Designing data-intensive applications

The big ideas behind reliable, scalable, and maintainable systems

Contents

CHAPTER 1: Reliable, Scalable, and Maintainable Applications

Preface reason for development of new database over last few decades.

- Big companies like Facebook, Google, Amazon, LinkedIn handling huge volume of data & traffic, which is forcing them to build/innovate new tools to effectively handle data at scale.
- Businesses are becoming more Agile and need shorter time-to-market.
- Free open-source software are becoming successful, and are preferred more than commercial and bespoke custom solutions.
- CPU clock speed is barely increasing, but the multi core & faster network are becoming more accessible leading to parallel processing.
- Raise of IaaS for allowing to build distributed service, spanning access multiple geographies.
- Demand for high availability increasing.

The goal of this book is to help you navigate the diverse and fast-changing landscape of technologies for processing and storing data.

In today's world there are more data intensive application then CPU(computation) intensive applications.

• Store data so that they, or another application, can find it again later (databases)

• Remember the result of an expensive operation, to speed up reads (caches)

• Allow users to search data by keyword or filter it in various ways (search indexes)

• Send a message to another process, to be handled asynchronously (stream processing)

• Periodically crunch a large amount of accumulated data (batch processing)

The three important concerns of any system are

- *Reliability*
- *Scalability*
- *Maintainability*

Reliability: "continuing to work correctly, even when things go wrong".

There cannot a situation where fault never happens; thus, one needs to build a system which is faut tolerant. There are three main types of faults – Hardware Fault, System Fault, Human Fault.

Scalability – Ability of the system to work correctly under load.

Response time – time that client needs to experience to receive a response. The response time is not a single number as each client experience different time to received it response based on different factors (internal – like size, External like loss of network packet, garbage collection etc.) hence it is good to represent the response time as percentile usually represented as p50 (50 percentile), 995 p99 p999 (99.99 percentile).

When there are requests made in parallel, the slowest response time is the one the affect the actual response time as the client needs to waits for the slowest response.

Latency – time it takes for a request to be addressed to be served.

Maintainability, the three main concept of Maintainability are

- *Operability – For the existing operation team to maintain the system. Good operability means having good visibility into the system's health, and having effective ways of managing it.*
- *Simplicity – For the new engineers to understand the system. Right amount of abstraction is the key for simplicity. Finding a right level of abstraction is difficult to identify, however there are algorithms that can help define the right abstraction level.*
- *Evolvability – For engineers to upgrade system in future to newer platform, system or use cases.*

CHAPTER 2: Data Models and Query Languages

The main driving force behind the NoSQL were

- *A need for greater scalability than relational databases can easily achieve, including very large datasets or very high write throughput*
- *widespread preference for free and open-source software over commercial database products*
- *Specialized query operations that are not well supported by the relational model*
- *Frustration with the restrictiveness of relational schemas, and a desire for a more dynamic and expressive data model*

Document database is effective managing documents type objects but its hard to create joins as its needs a top-level object to traverse through.

Schema-on-read – beneficial when there are different types of object in a single collection.

Data locality – when a large number of data that been read together it makes sense to store the data locality

Schema-on-write , implicitly enforce the schema while writing the data.

Graph-Like data, where the data is related with many-to-many relationship. For such kind of data, graph databases are more suited. IN a graph database there are two kinds of Objects – Vertices (also known as node or entities) and Edges (also know as relationship and arcs). Each of the vertices and Edge can have any objects, by cleanly define their relationships between them.

***Semantic Web [conceived in early 2000],** The semantic web is fundamentally a simple and reasonable idea: websites already publish information as text and pictures for humans to read, so why don ' t they also publish information as machine-readable data for computers to read? The Resource Description Framework (RDF) [41] was intended as a mechanism for different websites
to publish data in a consistent format, allowing data from different websites to
be automatically combined into a web of data—a kind of internet-wide "database of
everything."*

Difference Between Graph AND CODASYL?

- *In CODASYL, a database had a schema that specified which record type could be nested within which other record type. In a graph database, there is no such restriction: any vertex can have an edge to any other vertex. This gives much greater flexibility for applications to adapt to changing requirements.*
- *In CODASYL, the only way to reach a particular record was to traverse one of the access paths to it. In a graph database, you can refer directly to any vertex by its unique ID, or you can use an index to find vertices with a particular value.*
- *In CODASYL, the children of a record were an ordered set, so the database had to maintain that ordering (which had consequences for the storage layout) and applications that inserted new records into the database had to worry about the positions of the new records in these sets. In a graph database, vertices and edges are not ordered (you can only sort the results when making a query).*
- *In CODASYL, all queries were imperative, difficult to write and easily broken by changes in the schema. In a graph database, you can write your traversal in imperative code if you want to, but most graph databases also support high-level, declarative query languages such as Cypher or SPARQL.*

Datalog , used previously works similar to triple-store model for storing data with many-to-many relationship.

CHAPTER 3: Storage and Retrieval

The two common families of data storage engines – log-structure storage AND page-oriented storage engines.

In log structure storage engines, every records are logs as entry, and then then there are indexes creates to speed up the search results. Indexes degrades write performance of the DB as it needs to be updates every time the data is written, while improves the search (read) functions.

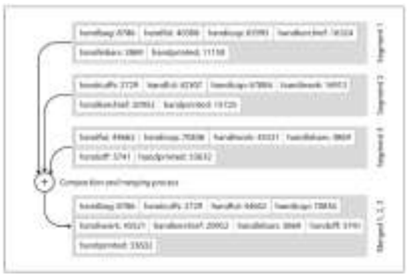
While designing database storage , there are many things that needs to be keep in mind

- File Format , CSV seems good and simple, but binary file format is better.
- Deletion of the records, especially when merging the log segments.This is done by adding a delete key, which helps to discard the records written for the delete key.
- Crash Recovery , if the system crash/restarted the in-memory hash map is lost, the map need to be recreated by reading the segment from the beginning. Here snapshot of each segments comes handy as they are faster to load.
- Partially written data, database maintain a checksum in case of server crash partial written data can be ignored with the help of checksum.
- Concurrency control, the segments are appended in sequential order this helps to give the much needed concurrency control.

SSTables (Sorting String table) and LSM Tree

In case of the SSTable the data are stored in Key-Value pair, stored in sequence sorted by key. Where each key can appear once within each merged segment file which is ensured by the compaction process. There are couple of benefit of storing the key in sequence.

Merging the segment gets simple : each file segment can be read side-by side and based on the smallest key , the file is re-written to a newer segments making the new merge segment sorted. (if a same key appear several time – in each segment there will be only one key, the process is so design if the same key appears while merging the process keep the recent key and discards any values of the older segment.



In order to search a particular key within a segment , all keys need not to be maintain in the read-only memory. As the key are stored in sequence, if a nearest key offset is found then the offset of the other key can be search by scanning the nearby offsets.

How one can construct and maintain a SSTable ?

1. *Maintain a stored structure in memory. Use any balance tree structure to store the keys within the in-memory structure.*
2. *Once the in-memory structure is reaching to its threshold limit , write the data into a segment . Here write operation is simple as the data is already in sorted format.*
3. *While reading , check on in memory structure first , then to the most recent segment then to the older segments.*
4. *Behind the screen run the merge and compaction process for combining the segments and discard the overwritten and deleted data.*

This has a major limitation , if the server is crashed the data written/maintain in the memeory is lost. This can be overcome by maintaining a log where are data will be written in un-sorted order.

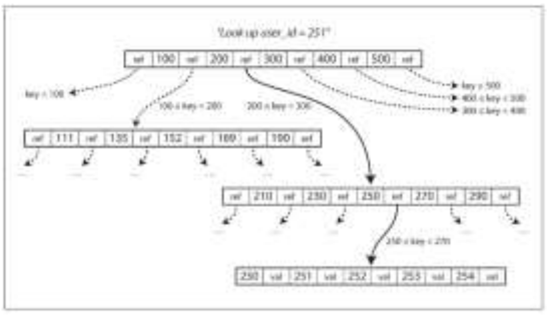
What is LSM Tree ?

Storage engines that are based on this principle of merging and compacting sorted files are often called LSM storage engines. (Log Sorting and Merging Tree)

Basic idea of LSM-trees¬keeping a cascade of SSTables that are merged in the background. Its simple and effective and can handle a dataset which is much bigger than that of the available memory. AND because the disk write are sequential it can handle a remarkably high throughput.

B-Tree

Log structure indexes are not common, the mostly widely used index type are tree-indexes (B-tree). Like SStable B-tree also store data in key-value pair sorted by key. Unlike log structure where the data are stored in segments, in case of the B-tree the database is breakdown into   block size or pages where the data is written to read from the pages one at a time. Each page can be identified with a location/address, one page is allowed to refer another page location in disk (instead of memory).


"Look up user_id = 251"
ref 100 ref 200 ref 300 ref 400 ref 500 ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
key ≥ 500
400 ≤ key < 500
300 ≤ key < 400
ref 111 ref 135 ref 152 ref 169 ref 190 ref
ref 210 ref 230 ref 250 ref 270 ref 290 ref
250 ≤ key < 270
250 val 251 val 252 val 253 val 254 val

Top Page is called Root of the tree, and last page is called leaf where data is read from/write to. The number of references to child page in one B-tee is called branch factor.

For updating an existing value, one need to traverse through the B-tree, and update the leaf page containing the key. For adding a new key, one need to find the page whose range encompasses the new page and add it to the page. If there is not enough space add the page, then split the page into two half the parent page is updated to account for the new subdivision of key ranges. This is done to keep the tree in balance, for n-keys there are is always O(log n) depth is maintain.

To make the B-tree more reliable, WAL (write ahead log aka redo.log) is introduced, where the all changes are written to append only file, before making the changes in an event of an crash the data can be re generated by replying the redo logs.

NOTE : There are several other optimization been done to improve the redo log efficiency , like in some databases like (LMDB), the new pages are written to a newer location, and a new version of the parent page is created pointing to the newer location. This approach is useful for concurrency control. Also, for the interior pages where the primary need to find the range of the reference pages, packing more keys into a page helps in getting higher branching factor.

Also, each of the page is made to store the information of its sibling page to its right and left, this allows scanning key in order without jumping back to the parent pages.

Additionally, factor-tree borrows some of the additional log-structure ideas to improve the disk seeking needs.

Comparing B-Tree with LSMTable

B-tree storage engine is faster to read while LSMTable based storage engine are faster to write.

LSMTable provide better write throughput for sequencies data write on B-tree as B-tree need the data to be written twice one in redo logs and other in pages, also writing into page are closely then writing into segments as in case of the LSMTable.

NOTE: In case of LSMTable compaction process can impact the read write throughput of the storage engine.

In case of the B-tree each key is existing only one in one place, unlike in case of the LSMTable where the key is spread across multiple segments. This feature of the B-tree to help maintain a strong transactional semantics, which is more desired for a relational database.

While B-tree storage engines are ingrained to the database architecture, while with the newer databases log-structure indexes are getting increasingly popular in newer databases. There is no hard and fast rule to identify the best storage type for a given use cases.

Storing value in indexes

Primary indexes: A primary key uniquely identifies one row in a relational table, or one document in a document database, or one vertex in a graph database.

A secondary index can easily be constructed from a key-value index. The main difference is that keys are not unique; i.e., there might be many rows (documents, vertices) with the same key. This can be solved in two ways: either by making each value in the index a list of matching row identifiers (like a postings list in a full-text index) or by making each key unique by appending a row identifier to it. Either way, both B-trees and log-structured indexes can be used as secondary indexes.

Indexes can store the either of the two things – actual value (row/document / vertex) etc. OR the reference to the row storing the actual value somewhere else know as heap file.

Cluster indexes – when hoping form index to heap file is more expensive then, cluster indexes are used, they use to save index rows directly within indexes. These are known as cluster indexes.

Concatenation Indexes, unlike primary key and secondary key indexes where only a single key is considered, in case of the concatenation indexes multiple keys (column) are appended to one another to create a single key. Example for storing geospatial data where longitude and latitude information are often combined together, in an ecommerce website where combination of red/blue/green can be combine together to search a product of a particular colour.

Full text Search and Fuzzy indexes

In case of the full text searching, and fuzzy indexes instead of searching with exact value of the key or the range of the keys, it allows a search for one word to be expanded to include synonyms of the word, to ignore grammatical variations of words, and to search for occurrences of words near each other in the same document, and support various other features that depend on linguistic analysis of the text.

In Memory database

As the cost per GB of RAM is going down, it’s possible to have entire database to be loaded into RAM as some of the datasets are not that large enough AND can be managed to be store in memory. AS memory is much better in managing data then disk, it largely improves the performance of the databases.

OLTP (online transaction processing) , OLAP (online analytics processing) .

Mostly of the databases are now, planning their products target to either to OLAP or OLTP but not both. Some of the databases does support both using a same SQL interface but they have different storage engines and query engines.

Star-schema: where the main table is surrounded by multiple dimension schema then, it popularly known as star schema.

Snowflake schema is the variation of the star-scheme where the dimensions table are further broken down into subdimensions tables. Snowflake schemas are more normalized than that of the star-schemas.

In OALP databases, usually all columns are query together hence it would be more performance oriented to have the data stored (group) in columns than it usually stored in OLTP in from of rows.

Memory bandwidth and vectorized processing. *Its important to efficiently using the bandwidth from main memory into the CPU cache, avoiding branch mispredictions and bubbles in the CPU instruction processing pipeline, and making use of single-instruction-multi-data (SIMD) instructions in modern CPUs.*

Since the OLAP data are stored in columnar fashion, it's not stored with sorted column instead its been sorted using the target column so that entire data is sorted. Another advantage of the sorting is , if the primary column done not have too many unique values then after sorting there will be a long-sequence of the same value been repeated multiple times which would help in compressing the data into a much smaller space, like in case of the bitmap.

The compression technique makes the reading data more effective, but while writing data in the middle of the column then possibly all the column data needs to be rewritten.

Not all the data warehouse keeps their data in column store, there are other storage view which are often used to boost performance. One of these is Materialized view where the data are stored as standard (virtual) view – where a table like object where the content are some kinds of a query. When a standard or the virtual view needs to be read , the sql expands into a the view's underline query and then process the underline query. BUT when the underline data is changed the view needs to be updated , the database can do this automatically but its bit expensive to do so hence its not used in OLTP databases.

The main advantage of the Materialized cube is that some of the queries run much faster as its precomputed.

CHAPTER 4: Encoding and Evolution

In this chapter we are going to look through the different format for encoding including – JSON, XML, Protocol Buffers, Thrif and Avro

The data are usually been represented in two different forms

1. ***When the data is written within a memory:*** *Within memory the data are represented in objects like struct List, hash tables, tree etc. which are optimized for effective access and manipulation by the CPU.*
2. ***When the data is transmitted within a network or written to a file:*** *Within network or while its written to a file, the data needs to be represented by some self-contained sequence of bytes.*

Both the forms are quite different hence they need some kind of translation within these two formats. The translation from the in-memory data to sequence of bytes is called as encoding (aka. marshalling or serialization), and the reverse process is called decoding (aka unmarshalling, de serialization).

Language specific encoding often tried down a particular language, they also need to be instantiate to an arbitrary classes which can be hacked to elevate for remote execution, versioning of the data is always an afterthought for these libraries which makes forward and backward compatibility difficult, last but not the least language specific encoding are not optimized for CPU efficiency. Hence XML, JSON and CSV are the common form of representation of the language independent data format which are widely used. They too have issues like

- *JSON format cannot correctly represent large number (number larger then 2^{53}) in IEEE 754 double-floating point precession format.*
- *JSON and XML format works well with the Unicode character set, however when used for binary data using BASE 64 encoding it bump up the data size by 33%.*
- *JSON and XML supports schema, but the applications that done used JSON or XMLs need to hardcode appropriate encoding and decoding logic.*
- *CSV have a vague format.*

Json is less verbose as compare to XML, but in comparison to any binary format its still usages lots of spaces. This leads to the development of profusion of binary into JSON messages like BSON, BJSON, UBJSON, BISON formats, similarly for XML we have WBXML etc. Though for few datatypes binary JSON extends additional datatypes but for large part it keeps the JSON/XML datatypes unchanged.
Also, since the schemas are not prescribed, all the object field names are included within the encoded data.

Thrift and Protocol buffers
Thrift is a binary communication protocol and interface language that has been developed by Facebook, it supports two different binary encoding format – compact protocol and binary protocol.

Protocol buffer was developed by Google and unlike Thrift it only supports binary protocol – binary encoding format.

Schema evolution: Forward compatibility means newer version can be used as a writing while older version can be still use for reading. Backward compatibility means older version can be used for writing and newer version can be used for reading the value.

In case of the Avro there are two diffident schema – reader schema and writer schema. The reader and writer schema need not to be same, they need to be compatible.

- *The reader schema and writer schema fields can be re-arranged in different order , the reader schema would match the fields using the field name and arrange it accordingly.*
- *If the field is missing in reader schema and present in writer schema then the field is ignored while writing.*
- *If the field is missing in writer schema and present in reader schema then the field is read with the default value define in the reader schema.*

Dataflow

When the data need to be shared from one process to another which does not share the same memory then one of the following processes can be used

1. *Via database*
2. *Via service call*
3. *Via asynchronous message passing*

Dataflow through databases: *the data are written into database using multiple version, the data written into a database, using a older version may not include the newer field while writing new data.*

When the data are written by new version of the code, the older data still remain in the database unless they are rewritten – this is known as outlives code.

While taking a snapshot of the data, the data dump will be typically encoded with the newer schema, even if the data is from older schema. It’s better to use formats like Avro object container files to encode the data in an analytics-friendly column-oriented format such as Parquet.

Dataflow through services (RPC or REST)*: In case of dataflow through services – there is a server and a client, and the data flow between then in a mutually agreed format. Server can be a client to any other service. (This way of building application is calling as SOA, recently it’s been rebranded as microservices).*

Each of the services are called as WebService – there are two popular form of webservices protocol for communication between services SOAP and REST. SOAP uses WSDL to established contract between server and the client , while REST there is no specific format – recently OpenAPI (a.k.a swagger) is becoming famous.

The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process (this abstraction is called location transparency).
Problem with RPC

- *RPC call passes through network, which is un-predictable*
- *If RPC call response timeouts you would not know, what happened – failed completely or just response failed.*
- *Retrying PRC call is difficult, as its difficult to predict if the request is timeout due to failure ore response failure, hence one need to build idempotency to retry the request without duplicating the request.*
- *In case of the RPC procedure call, the parameters need to be pass as sequence of bytes , which is fine for primitive datatypes but becomes problematic when it comes to large objects.*
- *RPC client and services implemented in different languages – case problem in translating the datatypes from one language to another, as same datatypes may not be available on both the languages.*

For these reasons, REST seems to be the predominant style for public APIs. The main focus of RPC frameworks is on requests between services owned by the same organization, typically, within the same datacentre.

Message based communication for asynchronous message-based data passing. There are several advantages of message based data passing over service based data passing.

- *It can buffer the message if the recipient is unavailable.*
- *It can redeliver the messages if the recipient process is crashed.*
- *Allows one messages to be sent to multiple recipients.*
- *Logically decouple sender from its recipients.*

Distributed actor model (framework): *Actor model is a programming model for concurrency in a single process. Rather than dealing with threads, logic is encapsulated in actor logic. Each if the actor represented an entity or a client, they can have their own local state which is not shared with other actors, each actor can communicate with another actor using synchronous messages. Message delivery is not guaranteed, each of the actor is schedule independently and need not to worry about threads.*

Distributed actor framework*, the programming model is used for scaling application across multiple nodes, same message parsing mechanism is used to send message running on a same node or a different node – if it needs to send a message to a actor running on a different node, then message will be encoded into Byte sequence and sent over the network, and decoded on receiving on the targeted node. Usually, message broker is integrated with the distributed actor frameworks.*

Popular Distributed actor framework are

1. *Akka framework*
2. *Orlean framework*
3. *Erlang OTP framework*

CHAPTER 5: Replication

The reason for replication

- To keep the data closer to the user (data geographically close to user).
- To Keep the system continue to operate even if some of the components / parts failed (to increase high availability)
- To improve the system read performance (throughput).

Three common replication algorithms are – single leader, multi leader and leaderless.

Leader-follower

- The client sends the data to the leader node to write, once the leader node writes the data into its local store.
- The leader node then sends the data to its follower as replication log/ change stream for the follower to replicate the data in to the follower node in the same order as its processed by the leader.
- When client want to read the data, it can read it from the leader or from any of its follower, but when it need to write the data it needs to send it only to the followers.

Replication mode

- Synchronous replication – leader and the followers need to wite the data synchronously, if one of the node failed , the entire write operation will be halted until all the follows replicate the data within their nodes.
- Semi-Synchronous replication means leader and one of the follower need to write the data synchronously, while the remaining follower replicate the asynchronously.
- Asynchronous replication means the leader write the data and follower asynchronously replicate the data within their nodes.

Failures

Follower: Fails: If a follower fails, its restarted and then sync up with the leader from the last know of its successful transactions.

Leader: Fails: If a leader fails, a new leader needs to be identified from the active followers. The follower with the latest transaction qualifies. It can be done automatically with consensus or by manually identifying a follower as a leader. Finally, the clients write requests needs to be routed to new leader.

Problem with failover process

- In case of asynchronous replication, if a leader fails and followers have not completed their replication then the new leader will be losing some of the earlier committed transactions. In case the leader is back quickly, it will create a discrepancy between the new leader and old leader. Usually, in this situation its old leader transaction are ignored, this may violate client durability needs.
- In certain scenarios, two of the followers are simultaneously promoted as leader which can lead to a problem call as *"split brains". Usually, this can be resolved y shutting down one of a leader.*
- *If a leader is incorrectly considered as offline, if it done too quickly then it causes un-necessary failover. If it delayed there is a changes of missing transaction data.*

How replication happens?

- *Statement base replication*: In leader log all statements, every write statements is send to its follower. In case of the relation database INSERT, UPDATE, DELETE statements are forwarded to the follower node for replication.
  Problems with type of replication are
  - Any statement like NOW () [to get the current system date], RAND () [generate random number] going to have a different value.
  - Any autoincremented column or any condition that depends on the existing value needs to be executed in the same order, else it would result in different output.
  - Statement side-effects (trigger, user define functions, store procedures) may results in side effects when running in different nodes.
- **Write-ahead log (WAL) shipping**: In both the type of storage engines – SSTable and LSM Tree as well as B-tree storage engine, the logs are first written to WAL which leader can shipped (sends) it to its follower to replicate. Follower replicates the data at the disk level. However, this tightly couple the replication with the storage engine type. The database can't change from one storage engine to another with this type of replication.
- **Logical (row-based) replication:** In this type of replication is decoupled from the storage level log replication instead the replication is performed at the granularity of the row. Each time a row changes in the leader node, leader node send the data to follower node
  - That can uniquely identify the newly inserted row

    - That can uniquely identify the deleted row (in case of primary key, primary key is sent else all the column of the deleted row is send.)
    - That can uniquely identify the updated row (in case of primary key, primary key is sent else all the column of the deleted row is send.)
- **Trigger based replication**: When one needs bit more flexibility in replicating data, like replicating a subset of the data or replicating data to another type of a database or it needs for conflict resolution then trigger based replication is helpful. This can be achieved, using a different too or using database triggers and stored procedures. This type of replication is more bug prone but have high flexibility.

Problems with the Replication Lag

Another reason for adding replication is to scale the database, usually the read operation is much higher than write operation. Redirecting the read operation away from the Leader node helps scale better.

Since the follower nodes will be lagging behind the Leader node, if client queries both the nodes (leader & follower) there can find some discrepancies due to replication lag, which will be eventually cleared – hence its known as eventual consistency.

The replication lag causes due to following:

- ***Reading your own writes****: When a user writes some data into a database, and try to reload the data, in this case if the read is performed from another node, then the data the user have just written will not be visible. To prevent this scenario following measures can be taken*
    - *When reading own written data, read it from leader node and for reading data written by another user read it from the follower node. For example, in case of social media profile update – when user is reading its own profile read it from the leader node, when user is reading other profile read it from the follower node.*
    - *Client is looking for the last updated timestamp from the follower node then there is chance for reading stale data due to replication lag, If client maintain its own last write timestamp instead of reading it from the database, then this can be overcome.*
    - *If the nodes are geographically spread across different datacentre, then all the queries routed to the leader node, needs directed to the datacentre where leader node is hosted.*
- ***Monotonic Read:*** *When the client query multiple times and in each of the time, the query is been severed by a follower which has more replication lag then the previous follower. This may appear like client is going back in future. This problem can be addressed (not fully) if user is made to read data from a fix replica every time it request to read a data. This can be achieved by selecting the replica node based on user hash value instead of selecting it randomly.*
- ***Consistent Prefix Read:*** *if there is a sequence is writing operations, then read operation should follow the same sequence to maintain the original sequence.*

With strong consistency, the problem with Replication lag can be reduce. Transaction on a single node is easy to achieve, however for distributed nodes its difficult to implement transactions.

Multi Leader Replication

Muti leader replication is usually performed on geographically separated datacentres, in a single datacentre it's no helpful. The following are the benefits of the multi leader replication.

Performance – latency will be reducing to replicate on the follower node as the follower nodes will ONLY replicate from their local datacentre leaders.

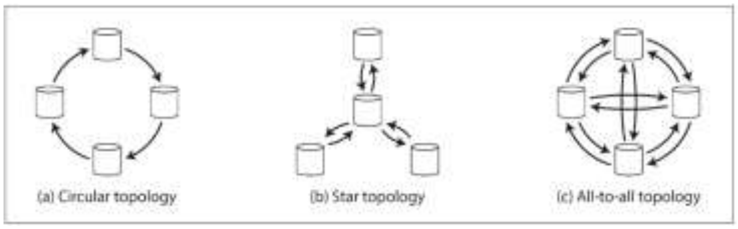
Better fault tolerance over datacentre outages – as there are multiple leaders, each leader will be working independently thus improving on fault tolerances.

Better fault tolerance over network outage – since a large part of the replication would be done within data centres

Conflict resolution is a key when there is multiple leader the following are the strategies to resolve conflicts.

- *Give every write a unique ID (timestamp + long unique UUID + hash key + value), in case of conflict honour the highest ID value. This strategy is known as, last-write-win.*
- *Order each replica with a number, and let the highest number replica always win. This can lead to some amount of data loss.*
- *Define logic to some-how sort the write in any specific order and resolve the conflict automatically.*
- *Record the conflicts on an external data structure and let the user resolve it at later point in time.*

Different types of Replication Topologies



The drawback of circular topology is if one node fails, the replication is broken until the nodes are reconfigure, all-to-all topology is best here because even if one node fails, the other nodes continue to operate.

Leaderless replication

In case of leaderless replication, when a write is performed it sent to all the node, the write is considered successful when all the nodes (or a threshold of nodes) respond it as successful.

While reading the data from a leadless replication, client needs to send read request to multiple nodes, based on the response from multiple nodes, the data with the latest version is considered.

In case of the leaderless, the probability of reading the stale data is calculated using the below formula

$r + w > n$

where n is the number of nodes, where w is number of nodes that needs to confirm for a write to be successful and r is the number of nodes that needs to be query for a read operation. If r and w follows the read and write values then it's known as quorum read. Usually, the optimal values is calculated as w= r = (n+1)/2.

Even If r + w < n then there will be high probability of state data read, under following conditions does not guarantee the quorum will return latest values

- *Sloppy Quorum, where w + r > n but still there is a state data in the read, this occurs when w ends up to a different node then r and there is no longer any guaranteed overlap of w and r.*
- *When there is concurrent write, and the writing operation is selected based on latest timestamp the previous write may have been lost.*
- *When the write operation is successful in fewer replicas w, and fails to be removed from the successful replicas.*
- *When the replica was replicated from an old version, when it already has latest version.*
- *Under certain edge scenarios where one is unlucky with the timing, these are known as linearization and quorum.*

Concurrent operations are not the operation that happens in the same time, instead they are the two operation which is unaware of each other.

How to effectively store data for concurrently running processes. One of the way to achieve this is using Version Vector – is a collection of version numbers, client needs to send version vector to the server every time a data is written and server will be sending version vector every time the data is read, thus effectively tracking overwrites and concurrent writes.

CHAPTER 6: Partitioning

It’s not enough for a very large dataset to have multiple replicas to meet their required throughput benchmarks, with replication of data its also important the data needs to be partitioned so that query load can be distributed across multiple loads over a relatively smaller data store. Each partition database is a database withing itself, a single query on a specific portioned data can be successfully executed on a single node, even a complex query can be made to run in parallel across multiple nodes with some effort.

The main goal of partition is scalability.

Always the replication and partition are use together, hence a single partitioned data can be store across multiple nodes for fault tolerance. While its possible, a same mode can be leader for particular partition and follower for another partition. *However, the choice of the partition scheme and replication scheme is not interdependent.*

Its important that the partition data are store evenly across all the nodes. There are few ways to achieve this

Partition by key range, in this approach entire data’s key is assigned a unique node, and their related data are stored only on these nodes. However, this type of partition are not effective, as the data are evenly distributed across each nodes.

Partition by hask key, in this style of hashing the key is hashed and based on the hash value partition is identified. The downside of using this technique is , all the sorting order is lost, now for range queries one need to query all the shards at a same time.

When partitioning based on the hash key, if a single value has increasingly large number of values the hash key fails to prevent skewing resulting in Hot node, for which a random number can be appended to distribute the values across different nodes.

Secondary indexes usages in relational database are common, but they are also used in document database too.

However, searching with secondary index is costlier when the data is partitioned into more than one node, as it needs to combine the data before returning it back to its client.

Instead of local indexes within each data partition, term-partitioned indexes offer a global approach for searching across all partitions based on specific terms. While a single index node becomes a bottleneck, partitioning the index itself avoids this while maintaining flexibility for different data distributions.

A secondary index also needs to be partitioned, and there are two methods:

• Document-partitioned indexes (local indexes), where the secondary indexes are stored in the same partition as the primary key and value. This means that only a single partition needs to be updated on write, but a read of the secondary index requires a scatter/gather across all partitions.

• Term-partitioned indexes (global indexes), where the secondary indexes are partitioned separately, using the indexed values. An entry in the secondary index may include records from all partitions of the primary key. When a document is written, several partitions of the secondary index need to be updated; however, a read can be served from a single partition.

Rebalancing Partitions

The process of moving data across the nodes of the cluster is call rebalancing, the main objective of replacing is to evenly distribute the load across all the nodes of the cluster. The main requirements of rebalancing are

- After rebalancing the data needs to be fairly distributed across all the nodes of the cluster.
- While rebalancing happing, the system should be excepting read and write requests.
- No more data than what is required shouldn’t be move as part of the rebalancing act, as data movement are costlier.

Different strategies for Rebalancing:

In range value partition, based on the predefine range of primary key are assigned to a specific node, this type of partition is called as range value partition.

For assigning the node, the easiest approach is to assign key using the following approach, **partition key = hash (Key) % n** , where n is the number of nodes in the cluster. This type of partition are called as *HashValue* partition. The problem with this approach is as the number of node increases/decreases, the partition key changes. This partition is not suitable for growing data. Splitting of the partition is complex, hence many databases does not perform partition Splitting.

Another approach is, to assign partition dynamically, start with single partition once the data grew to a specific threshold the partition is separated into halves, and so forth. This approach can be adapted into range-value partition as well as hash value partition.

Rebalancing can be done in a fully automated way or can be done by manual intervention, it's an expensive task, it better been done under manual intervention.

There are three distinct approaches when routing an incoming request,

1. Allowing all client to contact all nodes, if the node is serving the request from its hosted partition then server the request else forward the request to another node.
2. Let the client contacted routing tier, the routing tier is responsible for assigning the correct node and correct partition to the client request.
3. Client been ware of the partition and node details and they themselves route the request to the correct partition or node.

CHAPTER 7: Transactions

In a database many things can go, thus transactions are device to ensure *safety guarantee. Transaction safety guarantee makes the application code look simpler by taking care of the complexity of the transactions within databases, but this cause considerable drop in the database performance.*

ACID – Atomicity, Consistency, Isolation, and Durability. The system that does not meet ACID criterion are called as BASE – Basically, Availability, Soft state, and Eventual Consistency.

The problem with read isolation lock – at the object level there can be a lock mechanism implemented, each transaction needs to acquire the lock before making any change to the object this helps in preventing concurrent write to corrupt the data, however this doesn't help in preventing dirty reads. Read transactions still able to read uncommitted (partially committed) data, to overcome this if the lock is extended to read transactions also, then this would significantly impact the system performance as all read transaction will be blocked by long running write transactions for acquiring locks. To overcome this, the database would maintain two values of the same object (before change value & after change value), when a read transaction is looking for a value of an object which is under lock state, it returns the old (before change) value of the object instead of holding the read transaction to acquire the lock. This server two benefits – clean read, no performance bottleneck.

Read Skew – during a temporary phase the database is inconsistent, this is known as read skew. Though read skew is acceptable in many cases, here are some of the case where read skew is not acceptable

- While taking backups
- While running reconciliation queries (analytical query / integrity check)

Read Skew can be prevented using Snapshot isolation, where database maintains a consistent snapshots and same snapshot is used for all read considerations, keep the database in a same consistent level. Since this approach needs to maintain multiple version side-by-side its also know as MVCC (Multi version concurrency control).

In case of MVCC, there will be multiple snapshot version need to be maintain for the write consistency as there would be multiple transaction is in progress sate. Each of these transactions are numbered with unique transaction id, in incremental fashion. Every database defines their own transaction visibility rules, based on these (generic) rules a given transaction is visible or invisible.

1. Database, maintains a list of in-progress transactions (not committed or aborted transactions) at any given point in time, all these transactions are ignored.
2. Any write made to the aborted transactions
3. Any write made to the transaction which has higher transaction ID are ignored.

These generic rules are applied to both creation and deletion of objects.

How indexes work in MVCC?

One of the options is to have the index filter out the version that are not visible. This way consistency can be achieved. However, different databases implement their own approach to achieve the same.

Lost updates arise when concurrent transactions try to read and modify the same data, potentially overwriting each other's changes. This section explores various techniques to prevent this:

Atomic Write Operations:

- Databases provide built-in operations that update data without allowing concurrent reads, eliminating the need for manual read-modify-write cycles.
- Examples include incrementing counters in relational databases or modifying parts of JSON documents in document databases.
- This is the optimal solution when code can be expressed using these operations.

Explicit Locking:

- For scenarios where atomic operations lack functionality, applications can explicitly lock objects before updating.
- This ensures other transactions wait until the locked object is free, preventing concurrent modification.
- Useful for complex updates like multiplayer game moves where rules validation is needed beyond database queries.

Automatic Lost Update Detection:

- This approach allows concurrent read-modify-write cycles but identifies and aborts transactions causing lost updates.
- Snapshot isolation levels in Postgres, Oracle, and SQL Server implement this automatically.
- Requires no special application code but may not be available in all databases (e.g., MySQL).

Compare-and-Set:

- In databases without transactions, compare-and-set allows updates only if the data hasn't changed since the last read.
- If the value differs, the update fails, and the cycle needs to be retried.
- Useful for scenarios like concurrent wiki page edits, but the safety depends on the specific database implementation.

Conflict Resolution and Replication:

- In replicated databases with copies of data on multiple nodes, preventing lost updates involves additional considerations.
- Locks and compare-and-set don't work well due to potentially asynchronous updates on different nodes.
- Conflict resolution techniques merge conflicting versions of data ("siblings") created by concurrent writes.
- Atomic operations can work if commutative (applying them in different orders yields the same results).
- Last-write-wins approach is prone to lost updates and often the default, requiring careful use in replicated settings.

Write Skew when multiple concurrent transactions modify two different objects resulting in db. constraint violation. like doctors taking leave from hospital duty. Both initially see two doctors on call (snapshot isolation) and proceed to remove themselves. This results in no doctors on call, violating the requirement of at least one.

Phantoms arise when a write in one transaction alters the results of a subsequent search query in another.

Write skew and **phantoms** are subtle race conditions beyond lost updates in concurrent writes. Preventing them requires careful consideration of isolation levels, explicit locking, and potential complexity of conflict materialization. Serializable isolation offers the best protection against both, though alternative approaches may be suitable depending on the specific scenario and trade-offs.

Serialization

It's hard to achieve isolation, there are certain scenarios which are even harder to prevent which are raised due to isolation problem.

Serializable isolation is usually regarded as the strongest isolation level. It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, serially, without any concurrency. Thus, the database guarantees that if the transactions behave correctly when run individually, they continue to be correct when run concurrently—in other words, the database prevents all possible race conditions.

There are different ways to implement serialization.

- Execute transaction in serial order.
- Two phase locking
- Optimistic concurrency control technique such as snapshot isolation.

Single loop execution: Until 2007 there was no clear way to do this , this was mostly possible because of the following two factors

- The RAM became cheaper, now it's possible to load the full dataset into memory and execute transaction becomes faster as compare to loading the transactions from disk.
- It's been recently realized that the OLTP transaction are short and only makes small number of read & write transactions as compare to OLAP transactions which are mostly long running read only queries.

In order to achieve single threaded design for running transaction in serial then instead of making multiple follow-up transactions, entire logic needs to be developed as a store procedure and execute as a single short transaction.

With stored procedures and in-memory data, executing all transactions on a single thread becomes feasible. As they don't need to wait for I/O and they avoid the overhead of other concurrency control mechanisms, they can achieve quite good throughput on a single thread.

Here are they pointer to use serial execution of the transactions

1. Transaction needs to be small and fast , single slow transaction can stall all the transactions.
2. It should be limited to those use cases where the entire dataset can be loaded to the memory & rarely access the data from the disk.
3. Transactions should be limited to run on a single partition, if the transactions needs to span through multiple partition then cross partition coordination is required which has its own hard limit.

Two Phase Locking

Over 30+ year two phase locking was the only available. The 2 phase locking is different from the snapshot level isolation where write transaction don't block read transactions and vice versa , in case of 2 phase locking read transaction blocks following write transactions and vice versa. This provides the required support needed for serialization.

Working for two phase locking.

In case of the two-phase locking there are two types of locks – shared lock and excusive lock, read transactions need to acquire a share lock, while a write transaction needs to acquire an exclusive lock. The process of acquiring the lock is consider as a first phase, and process of releasing the lock is consider as the second phase hence its named as two-phase locking.

Two phase locking has to acquire lock and then release the lock it adds additional operational overheads, on the top these overheads the main performance overheads is due to the wait that a transaction needs to undergo due to other transactions holding the lock, hence the two phase lock is having significant overheads, as compare to an weak isolation.

Serializable Snapshot Isolation

Serializable Snapshot Isolation (SSI), a promising algorithm. It guarantees serializability – ensuring transactions appear to execute one after another – but with minimal performance overhead compared to snapshot isolation. Its comparatively recent development as compare to other serializable options.

Optimistic Vs Pessimistic Concurrency Control

In case of the pessimistic concurrency control assumes potential conflicts and acquires locks to prevent them.

While in case of the optimistic concurrency control transactions proceed without blocking, hoping everything will work out. If anomalies arise during commit, the transaction is aborted and retried.

SSI builds upon snapshot isolation, where all reads within a transaction get a consistent view of the database from a specific point in time. This differentiates it from earlier optimistic techniques. On top of snapshot isolation, SSI adds an algorithm to identify and abort transactions violating serializability by conflicting writes.

SSI is effective in both the Write Skew and Phantoms scenarios.

In case of the Sate MVCC read scenario, SSI ensure if there is any write transaction that's committed before any read transaction commit, it aborts the read transaction and hence preventing the stale reads.
In case of the write affecting prior read scenario, SSI ensure by tracking reads using indexes or table-level information. Writes then check for affected reads and notify them, potentially triggering aborts if conflicting writes have committed.

SSI strikes a balance between the guarantees of serializability and the performance of snapshot isolation. Its optimistic approach minimizes blocking while detecting and resolving potential anomalies through aborts. While still under development, SSI demonstrates promising potential for future widespread adoption due to its efficiency and accuracy.

The SSI works best with short write transactions and longer read transactions, with minimum transaction abort scenarios.

CHAPTER 8: Trouble with Distributed System

Fault and Partial Failures

In building a distributed system, there are many things that can go wrong. One need to consider failure and partial failure when designing a distributed system.

Unreliable Network

UDP vs TCP – where there is a trade-off between the reliability vs variability of delay then it's advisable to have UDP over TCP as UDP does not do flow control or retransmit lost package.

Unreliable Clock

The clock within a machine is a hardware device, it uses quartz crystal oscillator. It can't be perfectly synchronized within a distributed system, however in distributed systems NTP (network time protocol is used sync the time based on group of servers.

In a distributed system there are two different clock – time-of-the-day clock and monolithic clock, time-of-the-day clock represent the time in milliseconds from epoch time (1 Jan 1970 UTC).

In case of the monolithic clock, it the time lapse between two checks. It's tied up to the CPU, and does not make any sense to compare two monolithic clock time from two different system as they are not the same thing, however NTP tries to synchronized the lock by compensate the frequency of the clock, but as such it doesn't now move forward / backward the clock time. Because of this reason, its OK to consider the monolithic clock to measure time elapsed in a distributed system.

Monolithic clock doesn't need any synchronization.

Day-in-a-time clock needs to be synchronization, as per the google research, as much as 17 sec a clock can differ in 24 hours, hence it needs to be reset from NTP time.

MiFID II draft European regulation for financial institutions requires all high-frequency trading funds to synchronize their clocks to within 100 microseconds of UTC, in order to help debug market anomalies such as “flash crashes” and to help detect market manipulation.

Sudden Pause

A process may pause for a substantial amount of time at any point in its execution
(perhaps due to a stop-the-world garbage collector), be declared dead by other nodes, and then come back to life again without realizing that it was paused.

Knowledge, truth and lies related to fault there are different type of fault, it's at times difficult to understand there is a fault occurred as in case of the distributed system there is no global variable, no shared memory, no common knowledge or any other kind of shared state between the machines. Nodes can’t even agree on what time it is, let alone on anything more profound. The only way information can flow from one node to another is by sending it over the unreliable network. Major decisions cannot be safely made by a single node, so we require protocols that enlist help from other
nodes and try to get a quorum to agree.

However, it's also need to understand that, more than the stability of the single node, it equally important in these days one need to consider distributed system for fault tolerance and low latency by placing the data geographically closer to the user.

CHAPTER 9: Consistency and Consensus

Linearization (Strong Consistency, immediate Consistency, external Consistency) : The basic idea behind linearizability is simple: to make a system appear as if there is only a single copy of the data. A hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability. Other kinds of constraints, such as foreign key or attribute constraints, can be implemented without requiring linearizability

CAP is sometimes presented as Consistency, Availability, Partition tolerance: pick 2 out of 3. In reality, the network is not reliable, hence we have to make a choice between Consistency and availability.

There is a deep connection between ordering, linearization, and consensus.

Lamport timestamps – where each of the node is assigned with a unique number, and the Lamport timestamps is combination of the node number and timestamp e.g. timestamp + nodeID. Even for event that are occurred at a same time (having same timestamp value), can be easily ordered based on the nodeID. One with the greater NodeID will be consider as higher.

Lamport timestamps does not help in preventing problem like finding uniqueness – as for finding uniqueness one also need to know the order in the final state. Lamport timestamps only helps to establish the total order. That is where total order broadcast helps.

In distributed systems, data is often replicated across multiple nodes for fault tolerance and performance. However, this replication can introduce challenges in maintaining consistency.
Eventually Consistent Systems: These systems allow for temporary inconsistencies between replicas, which can lead to clients seeing different data depending on which replica they access. While this can improve performance, it can also create confusing and unpredictable behaviour.
Linearizability: This concept offers a stronger consistency model that guarantees all clients see the same data, as if there were only a single, atomic copy. It ensures that operations appear to execute one after another in a global order, even when multiple clients are interacting with the system concurrently.
Key Characteristics of Linearizability:

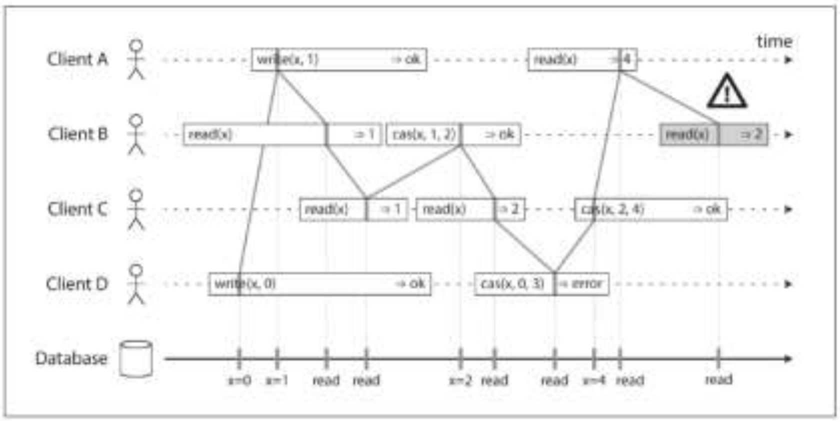
- *Single Copy Illusion*: Clients perceive a single, consistent view of the data, even though it's replicated across multiple nodes.
- *Recency Guarantee*: Reads always return the most recent value written, even if that write was only just completed.
- *Atomic Operations*: Operations appear to execute instantaneously at a single point in time, without any intermediate states visible to clients.

Timing Dependencies of Linearizability:

- *Concurrent Reads and Writes*: If a read operation overlaps with a write, it can return either the old or new value.
- *Strict Ordering:* Once a client reads the new value, all subsequent reads must also return the new value, even if the write operation hasn't fully completed.

Visualizing Operations:

- *Timing Diagrams*: The below diagrams illustrate operation timings and dependencies, helping to visualize linearizability.



- *Atomic Execution Points*: Operations are visualized as taking effect atomically at specific points in time.
- *Forward-Moving Progress*: The lines connecting operation markers in the diagrams always move forward in time, enforcing the recency guarantee.

Linearizability in Practice:

- *Testing for Linearizability*: While computationally expensive, it's possible to test a system's behavior for linearizability by recording request/response timings and checking for valid sequential ordering.
- *Implementation Challenges*: Achieving linearizability in distributed systems often involves complex algorithms and careful coordination between nodes.
- *Performance Trade-offs*: Linearizability can sometimes come at the cost of performance, as it often requires stricter synchronization and coordination.

Linearizability provides a powerful consistency guarantee for distributed systems, ensuring that clients always see the most up-to-date data. While it can be challenging to implement, it's essential for applications that require strong consistency and predictability in their data interactions.

For following conditions Linearizability is important

Locking and Leader Election:

- Single-leader replication systems: Linearizability guarantees a single, unambiguous leader, preventing "split brain" scenarios where multiple nodes believe they're the leader.
- Distributed locks: Implemented using coordination services like ZooKeeper or etcd, they rely on linearizable operations to ensure consensus on lock ownership.

- Granular locking in databases: Oracle RAC uses linearizable locks at the disk page level for efficient shared access to storage.

Constraints and Uniqueness Guarantees:

- Enforcing uniqueness: Creating users with unique usernames or preventing duplicate files in a file system requires linearizable operations to ensure consistency across concurrent requests.
- Username registration as "locking": Conceptually similar to acquiring a lock on a chosen username to guarantee its exclusivity.
- Data integrity constraints:
    - Preventing negative bank account balances
    - Ensuring accurate stock levels
    - Avoiding double-booking of seats
    - These constraints rely on a single, agreed-upon value for critical data, necessitating linearizability.

Loosely Interpreted Constraints:

- Flexibility in some cases: Applications might tolerate temporary inconsistencies or offer compensation for issues (e.g., overbooked flights). Here, linearizability might not be strictly required.

Relational Databases:

- Hard uniqueness constraints: Linearizability is essential for enforcing strict uniqueness guarantees in relational databases.
- Other constraints: Foreign key or attribute constraints can often be implemented without linearizability.

Distributed Transactions and Consensus

Consensus goal is to make several nodes to agree on something. There are multiple scenario where we would need multiple nodes to agree upon something.

1. Leader election
2. Atomic Commit

Fischer, Lynch, and Paterson¬which proves that there is no algorithm that is always able to reach consensus if there is a risk that a node may crash.

Consensus algorithm

Two-phase commit
In case of single node, the data is first updated into the disk and then into the commit logs, if there is anything happen before written into the commit logs the transaction will be rollbacked

In case of the multiple node, the transaction are committed to multiple node on successful commit – the distributed transaction will be committed, else the distributed transaction will be rollback.

Under the following conditions a distributed transactions needs to be rollback

- Commit encounter a constraint violation causing the transaction to be rollback
- Due to network, the commit request is lost in certain nodes resulting in timeout which triggers a rollback.
- Some node crashed before committing, resulting in rollback.

Once the commit is executed, it cannot be undone. It needs to be compensated in case the transaction needs to be undone.

In case of two-phase commit, which is commonly used to achieve transaction across distributed nodes, and its supported by Java transaction API and WS-Atomic transaction for SOAP web service. As name suggested the 2PC have two distinct phases – within the application there is a component call as transaction manager that coordinated the transactions across multiple nodes, data is sent to all node for committing, then transaction manage sends Prepare signal, once all the nodes response to the prepare signal it sends the commit signal. In case prepare signal is not responded by any of the nodes then the rollback will be initiated in all the nodes.

How 2PC commit works

1. For each of the transaction there is a globally unique global transaction id is assigned.
2. The application begins a single node transaction for each of the nodes, all read and write will be performed on the single node.
3. Application the send prepare message to all nodes , if the transaction un-committed or timeout in any node then the application manger sends a rollback message to all nodes.
4. On receiving the prepare message, each node ensures the transaction can be completed – this is call as point of no return.
5. Once the node responded as "yes" in response to prepare request – transaction manager must write the transaction to a disk, call as point commit.
6. Once the transaction coordinator (manager) successfully writes the transaction to point commit, it then send commit / rollback message to all the nodes. The node must commit / rollback as per the received message, in case the node crash – transaction coordinator continues to retry until it receives success.

In case the transaction coordinator (manager) fails before sending commit/rollback messages, then the node needs to wait for the transaction manager to come online and reprocess the transaction decision based on the written commit log decision.

As 2PC commit are having a significant overhead as compare to single node transaction commit.

Database internal distributed commit: All nodes running same database software , when the database transaction is taking place.

Heterogeneous distributed transactions: different nodes running different software's from different software vendors, when the database transaction is taking place. XA Transaction API which is an C API, with bindings available on multiple languages supports heterogeneous distributed transactions, JTA (Java transaction API)support heterogeneous distributed transactions across multiple java platform using JDBC connection.

Heuristic decisions: Some time due to node failure, transaction is locked. There is no alternative way to release the lock, apart from manually committing/rollbacking each of the hang transactions. Alternatively, many XA implementation implements emergency escape from this situation by implementing Heuristic decisions, where it allows node(s) to take unilaterally decide to abort or commit an in-doubt transaction without a definitive decision from the coordinator. Though this breaks the atomicity and 2PC commit purpose.

XA Limitation

- Transaction Coordinator are single point of failure, as there are mostly not developed as high available.
- Application are design as stateless, with transaction coordinator implemented – coordination logs becomes the important part of the system, along with the database itself.
- Since XA needs to be compatible wide range of heterogeneous systems for communicating the transaction , implementing Serializable Snapshot isolation (SSI) in not possible to achieve as its needs protocol to identify conflict across different systems.
- Within database internal distribution also the 2PC commit has an tendency to amplify the single node problem to all the nodes – if one node fails to commit a transaction due to un-availability, the transaction cannot be completed on the other healthy nodes hence reducing the overall system fault tolerance level.

Fault tolerant Consensus

Consensus means – all nodes agree on something. It should have the following properties

1. *Uniform Arrangement – No nodes decide differently.*
2. *Integrity – No nodes decide twice.*
3. *Validity – If Node decides on v value, the v value was proposed by some other node.*
4. *Termination – every node that does not crashed, should eventually decides on some value.*

1, 2 and 3 are the safety property.

Leader section – doctorial leader selection where one leader is selected and other followers need to follow the selected leader.

Every time a current leader is dead, each of the follower are allocated Epoch number, epoch numbers are totally ordered and monotonically increasing. If there is any conflict among the leaders then the leader with the higher Epoch number decision in selected.

Instead of vote for quorum, the leader node send proposal for quorum, and wait for quorum of nodes to vote in favour of the quorum value in response. A node votes in favor of a proposal only if it is not aware of any other leader with a higher epoch

Consensus algorithms are a huge breakthrough for distributed systems: they bring concrete safety properties (agreement, integrity, and validity) to systems where everything else is uncertain, and they nevertheless remain fault-tolerant (able to make progress as long as a majority of nodes are working and reachable).

Consensus problem

Linearizable Atomic Lock – When multiple nodes try to perform a same operation, then the it needs to acquire a lock, prior to performing the operation. It also implements lease mechanism, where the lock is automatically release after certain time.

Total Order of operation – When there are multiple transactions are made, the lock is acquired based on some number that monotonically increases every time the lock is acquired. These are known as fencing token; this helps protecting a lock and lease. The higher the value of the fencing token it will be more prioritized.

Failure Detection – a long session is maintained which periodically take heartbeat of all the nodes, after a speculated time if there is no heartbeat received the node is declare dead and all the lock acquired by the node will be released.

Unique Constrain – when there are several transactions trying to update the database with conflicting records, then the database should be able to identify which transaction to be allowed and which once to be failed.

Part – III Derived Data

There are two types categories a system can be grouped that store and process data.

> ***System of records*** *- A system of record, also known as source of truth, holds the authoritative version of your data. When new data comes in, e.g., as user input, it is first written here. Each fact is represented exactly once (the representation is typically normalized). If there is any discrepancy between another system and the system of record, then the value in the system of record is (by definition) the correct one.*
>
> ***Derived Data*** *– Data in a derived system is the result of taking some existing data from another system and transforming or processing it in some way. If you lose derived data, you can recreate it from the original source. A classic example is a cache.*

CHAPTER 10: Batch Processing

There are different types of system based on the data it processes

- **Services (Online System):** In these types of system, as soon as the request is received, the service process the request and return the response to the requester. The response time is very important in this type of processing. Along with the response time, the availability of the service is also very important.
- **Batch Processing (Offline System)**: In these types of systems, a large amount of input data is process to produce output data, usually the batch processing is done offline hence there is no requester awaiting of the

response to be received. Once the output data is ready, the data is made available to the requester. For batch processing system it is the throughput (amount of data it can process over a period) that counts.

- **Stream Processing system (Near-Real time):** These types of systems are the in-between Services and Batch processing, like batch processing it accepts data offline, and a stream processor processes input data and produces output data in a near-real-time manner. Unlike batch processing systems, stream processing systems have lower latency.

Unix Philosophy

1. *Make each program do one thing well. To do a new job, build afresh rather than complicate old programs by adding new "features."*
2. *Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input.*
3. *Design and build software, even operating systems, to be tried early, ideally within weeks. Don't hesitate to throw away the clumsy parts and rebuild them.*
4. *Use tools in preference to unskilled help to lighten a programming task, even if you have to detour to build the tools and expect to throw some of them out after you've finished using them.*

Much of the Unix tool that build over decade work effectively till today, as they follow the Unix Philosophy – ability to join functions using pipe, existence of common file interface, transparency, and experimentation; however it lacks one thing, it cannot be scale beyond one machine.

Like Unix tool, MapReduce too follows the Unix Philosophy, however it can be scales to thousands of nodes (computer) – Hadoop implementation of MapReduces usages – Hadoop Distributed File System (HDFS) which is an open-source implementation of Google file system. HDFS is based on the shared-nothing principle (each of the independent nodes to have their own resources – share nothing).

HDFS consists of a ***daemon process*** running on each machine, exposing a network service that allows other nodes to access files stored on that machine (*if every general-purpose machine in a datacenter has some disks attached to it*). A central server called the ***NameNode*** keeps track of which file blocks are stored on which machine. Thus, HDFS conceptually creates one big filesystem that can use the space on the disks of all machines running the daemon.

MapReduce has two main functions –a) **Map** & b) **Reduce**.

MapReduce Workflows – The range of problem that can be solved by single mapReduce is limited, hence multiple mapReduce can be join to address larger problems, for which MapReduce workflow is used. As such MapReduce workflow framework do not provide any (particular) support hence two mapReduce functions needs to be chained using directories. Output of one mapReduce functions go one directory, from where the second mapReduce function reads the data.

mapReduce Join – to resolve complex problem, it is important to join two or more datasets. For example, if one needs to know from the website access logs, which pages are more popular among which age groups in this scenario along with website access logs datasets one also needs to join user data sets (website access logs datasets would have only userID not complete demographics of the login user).

To make this join (*join user data table for each of the log entries*). It would be not effective as the processing throughput would be limited by the round-trip time to the database server, reading the data from cache would also be ineffective as the effectiveness of a local cache would depend very much on the distribution of data, the most effective way to achieve good throughput in a batch process, the computation must be (as much as possible) local to one machine. Thus, a better approach would be to take a copy of the user database into the same distributed filesystem as the log of user activity events.

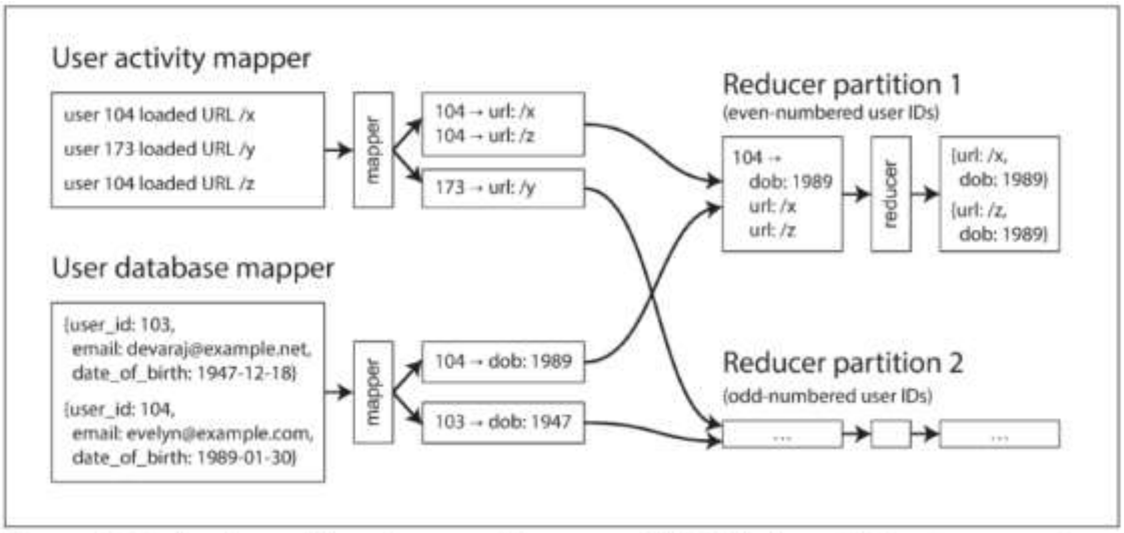


Figure 10-3. A reduce-side sort-merge join on user ID. If the input datasets are partitioned into multiple files, each could be processed with multiple mappers in parallel.

Since the reducer have all the records for a particular userID , it doesn't need to store multiple userID into memory – this type of join is call **sort-merge-join**.

Another advantage of the bringing all related records together is to perform action on group of similar records. Handling Skew – in most of the scenarios the records are similar number of related records, but under some special scenarios, there are some records which has disproportionately large related records, such disproportionately active database records are known as **linchpin objects** or **Hot Keys**.

In case of map-reduce until mapper and reducer has successfully processed all the records, in case of Hey Key scenario, its effects the performance as the performance of the entire operation on the slowest hot key reducer action.

Different frameworks handles Skewing problem differently, in case of Pig , it run a sampling job to determine which are hot keys, the hot key record is send to one of the several reducer (*and not based on key hash value*) , For the other input to the join, records relating to the hot key need to be replicated to all reducers handling that key. This way it can handle parallelization better.

In case of Hive, it requires hot keys to be specified explicitly in the table metadata, and it stores records related to those keys in separate files from the rest. When performing a join on that table, it uses a map-side-join for the hot keys. When grouping records by a hot key and aggregating them, one can perform the grouping in two stages. **The first MapReduce** stage sends records to a random reducer, so that each reducer performs the grouping on a subset of records for the hot key and outputs a more compact aggregated value per key. The second Map‐Reduce job then combines the values from all the first-stage reducers into a single value per key.

In case of the **map-side-join** approach, it uses a cut-down MapReduce job in which there are no reducers and no sorting. Instead, each mapper simply reads one input file block from the distributed filesystem and
writes one output file to the filesystem.

Broadcast hash joins

This type of join mechanisim , is mostly been used when a large dataset needs to be join to a small dataset(small dataset should be small enough to be load into memory)

This simple but effective algorithm is called a broadcast hash join: the word broadcast reflects the fact that each mapper for a partition of the large input reads the entirety of the small input (so the small input is effectively "broadcast" to all partitions of the large input), and the word hash reflects its use of a hash table. This join method is supported by Pig (under the name "replicated join"), Hive ("MapJoin"), Cascading, and Crunch. It is also used in data warehouse query engines such as Impala.

[Page # 408 to Page # 410 – summary pending]

Mape Reduce workflow with map-side join:

Out put of the batch processing – in case of the OTLP small set of number of records looked up by key, using some index to present it to the user. In case of the OLAP a large set of number of record are grouped and aggregated to produce some sort of report, in case of batch processing the result is not a report, but its some other kind of structure.

Common examples of batch processing are

1. *Building full text search indexes.*
2. *Building machine learning classifiers*

In both these examples the out of the map-Reduce is copied to a database, from which generic application search its results. Writing a Hadoop cluster results into a production db is not a good idea – 1) every map data needs to be written into the db, which itself is a overwhelming for a db to handle. 2) most of the map_redue jobs runs in parallel which furfur increase the number of open connection to the db, 3) some of the batch jobs task failed – it needs to be retired and for some it simple produce half-baked results, db needs to address these incomplete jobs. In order to overcome these challenges – there has been a dire need for a new type of database inside the batch job, its data-files are immutable and once written can be load in bulk into server that server read-only queries. Voldemort, Terrapin, ElephantDB, and HBase bulk loading are example of such dbs.

The main difference between MPP (massively parallel processing) database and Hadoop is – in case of Hadoop, it doesn't discriminate the data by its type, or ability to use it loads the data quickly as possible into a distributed HDFS file store and then later decides how to transform the data into a meaningful insight.

Since, MPP are designed for a intended datatype its always easy to tune it for performance and efficiency , where as HDFS are difficult to tune. Over the time , it was found that MapReduce was too limiting and performed too badly for some types of processing, so various other processing models were developed on top of Hadoop, making it feasible to implement a whole range of approaches, which would not have been possible within the confines of a monolithic MPP database.

[Page # 417 – 429 Summary pending]

CHAPTER 11: Stream Processing

The problems with the batch processing are

1. It can work on bounded data, whereas the true nature of the data is unbounded.
2. Data are process with a delay.

Event Streaming as a data management mechanism, overcomes the shortcoming of the batch processing.

Event *– an event is essentially the same thing: a small self-containing, immutable object containing details of something that happened at some point in time.*

Messaging System *– system the carries the message produce by the producer to the message consumers.*

In case of the batch processing there is a strong reliability guarantee, the failed jobs are automatically retried and output of the partially failed job are discarded. But in case of event processing, one need to consider that

1. What happens when the consumer fails to accepts event
2. What happens when the node itself fails.

Stream as a database

Data base	Message Broker

In case of database, the data is retained until its explicitly deleted.	In case of message broker, the data is deleted soon after its successfully delivered to its message consumer.
Database are design as large datastore, and the performance does not degrade much when the data store working set size increases.	The message brokers are design to have small working set, when the data working set increases the performance get impacted as message broker need to swipe some of its data to disk as its memory started to get fill.
Database supports secondary indexes providing various ways to search a data within the database.	IN case of a message broker, there is only certain limited way for its subscriber to filter the data before consumption.
In case of database, the query returns the data based on a point-in-time snapshot, user will not be informed if the new data matching the query criteria arrives.	Message broker notify the consumer for consumption of any new event that match to the consumer criteria.

Different ways of connecting to a consumer

1. **AMQP/JMS-style message broker (Load-balancing)**: Message broker will arbitrarily deliver the message to one of its consumers from the consumer group.
2. **Log-based message broker (Fan-out)**: Message broker will push the message to all its consumers with a consumer group.

In situation where parallel process of the message is more important and the order of the message is not significant in such cases Fan-out style messaging is preferred, whereas in situation where higher through put is desired and message order is important in such cases, Load-balancing style or log bases style is very preferred. Where message producer appends to the log, and message consumer reads the log sequentially using consumer offset. consumer offset is sequential number assigned by the message broker, where broker periodically reads the consumer offsets which tell the broker which message are read and which message are yet to be read. Message having offset higher than the consumer offset is yet to be read, where message will lower offset then consumer offset is already read by the consumer.

To ensure, the messages are delivered to the consumer. Message broker uses acknowledgement mechanism, where once the message is delivered consumer and the consumer successfully process the message it sends an acknowledgement back to message broker and the message broker would remove the message from its queue. In case the consumer is down or failed to process the message within a pre-define time (within define timeout) it would not send the acknowledgement which would trigger message broker to re-delivered that message to another consumer for processing.

Maintaining Message Ordering – With load-balancing message brokering and re-deliveries of messages it is inevitable to maintain the message order. However, this can be address by maintaining separate partition for each of the consumers. The message order cannot be guaranteed across partitioned.

Change Data Capture (CDC) - For decades, many databases simply did not have a documented way of getting the log of changes written to them. For this reason, it was difficult to take all the changes made in a database and replicate them to a different storage technology such as a search index, cache, or data warehouse. More recently, there has been growing interest in change data capture (CDC), which is the process of observing all data changes written to a database and extracting them in a form in which they can be replicated to other systems. Parsing the replication logs to capture change is more practical approach *(though it too has certain challenges like – schema changes)*. LinkedIn's Databus,Facebook's Wormhole , and Yahoo!'s Sherpa ,use this idea at large scale. CDC for PostgreSQL using an API that decodes the write-ahead log , Maxwell and Debezium do something similar for MySQL by parsing the binlog, Mongoriver reads the MongoDB oplog, and GoldenGate provides similar facilities for Oracle. Increasingly, databases are beginning to support change streams as a first-class interface, rather than the typical retrofitted and reverse-engineered CDC efforts.

Kafka Connect is another initiative to integrate change data capture tools for a wide range of database systems with Kafka. Once the stream of change events is in Kafka, it can be used to update derived data systems such as search indexes, and feed into stream processing systems as discussed later in this chapter.

Event Sourcing

Event sourcing involves storing all changes to the application state as a log of change events. A technique that was developed in the domain-driven design (DDD) community.

Command Query Resource Segregation

Form the data is store is different than the form the data will be quired , this is known as Command Query Resource Segregation.

Event Processing:

An event can be processed in one of the following ways:

1. Store the event into some kind of database, cache, indexes etc.
2. Send the stream directly for the end user to consume – generating email alert bases on the event , sending event to be viewed on a dashboard for end user to view/consume it.
3. Process a stream to another stream – combining multiple streams to produce a output streams using stream processing pipelines.

Processing Streams

Streams can be processed by complex event processing tooling, where a static query is getting performed on an incoming stream, when a match is found it generates a new event hence the name – **Complex Event Processing**.

Stream Analytics: Alternatively, steams can be used to run probabilistic query which are compartivitly less accurate but far for less expensive to run and generate a report.

Materialized view: Delivering an alternative view which can be query effectively and updating the view when the underline data changes.

Seach Stream: Like CEP, its sometime need to search a particular event from the streams.

Stream Joins

Since the stream is an unbounded data flow, applying join is more difficult than in batch processing jobs where the data is bounded. In Streams there are three major types of joins

1. **Stream-Stream Joins (Window Join):** As name suggested, two or more streams are joined to produce a meaningful insight. To implement this type of join, stream processor needs to maintain a state where it needs to perform the join. Example – join search query stream & click event streams to find a meaningful insight to understand which search results has been clicked. The two different streams can be correlated based on the user session id.
2. **Stream-Table Joins (Stream Enrichments):** Certain times it's important to enrich the streams with some meaningful insights where stream data is joined with the table data to enrich the stream information. Since stream processor is long running the content of the table should be also changing over time. A Stream-table join is also a form of Stream-Stream join, the biggest difference is that for the table changelog stream, the join uses a window that reaches back to the "beginning of time" (a conceptually infinite window), with newer versions of records overwriting older ones. For the stream input, the join might not maintain a window at all. Example of Stream-Table join, joining the user click stream with user-table data to derive a meaningful insight which URL is famous among which user demographic.
3. **Table-Table Joins (Materialized view maintenance):** In certain time a Materialized view maintenance needs to be maintain by joining two or more streams together for example: in case of twitter timeline example, it would be too expensive to iterate over all the users the user is currently following to create a home timeline, instead a timeline cache: a kind of per-user "inbox" to which tweets are written as they are sent, so that reading the timeline is a single lookup.
4. **Time Dependent Join:** In certain scenarios, the join need to made based on the time. This type of time dependent join can be seen in multiple places, for example when joining a sale with the tax implication; since

tax keeps changing time to time, when join the sales stream we need to join the tax that are applicable at that time, hence these jobs cannot be rerun without considering the tax of the time the sale take place. In data warehousing this problem is known as slowly changing dimension (SCD) , and this is been take care by introducing a unique identifier to the particular version of the join record. In the above example, every-time tax changes it allocates a unique identifier

Fault Handling in Streams: Unlike Batch jobs, streams jobs cannot be rerun to fix the fault , as stream are continuous stream of data , reprocessing is not an option. To handle fault, streams uses micro-batch size which can be retried on failure OR create rolling checkpoint on failure the job can start processing from the previous checkpoint.

[1] Google Big table paper – which introduce the term SSTable and Memtable.
leap second on June 30, 2012, where may application got stuck due to a bug in the Linux Kernel.

#NoSQL was first used a hashtag on 2009 for open source distributed, non-relational database meetup. It was only used to represent Not Only SQL.

Byzantine-faulty