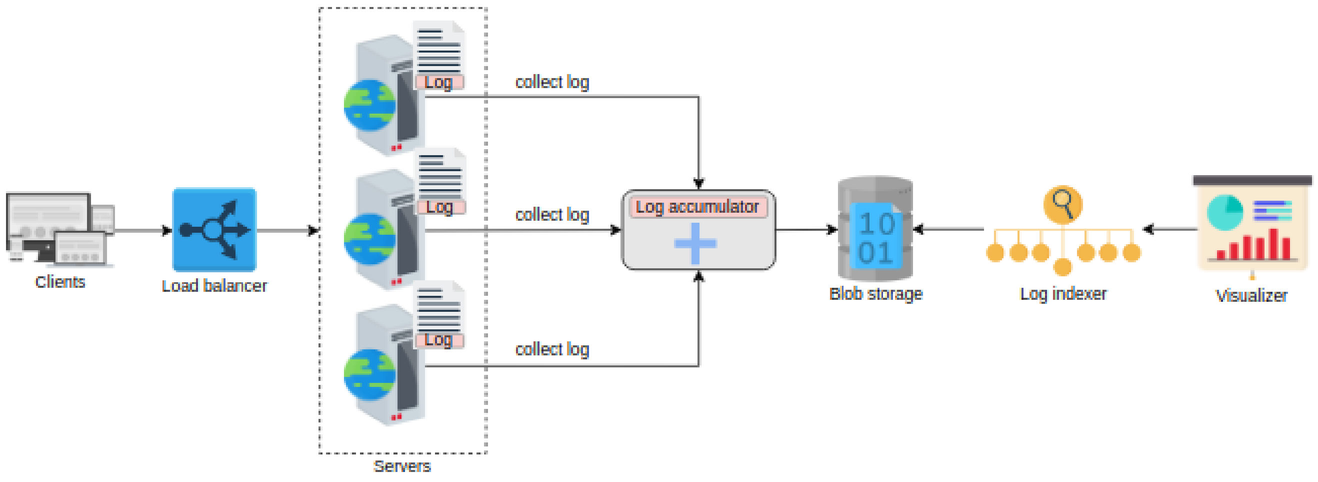
Parameter	Description
unique_ID	It is a numeric ID containing application-id, service-id and time stamp.
message_to_be_logged	It is the log message stored against a unique key.

```
searching(keyword)
```

Parameter	Description
keyword	It is used for finding the logs containing the keyword.

Initial Design

In a distributed system, clients across the globe generate events by requesting services from different serving nodes. The nodes generate logs while handling each of the requests. These logs are accumulated on the respective nodes.



The Major components of our system :-

- **Log accumulator**: An agent that collects logs from each node and dumps them into storage. So, if we want to know about a particular event, we don't need to visit each node, and we can fetch them from our storage.
- **Storage**: The logs need to be stored somewhere after accumulation. We will choose blob storage to save our logs.
- **Log indexer**: The growing number of log files affects the searching ability. The log indexer will use the distributed search to search efficiently.
- **Visualizer**: The Visualizer is used to provide a unified view of all the logs.

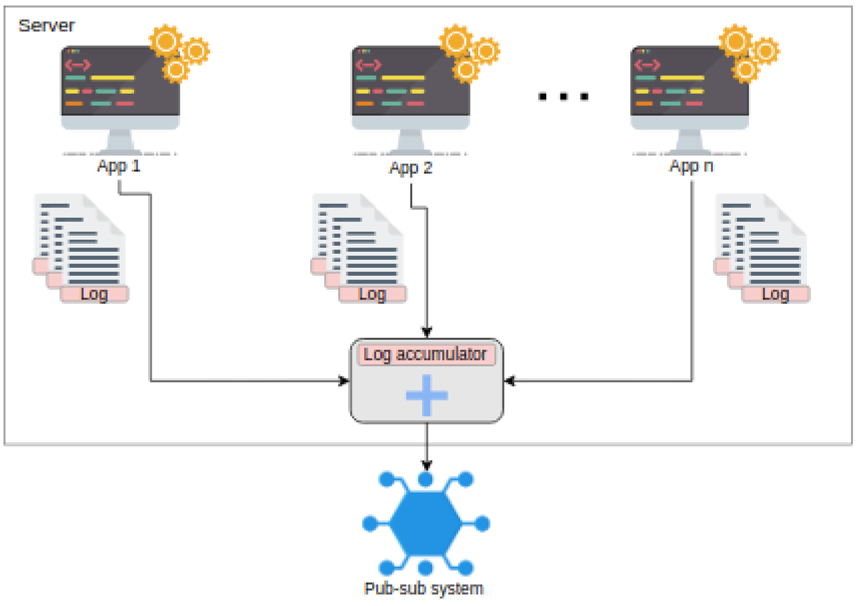
Logging at Server level Consider a situation where we have multiple different applications on a server, such as App 1, App 2, and so on. Each application has various microservices running as well. For example, an e-commerce application can have services like authenticating users, fetching carts, and more running at the same time. Every service produces logs. We use an ID with application-id, service-id, and its time stamp to uniquely identify various services of multiple applications. Time stamps can help us to determine the causality of events.

Each service will push its data to the ***log accumulator*** service. It is responsible for these actions:

- Receiving the logs.
- Storing the logs locally.
- Pushing the logs to a pub-sub system.

We use the pub-sub system to cater to our scalability issue. Now, each server has its log accumulator (or multiple accumulators) push the data to pub-sub. The pub-sub system is capable of managing a huge amount of logs.

To fulfill another requirement of low latency, we don’t want the logging to affect the performance of other processes, so we send the logs asynchronously via a low-priority thread. By doing this, our system does not interfere with the performance of others and ensures availability.



Log accumulator pushes logs to a pub-sub system

We should be mindful that data can be lost in the process of logging huge amounts of messages. There is a trade-off between user-perceived latency and the guarantee that log data persists. For lower latency, log services often keep data in RAM and persist them asynchronously. Additionally, we can minimize data loss by adding redundant log accumulators to handle growing concurrent users.

Logging at Data center level

All servers in a data center push the logs to a pub-sub system. Since we use a horizontally-scalable pub-sub system, it is possible to manage huge amounts of logs. We may use multiple instances of the pub-sub per data center. It makes our system scalable, and we can avoid bottlenecks. Then, the pub-sub system pushes the data to the blob storage.

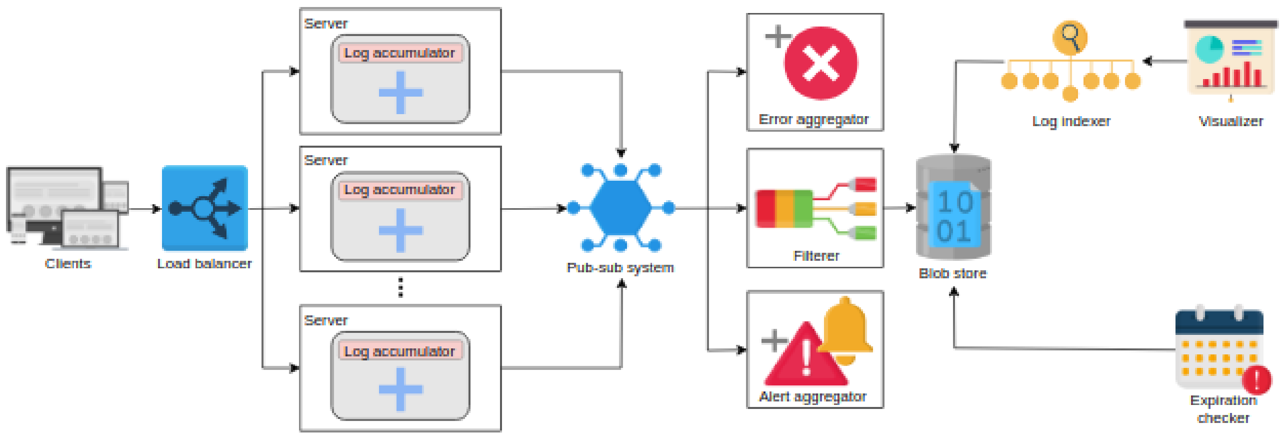
The data does not reside in pub-sub forever and gets deleted after a few days before being stored in archival storage. However, we can utilize the data while it is available in the pub-sub system. The following services will work on the pub-sub data:

Filterer - It identifies the application and stores the logs in the blob storage reserved for that application since we do not want to mix logs of two different applications.

Error aggregator - t is critical to identify an error as quickly as possible. We use a service that picks up the error messages from the pub-sub system and informs the respective client. It saves us the trouble of searching the logs.

Alert aggregator - Alerts are also crucial. So, it is important to be aware of them early. This service identifies the alerts and notifies the appropriate stakeholders if a fatal error is encountered, or sends a message to a monitoring tool.

Expiration checker - Verifying the logs that have to be deleted. Verifying the logs to store in cold storage.



Logging service design

Conclusion

- We learned how logging is crucial in understanding the flow of events in a distributed system. It helps to reduce the mean time to repair (MTTR) by steering us toward the root causes of issues.
- Logging is an I/O-intensive operation that is time-consuming and slow. It is essential to handle it carefully and not affect the critical path of other services' execution.
- Logging is essential for monitoring because the data fetched from logs helps monitor the health of an application. (Alert and error aggregators serve this purpose.)

15. Distributed Task Scheduling

Requirements

System Design

Evaluation

16. Shared Counters

High-level Design

System Design