Reliability

- Hardware faults

- Software errors

- bugs

- process used up resources

- Human Errors

- Abstractions

- Testing

- Monitoring

- Training

Scalability

- How to measure performance?

- response time percentiles

- Vertical/Horizontal scale

Maintainability

- Operability

- Monitoring

- Track down problem

- Perform update/regular maintenance

- Simplicity

- Good abstraction to remove complexity

- Evolvability

- Agile

- TDD

Data Model

- Relational

- One to many, many to many

- Strong declarative SQL language, query optimizer

- fix schema

- Non-relational

- Flexible data structure, validate on read

- Horizontal scalable

- Document

- JSON like

- Denomolized, redundant data

- Good for one to many read, bad for many to many

- extra request to find reference

- Graph

- Store vertexes and edges

- Good for complex relationship, relationship schema not fixed

- Query can be much simpler than SQL

# Data structures

- Log key-value pairs

- Hashmap

- Pros

- Fast read and write

- Cons

- Must fit in memory

- Range queries not efficient

- Balanced BST (SSTable)

- Pros

- Merging is simple

- Lookup is efficient

- Save memory/disk space

- Issues

- Lose data in memory if db crash

- Maintain a separate log on disk

- Slow when key does not exist

- Bloom filter

- Size-tiered and levelled compaction (determine the order and timing of how SSTables are compacted and merged)

- B Tree

- Fixed-size blocks or pages

- Similar to hardware

- Issue

- Crash

- Write ahead log

- Concurrency

- latches (lightweight locks)

- Optimization

- Copy-on-write

- Handle concurrency control, crash recovery

- Abbreviate key

- Leaf pages in sequential order

- Faster disk seek when scanning large key range

- Hard to scale

- LSM vs. B Tree

- Faster write

- Sequential faster than random write

- Smaller files on disks

- B tree use pages, leave some space unused

- Problem with compaction:

- Compaction process taking resource, affect read/write response time

- Affect small on average, but big on higher percentile. Thus more unpredictable

- Compaction share disk bandwidth with initial write

- The bigger the db, the more disk bandwidth is required

- Eventually compaction cannot keep with write

- Unmerged segments increased

- Run out of disk space

- Read slow down

- Explicity monitoring is required

- B tree has each key exactly in one place, LSM has multiple copies of same key

- Transaction isolation locks can be directly attached to the tree

Other indexing structures

- Secondary index

- Not unique

- List of matching rows as value

- Add unique id to key

- Storing value

- Heap file (store reference to value)

- Multiple indexes point to same location in heap

- Overwrite in place if small value, not if value is larger

- Update all index pointers or create a forwarding pointer

- Cluster index (store all value)

- Value stored within index

- Faster read

- Overhead on write

- Additional storage

- More effort to enforce consistency

- Mutil-column indexes

- Combine several fields into one key

- Fuzzy search

- Maintain a in-memory term dictionary to calculate offset for queries

- In-memory db

- Durability

- Write log/snapshot to disk

- Replicate state to other machines

- Better performance than disk

- Avoid overheads of encoding in-memory data structures in a form that can be written to disk

- Supports more data structures

Data analytics (OLAP vs. OLTP)

- Scan huge number of records, read a few columns, calculate aggregate stats

- Data warehousing

- Separate db for analysts to query

- Extract(copy), transform and load from all OLTP db (ETL)

- Optimized for analytics access pattern

- Schema (star, snowflake)

- One fact table (for each event), multiple dimension table (for details)

- More sub-dimensions in snowflake, more normalized, but harder to work with for analytics

Column-oriented storage

- Useful when read only a few columns out of large number of columns (such as data warehousing)

- Store values by columns instead of by rows

- Good for compression as values are usually repetitive in same column

- Use less CPU cache (vectorized processing)

- Sorting

- Sort entire row at a time

- Help with compression (run-length encoding)

- Multiple sort orders for multiple replications

- Faster to read, but harder to write

- Cannot update-in-place with compression columns

- Writes first go to in-memory store, write to disk when full

- Cache most often used aggregation results

- Denormalized copy, needs to be updated on write, make write more expensive

- Not useful for OLTP

- Fast read, not flexible, use as performance boost

# Encoding

- Language built-in encoding

- Bad to be compatible with other languages

- Bad security

- Neglect forward/backward compatibility

- Bad efficiency

- Standardized format (JSON, XML, CSV)

- Encoding number is hard

- Binary string is not supported

- Schema is not supported

- Binary encoding

- Lose human readability

- No schema

- Encode field names

- A little space reduction

- Has Schema (Thrift, Protocol Buffers)

- Encode field tags, compact version of field names

- Change schema

- Forward compatibility

- Add new fields with new tag number, ignored by old code

- Backward compatibility

- Cannot make a new field required

- Remove optional field is ok, but tag number cannot be reused

- Change field type

- “Repeated” for list type

- Allow changing from optional to repeated

- Use list

- Support nested list

- Avro

- Has schema, no tag numbers, more compact

- Value concatenated in order

- Use two schemas: reader and writer

- Change schema

- Read/write schemas don’t have to be the same

- Add/remove fields with default values

- How does reader know who’s the writer?

- Large files: add writer schema to the end

- Individual records: add version number

- Network connection: negotiate schema version at the beginning

- Good for dynamically generated schemas

- Thrift and Protocol Buffers requires manually mapping of tag number

Dataflow through db

- Rolling upgrade

- Value in db is written by different version of code at the same time

- Need to be taken care at application level

- Avoid schema migration

Dataflow through services

- SOUP

- User rely heavily on tool support, code generations and IDEs

- Integration with SOAP services is difficult

- Interoperability between different vendor’s implementations cause problems

- REST

- Less code generation and automated tooling

- Swagger is useful for documentations

- Good for debugging (by just using web browser or curl)

- Good for public APIs

- Compatibility

- Add version number to url/header to identify clients

- RPC

- Function call through network

- Problems:

- Network request is unpredictable

- Parameters can be large after encoding

- Translate data types to match servers’ language

- Better performance sometimes

- Focus on requests between services within the same org and the same datacenter

- Async message queue

- Act as buffer

- Hide recipients details (IP address, port number etc.)

- Don’t worry about results

# Replication

- Pros

- Data geographically close to users

- Fault tolerance

- Increase read throughput

- Leaders and followers (master-slave)

- Writes are only accepted by leader

- Send update stream to all followers

- Reads can be accepted by any nodes

- Sync vs. Async

- Sync

- One leader and one follower are sync, others are async

- Async

- Week durability

- Widely use

- Add new followers steps:

1. Copy data first with snapshot

2. Continue asking leader for changes after snapshot was taken

- Handle node outages

- Follower failure

- After recovery, continue asking leader for changes after last transaction

- Leader failure

- Steps:

1. Determine that the leader has failed (usually timeout)

2. Choose a new leader (election process, usually with the most up-to-date data changes)

3. Reconfigure the system to use the new leader

- Client should send write requests to new leader

- Ensure old leader becomes a follower

- Problems

- Async replication, out-of-date follower is promoted

- Discard conflict writes

- Violate durability

- Inconsistency with outside system

- Two nodes both believe they are leaders

- Some teams prefer to perform fail-overs manually

- Implementation

- Statement-based replication

- Leader forwards each queries to followers

- Side effect from non-deterministic call

- Too many edge cases

- Write-ahead log shipping

- Send log to followers

- The log describes data on a very low level

- Does not allow different versions of db

- Logical log replication

- Use different log formats for replication and for storage engine

- Backward compatible, allow different version/storage engines

- Easier for external app to parse

- Triggered-based replication

- Happens at application layer

- Register application code that is triggered when a data changed

- Greater overheads, but more flexible

- Problems: eventual consistency

- Latency to get latest data depending on network and system workload

- Read your own writes

- Read-after-write consistency

- Read use modified data only from leader

- e.g. user profile data is likely to be modified by user

- Not efficient if most data is editable by the user

- Remember timestamp of last write, only retrieve data after the timestamp

- Suspend or let it handled by another follower if couldn’t find the data

- Cross-device read-after-write

- Last update timestamp is not shared across devices

- Different devices might be routed to different datacenter

- See data moving backwards in time

- Monotonic reads

- Read data in sequence

- Achieved by always read from the same replica

- Hash user ID

- Violation of causality

- Consistent prefix reads

- User should see writes in the same order as they happened

- Make sure causally related writes are written to the same partition

- Algorithms to keep track of causal dependencies

- Particular problem in distributed database

- Solution

- Transaction: application doesn’t have to worry about consistency issue

- Multi-leader replication

- Multiple leaders accept writes and replicate to other nodes

- Reduce write latency

- Use cases

- Multi-datacenter

- One leader in each datacenter

- Every write can be processed by nearest leader

- Tolerance of datacenter outage

- Tolerance of network problems

- Single-leader uses public internet, whereas multi-leaders use local network

- Downside: conflict writes in different datacenter

- Clients with offline operations

- e.g. calendars, google drives

- Every device has a local db that act as a leader

- Collaborative editing

- Handle write conflicts

- Single-leader can force user to retry write

- Avoid conflicts

- Always route to the same datacenter

- Last write wins

- Data loss

- Merge/concatenate the values

- Resolve conflict later (by prompting the user)

- Custom conflict handler

- Replication topology

- Circular, star

- Has single point of failure

- All to all

- Causality problems (update before insert could happen)

- Use version vectors

- Leaderless replication

- Clients send writes/reads to several replicas in parallel

- To prevent reading outdated data from failed node

- Without a leader, nodes won’t know by itself if they have the latest data or not

- Version number is used to determine which response value is newer

- No fixed order of writes

- How to repair outdated node?

- Read repair

- Client write newer value back to the outdated replica after getting responses

- Works well for values that are frequently read

- Anti-entropy process

- Background process which constantly copy missing data to outdated nodes

- Can have significant delay before copy

- Quorum reads and writes

- If there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each read. w + r must be greater than n to guarantee a up-to-date value when reading.

- We can tolerate n - max(w, r) number of nodes to be failed

- Setting smaller w and r values allows lower latency and higher availability

- More nodes fail can be tolerant

- Edge cases

- Sloppy quorums

- Write conflict

- A write happens concurrently with a read

- Write succeeded in some replicas but failed in others

- Node recover from old value

- Timing issue

- No absolute guarantee on not reading stale value

- Monitoring staleness

- Leader-based can compare replication logs between leader and followers

- Because writes are applied to them in the same order

- Difficult to monitor leaderless system as there’s not fixed order for writes

- Sloppy quorum

- Requests to temporary node when having network interruption

- Ensure durability of data

- Multi-datacenter operation

- Only wait for responses from a quorum of nodes within its local datacenter

- Handle write conflict

- Last writer wins

- Eventually convergence achieved, but lose durability

- Define concurrent

- Two operations are either one happened before another or they are concurrent

- They are concurrent if they are both unaware of each other

- If it’s not concurrent, the earlier one should be overwritten

- Otherwise we need to resolve the conflicts

- Merge is required

- Single replica

- Use version number to determine if they are concurrent or not, sent from db to client and sent back to db on write

- Multiple replicas

- Use a version number per replica and per key

- Version vector

# Partitioning

- Main purpose is scalability: a large dataset can be distributed across many nodes

- Combine with replication: each partition has a leader

- We should partition data evenly, otherwise it becomes less effective

- **Partitioning by key range**

- Need to know the boundaries of key range

- The boundaries need to adapt the data in order to make it distribute evenly

- Key data in sorted order within each partition (easy for range scan)

- Cons

- Leads to hot spot (unevenly distributed)

- Use additional parameter together as partition key, but makes range query more complicated

- **Partitioning by hash of key**

- A good hash function takes skewed data and makes it evenly distributed

- Consistent hashing

- Compute hash on key and assign it to a range partition

- Cons

- Lose the ability to do range query

- Concatenated index approach

- Only first part of the key is hashed to determine partition, other columns can be sorted.

- Good for one-to-many relationships

- Extreme case: all requests aim for the same key

- e.g. social media one user has millions of followers

- It is the application responsibility to reduce the skew

- Add a random number to the key

- Able to write to different partitions, but reads need to scan all partition

- Additional bookkeeping: need to keep track of which keys are being split

- Secondary index

- Partition by document

- Each partition maintains its own indexes (local indexes)

- Pros

- Writes only need to deal with the one partition

- Cons

- Reads need to scan all partitions and combine the result

- Partition by term

- Global index that covers all partitions

- A global index must also be partitioned (by hash)

- Pros

- More efficient reads

- Cons

- Slower and more complicated writes

- Write to multiple partitions

- Indexes are updated async

- Rebalancing

- The process of moving load from one to another

- To increase throughput

- To fit more data

- To handle a machine failure

- What not using hash(key) mod n?

- That requires too much data move when n changed in rebalancing

- Fixed number of partitions

- **Size of each partition is proportional to the size of dataset**

- Since a node has many partitions, new node can steal a few partitions from each old node.

- The partition assignment does not change

- Thus max number of nodes = number of partitions

- Choosing number of partition is hard

- Too large

- Too much management overhead

- Too small

- Recovery and rebalancing becomes expensive

- Dynamic partitioning

- **Number of partitions is proportional to the size of dataset**

- Suitable for both key range partitioning and hash partitioning

- When a partition grows to exceed a configured size, it’s split into two.

- Similarly if it shrinks below a threshold, it can be merged with an adjacent partition.

- If a node have too many partitions, some partitions can be moved to a new one. (Rebalancing)

- Pros

- For small amount of data, small number of partitions is used.

- Low overhead, no need to worry about configuring the number

- Cons

- When the first partition is splitting, only one node can process the writes

- Solution is to create a initial set of partition on an empty db at the beginning

- Third option

- **Number of partitions is proportional to the number of nodes**

- Number of partitions per node is fixed

- Good for keeping partition size stable

- New node randomly pick some partitions in old node, split and take ownership on half of them.

- To randomly pick partition boundaries, it only used hash-based partitioning.

- Automatic or manual?

- Automatic

- Convenient

- Less operational work

- Unpredictable

- Manual

- Slow

- Prevent operational surprises

- Request routing

- Allow clients to contact any node via a load balancer, forward if necessary

- Routing tier to determine which node should handle it

- Client keeps partition information locally

- **Zookeeper**

- Keep track of partition information in the system

- Notify the routing tier when there are changes

- DNS is sufficient for finding routing tier or a random node

- Parallel query is required for analytics

# Transaction

- A way of **grouping multiple** reads and writes together into a single unit

- Then the system doesn’t have to worry about partial failures

- Simplify the programming model for applications accessing a db

- Sometimes weakening or abandoning transactional guarantees can achieve high performance and high availability.

- ACID

- Atomicity

- If writes are grouped together into an atomic transaction and it cannot be completed due to a fault, it must discard any writes that has been made so far. So, then a safe retry can be performed.

- If a transaction is aborted, the application can be sure that it didn’t change anything

- Consistency

- An application-specific notion of the database being in a “good state”

- It’s the application responsibility to guarantee consistency

- Isolation

- Concurrently executing transactions are isolated from each other

- Serializable isolation has performance penalty

- Durability

- Once a transaction has been committed, any data it has written will not be forgotten.

- Single object writes

- Almost all storage engines provide atomicity and isolation on single object

- Isolation can be implemented using a lock

- This is light-weight transaction

- Multi-object transactions

- Abandoned by many distributed datastores because they are hard to implement.

- However, error handling becomes much more complicated without atomicity, and lack of isolation can cause concurrency problem.

- Handling errors

- Atomicity enable safe retry after fail

- Problem of retry

- Retry the transaction cause it to be performed twice if it’s actually a network failed

- Make it worse if it’s because of overload

- Not worth retrying after transient errors (deadlocks, temporary network failure, failovers)

- Has side effect (send an email)

- **Weak isolation level**

- Because serializable isolation has a performance cost

- **Read committed**

- No dirty write/read

- When reading from the database, you will only see data that has been committed.

- When writing to the database, you will only overwrite data that has been committed.

- It does not prevent the race condition between two counter increments.

- **Implementations**

- Prevent dirty writes by using row-level lock

- Prevent dirty reads by using row-level lock

- Not good, long-running writes can block read-only transactions, harms the response time

- Database remembers both old and new value of an object, give old value to read. (two-versions MVCC)

- **Snapshot isolation**

- Two reads get different state of data because a write is interrupted in between, thus seeing inconsistent result.

- Read skew, eventually consistent

- Read committed only guarantee that write transaction is not interruptible, but not reads.

- Sometimes not acceptable

- Recover from inconsistent backup

- Analytics query

- Each transaction reads from a consistent snapshot of the database

- Good for long-running, read-only queries

- Implementations

- Write lock

- Keep several different committed versions of objects for reads. (MVCC)

- Rows are marked as deletion using transaction ID during transaction, removed by garbage collector afterwards.

- Visibility rules

- Transaction IDs are used to decide which objects a transaction can see.

- Followings are hidden

- In progress write transactions

- Aborted write transactions

- Write transactions that started later than current one

- Indexes

- Index can simply point to all versions of objects, allow filters on query.

- For append-only b-tree, every write create a new root, a particular root is a consistent snapshot.

- No filter needed

- Require a background process for compaction and garbage collection

- Repeatable read

- Similar but flawed

- **Preventing lost update**

- Happens when two transactions write concurrently (read-modify-write)

- Solution

- Atomic update

- Remove the need to implement read-modify-write

- Implemented by taking an exclusive lock on the object

- Cons

- Some framework makes it easy to not use this feature (Django)

- Not all write queries can be implemented as atomic write

- Explicit locking

- Lock the object before updating

- Automatically detecting lost update

- Transaction manager can monitor, abort and retry lost updates

- Pros

- Works efficiently with snapshot isolation

- Doesn’t require application code to use any special database features

- Compare-and-set

- Allow an update to happen only if the value has not changed since you last read it.

- It won’t work if the “where” clause reads from old snapshots.

- Replicated/distributed database

- Add version numbers and use application code or special data structure to solve conflict

- Last write win is prone to lost update but is default to many distributed databases.

- **Write skew and phantoms**

- Happens when two transactions read the same objects, and then update some of those objects. (different transactions may update different objects)

- Solutions

- Configure constraints (uniqueness, foreign key)

- Use serializable isolation level

- Explicitly lock the row

- **Phantom**: a write in one transaction changes the result of a search query in another transaction.

- Materializing conflicts

- Introduce a lock object into database

- Ugly to let a concurrency control mechanism leak into the application data model

- Serializable isolation level is better than this one.

- **Serializability**

- **The key difference between serializable and non-serializable isolation level is that serializable only allow one transaction at a time, whereas non-serializable makes operations in multiple concurrent transactions happens in between each other.**

- **Actual serial execution**

- Execute only one transaction at a time, in serial order, on a single thread.

- Reason of using this

- RAM became cheap, use in-memory database

- Separate OLTP and long-running queries, so that read-only query can be run on a consistent snapshot.

- No need to worry about detecting and preventing conflict problems.

- The throughput is limited to a single CPU core.

- **Stored procedures**

- Useful for single thread

- Using stored procedures can avoid too much I/O between application and disk, everything is executed within a stored procedure

- **Partition**

- Partition the data in a way that each transaction can have its own thread running independently, as long as each transaction only need to read/write data within a single partition.

- But if a transaction has to access multiple partitions, it has to **coordinate and** **lock all of them** to ensure serializability.

- It has additional coordination overhead, much slower than single-partition transaction

- Single key-value stores can be partitioned easily, but data with multiple secondary indexes requires a lot of cross partition coordination.

- Summary

- Transaction must be small and fast

- Active dataset should fit into memory

- Write throughput must be low enough to be handled in a single CPU core, otherwise use partition

- Cross-partition transaction is very inefficient.

- **Two-phase locking (2PL)**

- Similar to the row-level lock that prevents dirty writes, but much stronger.

- **Difference to snapshot isolation**:

- Writers don’t just block other writers, they also block readers and vice versa.

- Snapshot isolation has “readers never block writers, writers never block readers” principle.

- Reading an old value is acceptable in read committed but is not acceptable in 2PL.

- Implementation

- Readers acquire shared lock

- Writers acquire exclusive lock

- Exclusive lock is used to block shared lock

- Deadlock happens easily here. System automatically detects that and abort, retry one of them.

- Performance

- Significantly worse than weak isolation level

- Overhead of acquiring and releasing locks

- Reduced concurrency

- Unstable latency because long-running queries can block others, slow at high percentile.

- Deadlocks make a lot of aborts and retries. Frequent deadlocks mean significant wasted effort.

- Predicate lock

- To solve “phantoms causing write skew” problem

- Similar to above locks but belong to all objects under certain conditions rather than one object.

- The key idea is the lock applies to objects that do not yet exist in the database (phantoms)

- Cons

- Bad performance. Checking for matching locks takes time if there are many locks.

- Index-range locks

- A simplified approximation of predicate locking.

- It is used to match a greater set of objects. (attach shared locks to index).

- e.g.:

- Predicate: lock room 123 between noon and 1 pm

- Index-range lock:

- Lock room 123 at all time

or

- Lock all rooms between noon and 1 pm

- Not as precise as predicate locks, but much slower overhead.

- If no suitable index where a range block can be attached, the database can fallback to a shared lock on the entire table.

- **Serializable snapshot isolation (SSI)**

- Recall

- Serial execution doesn’t scale well; 2PL doesn’t perform well.

- Weak level isolation has good performance but has many race conditions. (lost updates, phantoms, write skew etc.)

**- SSI provides full serializability, but only a small performance penalty compared to snapshot isolation.**

- Used in both single and distributed databases.

- **Pessimistic vs. Optimistic concurrency control**

- Pessimistic

- If anything might go wrong, wait until it’s safe before doing anything. (2PL, serial execution)

- Optimistic

- Transactions continue anyway in the hope that everything will turn out all right. (Serializable snapshot isolation)

- When a transaction wants to commit, the database checks whether anything bad happened. If so, abort and retry.

- Pros

- Perform better than Pessimistic when space capacity is enough, and contention is not too high.

- Cons

- Many transactions trying to access the same objects (contention) leads to a lot of transactions to be aborted.

- Retry will make it worse if system is closed to its maximum throughput

- Contention can be reduced by avoiding conflicts by commutative atomic operations.

- All reads are made from a consistent snapshot of the database.

- On top of snapshot isolation, SSI adds an algorithm to detect serialization conflicts among writes and determine which transaction to abort.

- **Decisions based on outdated premise**

- Reading from a consistent snapshot may get outdated data because it might be modified in the meantime. (causing write skew)

- Approaches

- Detecting reads of a stale MVCC object version

- When the transaction wants to commit, the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted.

- Detecting writes that affect prior reads

- Similar technique as predicate lock

- Reading an object leaves an index entry record. Writes will look in the index for any other transactions that read this data.

- It simply notifies the transaction that the data they read may no longer be up to date.

- Performance

- Compared to 2PL, the big advantage is that transactions are not blocked by locks.

- Affect by the rate of aborts.

- Thus, requires read-write transaction to be short

- Still less sensitive to slow transactions than 2PL and serial execution

- Summary

- A lot of errors are reduced down to a simple transaction abort, and the application just need to try again.

- Weak isolation levels protect against some of anomalies but leave the application developer to handle others manually (explicit lock), unless using serializable isolation level.

# The trouble with distributed system

- The nondeterminism and possibility of partial failures is what makes distributed systems hard to work with

- Single computer deals with partial failure by escalate into total failure

- **Unreliable network**

- Shared-nothing systems

- Cheap

- Make use of cloud computing services

- Achieve high reliability through redundancy

- In datacenter, requests through internet is unreliable because

- Request may have been lost (someone unplugged the cable)

- Request waiting in a queue (network/recipient overload)

- Remote node failed/crashed

- Remote node not responding (pause)

- Response lost

- Response delayed

- Detecting faults

- System needs to detect failed nodes

- e.g. load balancer needs to detect dead node; distributed database needs to detect failed master

- Rapid feedback

- Other process in the node

- Management interface of the network switches in the datacenter

- Retry a few times, wait for a timeout to elapse, and eventually declare the node is dead.

- Timeouts and unbounded delays

- Long timeout => long wait time

- Short timeout => risk having false positive

- Perform same action twice

- Overload the system

- System does not have guarantee about maximum network delay (unbounded)

- Network congestion and queueing

- Reasons for variability of network delays:

- Network switch queue network requests if requests are too many

- Operating system queue requests if CPU cores are all busy

- VM is paused when other VMs are using CPU cores

- TCP use flow control to avoid overloading a network link or the receiving node

- TCP vs. UDP

- UDP does not have flow control and does not retransmit lost packets

- Avoid some network delays

- Unreliable

- Good for situations when delayed data is worthless

- Resources are shared, network delays can be highly variable if someone near you is using a lot of resources.

- Timeouts should be chosen experimentally, or automatically adjusted using tools.

- Synchronous vs. asynchronous delay

- Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays.

- Why using packet-switching?

- Optimized for bursty traffic

- TCP dynamically adapts the rate of data transfer to the available network capacity

- Complete as quickly as possible

- Static partition

- Latency guarantee

- Reduced utilization

- More expensive

- Dynamic partition

- No latency guarantees. Suffer from queueing, variable delays.

- Better utilization

- Cheaper

- **Unreliable clocks**

- Hard to know the order of thing happening when multiple machines are involved due to variable delays

- Each machine has its own clock

- Sync with Network Time Protocol (NTP)

- Monotonic vs. time-of-day clock

- Time-of-day clock

- Return current date time

- Sync with NTP

- Sometimes needs to reset the time to sync with NTP

- Therefore, this clock is not good for measuring elapse time

- NTP synchronization can only be as good as network delay

- Monotonic clock

- Guarantee to always move forward

- Good for measuring durations

- Absolute value of the clock is meaningless

- Makes no sense to compare monotonic clocks between different machines

- NTP can be used to adjust frequency

- Relying on synchronized clock

- Although they work quite well most of the time, robust software needs to be prepared to deal with incorrect clocks.

- Clock drift is common, easily got unnoticed.

- Timestamps for ordering events

- Earlier value may overwrites later value in last write win rule because of a lagging clock.

- Logical clocks are a safer alternative for ordering event

- Clock readings have a confidence interval

- Depending on how long it has been sync with NTP last time; The NTP server uncertainty; The network round-trip time

- Synchronized clocks for global snapshots

- In snapshot isolation, if a write happened later than the transaction, that write should be invisible to the transaction.

- Later write must have larger transaction ID

- Hard to determine the order in distributed system because of clock accuracy.

- Spanner approach: Compare confidence interval.

- If the interval does not overlap, then one must be later than another

- If it does, wait for the length of interval time before committing a read-write transaction so that any transaction may read the data is at a sufficient later time.

- Process pauses

- In distributed leader-follower based system, how does a leader know he has not been declared dead?

- Leader get a lease from followers. During the lease period it knows it must be leader. It must renew the lease when it expires.

- Problem

- It’s relying on synchronized clock

- We can use monotonic clock instead

- It’s possible that the program got paused for to long, meanwhile the lease is expired and another node claims to be a leader.

- e.g. Garbage collector in Java stops all threads occasionally; VM paused; Context switching;

- Response time guarantees

- Useful in some safety-critical systems that needs guarantee response

- e.g. aircraft, rockets, cars

- How to achieve this?

- Guarantee CPU allocation

- Library functions worst-case

- Dynamic memory allocation is restricted

- A lot of testing

- Expensive, may be lower performance

- Limit the impact of garbage collection

- Approach 1: Treat garbage collecting nodes as failed node, so they won’t accept new requests

- Reduced high percentiles of the response time

- Approach 2: Use garbage collector only short-lived objects (faster to collect) and restart processes periodically.

- Knowledge, truth and lies

- In distributed system, a node cannot necessarily trust its own judgement of a situation.

- No response from a node, declare it being dead. But it might be network issue or paused by GC.

- The truth is defined by the majority

- Many distributed algorithms rely on quorum, that is, voting among nodes.

- e.g. declaring node dead

- Fencing token

- A file can be accessed by one client at a time.

- Obtain a lease/lock from a lock service, renew when expired. While holding the lock, it can access the file

- Problem

- The client can be paused by garbage collection process. When it wakes up it thinks it still have the lock.

- Solution

- Lock service returns a lock and a fencing token, which is a number that increments every time a lock is granted.

- Client send the fencing token to the storage, storage might reject the token if it is lower than previously processed token.

- Byzantine faults

- A node may lie

- Byzantine fault-tolerance

- Protect system against attackers, tolerate physical corruptions

- Most Byzantine fault-tolerant algorithms require more than 2/3 of the nodes to be functioning correctly

- Very expensive

- If an attacker can compromise one node, he can compromise all of them

- So traditional mechanisms (authentication, firewalls, encryptions…) are still necessary

- Weak form of lying

- e.g. Invalid message due to hardware issues, software bugs, and misconfiguration

- Byzantine doesn’t work here. Need to take pragmatic step.

- System model and reality

- Abstract system model: make some assumption to the system, theoretically solve the problems.

- Algorithms need to be written in a way that does not heavily depend on software config and hardware details

- To solve the timing issue

- Synchronous model

- Assumes bounded network delay, bounded process pauses and bounded clock error

- In reality, unbounded delay and pauses do occur.

- Partially synchronous model

- Sometimes bounds network delay, pauses and clock drifts.

- Realistic model for most systems

- We must assume most of the time system and networks are well-behaved

- Asynchronous model

- Algorithm is not allowed to make any time assumptions, does not even have a clock

- Used in very restrictive case

- To solve node failures

- Crash-stop faults

- Assume that node always crashes and stops suddenly, and then gone forever

- Crash-recovery faults

- Assume that node always crashes and starts responding again after some unknown time

- Nodes always have stable storage, in-memory storage is lost

- Byzantine faults

- Node may be anything, including trying to lie to others.

- Most used model: partially synchronous model with crash-recovery faults

- Algorithms properties

- Properties are used to define whether an algorithm works or not

- Safety: Nothing bad happens

- When safety property is violated, we can point at a particular point in time at which it was broken

- Cannot be undone

- Must always hold

- Liveness: Something good eventually happens

- There is always a hope the issue will be fixed in the future.

- Allow to make caveats

- In reality without byzantine, we always have to assume some faults can or cannot happen. However, we also need to implement code that handle seemly impossible cases.

- Summary

- Problems

- Unreliable internet: lost packet

- Unreliable clock: clock drift and jump

- Process pause: due to garbage collection

- First step: detect them

- Second step: build fault tolerance

- A problem is trivial if it can be solved on a single machine

- Hard real-time response guarantees and bounded delays in network is expensive and results in lower utilization of hardware.

# Consistency and consensus

- The best way of building fault-tolerant systems is to find some general-purpose abstractions with useful guarantees, implement them once and let the application rely on those guarantees.

- Consensus

- One of the most important abstractions in distributed system: getting all nodes to agree on something.

- Consistency guarantee

- In replicated database, two nodes might have different value at the same time (eventually consistency)

- Eventually consistency makes no guarantee on when it will be consistent

- **Linearizability**

- Strongest consistency model in common use

- Make the system appear as there is only one copy of the data, and all operations are atomic

- All clients must see the latest data upon completion of a write

- What makes a system linearizable?

- If one client read the new value, all subsequent reads must also return the new value, even if the write operation has not completed yet.

- Each operation is marked with a vertical line (inside the bar for each operation) at the time when we think the operation was executed. Those markers are joined up in a sequential order, and the result must be a valid sequence of reads and writes for a register

- Linearizability vs. Serializability

- Serializability

- An isolation property of transaction

- Guarantee multiple operations are executed in some serial order

- Prevent race condition

- Linearizability

- Guarantee there only exists one copy of data

- Prevent inconsistency

- These two can be used together (2P locking, serial execution)

- Consistent snapshot doesn’t guarantee linearizability

- Relying on linearizability

- Locking and leader election

- All nodes must agree on the one leader at any time

- Leader election is implemented by acquiring lock

- Manage lock using Zookeeper

- Zookeeper and etcd use consensus algorithms to implement linearizable operations

- Uniqueness constraint

- e.g. username, email address, primary key

- Cross-channel timing dependencies

- Image resizer

- Upload an image -> image stored full-size in storage -> image info sent to resize service via a message queue -> resize service fetch image from storage and start resizing

- Most message brokers are designed for smaller message

- If it’s not linearized, the message queue might be faster than the internal replication inside the storage service.

- Implementation

- Single-leader replication

- Potentially linearizable

- Might not me linearizable if

- It uses snapshot isolation

- It has concurrency bug

- Node thinks it is leader but actually it is not

- Data loss due to failover last write wins

- Consensus algorithm

- Multi-leader replication

- Generally not linearizable

- Leaderless replication

- Sometimes not linearizable if

- Use last write wins conflict resolution based on time-of-day clock due to clock skew

- Sloppy quorum

- Strict quorum can also produce nonlinearizability

- In order to make it linearizable, reader must do read repair synchronously before returning result to application, and writer must read the latest state of a quorum of nodes before sending its write.

- Performance is reduced

- But most leaderless system does not implement this due to performance cost, and it requires linearizable read and write operation

- So it’s safe to assume that leaderless system does not provide linearizability

- The cost of linearizability

- Network interruption between two datacenter

- Multi-leader

- Operate normally. Writes can queue up and exchange when network is restored

- Single-leader

- Either client has to read stale data (lose linearizability) or client has to wait for network to be restored (lose availability), to those clients who can only access a follower datacenter

- The CAP theorem

- Applications that don’t require linearizability can be more tolerant of network problem

- Consistency, availability and network partitions

- Misleading because network faults always happen.

- When a network fault happens, you have to choose between availability and consistency.

- It has little practical value today because

- It only considers one kind of consistency model (linearizable) and one kind of fault (disconnect network)

- It doesn’t consider dead node, network delays.

- Linearizability and network delays

- In practical, few systems are linearizable because of performance cost but not fault tolerance.

- If you want linearizability, the response time of read and write is at least proportional to the uncertainty of delays in the network.

- A faster algorithm for linearizability does not exist, but weaker consistency models can be much faster.

- **Ordering guarantees**

- Recap on ordering

- Main purpose of leader in single-leader replication is to determine the order of write in replication log

- Without ordering in multi-leader system, conflicts can occur.

- Serializability, transactions behave like they are executed in some sequential order.

- Use clock and timestamp to determine which one of two writes happened later

- **Causality**

- Ordering helps preserve causality

- Causality imposes an ordering on events: causes come before effect

- If a system obeys the ordering imposed by causality, we say that it is causally consistent.

- e.g. snapshot isolation

- The causal order is not a total order

- Total order: allows any two elements to be compared

- e.g. 13 > 5

- Linearizability: one operation must happen before or after another

- Partial order: sometimes one is greater than another, sometimes incomparable

- Causality: two operation can be concurrent so incomparable

- Concurrency means that timeline branches and merges again

- Linearizability is stronger than causality

- Linearizability implies causality

- A system can be causally consistent without incurring the performance hit of making it linearizable

- Causal consistency is the strongest possible consistency model that does not slow down due to network delays, and remain available in the face of internet failures

- Systems that appear to require linearizability in fact only require causal consistency, which can be implemented more efficiently.

- Capturing causal dependencies

- If some preceding operations are missing, the later operation must wait until the preceding operation has been processed.

- Track causal dependencies across the entire database. Version vectors can be generalized to do this.

- The database keeps track of which data has been read by which transaction

- The version number from the previous operation is passed back to the database on write.

- Sequence number ordering

- Instead of tracking all causal dependencies, use sequence number of timestamps to order events. (logical clock)

- Every operation has a unique sequence number that can be used to compare.

- Concurrent operations may be ordered arbitrarily

- In single-leader database, replication log defines the total order of write operations.

- Non-causal sequence number generators

- Not single-leader or single-leader with partitions, not easy to generate sequence number for each operation

- Three way

- Each node generates its own sequence numbers (reserve bits)

- Attach a timestamp

- Pre-allocate blocks of sequence numbers

- Problems

- Different sequence number generators do not capture cross-node ordering

- Timestamps are subject to clock skew

- Lamport timestamp

- Generate sequence number that is consistent with causality

- It uses (counter, nodeID)

- Every node and every client keeps track of the *maximum* counter value it has seen so far, and includes that maximum on every request. When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.

- Timestamp ordering is not sufficient

- The total order of operations only emerges after you have collected all of the operations

- e.g. Two operation tries to create a unique username concurrently

- Need to know when the order is finalized

- Total order broadcast

- Usually implemented in Zookeeper and etcd

- Properties

- Reliable delivery: no messages are lost

- Total ordered delivery: messages are delivered to every node in the same order

- If every replica processes the same write in the same order, then the replica will remain consistent with each other.

- Every message represents a write

- Can be used to implement serializable transactions

- Every message represents a transaction

- Also useful for implementing lock service

- Implement linearizable storage using total order broadcast

- Implement total order broadcast using linearizable storage

- Distributed transactions and consensus

- Goal: get several nodes to agree on something

- Example

- Leader election

- Atomic commit: distributed transactions should all abort or all commit

- Atomic commit and **two-phase commit**

- The key deciding moment for whether the transaction commits or aborts is the moment at which the disk finishes writing the commit record: before that moment, it is still possible to abort (due to a crash), but after that moment, the transaction is commit‐ ted (even if the database crashes).

- In distributed system, there are problems when simply sending commit request to each node

- Some nodes have constraint which fails the commit while others succeeded

- Commit lost in the network

- Node crash

- After a transaction is committed, it shouldn’t be aborted.

- Two-phase commit is an algorithm for achieving atomic transaction commit across multiple nodes

- 2PC uses a coordinator (transaction manager)

- Phase 1: Coordinator sends a prepare requests to each of the node

- Phase 2: Send commit request if all nodes responded yes; Otherwise send abort request

- The coordinator write decision to its transaction log before sending requests in phase 2.

- So if one of the node failed in phase 2, retry forever until it succeeds.

- What if coordinator crashes while sending commit/abort requests to all nodes?

- Approach 1: In 2PC, nodes should wait for coordinator to recover.

- Approach 2: Three-phase commit

- But 3PC assumes networks and process pauses having bounded delay.

- In practical, 2PC is used.

- Distributed transactions in practice

- Distributed transactions carry a heavy performance penalty.

- When coordinator is down, locks is held in each nodes blocking other reads/writes.

- After a coordinator is recovered, transaction logs might be lost.

- The only way is for an admin to manually decide whether to commit or abort the transactions.

- Fault tolerance consensus

- A consensus algorithm cannot simply sit around and doing nothing forever – in other words, it must make progress.

- The best-known fault tolerant consensus algorithms are Raft, Paxos and Zab.

- Epoch numbering and quorums

- Limitation

- Async replications, some committed data can be lost during failover.

- Requires majority to operate

- Assumes fixed set of nodes in cluster

- Rely on timeouts to detect failed nodes, does not work well in highly variable network delays environment.

- Edge case: switch leader between two nodes frequently when having network problem

- Membership and coordination services

- Hold small amount of data that can fit entirely in memory

- Use cases

- Linearizable atomic operations

- e.g. lock service, lease

- Total ordering of operations

- e.g. fencing token

- Failure detection

- Release lock from dead node

- Change notifications

- Allocating work to nodes

- Failover on stateful service

- Partition rebalancing

- Service discovery

- Find out which IP address you need to connect to in order to reach a service

- But sometimes service discovery does not need consensus, therefore DNS is used.

- Membership service

- Determine which node are currently live and active members of a cluster

Message queue

- Unbounded queueing

- Two phase commit to ensure consistency

- Redeliver message due to consumer crash through load balancer can lead to reorder of messages

- Using logs for message storage

- A producer sends a message by appending it to the end of the log, and a consumer receives messages by reading the log sequentially.

- Log can be partitioned to be hosted on different machine to scale

- Partitions of the same type can be grouped as topic

- Entire partitions (topic) are assigned to a consumer group, each consumer reads one partition

- Works well for small message

- Consumer offsets

- Consuming a partition sequentially makes it easy to tell which messages have been processed: all messages with an offset less than a consumer’s current offset have already been processed, and all messages with a greater offset have not yet been seen.

- Broker does not need to track acknowledgement for every single message, reduced bookkeeping overhead.

- News feed: use creation date + id as primary key

- No secondary index

- No need to filter on date

- Reduce write latency

- Two incremental db generate id (one odd, one even)

- Web server has a connection limit

- Better to have separate service to handle and scale connections

- Websocket/long polling connections, long running download/upload process

- Buffer write requests and batch write

- Files can be divided into chunks (4MB). Upload a file only compare chunks. Chunks can be reused on cloud storage.

- Load balancer

- Round Robin approach

- No overhead, easy to implement, balance traffic

- Does not consider overload

- Typeahead suggest

- Each trie node only store reference to terminal node to save space

- Update trie offline after a certain interval rather than update on every query

- Rolling upgrade so that reads are not blocked by update

- Rate limiter

- Algorithms

- Fix window

- User can exceed limit

- Sliding window

- Too much space

- Hybrid

- Reduce granularity on fix window

- e.g. limit 500 requests per hour, we can count request based on minutes

- Limit by IP or user?

- IP

- Multiple users share the same IP

- Hackers who have a lot of IPs

- User

- Attack against login API

- Hybrid

- Limit both IP and user

- Web crawler

- DFS or BFS?

- DFS can have one connection at a time, save handshaking overhead

- Path ascending crawling

- It can discover hidden resources

- Payment system

- To achieve eventual consistency, retry multiple times.

- Write repair: every write call from clients attempts to fix an inconsistent

- Idempotency

- For an API request to be idempotent, clients can make the same call repeatedly and the result will be the same

- Exactly-once delivery

- Pinterest

- No joins, no foreign key

- Mapping table for relationships

- Queries are all PK

- All tables exists on all shards

- New index => new table (schema changes cause shut down on db)

- Scripting

- Move from old db to new shared db

- Migrate data while application is writing data to both db

- SOA

- Security

- One team for each service (deployment, provide support)

- Connection limit (connections take memory usage)

- Multi-tenancy

- Create isolated environments in production to help performing integration testing

- Each request is attached with a context (test/production/…)

- Uber message queue

- Fault tolerance

- Each message is delivered to extent which has multiple replicas.

- If one failed, all replicas in the same extent is sealed (allow read, disable write) and new extent is created.

- Producers and consumers maintain WebSockets with input/output host

- e.g. When an input host receive a message, it has to wait until the messages are replicated to all extents to acknowledge to producer. The acknowledgement will be sent later in WebSocket connection.

- Scalability

- Automatically launch new extents to increase throughputs

- Consumer

- Output hosts keep track of:

- delivered but unacknowledged message

- timing for retry purpose

- After exceeding retry limit, the message it put to dead letter queue for manual investigation.

- Merge if it was because of bugs in consumer

- Purge if the message is invalid

- Eventual consistency (AP queue)

- When network partition occurs, consumers read from its own partition. Thus hard to maintain order of messages.

- CP queue

- Extent creation needs to be linearizable: only one partition can create an Extent to succeed the previous sealed one.

- Moving from monolithic to SOA  
 - Benefits

- Deploy and manage separately

- Easier to make changes

- Challenges: Weak contract

- Apache Thrift

- Publish packages to lib

- MySQL vs. Postgres

- Postgres

- Replicate row-based log

- MySql

- Choice between row-based or statement base

- Row-based: verbose but might be more efficient

- Statement-based: simple but query could be expensive

- Monitoring system

- Collect metrics

- Application logs

- Servers metadata agent

- CPU/memory usage

- API latency

- QPS

- Batch logs and send to Kafka

- Queue

- Decoupling producer (metric collectors) and consumers (logstash)

- Highly reliable and scalable

- Logstash

- Accept metrics from different sources, process and send to multiple destinations

- Elastic search

- Kibana

- Visualization

- Alert

- Scheduler: AWS lambda

- Notification service

- Another monitoring system to monitor this one

- Exactly-once operation

- Two phase commit

- Idempotence

- Dead letter queue

- Simple retry

- Can be blocked by long-running batch jobs

- Publish each retry to a new queue, total we have N queues if max retry limit is N

- DLQ is the last one

- Delay each retry to avoid spamming bad requests

- Kafka

- Pull model for consumer

- Avoid being overloaded

- Producer send message streams to topic

- Consumer send offset to broker to find message and read from stream

- No cache, only rely on page cache

- Little overhead in garbage collecting its memory

- Stateless broker

- Broker does not track which message has been consumed

- Message is automatically deleted after a certain period of time

- Old message can be re-consumed with the help of this design

- Partition within a topic is the smallest unit of parallelism

- All messages from one partition are consumed only by a single consumer within each consumer group

- No overhead on tracking locking, state.

- Coordinate is needed only when rebalance the consumer load.

- Not have a central master node

- Not have to worry about master failure

- Zookeeper

- Detecting addition and removal of brokers and consumers

- Trigger rebalance process for consumers when needed

- Maintain consumption relationship and track consumed offset for each partition

- Lyft

- Use expiration to sign off inactive user

- Hash per region per 30 seconds bucket

- Pull future bucket to avoid clock drift

- Pull previous bucket to read data that’s closed

- Geohash

- Split a map to a grid

- input lat and long, return a geohash location with arbitrary precision

- Go deeper level to get better precision

- Configure geohash level to get rid of hot shard

- Use single key to represent user position

- Deal with memory issue

**Ownership**

1.      Tell me about a project you've worked in, recently or not, that you played an important role in. What was the goal of it?

a.      (Dig in. What was your role? What was your responsibility here? Deliver Results, Ownership, Bias for Action...)

Pen-test

Owner and developer

Gather requirements; Research on different options; Run pen-test before each release; Recommend pen-test to other teams

b.      What was the outcome of the project? Did you meet your goals?

Earned trust from client and other teams

**Deep Dive**

**1.**Tell me about a time when you were trying to understand a complex problem on your team and you had to dig into the details? What steps did you take to better understand this task? What was the part that you found was the most complex? What steps did you take to understand the code? How did you test your change?

Endpoint optimization

1. Requirement
2. Identify bottleneck (new relic)
3. Research on best practices
4. Implement solution

Most complex: identify bottleneck

Looking at log; Debug db connections; Use tool to help

Performance testing; Average latency on new relic

**Earn Trust**

1.      Can you give me an example of a time when your ideas were strongly opposed? How did you react? So who pushed back on you? What was the final outcome?

Rewrite a front-end component from scratch

Try to explain to my manager the best practices and difficulties we are having

Hybrid approach: keep duplicates and implement generic components, meet deadline

**Invent and Simplify**

1.      Give me an example of a complex problem you solved with a simple solution. What was the problem and what made it complex?

o   How do you know your solution addressed the problem?

* Talk about BOTH front-end (if you worked on it) and back-end, debuggin and testing

Create a contract in manual testing is inconvenient.

Chrome extension to fill the form

**Learn and Be Curious**

1.      Tell me about a time when you didn’t know what to do next or how to solve a challenging problem? How do you learn what you don’t know? What were the options you considered? How did you decide the best path forward? What was the outcome?

Context vs. Redux

Too many components, need to manage global used state in a single place

Both satisfied our need, context was easier to learn and implement.

Redux has more functionality: manage state separately from other logics, state transition highly reusable. Good for long term project

2.      Tell me about a time where you disagreed with you manager or a co-worker? What Happened?

Improve the create user flow even if the old flow was committed to the client

**Coding Problem**

1.      Give an array, print the NEXT GREATER ELEMENT (NGE) for every element. The next great element fro an element x is the 1st greater element on the right side of x in an array. Elements for which no greater element exist, consider next greater element as -1

2.      Given an infinite sorted boolean array find index where 0 changes to 1. [0011..] 1 [000011...] 3

Increment index i exponentially until find 1. Then binary search.

3.      For a hotel with n rooms (1 <=n < 10^6) = 2 and m reservation requests [{StartDate, EndDate}] (1<= m, StartDate, EnddDate <= 10^6) where reservation has to be exclusive between startDate <= reservation <= endDate and room will be occupied for the entire day, check if all m reservation requests can be fulfilled.

Merge intervals

4.      Given an input N, return all prime numbers from 0 to N. A prime number is a number that is not divisible by any number other than itself and 1

o   What's the complexity? Can we improve the complexity? If we consider that all non-prime numbers are made up of smaller prime numbers, can we do something with that information to improve the complexity?

5.      Come up with ideal Data Structure to implement Bag/Locker problem.

o   Lockers Small(n), Medium(n), Large(n)

o   Bag S, M, L

o   Token putBag(Bag bag); //T(O(1)) S(O(n))

o   Bag getBag(Token token); //T(O(1)) S(O(1))

6.      Are you familiar with the Binary Tree data structure?

o   How would you represent this data structure in code?

o   Can you write for me a function that computes the sum of the values in the tree?

o   TALK ABOUT HOW YOUR GOING TO SOLVE IT FIRST!! TALK TALK TALK

o   Unit tests? What kind of inputs?

7.      Let's say that a Binary Tree is a 'SumTree' if the value of each node is equal to the sum of the values of all the nodes beneath it.

For example: 6 = 2 + 4, 3 = 3 + nothing, **AND** 18 = 4+2+6+3+3            24 = 6+6+4+2+3+1+2

Could you write a function for me that takes in the root of a tree and returns  back true if it IS a valid SumTree, and false if it's not a valid SumTree?

8.      What's the big-O Runtime of the code for a tree with N nodes? Any way to do better than that?

9.      Suppose you cannot fit the whole Binary Tree in memory. What could you do to still produce the sum of the entire tree?

10.   Let's say we can store subtrees and load them arbitrarily, how could we run this subtree