HLD - System Design

Type	Year-long goals
Status	In progress
Tags	

Author: Ajay Kumar

Github: ajay-panchal-099 (AJAY PANCHAL) (github.com)
LinkedIn:
AJAY KUMAR | LinkedIn

Resources:

- System Design Roadmap (whimsical.com)
- https://www.youtube.com/playlist?list=PLhgw50vUymycJPN6ZbGTpVKAJ0cL4OEH3
- https://github.com/codekarle/system-design/tree/master/system-design-prep-material/architecture-diagrams
- https://www.youtube.com/playlist?list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7

- https://www.youtube.com/watch?v=mhUQe4BKZXs
-

Communication:

1. Synchronous
2. Asynchronous

Message Based Communication:

1. P2P model: peer-to-peer like sending an email
2. Pub-Sub model: One consumer but multiple subscriber

Tools: Kafka, RabbitMQ

Communication Protocol:

Application Layer:

- Client Server comm:
  - HTTP
  - FTP
  - SMTP
  - Web Sockets
- Peer2Peer comm:
  - Web RTC → Uses UDP Protocol

Transport Layer:

- TCP/IP:
  - First create a connection and then send packets, ack is also there, no packet loss, ordering is also maintained.
- UDP:

- No connection, send data parallely via multiple connections, no ordering maintained, packet loss can be there. Fast... ex: live/video streaming

Communication Methods:

1. **Push:** s2c, After Long: It is like I subscribed and server will send response when new data available. It is server to client conn.
   The client open the connection with server and keeps it always active.
   The server pushes new events to the the client. This method reduces the server load.

   Sent notification even if the website is not open.
   Prob: what if server sends response at a time when i don't need it.
2. **Pull/Polling** ⇒ c2s, Client request, Server Respond (ask and get). Prob: asked 100 req but fulfill only 5, then comes long polling
3. **Long Polling** ⇒ c2s
   Client request, Server keep it open until it gives response. ⇒ ask 100, gives 5 now and keep open channel and provides once available.
   Prob:
   a. Ordering issue
   b. Server will always keep running
4. **Socket** ⇒ When we need continuous and frequent connection. 2-way channel conn. ex: chat application
5. **Server sent events(SSE or WebSockets)** ⇒ Ex: Cricbuzz
   Client subscribes to the server 'stream', and the server will send message ('stream f events') to the client until the server or the client closes the stream.

   Works as long as User is using the website.
   a. One way connection
   b. Long Lived Connection

Few Terms:

- Website: Read only
- Web Application: Read and write
- **Stateless server:**

  Subsequent requests should not depend on something stored on that server from a previous request. DB can store stuff but the server itself shouldn't because we have no idea where the previous request got routed to!
- **Stateful:** Can have info about previous requests on a given server.
- **Latency**: Time taken by a server to process the request i.e. network delay + computational delay.
  - Sol:
    - Caching
    - CDN
    - Upgrade Systems

- **Throughput**: Amount of data transmitted or information flowing through a system per second. Unit: bps(bits per sec)
  - Low thp reason:
    - Latency
    - Protocol Overhead
    - Congestion(large number of requests)
  - Sol:
    - CDN
    - Caching
    - DS (Distributed System)

    - Use LB
    - Upgrade Resources
- **Authentication vs Authorization:**

  Authentication: who are you?

  1. Basic Auth: For log-in, send user-name and password each time to the server in clear text
  2. Token Based Auth: First time generate a token with username and password and with every subsequent request use token.
  3. OAuth: ex: ex: sign up with google/Facebook on an app.

  Authorization: What can you do?

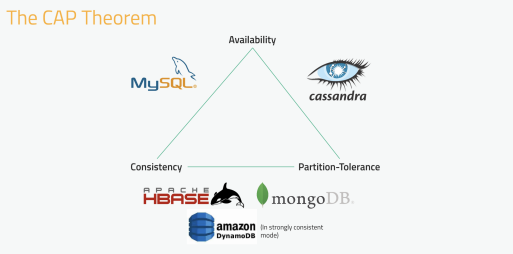
CAP Theorem:

You can only have 2, not three.

Consistency: Do I get the data right away from what I just wrote?

Availability: Is my system a single point of failure, even for a very short period?

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



Now, if we want system to work in partition tolerance situation, we need to choose either A or C. If we choose C, we need to stop dirty reads hence we are negotiating A but if we choose A, then C won't be there between nodes.

Most modern databases provide the best of all three. But it would help if you thought of trade while deciding the db for your system.

P is the most important in CAP theorem. Now, all decision comes to choose between A and C.

Notes:

If you want A and C, that system will always be centralized.

Choosing DB using CAP:

Mongo DB: trading off A → single master, single point of failure

Cassandra: trading off C → No single master, No single point of failure but it takes time to replicate data across all nodes.

MySQL: trading off P → not easy to Scale traditional DBS

Availability(replication vs redundancy):

Fault Tolerance ↔ HIgh Availability

How to increase HA:

1. Replication: includes redundancy but synchronization of nodes (in db). Types: Active-Active(read-write both),

Consistency(strong vs eventual):

If all the requests are getting the same output regarding of whose accessing it, is called consistency.

Factors to improve c:

1. Improve network bandwidth
2. Stop the dirty read operation
3. Replication based on distance aware strategies(keep the servers close)

Active-Passive(Master-Slave) (read-write by master only)

2. DS
3. Redundancy: duplication of nodes in case some of them are failing. Types: Active(which is currently providing service) and Passive(at stale).
4. Active -Passive Replication**:**
    a. Master-Slave
    b. Master-Master

Types of consistency:

1. Strong C: when system doesn't allow read operation until all the nodes with replicated data are updated. ex: train/flight seats.
2. Eventual C: We don't stop read operation when nodes are getting replicated. Some user will get old data and some new but eventually all data will be updated to latest after some time. ex: social media
3. Weak C: No need to update

Few Patterns:

- Strangler Pattern: (Refactoring to micro services)

  Slowly slowly moving monolithic service to micros services. Like A/B testing, send few traffic to new service and few old, once microservice looks good, send all traffic to that only.

- SAGA Pattern: (Data Management in micro service), Prob: how to maintain transactional property in microservices because a single trans. can touch multiple db? sol: SAGA pattern.
    - Shared DB among services → Not very common, scalability issues, Update/Delete issue etc. Benefits: table join is easy, maintain transactional property(single db) etc.
    - DB for each service → Maintain ACID property each, difficulty in joining(sol: CQRS).

  SAGA: for transactional property among service.

CQRS: for Joining operation among tables of different services db. This can be hug problem.

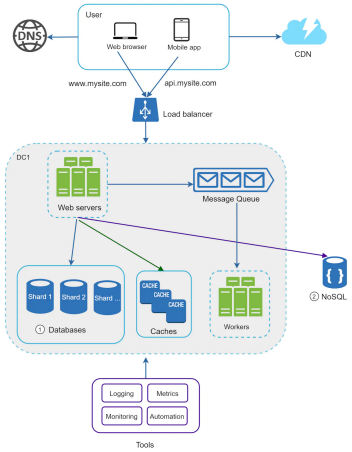
SAGA Types: https://www.baeldung.com/cs/saga-pattern-microservices (**IMP)

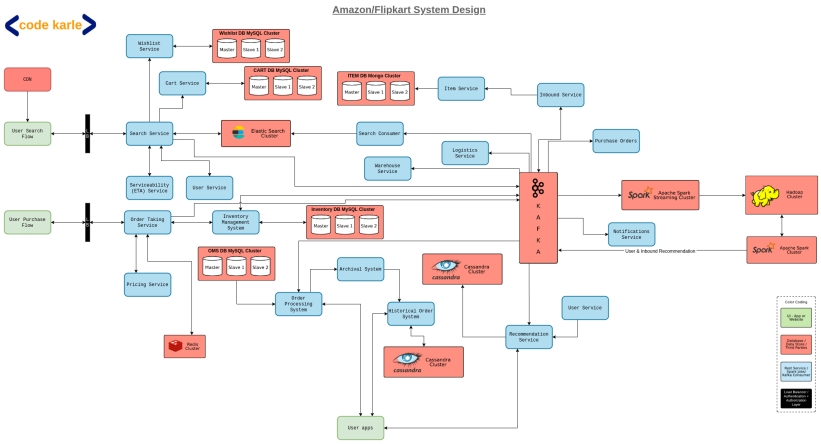
1. Choreography-based saga: Maintain 1/2 queues(noraml queue and failure queue) for publishing event, one service publish event and other one read the events and take steps.
   Drawback: There can be cyclic dependencies.
2. Orchestration-based saga: a middle man, call each service on the event and call next one based on events. Orchestra take care of everything.

CQRS: Command query request segregation. https://medium.com/design-microservices-architecture-with-patterns/cqrs-design-pattern-in-microservices-architectures-5d41e359768c
Maintain a common separate db for read operations and write in each services db. Each db will sync with common db via publish-event/db-trigger if they do any create/delete/update operations and common DB will have all the tables.

Scale From Zero To Million:

Basic Architecture:





Step by step to design system at scale:

1. Single Server (Avoid Single point of failure)
2. Application and DB server separation
3. Load Balancer with Multiple Application server
4. Database Replication
5. Database/App Scaling(Horizontal or Vertical)
6. Cache (Rabbit MQ, memcached)
7. CDN
8. Data Centre
9. Messaging Queue (like SQS, Kafka)
10. Database Sharding(Horizontal or Vertical)

Consistent Hashing:

Article: https://medium.com/@sandeep4.verma/consistent-hashing-8eea3fb4a598

video link: https://www.youtube.com/watch?v=jqUNbqfsnuw&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=7

Back of the Envelope Concept:

Video: https://www.youtube.com/watch?v=WZjSFNPS9Lo&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=10

docs: https://bytebytego.com/courses/system-design-interview/back-of-the-envelope-estimation, https://systemdesign.one/back-of-the-envelope/

Databases System:

File Based Storage System:

Where data is stored in form of file.

Challenges:

1. Data Redundancy: update/delete anamoly
2. Poor Security
3. Slow

SQL VS NO-SQL:

SQL:

- Managed structural data in forms of tables
- Pre Determined Schema
- Relations among tables
- Centralised Nature: Whole data needs to present in a single server
- Vertical Scaling is better in SQL (Sharding is there but...)
- ACID Property (Maintains data consistency and Integrity)
-

RDBMS:

A software for RDB which stores data in tables.

1. No data redundancy and inconsistency
2. Data Searching (built in using queries)
3. Data Concurrency (locking system to prevent abnormalities)
4. Data Integrity(add some constraints)

Prob:

1. Rigid Schema
2. High Cost

3. Scalability Issue (sharding is very difficult)

No-SQL:

- Non-Relational / Not only SQL /No-sql
- Horizontal Scaling
- Property: BASE
  - BA: Basically Available means HA db.
  - S: Safe State means state of data can be change(schema), sync is easy among them and can update themselves
  - E: Eventual Consistency means sometimes you might get stale data but after some time you will get correct data
- Better in managing Unstructured data, Distributed in Nature
  - Key-Value DB: like a dict, maintains key and their associated value. Value can be anything. Query on key, not value. Ex: Dynamo DB.
  - Document DB: same key, value type db. Value can be Json, Xml. You can query on value. Ex: Mongo DB
  - Columnar DB: Column wise data. A key can have dynamic no of columns. Ex: Apache HBase. Data Format: Parquet, Apache ORC.
  - Graph DB: Graph like db structure. Social networking, recommendation engines.
  -

1. Key Value db : stores in the form of key and value. ex: cache db i.e Redis db. [Use when everything is simple]
2. Document db: best of both RDBMS and No-SQL. relationship concept from RDB and dynamic schema and scaling from NoSql. ex: MongoDB [Use when nested items]
3. Columnar db: columns are stored together instead of rows. Aggregation is fast in such db. used for data analytics, machine learning. ex: cassandra [use

when machine learning]

4. Graph db: tores entities and relationship in the form of graph. ex: social network. [use when social graph]

Ex: shopping cart: Key -value(Redis), Map→ Graph, Player-score: key value, machine learning→ Columnar db Payment: need high consistency→ use RDBMS

When to use which?

- SQL can support complex/flexible queries but NoSQL.
- NoSQL - If in advance you know which column you are gonna search always.
- Data is relational, got for SQL otherwise No-SQL.
- Data Integrity(ACID) is a must, SQL is the only option. Ex: Financial transactions.
- High Availability High performance, Fast search query(with some inconsistency) → No-SQL

Indexing:

Organising the searching / sorting the things i.e implement binary search in tables. We can reduce TC from O(N) to O(logN)

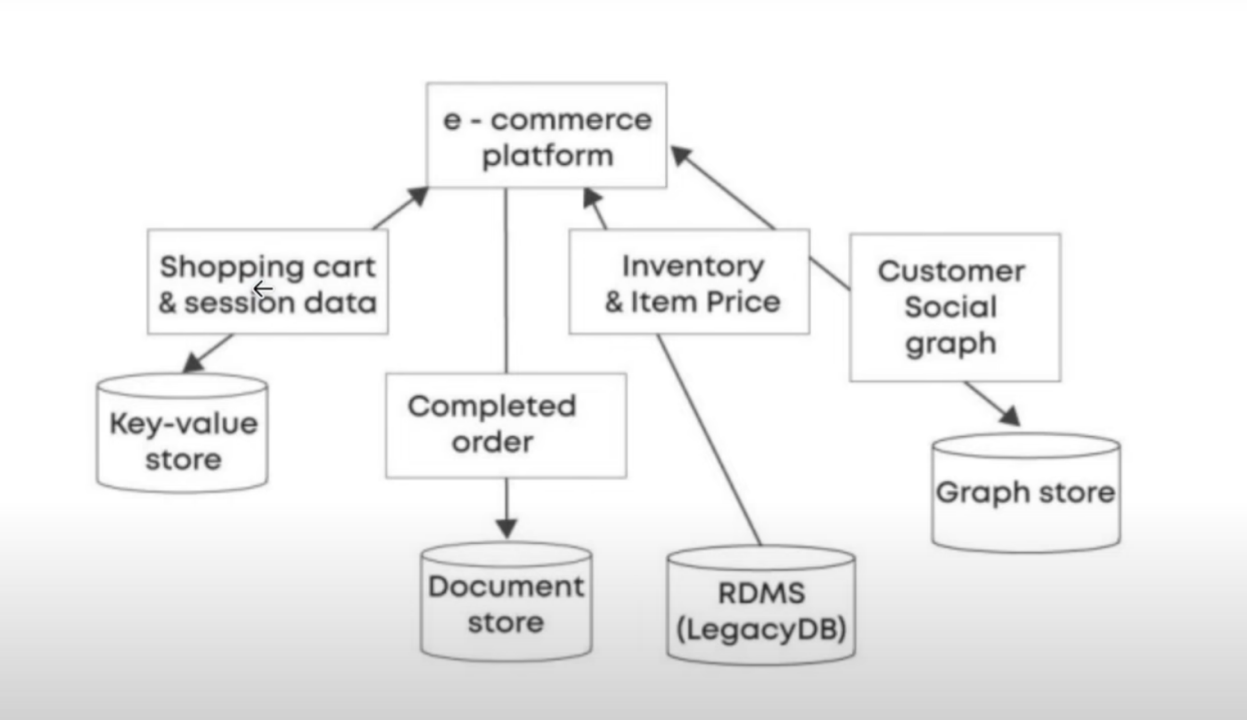
Def → Indexing creates the lookup table with the column and the pointer to the memory location of the row, containing the column.

DS: B-Tree data structure is used for indexing as it is multi-level format f the tree based indexing which has balanced binary search tree.
Use case:

1. Read intensive db only.
2. Not use with write intensive: 2 times entry→ main table + entry in indexing table + reshuffling/sort indexing table.

Polyglot Persistence:

When a single db is not able to fullfill application requirenment and we use multiple db called Polyglot Persistence/System



Database Scaling:

- **Vertical Scaling**
  1. Cold Standby: A standby database ready to take the place of failed db, but needs to backup data into the new server which we already stored at some place periodically.
     manually intervention is needed.
  2. Warm Standby: Constantly backing up the data into the new server. Instantly ready to take the place of failure db. **Replication**.
  3. Hot Standby: Write data simultaneously into all backup servers. No replication is needed.

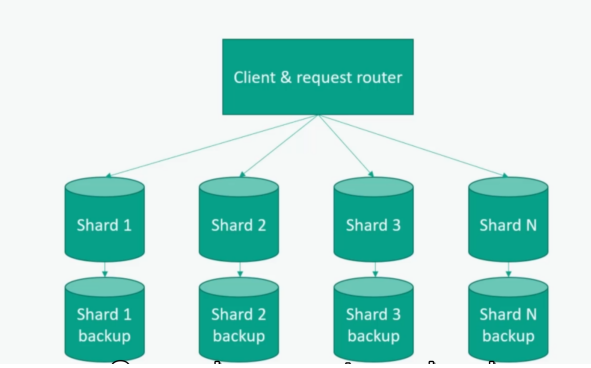
- **Horizontal Scaling:**
  1. Shards: Horizontal Partitioning of db

     At the top, we have a router that sends requests to each shard.

     Each shard can have a backup DB: like a hot standby

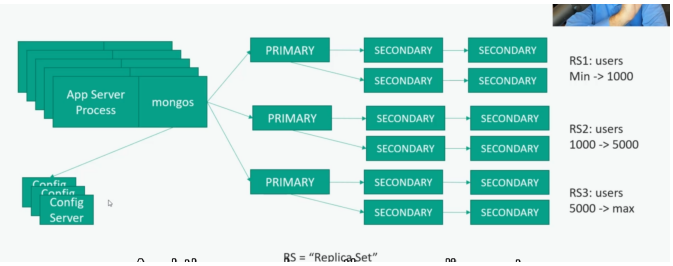
     Having a good hash fn can reduce joins and complex SQL queries.

  2.



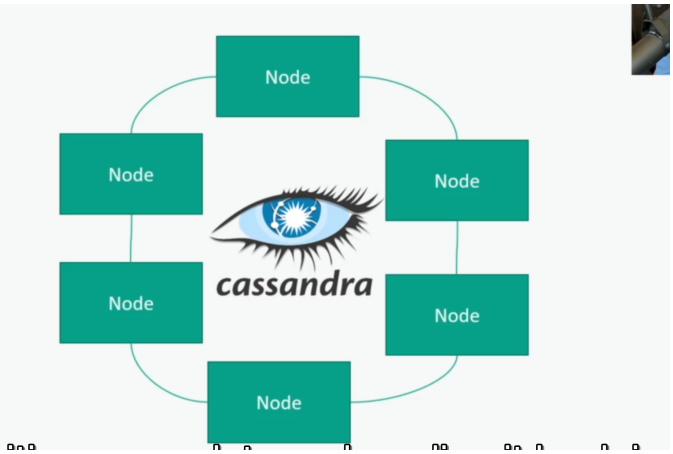
MongoDB:

- No-SQL DB
- It is a highly scalable database.
- No single point of failure, and Multiple config servers too.
- It can have multiple replication backup db across multiple regions or data centre
-



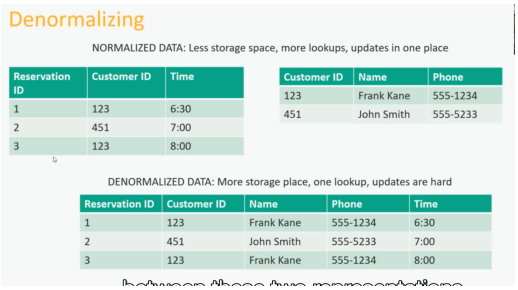
Cassandra

- No Single point of failure
- Any node can serve as the primary node
- Data might not be consistent because data needs to be written to all nodes and that could take time. So if we can have some time gap b/w reading and writing, then it is a good choice.
-



Few Terms related to dbs:

- Normalization: Splitting the data into multiple tables to avoid redundancy.
- DeN: Keeping all the data into single place.
- Sharding Databases are sometimes called “NoSQL”.
- Resharding: Once we scale up, we need to redistribute data to other DB, on what basics and how much etc.
- Hotspot: *Celebrity Problem* → what if some of the data is getting hit harder than other sections, how to redistribute db on actual traffic?
- Most Large-scale databases normally use *denormalised* data(All data in one place, no partition of tables like relational databases). One single hit is required. Otherwise, two or more in relational db to join the table and all.
- Finally, it depends on customer experience whether we require more updates of the database or more hits.
-



Database Indexing:

video: https://www.youtube.com/watch?v=6ZquiVH8AGU&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=22

Understanding How Data is Stored

- Data in a table is not stored logically but rather in data pages of a fixed size (typically 8 KB).

- Each data page has a header, data records (actual rows), and an offset array.
- The offset array is an index that points to the location of each row within the data page.

How Indexing Works

- Indexing uses a B-tree data structure to speed up searching for specific data.
- When a new row is inserted, the B-tree is traversed to find the appropriate data page for storing the data.
- Searching for data involves traversing the B-tree to find the data page containing the desired value.

Benefits of Indexing

- Improves search query performance by reducing the number of data pages that need to be scanned.
- Search time complexity is reduced from linear (O(n)) to logarithmic (O(log n)) with indexing.

Trade-offs of Indexing

- Indexing introduces additional overhead for maintaining the B-tree structure.
- Updating data (inserts, deletes) requires updating both the data pages and the B-tree.

Clustered and Unclustered Indexing in RDBMS

In Relational Database Management Systems (RDBMS), indexes are special data structures that act like lookup tables to improve query performance. They speed up searching for specific data by providing a faster way to locate relevant rows in a table. There are two main types of indexing: clustered and unclustered.

Clustered Indexing:

- **Concept:** A clustered index fundamentally reorders the physical storage of data rows in a

Unclustered Indexing:

- **Concept:** An unclustered index is a separate structure from the actual data table. It contains an additional

table based on the indexed column(s). This means the actual data itself is stored in the order dictated by the index key.

- **Structure:** The leaf nodes of the B-tree representing the clustered index contain the actual data records. When you create a clustered index, you essentially define the sort order for the entire table data.
- **Benefits:**
  - **Faster Ordered Access:** Retrieving data in the same order as the clustered index is very efficient. It's like having a sorted bookshelf - finding books in sequence becomes significantly faster.
  - **Reduced Disk I/O:** Since data is already ordered, the database engine can minimize disk seeks by efficiently traversing the B-tree to locate relevant data pages.
- **Drawbacks:**
  - **Limited to One:** A table can only have one clustered index because the data itself has a single physical order.
  - **Insert/Update Overhead:** Maintaining the sort order during inserts, updates, and

B-tree where the leaf nodes hold pointers to the actual data rows. These pointers can include the indexed column(s) and optionally other included columns from the table.

- **Structure:** The leaf nodes of the unclustered index B-tree do not store the actual data. Instead, they hold index entries with the indexed column values and pointers to the corresponding data rows in the table.
- **Benefits:**
  - **Multiple Indexes:** You can create multiple unclustered indexes on a single table, allowing for efficient retrieval based on various criteria. This is like having separate card catalogs for different book properties like author or genre.
  - **Flexibility:** Unclustered indexes don't impact the physical data layout, offering more flexibility for different query patterns.
  - **Reduced Update Overhead:** Maintaining unclustered indexes generally incurs less overhead compared to clustered indexes during data modifications.

deletes can introduce overhead compared to unclustered indexes.

- **Less Flexibility:** Choosing the clustered index column(s) is crucial as it dictates the physical data layout and impacts all queries on the table.

- **Drawbacks:**
  - **Slower for Ordered Access:** Retrieving data in a specific order not aligned with the unclustered index might require additional steps, potentially involving accessing the main data table.
  - **Increased Storage:** Unclustered indexes require additional storage space for the separate B-tree structure.

Choosing the Right Indexing Strategy:

The choice between clustered and unclustered indexing depends heavily on your typical query patterns and access needs for the table. Here are some general guidelines:

- **Use clustered index for:**
  - Queries that frequently retrieve data in a specific order based on the indexed column(s).
  - Tables with frequent range queries that leverage the sort order.
- **Use unclustered index for:**
  - Queries that frequently filter or search data based on specific column values (even if scattered physically).
  - Tables where you need multiple indexes for different query patterns.

Scalability:

Two types:

1. Horizontal Scaling

2. Vertical Scaling

Monolithic Architecture:

- Single Server Design
  - Single point of failure
  - Good for small website
- Single Server Design with Database scale-out
  - Database load splitter
  - Scale independently
  - Little better Resiliency
  - Still a Single Point of failure

Distributed System:

Vertical Scaling:

- Bigger Server (More Hardware)
- VS has its limit, it can’t extend infinite
- Still a Single point of failure

Horizontal Scaling:

- Multiple Server
- Load Balancer distributing the load on each server
- Not a Single point of failure
- Can scale infinitely
- More stuff to maintain

- This is easier if servers are stateless

DNS Server:

1. Load Balancing with BIND
2. Round Robin
3. Cause *Celebrity Problem*
4. Browser cache dns info
5. cache can cause CP

Good Solution: Let LB decide whom to send request

LB Problem:

1. Session problem → might need to be logged in on all servers
2. store session somewhere → new file server (but single point of fail)

LB types:

1. Round Robin(static→predefined)
2. Weighted Round Robin(static)
3. IP Hash Algo(static)
4. Source IP Hash(static)
5. Least connection algo: least active conn. (dynamic→ runtime)
6. Least Response time (dynamic→ runtime)

Failover Strategies

Servers:

1. On Premise Data Center

2. Cloud Provider ex:EC2
3. Serverless ex: Lambda, Athena

Caching

Servers can have their cache memory but that might not be enough every time. Use: when website is read intensive or static content.

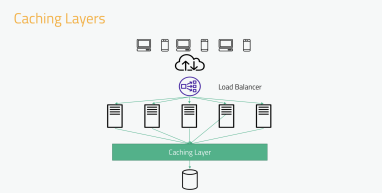
Two types:

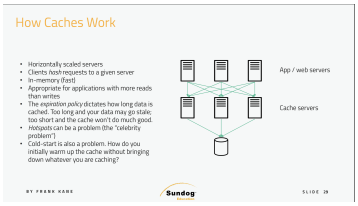
1. In memory/Local Cache
2. Distributed Cache

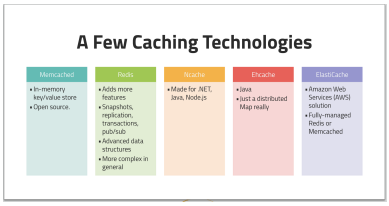
☐ TODO: Look for the LRU Cache code

Eviction Strategies for Caching

- LRU: Least Recently Used
- LFU: least Frequently used
- FIFO: First In First Out
- MRU: Most Recently Used
- MFU: Most Frequently used
- LIFO: Last In First Out
- RR: Random Replacement







Caching Strategies

Distributed Cache: It's just a consistent hashing technique.

Cache-Aside Caching Strategy:

First read through cache and if it is a miss, read from db.

Write Around Caching Strategy:

Directly write data into db and invalidate the cache. It does not

Pros:

- Good Approach For Heavy Read Application
- Even cache is down, req won't fail. It will fetch data from db.
- Cache data structure can be different from db. **Here Application is responsible for storing data into db and cache.**

Cons:

- For new data, we will always have miss-hit. we can pre-heat the cache.
- Without appropriate caching is used during write op, there can be data inconsistency.

Read Through Caching Strategy:

This application is very similar to Cache aside. The only difference is that in the read through Cache the **cache itself takes the responsibility to fetch the data from Db and store it back to Cache**.

Pros:

- Good Approach for heavy reading app.

update the cache.

Pros:

- Good for Read Heavy Application.
- Resolves Inconsistency problem between cache and db.

Cons:

- For new data, always cache-miss.
- If db is down, write operation will fail.

Write Through Caching Strategy:

Right through Cash will first write the data into the cache and then in ***synchronous manner*** write data into the Db. If either of them insertion got failed, you have to throw the exception.

Pros:

- Cache and DB will always remains consistent.
- Cache Hit chances increases a lot.

Cons:

- Alone is not useful, it will increase the latency. (Always used with Read through or ache aside)
- 2 phase commit, needs to be supported to maintain transactional property.
- If db is down, operation will fail.

- Logic of fetching the data from db and updating the cache is separated from application.

Cons:

- 2 cons points are common between this and above. Pre-heat the cache and inconsistency.
- Here Cache data structure will be same as db table.

Write Back Caching Strategy:

When we write data into the Db we will ***not write into a synchronous manner***. So we first write data into the cache, and then push the data into a queue, and then read it and put the data into the Db.

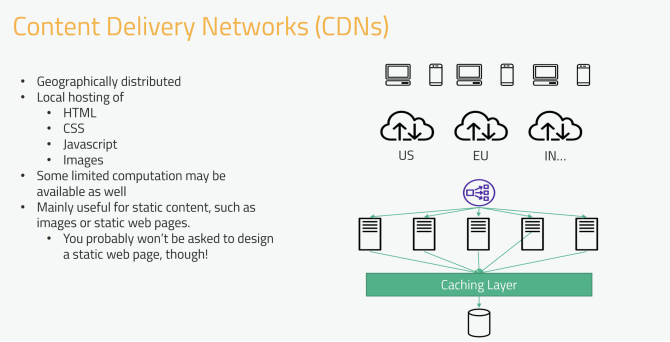
Pros:

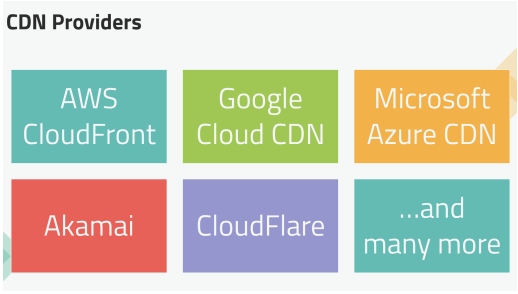
- Good for write heavy.
- Improves the write op latency because write happens asynchronously.
- Cache hit increases a lot.
- Gives much better performance when used with read through cache.
- Even if db fails, write operation will still work.

Cons:

- If data is removed from cache and db op still didn't happened. There can be issue. (Can solves this by keeping the TTL for like 2-3 days)

CDN: static content





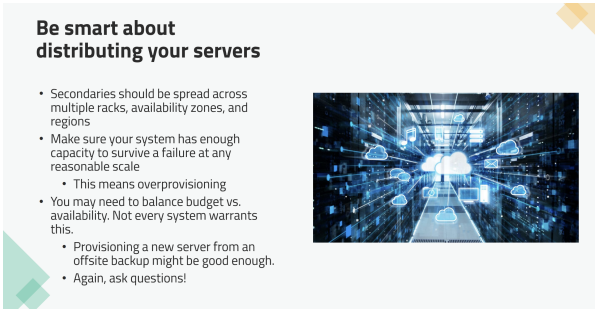
Resiliency

Resilience to Failure → How to handle failure on small or large scale

Data centre → Availability Zones

A Server, an AZ or an entire region could go down.

Overprovisioning → have excess capacity to handle any kind of failure



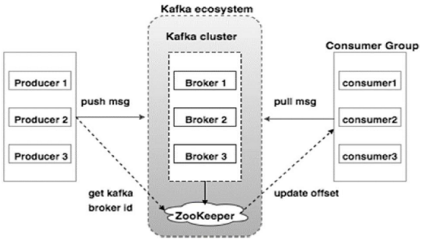
Message Queues

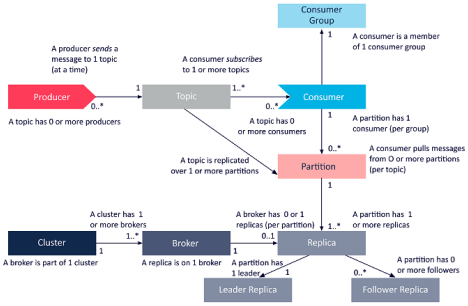
Advance stuff: https://www.youtube.com/watch?v=oVZtzZVe9Dg&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=17

Kafka:

components:

- Producer
- Consumer
- Consumer Group
- Topic
- Partition
- Offset
- Broker
- Cluster
- Zookeeper





P2P
queues:
SQS

Pub-Sub
queue:
SNS

Messaging Queue and its Advantages

The first use case is in an e-commerce application, where a user buys a product and needs to be notified as soon as the purchase is complete.

If E-commerce application directly talks to send notification application, then latency will increase, and if it goes in a synchronous manner, then definitely the latency will increase. But a queue brings retry capability, and this is another advantage of Q.

Second thing is the pace matching. If you have multiple applications that want to send a message, but the recipient application can only process 15 messages per second, you need to use a message queue to help you handle the messages.

Space matching through Q in the mid can be used to do pace matching, and this helps you understand why the message queue is very popular.

In a city, a cap service provider has many cars with Gps installed, and each car is sending its location every 10 seconds. A consumer application needs this data to create a dashboard, but it cannot consume this much data so frequently.

Point2Point and Pub/Sub Pattern

Point to point messaging services only allow a message to be consumed by one consumer. In Pub Sub architecture, a message can be consumed by multiple consumers.

The Pub Sub is a messaging system that broadcasts messages to all the queues based upon the logic. The Rabbit Mq is a similar messaging system that broadcasts messages to all the queues based upon the logic.

Kafka Messaging Queue in Depth

Kafka is a messaging system that talks with a **broker**. The broker is nothing but a server Kafka Server that is created once you started the Kafka.

Now this broker has Topic, and inside a Topic, it has **partition**. A partition can have many offsets, and each offset has a name, and each offset can have a different length.

Consumers read from a partition, and each consumer is part of one consumer group. Consumers inside a particular group do not share a partition, so if Consumer One uses partition zero, Consumer Two will take partition one, but this is not possible.

A **cluster** is a collection of Kafka Server running in different machine. Each Broker has a topic, which can be divided into several partitions, and each consumer group can read different partitions of a particular topic.

A message has key value, partition topic, value, and a message flow. A message flow takes care of all these things, including message sizes, limit, message goes down and consumer goes down, and all those things.

A **key** is something that can be string or Id or anything. A topic name is something that is mandatory, and a message that has passed certain Id can be published in a certain topic if the message has the right key.

It will push the data into a particular partition based upon the key hashing, but if the key is empty then it will look for the partition itself, but if the partition is empty then it will follow the round robin whenever the message comes.

Kafka consumers read messages from **partitions**, **consumer groups** and **Zookeeper**. The offset of a message tells how far a consumer has read a message, and the Zookeeper keeps track of this using a committed offset variable.

Zookeeper keep this data for all the consumers. If one consumer goes down, Kafka will pick another consumer from this consumer group from this group from this group and it will check if there is a free consumer.

Consumer groups are used to ensure that if a consumer goes down, another consumer can take over and consume the data from where the previous one left off. So now you know why committed offset is used and why consumer groups are used.

A **cluster** is a group of brokers, where each broker handles a topic. A cluster can have one broker or many brokers, and each broker handles a topic in a different partition.

The data is stored in our Brokers in partitions. If a partition goes down, the message inside it will still be there because we have a **replica** inside it.

When a **leader** goes down, the replica takes over and becomes a leader. So whenever somebody is reading from a partition particular partition, it always goes to the broker one because this is the leader.

Questions:

What happens when queue size limit is reached?

> We maintain different broker and host same topics there too.

What happens to messages when queue goes down?

> Nothing, we maintain replica. Now replica will be leader and will start reading message from offset. Read always happens only from leader.

What happens when consumer goes down?

> When a consumer goes down, another consumer takes over from the consumer group.

What happens when consumer not able to process message?

After retry, we push the message in DLQ(Dead Letter Queue).

How retry works and different ways of retry?

If we add a capability to the partition or topic to do three times retry, the system will not increase its commit offset when pulling data, but instead will increase its retry account.

Okay, so retries finished, so committed offset is not increased. Now it will put the message into failure queue that is dead letter queue, and move forward.

In RabbitMQ, for retry we push the message in the beginning again and later into DLQ if necessary.

How Distributed messaging queue works?

Different clusters, brokers and all.

RabbitMQ in depth

Rabbit Mq is push-based approach, whereas Kafka is pull-based approach. Rabbit Mq pushes the message as soon as the data message comes in, while Kafka is full-based approach.

To link all these things, we have a producer, a queue, an exchange, a routing key, and a consumer. The producer talks with the exchange, which is binded with the queue, the routing key, and the consumer.

Rabbitmq exchanges are made up of different types, and the Fan Out approach pushes a message to all the queues associated with a particular Exchange.

Direct Exchange means that if a message key and a routing key are exactly matching, the message will be pushed to Q1. You can use Wildcard in Topic Exchange to push messages to a particular queue if anything matches in the Wild Card. If it's a direct message, wildcard is not even allowed.

If any message consumer is not able to process a message, it will recue it to the back of the queue, and if it still cannot process the message, it will put the message into a dead queue.

We discussed different ways questions can come into the interview, and I hope we are good with this. If you have any doubts, please ping me on the comment

section or follow me on LinkedIn.

Design Rate Limiter:

source: https://www.youtube.com/watch?v=mhUQe4BKZXs, https://www.youtube.com/watch?v=X5daFTDfy2g&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=13

Algorithms:

1. **Token bucket:** maintain token counter and discard if token come down to zero
2. **Leaky bucket:** FIFO, Keep filling the queue, discard if queue is full and after each particular time, leak the bucket
3. **Fixed window counter:** Fix window. Prob: there can be overload if req are coming near end time or beginning time of window.
4. **Sliding logs:** store last one min request and maintain each req timestamp/logs
5. **Sliding Window counter :** store last one min request and maintain (timestamp + no. of req counter), this way we will have less entries.

Problem in distributed system: If we serve single user request with different different region application server, there might be inconsistency because if multiple service read the token data at the same time, they might get stale data or no of total request can exceeds as defined. Hence,

I**nconsistency:** Sticky session → Always server the same user request with one application server but there can be load and cause fault. To have fault tolerance, use locks

Race condition: Use locks but adds latency

There is not final best solution. Best methods are → relax rate limit, if system sometime server more than defined limit, it should be okay. Little performance optimization we can do is to use local memory instead of redis node and another service will load/sync the data with redis later.

Idempotent Handler:

resource:https://www.youtube.com/watch?v=mI73eTlSqeU&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=15

Idempotency: It helps in taking care of duplicate requests.

Concurrency: When multiple req are trying to access same resources.

Definition: From a RESTful service standpoint, for an operation (or service call) to be idempotent, **clients can make that same call repeatedly while producing the same result**. In other words, making multiple identical requests has the same effect as making a single request. Like http GET/PUT/DELETE request, it will always give the same result while POST create new object.

How to design POST API as Idempotent?
Scenarios where duplicate req can come?

1. Sequential: client send the req, server start processing, client got time out but server still processing. User again request , hence, a new entry.
2. Parallel: What if two POST req comes at the same time. It might be because of different browser or req is going on different servers etc.

Approach for Idempotency handling: Using Idempotency key which is UUID(universal unique key).

- Sequential requests: UUID is same for all req
    - Client generate UUID and send in req header, if first ime req, save the key status in db like created → claimed.
    - If duplicate req comes, key already present and claimed, it won't do anything. (STATUS 200)
    - If duplicate req comes, key already present and created(means still processing → conflict), return retry later or already in progress message. (STATUS 409)
    - If a req comes first time, we will add key top db and key status will be created then it will follow normal POST API path. (STATUS 201)

- Parallel:
    - Two req comes, not present in db, we add them and 2 resource gets created. This is a problem,
        - Solution: As always, Locking. Mutual Exclusion.

  Follow up question: What if req is going to different servers with their db. Solution: Use cache before db and connect to all servers, because cache is fast and they have fast synchronisation.

Design HA:

resource: https://www.youtube.com/watch?v=iL7_8TmrePM&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=16

1. Active-Passive
2. Active-Active

Proxy Vs Reverse Proxy

docs: https://httpd.apache.org/docs/2.4/mod/mod_proxy.html#forwardreverse

A Proxy server is a hardware or a software that is placed between the client and the application to provide intermediate service in the communication.

The Proxy server provides a gateway between the user and the internet.

1. Forward Proxy: Client → proxy server → Real Server

   Forward Proxy hides the client identity. Proxy server act as a client for Real server. Proxy server can act as a cache, vpn(means change location), filter requests etc.

2. Reverse Proxy: Load balancer connect with multiple server but we send request for LB ip only. Here server is hiding. Ex: API Gateway, Load Balancer.

   Reverse Proxy hides the server.

R-Proxy vs VPN

- So proxy act as a bypass, it can do caching, logging, and load balancing. So anonymity like it hides the Ip address and VPN can do encryption.
- The proxy can only do anonymity at the Ip address masking, but it cannot encrypt the data. VPN does encryption decryption, and creates a safe tunnel from where the data flows.

Proxy vs LoadBalancer

- The major difference between reverse proxy and load balancer is that reverse proxy can act as a load balancer, and load balancer can not be a proxy. Also, reverse proxy can be used when there is only one server, instead of load balancer.

Proxy vs Firewall

- Firewall is a software program that allows you to define what data can pass through it. It works on the packet level and checks the header, port number, Ip address, and destination address to determine whether to allow the packet or not.
- You can add rules based upon data, and also set up a firewall for each application. A proxy works on an application level, not on packets, and can also block certain requests based upon the rules.
- We can also develop a proxy which can also do a blocking right based upon the rules also, so proxy Firewall we can have right? But proxy Firewall works at application layer, so it is different from traditional firewall which works on packets.

Load Balancer and It's Algorithms

It helps in distributing the traffic appropriately to all the servers so that no single server get overburdened.

Types of LB:=

L4(Network) and L7(Application) Load Balancer

There are seven layer of Osi model, and the application layer load balancer is known as L7 balancer, while the network layer load balancer is known as L4

balancer, and it works at the transport layer. The application layer load balancer can read header session even data, and can do caching.

Algorithms:

All these load balancer algorithms come under the network load balancer type.

Static

- Round Robin
- Weighted Round Robin
- IP Hash

Dynamic

- Least Connection
- Weighted Least Connection
- Least Response Time

HLD Examples:

1. TinyURL system desing: https://www.youtube.com/watch?v=AVztRY77xxA
2. Design Rate Limiter: https://www.youtube.com/watch?v=X5daFTDfy2g&list=PL6W8uoQQ2c63W58rpNFDwdrBnq5G3EfT7&index=13

Other Imp Links:

1. SAGA design pattern: https://microservices.io/patterns/data/saga.html
2. Apache Spark: https://spark.apache.org/