Lecture 1 - ssds, lsm-trees, rocksdb

HDDs

-Rotating, magnetic disks

-Storing large amounts of data at low cost

-High access time that does not improve much over time

-200mb/s throughput

-The file system needs to be tuned specifically to rotating disks

SSDs

-Solid State Drives

-Purely electronic devices (no moving parts)

-Fast access time, low latency, lower power consumption, uniform random access speed

-Data is organized in pages (2-4kb), pages are then organized in blocks, where one block consists of 64 or 128 pages

-Reads and writes are done on page-level, while erase-operations are performed on complete blocks

-Data has to be erased before writing. All writing must happen on a clean area of the disk

-The only operation an SSD can do is to flip a 1 to a 0, not from 0 to 1.

-Throughput is many times better than HDDs, because they have parallell writes. GBs/s througput.

SSD wear leveling

-The SSD will eventually wear out when it is written to too many times. (limited number of writes possible)

-Wear leveling (spreading the writes around the whole disk, so that all areas of the disk wears out at the same time, maximizing the lifetime.

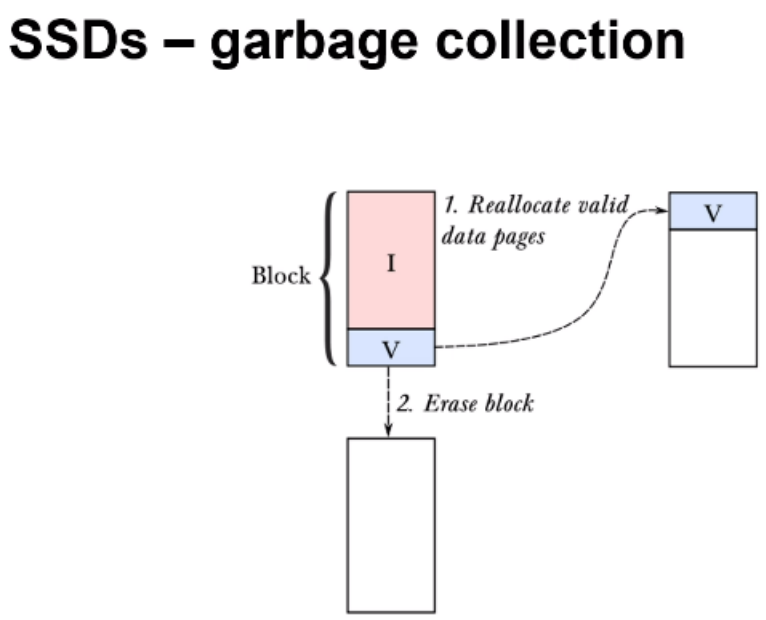
-Done by Flash Translation Layer

-Dynamic wear leveling: when performing updates, the old block is marked as invalid, and the block is relocated on the SSD.

-Static wear leveling: same as dynamic, but static (read only) datablocks are also periodically relocated, to ensure that the entire disk is being accounted for.

-Garbage collection (moving blocks around at writes and updates) is important in SSDs. Done in units of blocks.

-Today, array based FTL is always used, meaning that garbage collection is done on the disk itself.

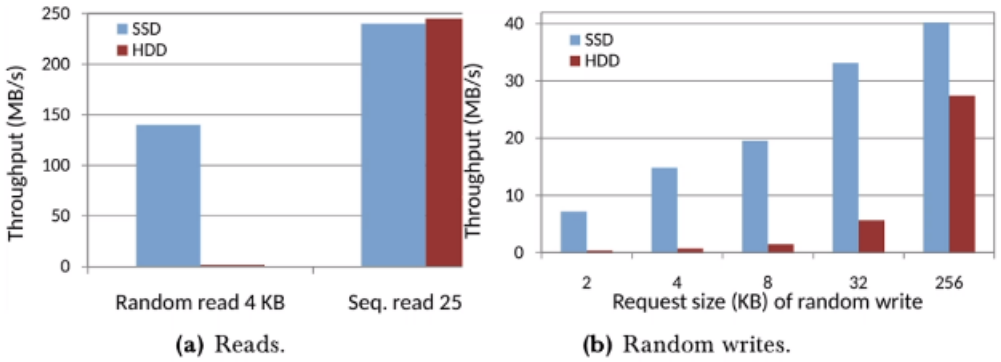


-SSDs have parallell I/O-operations. The more channels the SSD has, the more throughput it has, since channels can be used in parallell

-To take advantage of parallell writes, big blocks are preferred.

-Sequential writes has much better throughput than random writes

Sequential writes (2)



(a) Reads. **(b)** Random writes.

-We can see that the SSD performes very well with big blocks of data, because of parallellization

-To achieve great performace in SSDs, data should be organized so that you can pack several writes together, and write them at once. (LSM trees do this, which is why they are so good with large chunks of data

-Over-provicioning: about 7-28% of the SSD needs to be reserved for garbage collection

B+-trees

-Blocks are linked together with two-way-pointers

-Pointers on index-level as well

-There are a lot of pointers to each block, so if we want to split a block, we need to do a lot of writes to update all the pointers involved.

-Sequential inserts (inserting in ascending key-values) is better than random inserts, as you dont need to update as many pointers that way

Variants of B+-trees

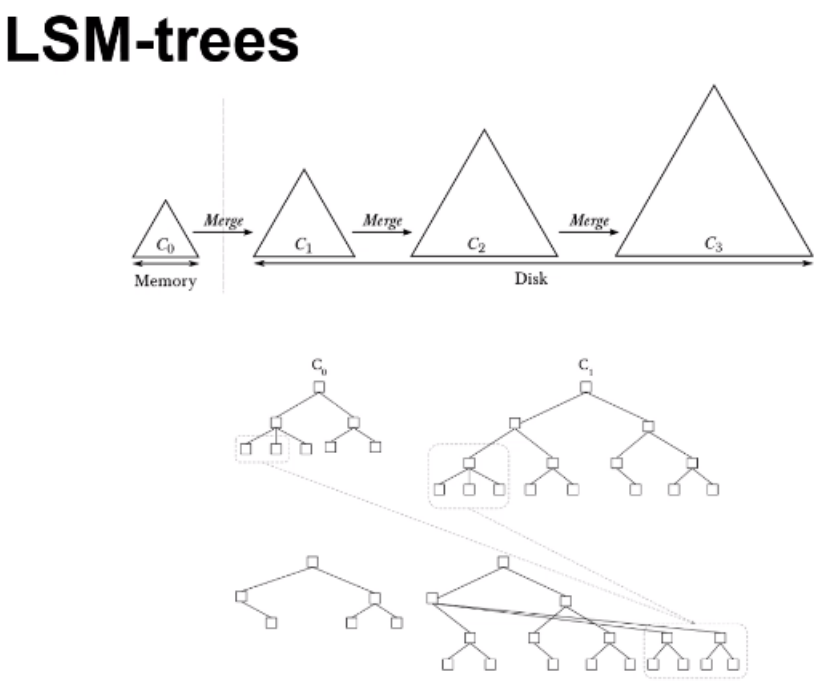
-FB-tree: simulating FTLs garbage collection (moving data when writing)

-Copy on write B-tree: copy block to new location at updates

-Write-Optimized B-tree: No sideway pointers, so you do not have to update as many pointers

LSM-trees

-Introduced in the 90s



-Rolling merge: when the first tree is full, parts of it is merge in to the next level tree, thats bigger.

RocksDB

-Made by Facebook, built on levelDB from google

-Multithreaded compaction

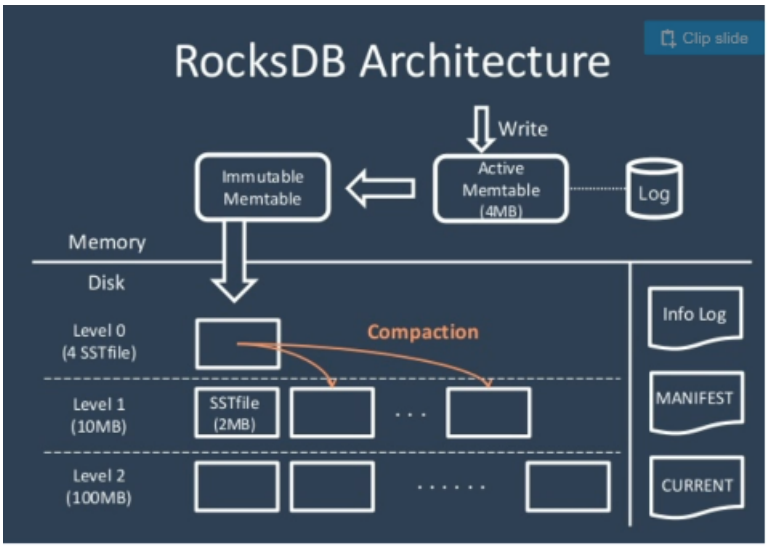
-Multithreaded insertions in to MEM-table (the part of the tree that resides in memory)

-Bloom filters: a way of answering whether or not the key you are looking for is possibly in a certain tree or not

-Because it has multithreading, it can utilize multi core-CPUs

-Large chunks of data, so it supports SSDs well

Layout:



-MEM-table is the same as SS-table (or SSFile), but mem is in memory, and SS is on the disk

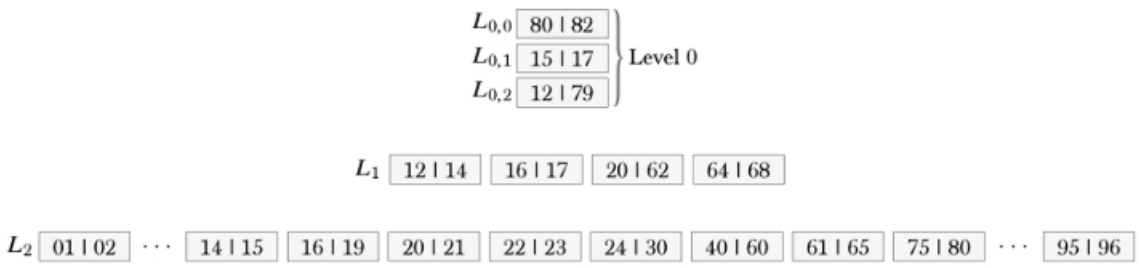
-Level below is about 10 times larger than level above

Leveled compaction

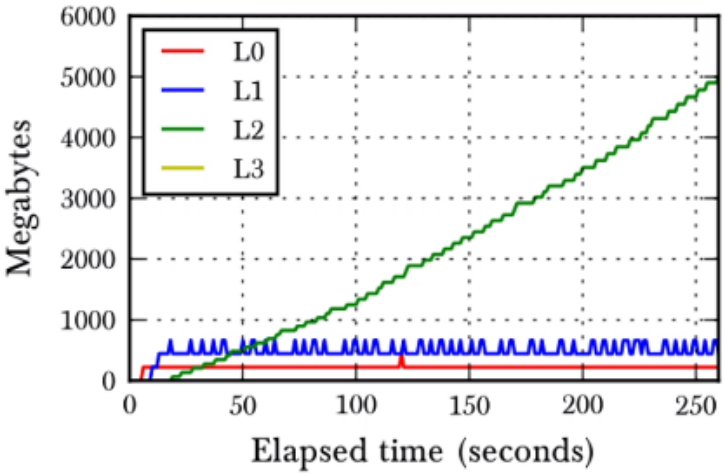
-Original compaction style in LevelDB

-SSTable files are stored in multiple levels

-Multiple overlapping SSTables at level 0 to get fast write of MemTables



-When deleting a record, you get a tombstone, showing that the key exists, but is deleted. So you dont really delete it, you just tell in a tombstone that it is deleted



(a) Leveled compaction.

-The levels increases until it reaches its size limit

Universal compaction

-Does not merge keys, but merges segments of time.

Bloom filter

-Data structure to check if a key **<u>may</u>** exist in a dataset

-A number of hash-functions are applied to each key that is inserted to the SSTable, and then insert them into a bit array

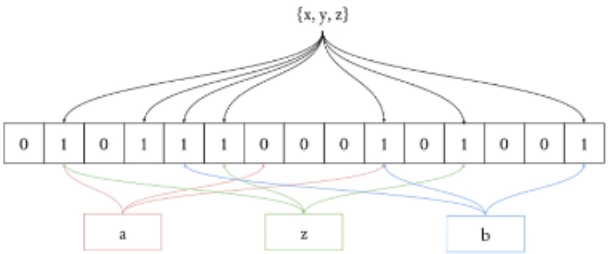


Figure 2.25: Example of a Bloom filter with the dataset x, y, z and $m = 15$ bits. There are $k = 3$ different hash functions.

-To check a in the figure above, three hash-functions are applied, and then we check if all the bits are 1. We can see that the second bit is 0, so a is not in the tree.

-If we check z, we can see that all the bits that it points to are 1, so z could be in the SSTable.

MyRocks

-Integration of RocksDB as a storage engine with MySQL

Write amplification

-Number of bytes written to api divided by number of bytes written to database

-B+-trees has high write amplification

-LSM-trees have less write amplification as long as the datachunks are big.

-Compaction is what causes LSM-trees to have write amplification,

-Write amplification is independent of record size in LSM-trees, because you are writing the complete SSTable

Write stalls and write stops

-Write stalls are needed in LSM-trees to have time to cope with compaction.

-Write stops are sometimes needed when so much data has been written that the entire disk has to be used for compaction for a while

Data structures that power your database

Hash indexes and datasegments

-You have a hashmap in memory, pointing to adresses on the disk

-When writing new data, it is appended to the end of data segment, writing sequentially

-Data segments is the compacted into smaller segments containing only the latest data

-If first data segment is full, we continue to the next segment, before merging them together and compacting

-Example with counting how many times four different videos are watched:

Merging + compaction

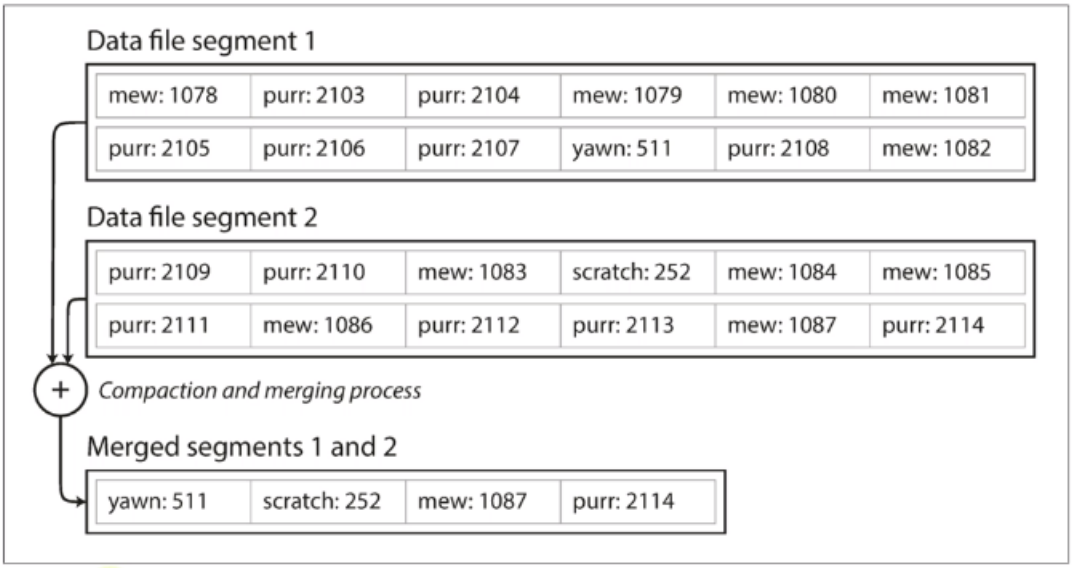


Figure 3 3. Performing compaction and segment merging simultaneously.

-File format: binary is better than cvs.

-To delete records in log-structured organization, tombstones are used to show that this record is dead.

-Crash recovery: store a snapshot of the hashmap on disk

-To avoid partially written records, checksums are usually used.

-Concurrency control: append only, so all updates are at the end of the log

-Problem: hashmap has to fit in-memory (usually not a problem with modern computers)

SSTables and LSM-trees

-Sorted String Table (basic storage unit)

-When doing writes in LSM-trees, you write complete SSTables, much larger than blocks or pages

-Sorted tables (can be sorted on strings for example such as in the figure

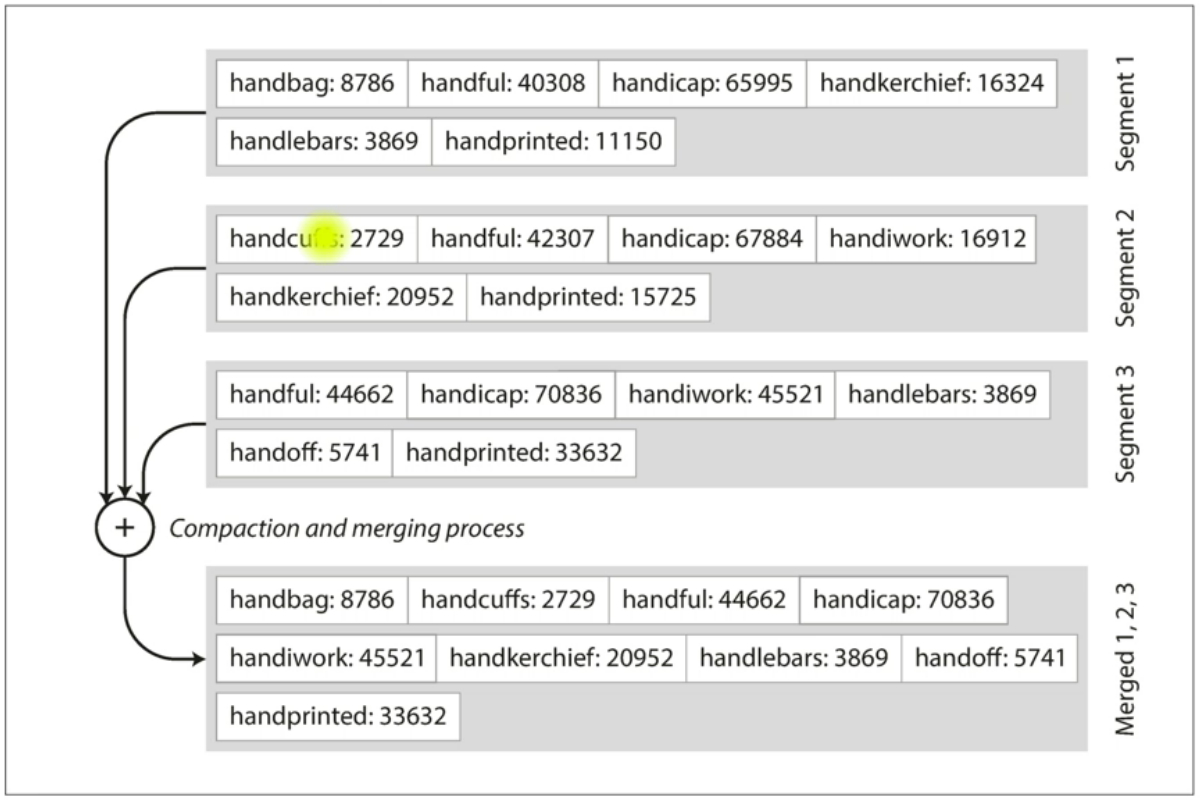


Figure 3-4. Merging several SSTable segments, retaining only the most recent value for each key.

-Above is three different SSTables being merged and compacted to a new SSTable.

-Since already sorted, merging is efficient.

-Each SSTable has its own index (e.g. SkipLists) both RocksDB and LevelDB use SkipLists.

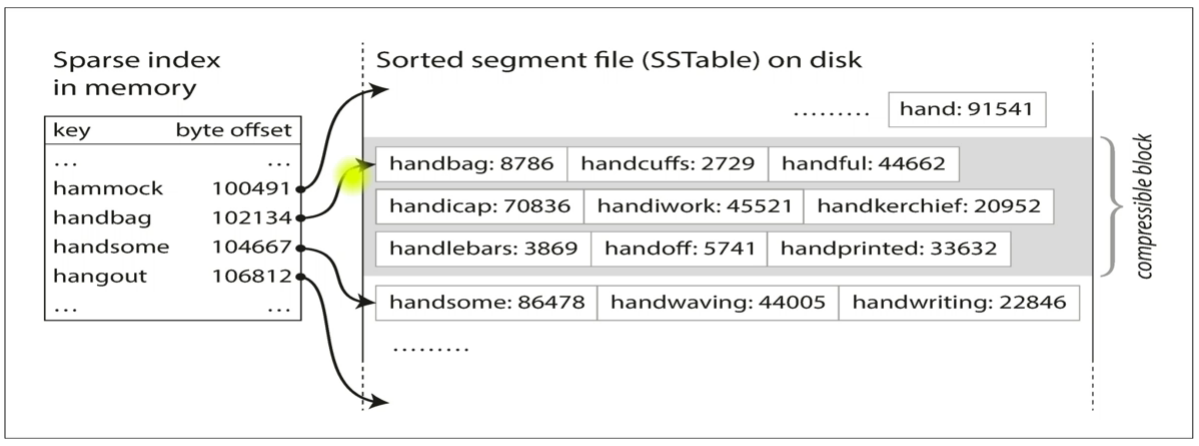


Figure 3-5. An SSTable with an in-memory index.

-This makes it easier to find data within SSTables

-SSTables are good when you have a lot of data being inserted (e.g. data from sensors)

-Also good that data you are searching for is new (so it probably lies in memory)

-Bloom filters are used in each MemTable and SSTable to make it easy to check if a key is NOT in the table.

B+-trees

-They are used everywhere. Standard database method

-Height-balanced tree with blocks as nodes, blocks contain records sorte don search key

-All records inserted by users are at leaf-level. (bottom)

-Typical height of tree: 2, 3, or 4 nodes

-Minimum 50% filldegree in blocks

-Average 67% filldegree in blocks

-Records are sorte don key

-b+-trees are good for most uses, except heavy right. B+-trees is not the solution for big data

LSM-trees vs b+-trees

-LSM-trees are much faster for writes because of low write amplification.

-b+-trees are usually faster at reads

-LSM-trees better for big data because compression is easier due to larges units (SSTables much larger than blocks)

-Compaction in LSM-trees might interfere with ongoing reads or writes (might have to stop compactions temporarily)

-B+-trees and LSM-trees have different use cases

Secondary indexes, heap files and clustered indexes

-Secondary indexes is indexing other parts of data than primary key.

Multi-column indexes

-Indexing two parts to be able to search base don a combination of the two (e.g. first and last name)

-Multiple dimensions (spatial index):

```
SELECT * FROM restaurants WHERE latitude  > 51.4946 AND latitude  < 51.5079
                          AND longitude > -0.1162 AND longitude < -0.1004;
```

-B+-trees do not support spatial indexes

-R-trees has multiple dimensions built in. Used as stardard solution. Slow at inserts, fast at reads.

Keeping everything in-memory

-Disks require careful layout to become performant. Data that is often needed together, should be stored close to each other.

-in-memory databases such as MemSQL are oriented towards ram and CPUs etc. and not the disk

Transaction processing or analytics

-Databases were traditionally used for business processing

-OLTP vs OLAP

- OLTP -- Online transaction processing is used for any access pattern where you need answers quickly
- OLAP – Online analytic processing is used for access patterns which do analysis of large datasets

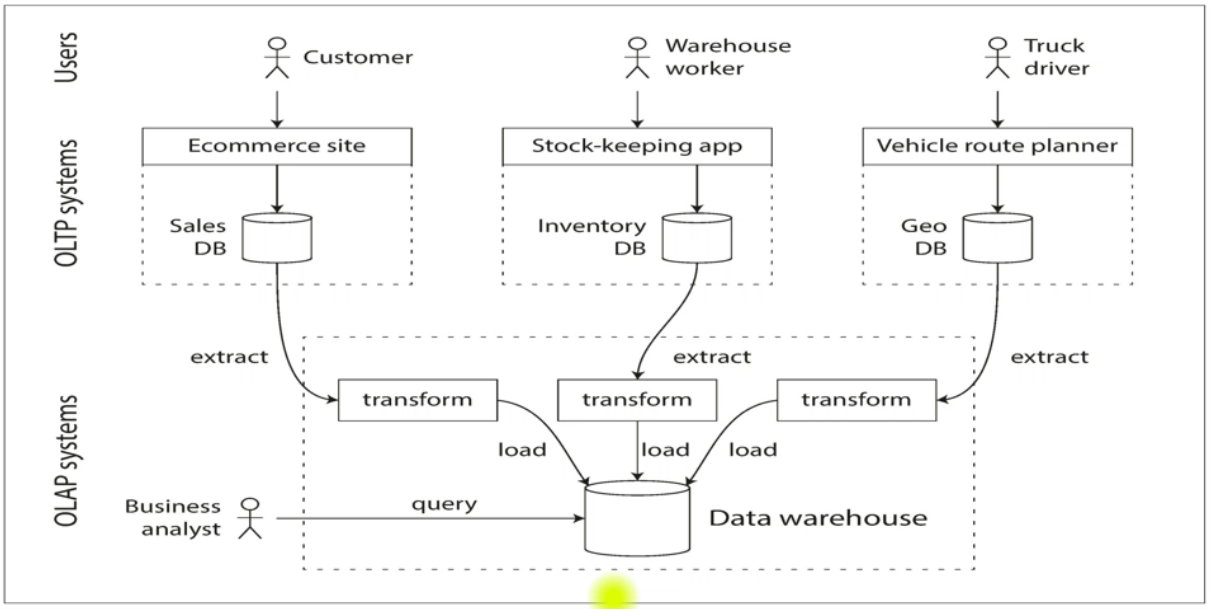
Table 3-1. Comparing characteristics of transaction processing versus analytic systems

Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End user/customer, via web application	Internal analyst, for decision support
What data represents	Latest state of data (current point in time)	History of events that happened over time
Dataset size	Gigabytes to terabytes	Terabytes to petabytes

Data warehousing

-Warehousing is useful for collecting data from different sources. An enterprise may have hundreds of OLTP systems

-Most of these are operated by separate teams



OLTP vs data warehouses

-Warehouses use SQL, but may use other types of indexes

-Data warehouse often store columns, while OLTP often store rows

-OLTP uses row storage, OLAP uses column storage

-Compression is very easy with column storage, as all values inside the same column are of the same datatype

-Bitmap encoding:

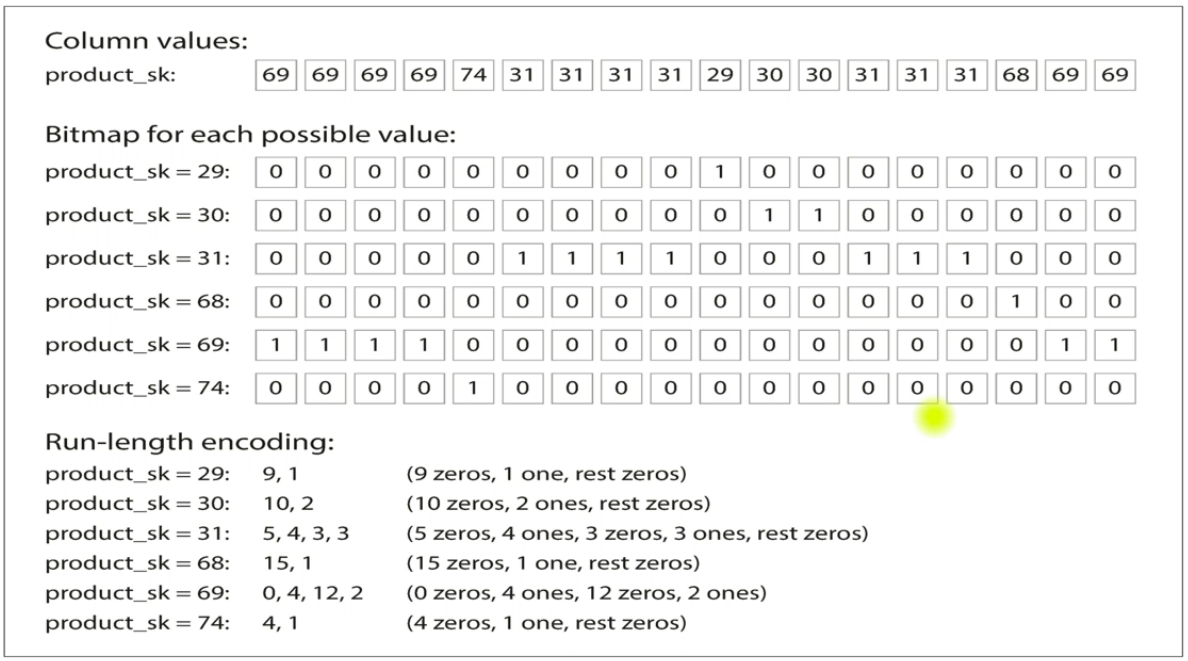


Figure 3-11. Compressed, bitmap-indexed storage of a single column.

Data warehousing techniques

-Data warehouses do some hardware-close optimalizations

-One trick is to not use jumps in the CPU, but instead of looping over one instruction, replicate the instruction as many times as you are supposed to loop over it. This is because jumping destroys cachelines etc.

-Sorting rows by date, because this is typical for queries

-Store the same data in different sort orders, so you can use the version of the data that fits the query best

-Materialized aggregates such as COUNT, AVG, SUM is computed in advance, so that when queries like that show up, it does not need to look through the entire column

Data cubes / OLAP cubes

- Aggregates in several dimensions: Making some queries faster

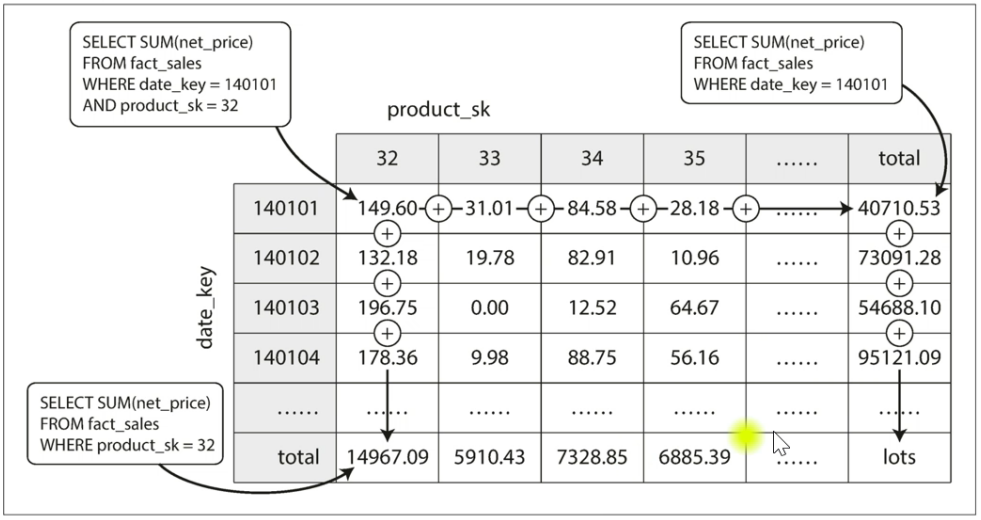


Figure 3-12. Two dimensions of a data cube, aggregating data by summing.

Summary OLTP vs OLAP

- OLTP databases: Online users
  - Log structured: LSM trees
  - Update in place: B+-trees
- OLAP databases: System analytics
  - Column stores
  - Data cubes

Evolvability

-SQL databases assume a schema, can be changed with ALTER TABLE

-Schema-on-read databases are document databases. You dont have a schema, but create the schema while reading databas. This means that the database may contain older and newer formats.

-Backward compatibility: newer code can read old data

-Forward compatibility: Older code can read new data

-Forward compatibility is way harder to achieve

Formats for encoding data

-Inside memory, data is kept in objects, structs, lists, arrays, hash tables, trees, etc. Use pointers

-When writing data to file or sending it over network, you cant use pointers in the same way. A pointer in one process may point to different data in another process.

-Endocing/decoding: sending and recieving data

-Built in language-support. Ex java has Serializable

Data formatting languages

-JSON (supported by browsers and javascript)

-XML (complex data formats)

-CSV (simple data formats)

-One problem of these formatting languages is that when for example sending a number, there is no way to know if it is supposed to be a number or string.

-Has support for UNICODE, but no support for binary strings.

Binary encoding

-Another way to do encoding

-Tyipcally used for internal data between programs that know about each other

-Compacter and faster than textual encoding

-Binary encodings for JSON: MessagePack, BSON, etc.

Schema evolution

-Thrift/Protocol buffers use schemas and fields are identified with tag numbers

-Field names can be changed and added, but the new fields must be optional or have default values. This is so that the new field can be inserted for all records. Important so new software may recieve messages from old software

Advantages o fusing a schema

-More compact as they dont use field names, but just the values and field tags

-Schemas are documentation, if you dont have them you dont know what the message contains unless you read the code.

-Storing different versions of the schemas can be used to achieve forward and backward compatibility

Dataflow through databases

-Storing data in database, and letting other applications read from it

-Data outlives code: old data may exist.

-Problems can occur when having same code read both old and new data

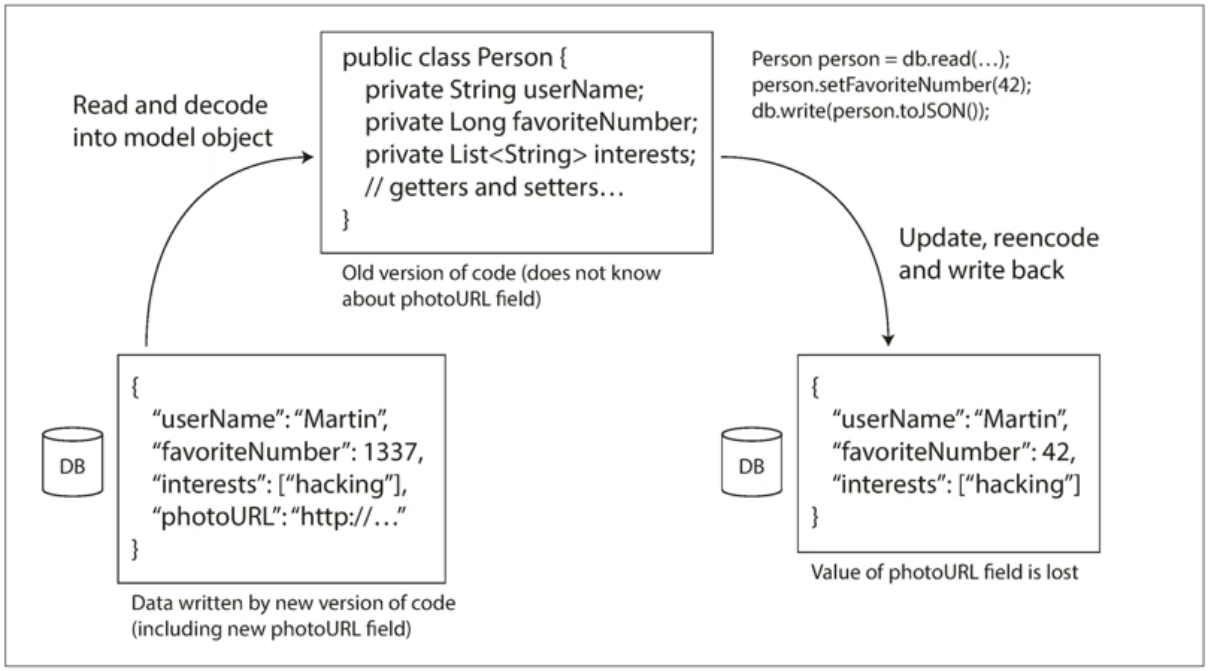


Figure 4-7. When an older version of the application updates data previously written by a newer version of the application, data may be lost if you're not careful.

-Here, the photoURL is not accounted for in the code.

Dataflow through services

-Clients and servers: Servers expose services using API

-Web browsers and servers:

    -GET to download HTML, CSS, javascript, images etc.

    -POST to submit data

-API uses standard formats: http, URLs etc.

-Applications may also use these protocols and formats (AJAX)

Web services

-Uses http as underlying protocol to communicate with service

-REST (a design philosophy)

    -uses http for cache control, authentication etc.

    -URLs to describe resources

-SOAP (base don XML and its API)

Summary

-Data encoding should support rolling upgrades

-Must ensure backward compatibility

-Programming language solutions such as Serializable are efficient, but locked because they require that reader uses same programming language

-Textual formats such as JSON, XML, and CSV support schemas, but are vague about data types

-Binary schema-driven formats like Thrift, prototol buffers and Avro allow compact, efficient encoding

-Modes of dataflow: Trough databases, RPC and REST API, message passing

Chapter 5 - replication

Distributed data (chapter 5-9)

-Scalability – partitioning – spread the load across multiple machines (the goal is linear scalability)

-Fault tolerance – replication – takeover when one machine fails (high cost since you need multiple copies, but necessary for some applications ex banking)

-Latency – locality to the application when data is distributed globally

-Shared nothing-architecture – every computer in the system has its own CPU, memory, disk etc.

-Shared disk – many computers share same disk through network

-NUMA – Many CPUs, but each CPU has its «local memory». The CPU could also access other parts of the memory than its own local memory, butt his would cost a lot more

Shared nothing-architecture

-Each machine is a node having its own CPU, memory, disks and network interface.

-Coordination between nodes done thourgh network messages. If you want to acces data on another computer, you have to send messages.

-No special hardware is needed

-There are some constraints and tradeoffs that occur in such distributed systems

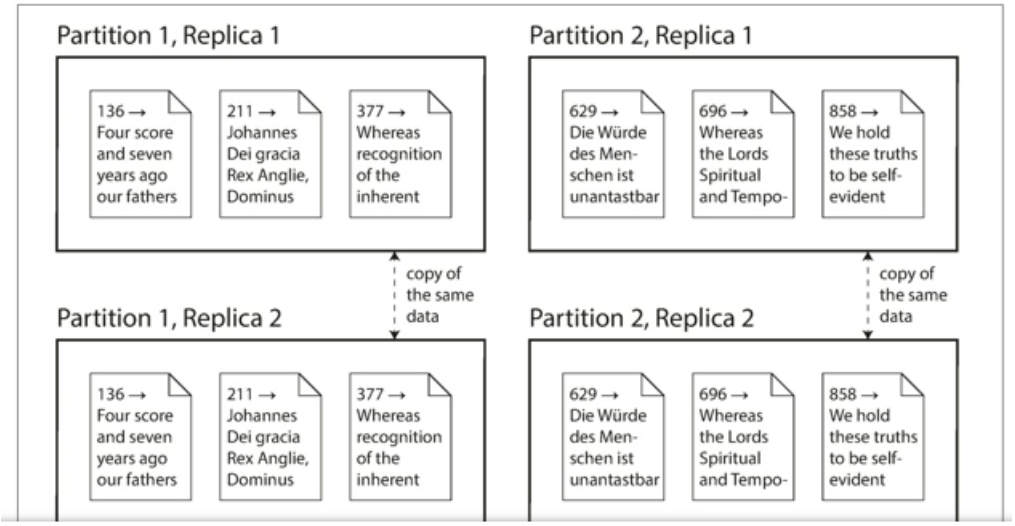
Replication vs partitioning

-Replication: keeping a copy of the same data on several different nodes (computers), potentially in different geographical locations.

-Provides redundancy (and therefore fault tolerance)

-Partitioning: Splitting a big database into smaller subsets called partitions so that different partitions can be store don different nodes.

-Provides scalability



Replication

-To keep data geographically close to your users to get lower latency

-Fault tolerance, increased availibility

-Can read from the closest copy at queries

-If the data being replicated is fairly constant, then replication is easy

-Replicating changes between nodes:

-single-leader

-multi-leader

-leaderless replication

-Synchronous vs asynchronous replication. In synchrounous, the user getting a reply awaits that the different copies are updated before it gets a reply, while in asynchronous, the update goes on after you have replied.

Leader-based replication

-Also named active/passive, master-slave, primary/backup, and primary/hot-standby

-Writes are sent to the leader, which writes the changes to local disk

-Followers are then sent the changes through a replication stream / change log

-The followers must apply all writes in the same order as they were processed on the leader

-Any replica may be read, but only the leader can accept writes

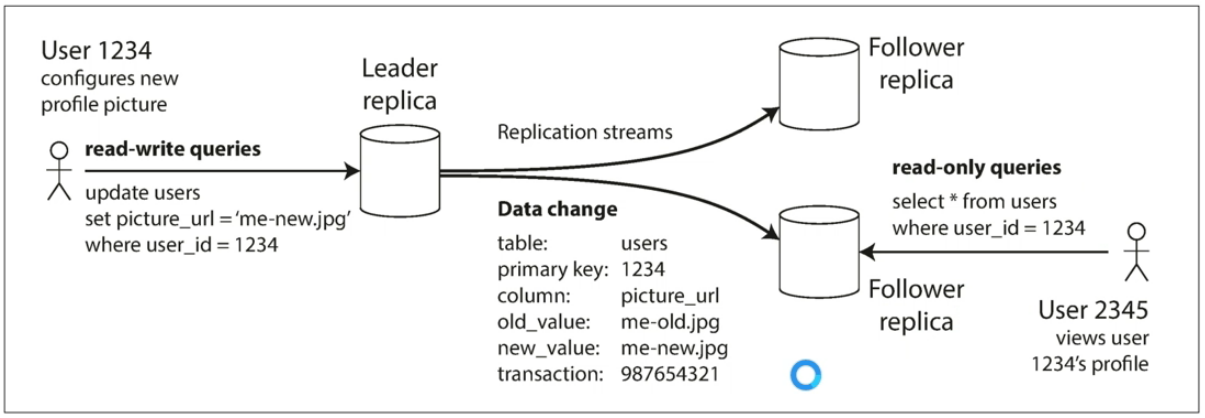


Figure 5-1. Leader-based (master–slave) replication.

Synchronous vs. Asynchronous Replication

- Is the replica update included in the client reponse (synchronous)
- More complicated failure scenarios for asynchronous

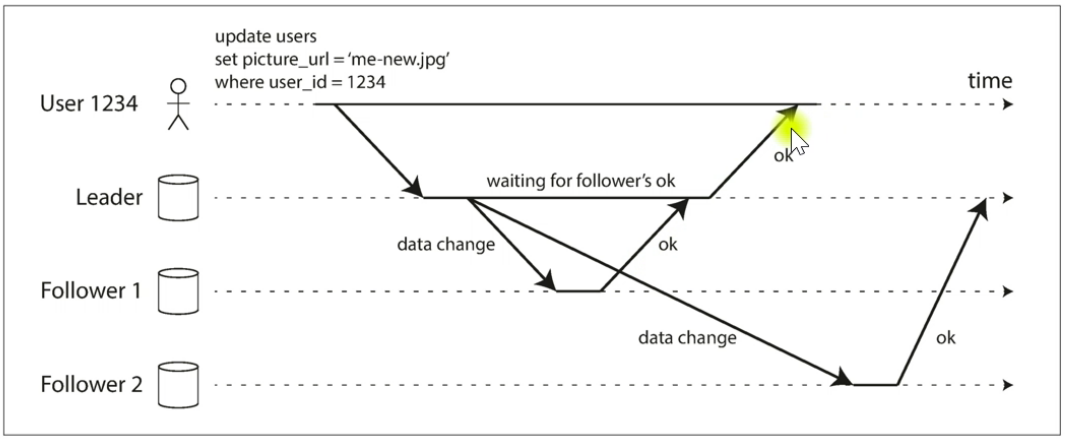


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

-Figure shows synchronous replication.

-In asynchronous, you dont wait for a reply from any of the followers before replying back to the client (meaning there is no waiting)

-Asynchronous is way faster, but more complicated failure scenarios.

Asynchronous replication

-Is more used than synchronous

-Often, leader-based replication is completely asynchronous

-If the leader crashes, operations can be lost because the leader could not manage to send the change to any of the followers before crashing.

-The leader can continue to process writes, even though all of its followers have fallen behind. This causes weakened durability. Asynchronous replication is still widely used because of its performance.

Adding new replicas (repair?)

-Adding new replicas must be supported, in case computers are being changed out etc.

-First take a consistend snapshot of the leaders database at some point in time

-Copy the snapshot to the new follower node

-Copy the log (all the data changes that have happened since the snapshot was taken

-Process the log, and we are now caught up to the leader

Takeover / failure handling

-You need failure detection (software that runs some protocol to see which other computers are alive and not alive

-Election of new leader, or pre-determined

-If a leader does not respond «i-am-alive», then the copy might do a takeover and become the leader.

-Problems:

> -Some of the updates residing in the computer that failed might not have been recieved
>
> by the followers before takeover. This means that the old leader may have «lost updates» when recovering.
>
> -In this case, the new leader (the one that took over) should «bumb up» anything used for identification (auto-increment keys and log sequence numbers) so that when the old leader is running again, the new leader has some room to implement the updated that were lost.
>
> -If you dont have a proper system for checking «i-am-alive», you can get split-brain, where two new nodes both think that they are the leader.

-What timeout to use? If the timeout is too small, you might detect false failures where the computer wasnt really down, but was just having some trouble causing it to use more time to respond with «i-am-alive». This can lead to doing unnecessary takeovers.

If the timeout is too big, there will be a long time of unavailability (time without leader)

Statement based replication logs

-For SQL: INSERT, UPDATE, or DELETE statement is forwarded to followers

-Problems:

-nondeterministic functions such as NOW() or RAND(), that gives different values for the leader and the followers

-Autoincrementing columns

-Used in MySQL prior to v5.1, but now row-based replication is used instead

-VoltDB uses statement based replication logs, but then you are not allowed to use nondeterministic functions such as NOW().

Write-ahead Log-shipping

-Log: append-only sequence of bytes containing all writes

-When the follower then processes this log, it builds a copy of the exact same data structures as found on the leader.

Logical (row-based) log replication

-Different formats on log and local storage (e.g. table names and primary keys are stored in the log, not BlockIds etc.)

-Inserts, delets, and updates generate different log records containing before/after images

-MySQLs binlog uses logical log replication, and also ships commit log records

-Leader and followers may have different storage formats and storage engines, you could for example use innoDB on the leader and RocksDB on a follower.

Trigger-based replication

- When flexibility is needed
- Replicate a subset of the data
- Need conflict resolution
- Triggers and stored procedures let «application code» be run when updates appear
- The trigger may log the update into a separate change table, which again may be read by a replication process
- General approach with higher runtime cost that special purpose replication approaches

Problems with replication lag

-If the replication takes too much time

-Leader-based replication is OK for loads having mostly reads and a few writes, so that replication is not that frequent

-Because of replication lag, you may read out-of-date info from a follower: Eventual consistency

-The lag ranges from milliseconds to minutes at high loads

Reading-your-writes-consistency

-Write-then-read-what you have written

-Problems occur when reading a not-updated replica/follower

-Example from a user inserting a comment:

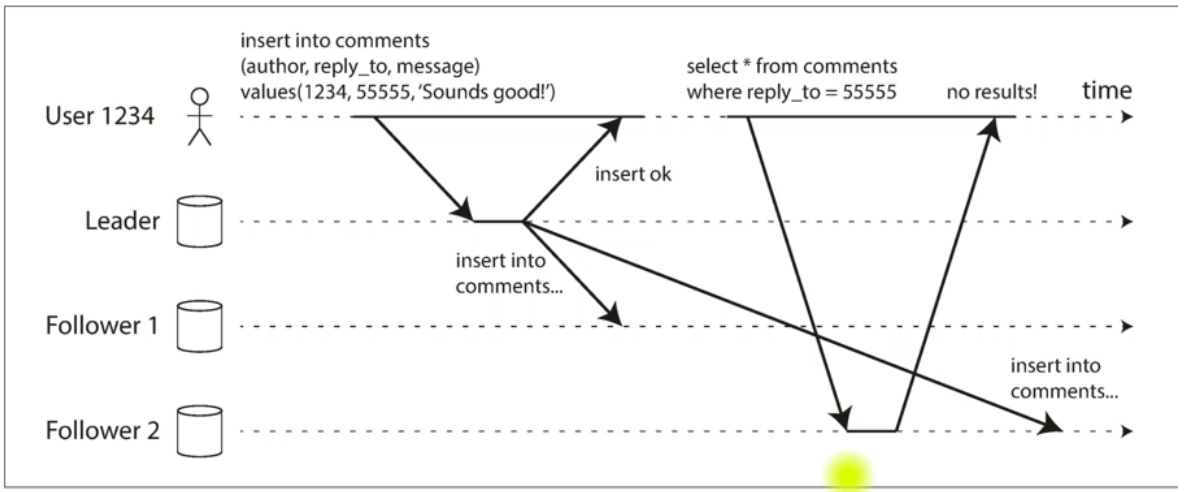


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

-Here, we can see that we are reading from follower 2 before the new comment has been inserted to that follower. The select-statement would therefore not return the new comment.

Reading-your-write-consistency

-One way to ensure reading-your-write consistency is that when reading data that you have access to change, you should always both write and read to/from the leader

-When reading data that you do not have access to change, you can read it from anywhere

-Another way is using logical timestamps: register write timestamps, and use this when reading replicas

Monotonic reads

-Monotonic reads is a guarantee that you never read older versions of data

-This may be ensured by making reads always to the same replica by hashing on the UserID

Consistent prefix reads

-If a sequence of writes happens in a certain order, then anyone reading those writes will read them in the same order: related data goes to the same machine

Solutions for replication lag

-Replication lag is typically not a big problem

-Use better guarantees such as read-your-writes

-Use transactions (high cost)

Multi-leader replication

-Allows more than one node (machine) to accept writes

-Also called master/master or active/active replication

-A leader in each datacenter

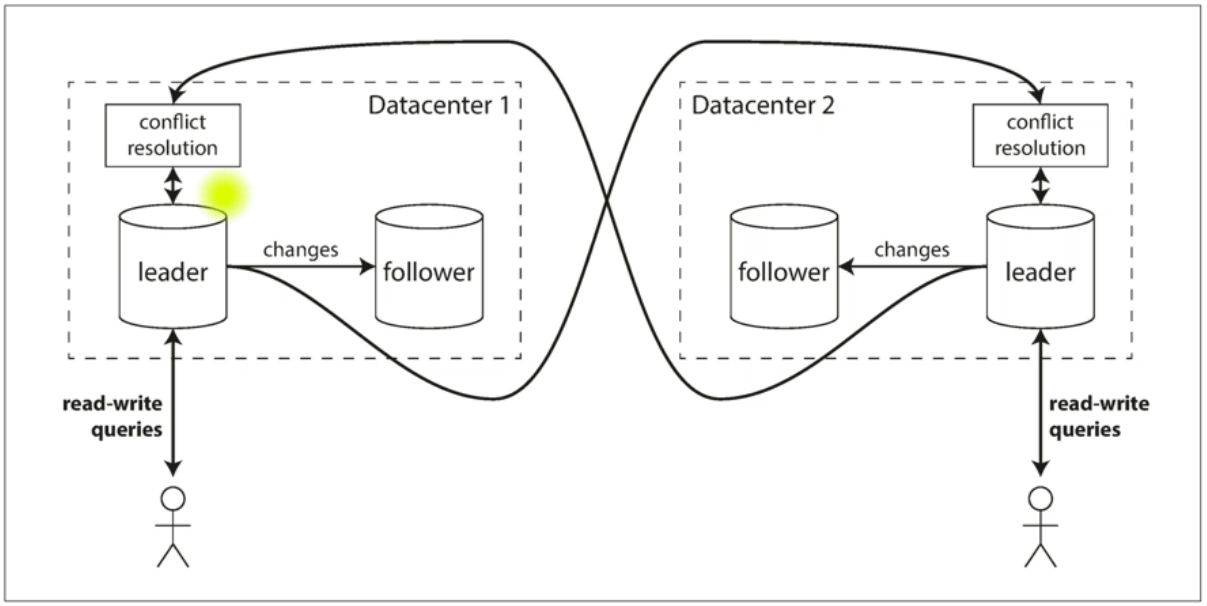


Figure 5-6. Multi-leader replication across multiple datacenters.

-Problem: conflicts when the same data is updated in multiple places at the same time

-Good performance because all writes are local

-Has fault-tolerance for complete datacenter-outages, as you can then use another datacenter

-Has fault-tolerance for network problems

-Needs conflict resolutions

Handling write conflicts

-Write conflicts are the biggest problem with multi-leader replication

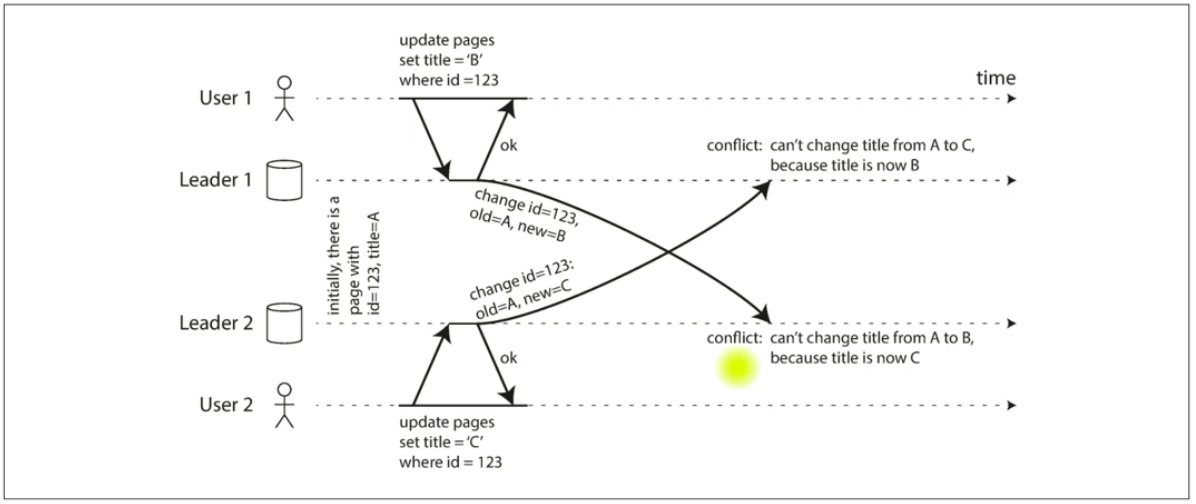


Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.

-Asynchronous conflict detection

-Conflict if not detected when doing the update, but someone might detect it later when they read it

-Synchronous conflict detection

-Conflict is detected when doing the update

-Converging to a consistent state

-There are no «correct» values when you have write conflicts, as both users trying to update the data are worth as much as each other.

-No correct value, but consistent values are the goal

-Each write can have a unique timestamp, and last write wins

-Each replica has a unique ID, and the highest ID wins

-Merge values and keep all of them (B and C in the example), and then do come conflict resolution on them

Conflict resolution

-On-write conflift resolution: As soon as the database detects a conflict in the log of replicated changes, it calls the conflict handler

-On-read conflict resolution: When reading and detecting that there are multiple conflicting writes stored, a flag is set that tells user that conflicting writes are stores, and something has to be done

-Conflict-free replicated datatypes (CRDTs): standard data structures to be used concurrently without locking

-Mergeable persistent data structures: Utilizes history and uses 3-way merge

-Operational transformation: Concurrent editing of documents (Google Docs).

Multi-leader replication topologies

-How are the leaders connected?

-A replication topology describes the communication paths along which writes are propagated

-Stop propagating a message when it is recieved at the sender

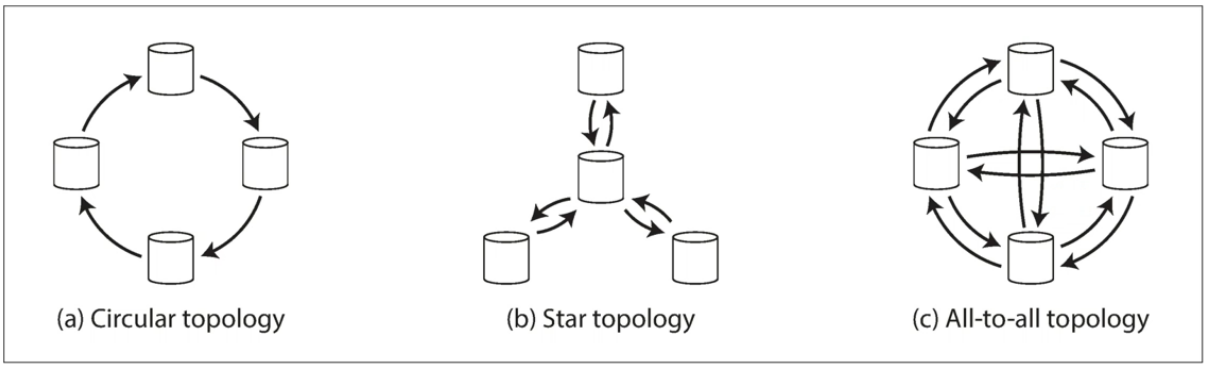


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

-One problem with multileader replication is that messages may be recieved in the wrong order

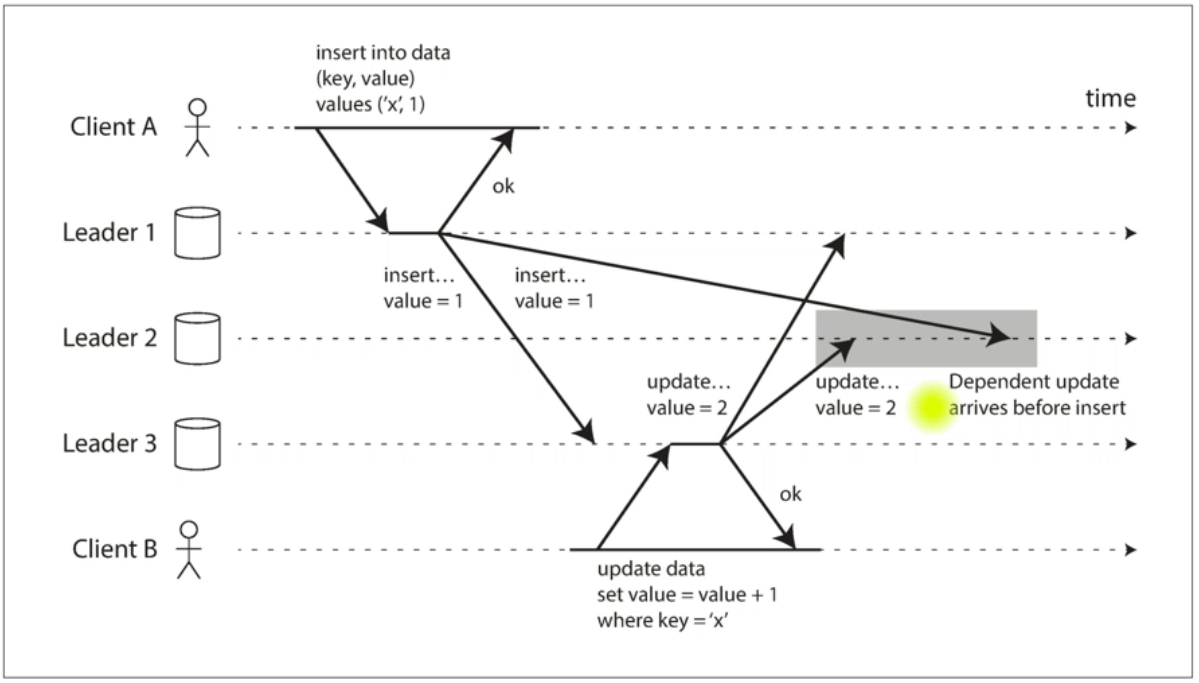


Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.

-May be solved by version vectors

Leaderless replication

-Dynamo-style of replication

-Several different machines, but no one is the leader

-Uses Quorums, which means that every write or read writes or reads a number of copies.

-This makes it so it can use read repair:

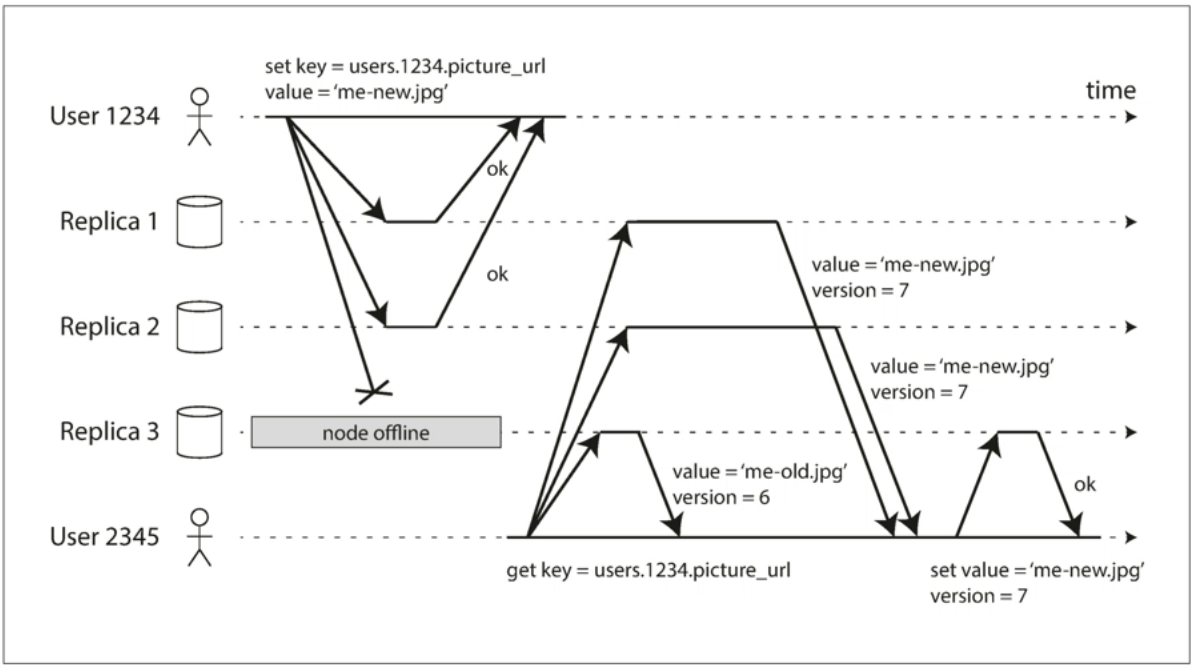


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

-Here, we can see that the first user sets a new value for a picture, and the write is done both in replica 1 and 2, but replica 3 is offline at that moment. Then, when a new user tries to read the data, it can see that replica 3 has the old value, while 1 and 2 has a new value, so it then updates the value for replica 3 as well. This is calles read repair.

Qourums

-You have N number of replicas, and every write must be confirmed by W number of nodes (machines)

-Query at least R number of nodes for each read

-As long as w+r > n, we expect to get an up-to-date value

-In dynamo-style databases, you can usually configure the w, r, and n parameters to suit your availibility needs.

-Most common approach: w = r = (n+1)/2 (rounded up)

-Another approach: w = n and r = 1: fast reads, but all nodes are written

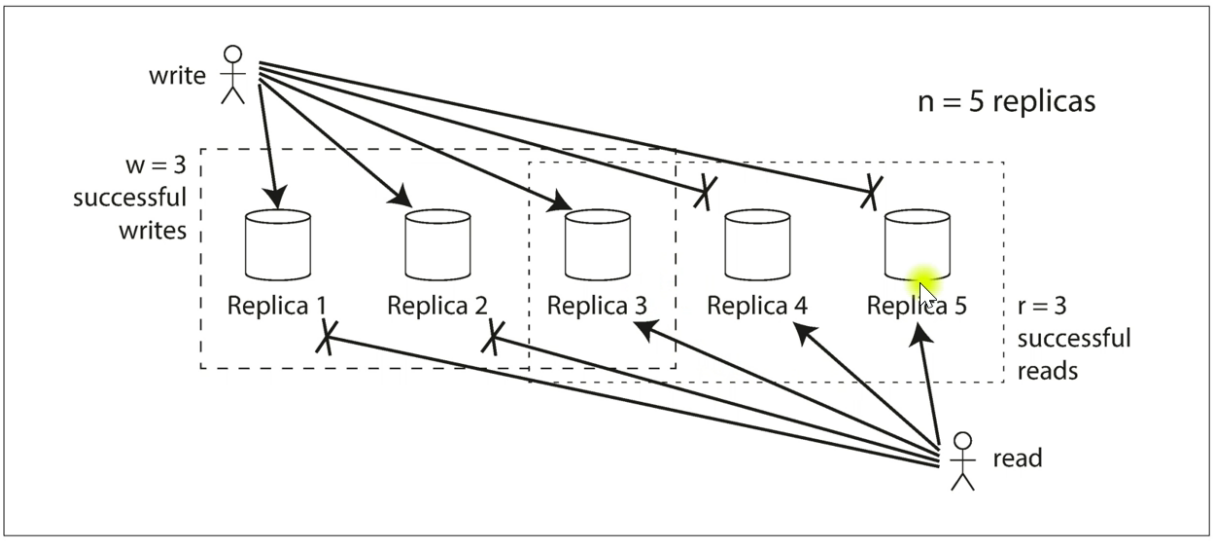


Figure 5-11. If w + r > n, at least one of the r replicas you read from must have seen the most recent successful write.

Sloppy quorums and Hinted handoff

-Dynamo-style systems, with some unreachable nodes

-Dynamo trades consistency away for availibility

-In dynamo, if trying to write to 3 nodes, but one of them is down, you still write to 3 nodes, but one of them is not among the n nodes on which the value usually lives

-Sloppy quorum means that it can be some extra nodes that are being used when some of the n nodes are down

Multi-datacenter support

-Multi-leader replication is used in multi-datacenters.

-Cassandra and Voldemort: writes to all replicas, but usually awaits only response from local replicas

Detecting concurrent writes

- Problem: Events may arrive in a different order at different nodes, due to variable network delays and partial failures.

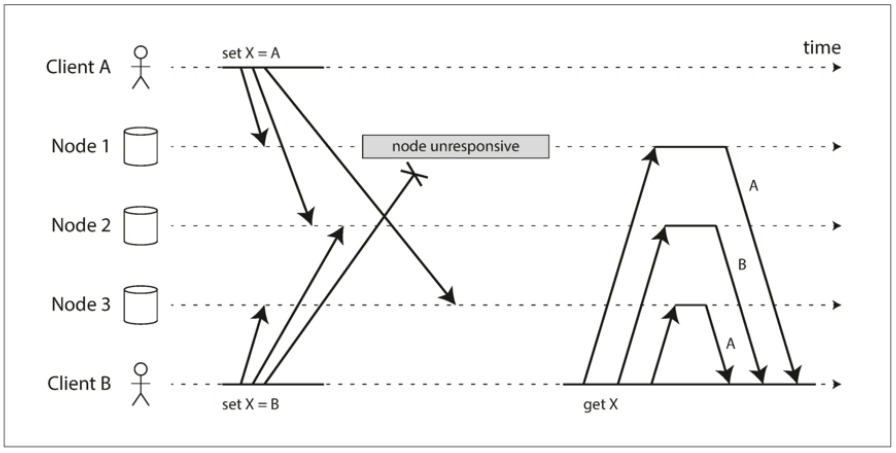


Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

Solutions to concurrent writes

-Last write wins (every update has a timestamp, and the newest timestamp always wins=

-In LWW, the problem is solved in the sense that the data is consistent in all replicas, but the client that lost its data will not be happy

Causality: Happens before and concurrent

-Happens before: one operation is dependand on another operation.

-An operation B depends on another operation A, when the user has read A's value

-A happens before B, or B is causally dependant on A

-If two operations are concurrent, then none of them happens before the other

-In this context, exact time does not matter. Two operations are concurrent if they are both unaware of each other

Capturing the happens-before relationship

- Single-node case: Two clients adding items to the same shopping cart

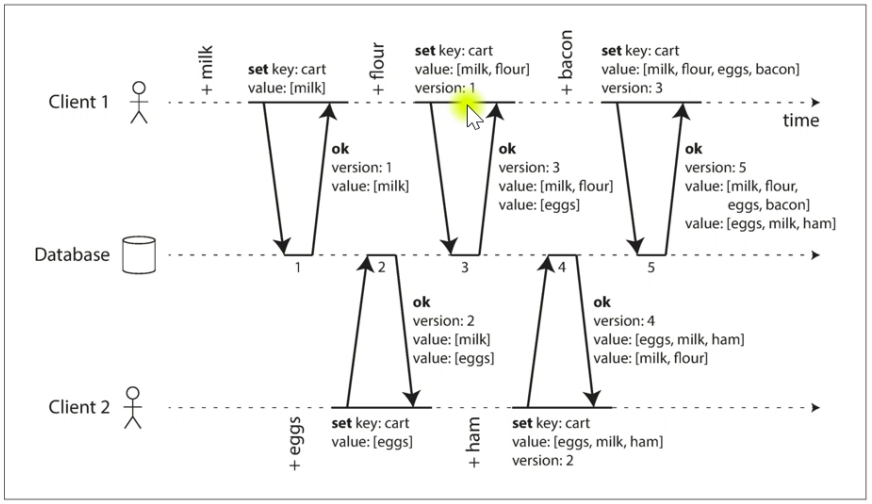


Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.

Algorithm to decide overwrite or «concurrent»

- Server maintains version number for every key, increments version number at writes
- At read, the version number is returned. A client must read before writing.
- When writing, it must include the version number of the read.
- When the server receives a write with a (read) version number, it can overwrite all values with that version number and lower.
  But it must keep all values with a higher version number (they are «concurrent»).

Summary chapter 5

-Why replication?

- -High availibility
- -Disconnected operation
- -Better latency (because of geographical closeness)
- -Scalability

-Types of leadership

- -Single-leader replication
- -Multi-leader replication
- -Leaderless replication

-Types of replication

- -Synchronous
- -Asynchronous

Chapter 6 – Partitioning

Partitioning concepts

-For very large datasets or very high query throughput

-We break the data into partitions

-The main reason for partitioning is scalability

Query execution and partitioning

-For single partition queries, each node can independently execute the queries

-Large, complex queries (with for example joins) can potentially be parallellized across many nodes, and by shipping SQL and/or data across the network

Range partitioning

-Partitioned by the order of the search key

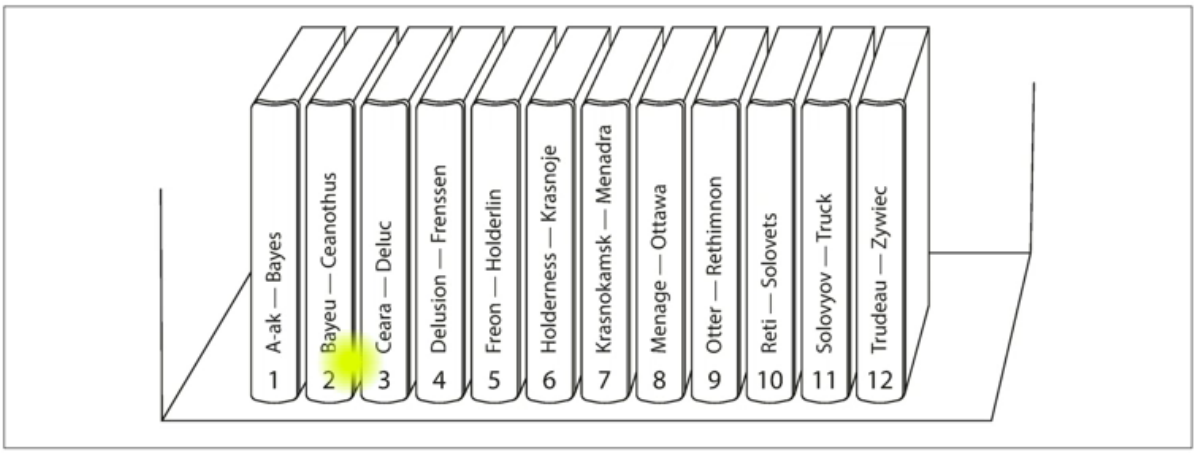


Figure 6-2. A print encyclopedia is partitioned by key range.

-May lead to some hot spots (e.g. newest items if partitioning by timestamps, which you should not do)

-Repartitioning may be necessary when scaling

Hash partitioning

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

-Gives good load balance, but no range scans. You have to do range scans on all partitions

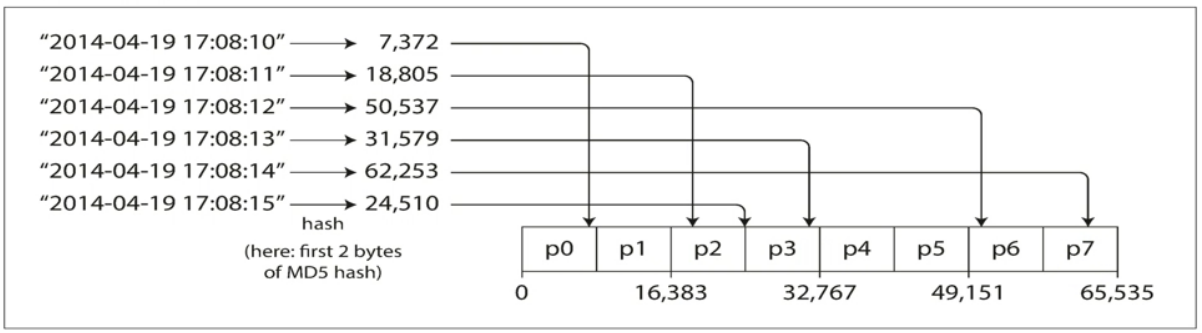


Figure 6-3. Partitioning by hash of key.

Skewed workloads and Hot spots

-Sometimes hashing does not load balance well

-A technique to solve this is to let the application use an additional random key to the start or end of the shared key

-It is also possible to try a different hash function, if the current one you are using does not load balance well.

Document-based indexing

-Also called local indexing

-Each partition has some data, and is partitioned by the primary key. The index for each partition is stored locally, connected to each partition

-Read from all nodes

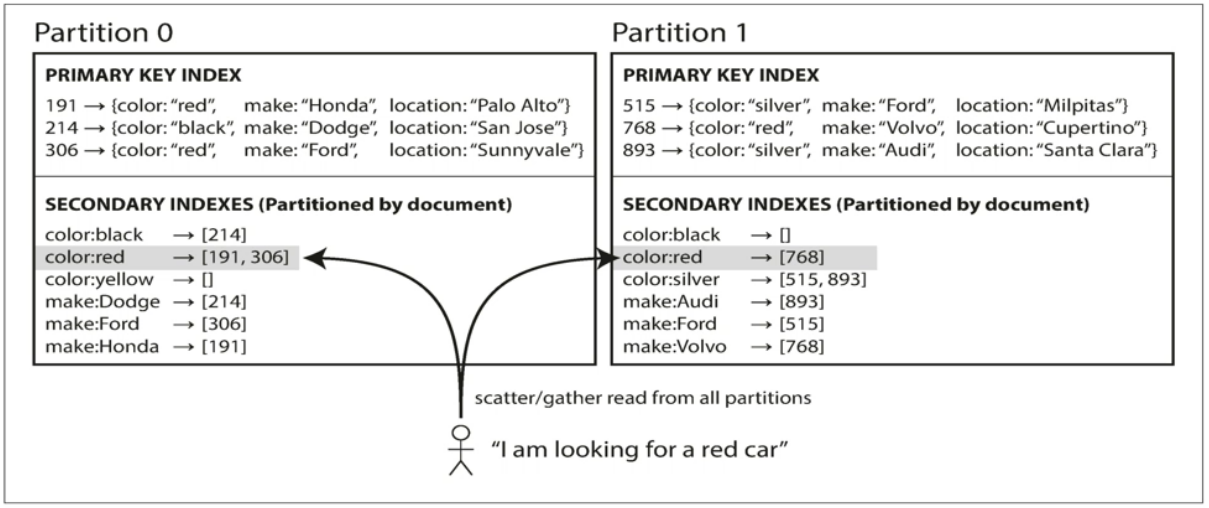
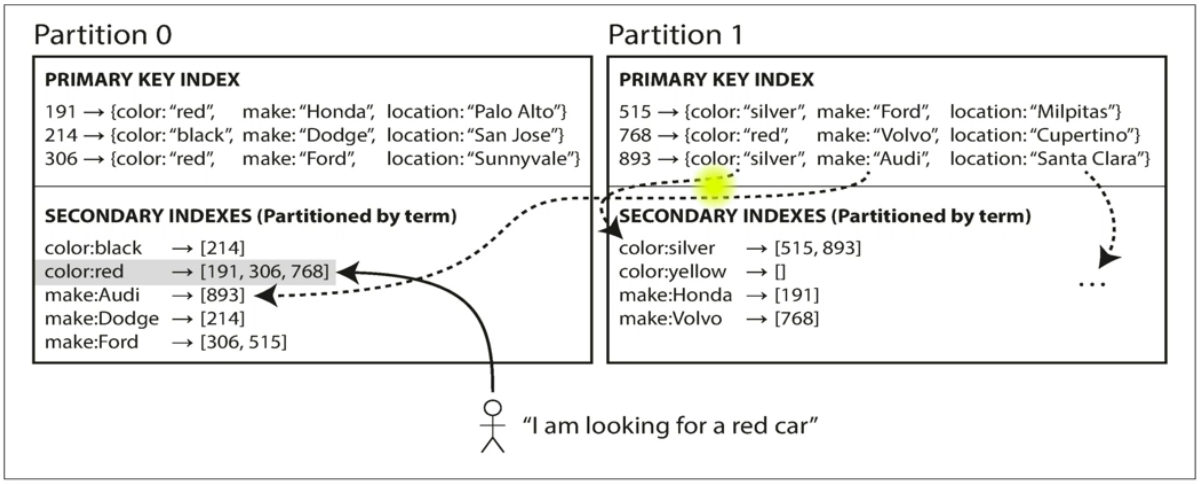


Figure 6-4. Partitioning secondary indexes by document.

Term-based indexing

-Also called global indexing

-You store your index some places in the system, not on every node.



Rebalancing partitions

-Reasons for rebalancing/repartitioning

-Increased load

-Increased data volume

-Failed nodes need to be replaced

-Use a fixed, high number of partitions

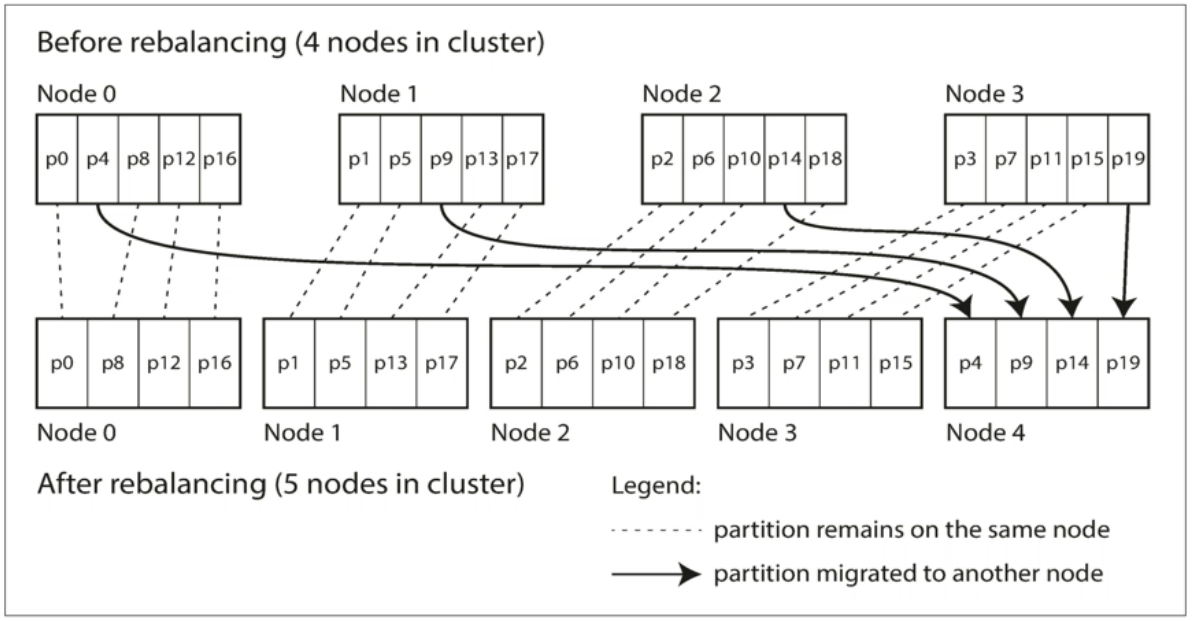


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

-This makes it so that you only need to move the data that you NEED to move.

Dynamic partitioning (range part.)

-When a partition grows to exceed a configured size, it is split into two partitions

-Can be proportional to the size of datavolume

-Or can be proportional to the number of nodes

Request routing

-How do you contact the right partition

1. allow clients to contact any node, then check the distribution dictionary, then connect to the correct node

2. Send all requests from clients to a routing layer first

3. Require that clients be aware of the partitioning, so that they can directly connect to the correct node

Coordination of services

-Every node should know about every other node, and what kind of services they provide

-Gossip protocol: when for example adding a new node, the new node notifies another random node that it is added. That node then notifies another random node, that notifies another random node and so on until all nodes are notified

Chapter 7 – Transactions

Why do we need transactions?

-Crash recovery when something is going down, or being able to abort transaction i fuser feels something is wrong

-Concurrency: several clients may have concurrent access that may destroy each others effects.

-A client may read data that doesnt make sense due to partial updates

-Race conditions between clients may cause suprising bugs

Transactions

-Used to group several reads and writes together into a logical unit. Commits or aborts as a unit.

-If transaction is aborted, you could retry until it succeeds (basic idea behind transactions)

-Used to simplify the programming model for applications accessing a database

-Some applications dont need transactions (NoSQL does not use transactions as it is hard to implement)

-Concepts to learn: read committed, snapshot isolation, and serializability

ACID – properties of transactions

A transaction is a sequence of database-operations which are:

A – Atomic: completely run or not

C – Consistency: (primary key, references, check etc.)

I – Isolation: does not notice or care about other transactions

D – Durability: nothing lost after commit

Atomicity (or abortability)

-Cannot be broken into smaller parts

-Run completely or not run at all

-Atomicity simplifies the problem of half-done operations. If transaction is aborted, the application can be sure that the transaction did not change anything so it can safely be retried

Consistency

-In ACID, consistency refers to an application-specific notion of the database being in «a good state». Also, supported by the database by having some consistency-constraints that can be defined in ex SQL, such as foreign keys.

Isolation

-In ACID, isolation means that concurrently executing transactions are isolated from each other

-The goal is serializability: ecah transaction can pretend that it is the only one running on the entire database

-The database ensures that when the transactions have committed, the result is the same as if they had run serially.

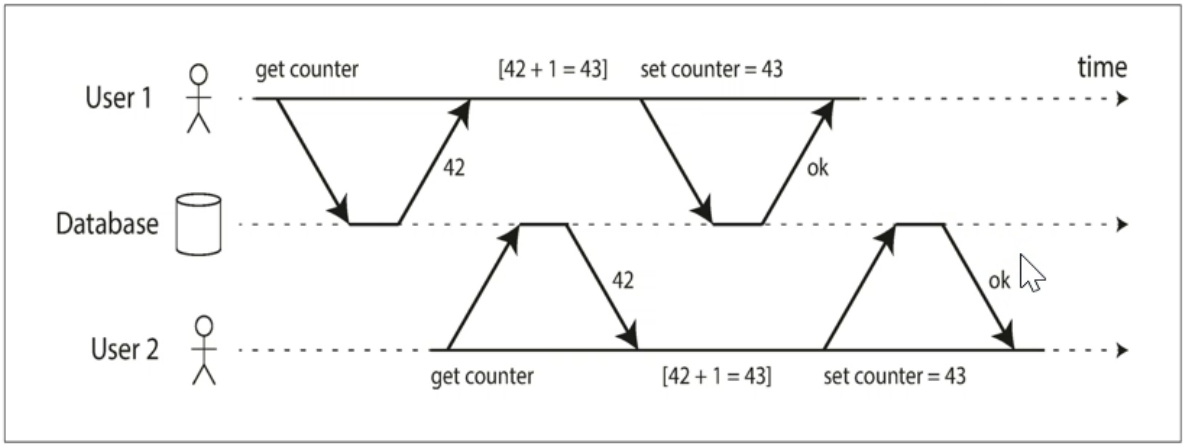


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

The figure above shows a race condition, a problem that can occur if serializability is not implemented. The counter should have been 44 at the end, as two different clients each incremented one from 42.

Durability

-Durability is the guarantee that once a transaction is successfully commited, then any data will not be forgotten

-Usually done by write-ahead-logging (WAL), meaning that as soon as you commit, everything that is intended to be done is written to a log. Then force-log-at-commit

Single-Object and multi-object operations

- Multiple writes could be rolled back (atomicity)
- Concurrent transactions may see all writes or none (isolation)
- SELECT COUNT(*) FROM emails WHERE recipient_id = 2 AND unread_flag = true
- Slow, so use a cached counter: Anomaly may appear

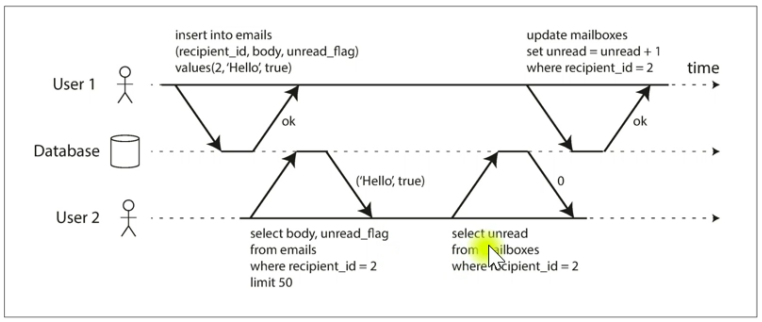


Figure 7-2. Violating isolation: one transaction reads another transaction's uncommit-

Single-object writes

-All databases try to make single-object writes atomic. Ex writing large json strings may fail halfway, so logging and concurrency control is therefore used to ensure atomicity.

-Some databases support increment instead of read-modify-write, because with read-modify-write, there is some time between the read and the write, while increment can be done «atomically»

The need for multi-object transactions

-Rows may contain foreign keys, so several rows in different tables need to be updated

-The same is true for document databases (NoSQL) using normalization, as documents references other documents

-Operations belonging to the same logical task (such as a bank transaction. Two accounts need to be updated at the same time.

Handling errors and aborts

-In leaderless replication, of the database runs into an error, it wont undo something that is already done. It is therefore the applications responsibility to recover from errors. This means that Atomicity/abortability is not supported in leaderless replication.

-Retries are not always good:

-For example if a transaction has succeeded, but then the network fails. The client will not know that the transaction succeeded, and therefore might try the same transaction again, meaning we get a double-transaction

-Overload: If the database is overloaded and that is why the transaction did not complete, then automatic retries will only make the problem worse.

-Transactions could have side effects outside of the database that happens regardless of whether the transaction is aborted or not. This side effect would then happen several times in the case of retries.

-Retries are only good for transient errors (deadlocks, concurrency etc.) meaning that it is the concurrency control that has aborted the transaction because someone else is using the data that you tried to update

Weak isolation levels (1)

-Isolation levels are different levels supporting concurrency. Some levels support more concurrency than others. How are the different levels implemented?

-Isolation levels deal with race conditions (concurrency issue) where one transaction reads data that is concurrently modified by another transaction

-Concurrency bugs are very hard (almost impossible) to detect by testing. Concurrency support is therefore important.

Read committed

-Lowest isolation level

-Ensures no dirty reads and no dirty writes

-You are only allowed to read data that has been fully committed, and you are only allowed to overwrite data that has been fully committed.

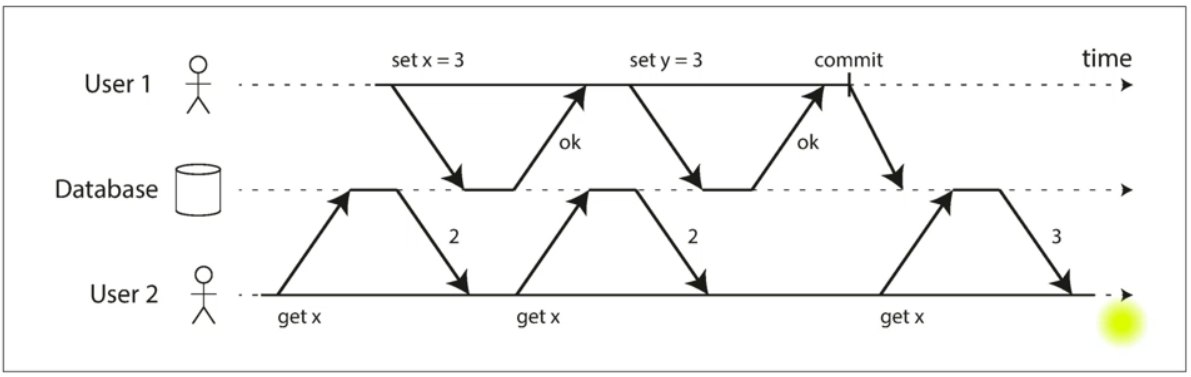


Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

-If a transaction needs to update several objects, a dirty read would mean that another transaction may see some of the updates, but not all. This could lead to the second transaction being able to see and use data that will later be rolled back.

-In the case of a dirty write, the earlier write is part of a transaction that has not yet committed, so that the later write overwrites an uncommitted value.

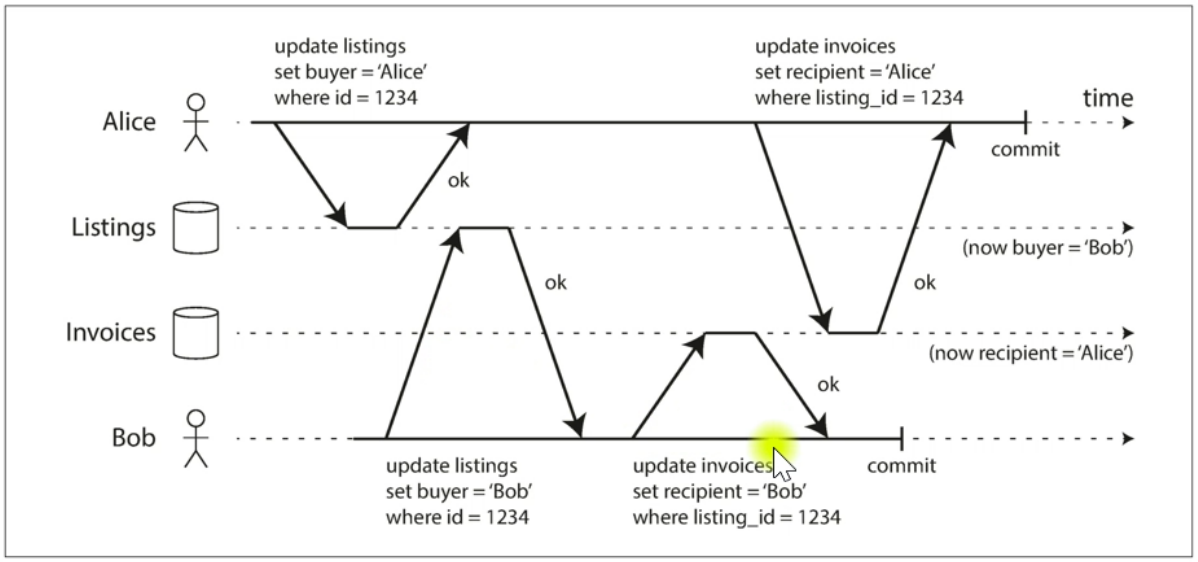


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

The figure above shows a diry write that results in the buyer being set to «bob», while the recipient of the invoice is set to «alice». This happened because the first transaction did not commit before the second transaction started writing. If we did not have a dirty write, the bob-transaction would need to wait until after the alice-transaction was committed before being able to overwrite.

Read committed

1. When reading from the database, you will only see data that has been fully committed (no dirty reads: reading uncommitted data)
2. When writing to the database, you will only overwrite data that has been fully committed (no dirty writes: overwriting uncommitted writes)

-Read committed is a very popular isolation level.

-Most databases now adays prevent dirty reads by keeping multiple copies of the data that is being updated. Read transactions may read the old value until the write transaction fully commits.

-When the new value is fully committed, then other transactions can read the new value

-It is done this way because keeping single record locks would cost too much, as one writer may cause multiple readers to wait. The advantage of preventing dirty reads by keeping multiple copies compared to using locks, is that readers can read old data instead of having to wait for new new data to be committed

Snapshot isolation and Repeatable read

-Snapshot isolation prevents dirty reads and unrepeatable reads

-Snapshot isloation uses exclusive write locks, meaning that when someone is writing to a record, no one else is allowed to touch that record. Snapshot isolation however, does not use read locks. Therefore, readers never lock out writers, and writers never lock out readers, but writers lock out writers.

-Snapshot isolation uses Multiversion Concurrency Control (MVCC). This means that it stores several copies of an updated record, and the newest is only made available to other transactions as soon as the entire update is committed. Until then, other transactions can only see the old copy of the record. Once the update is committed, the old copy is deleted

-Storage engines that support snapshot isolation typically use MVCC for their read committed isolation level

PostgresSQL's MVCC

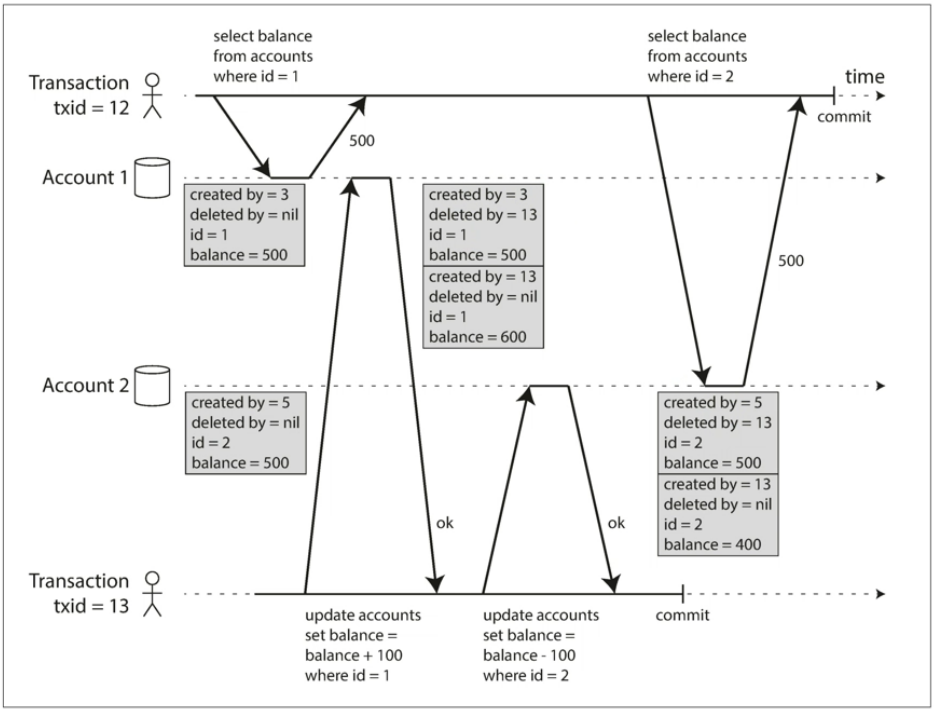


Figure 7-7. Implementing snapshot isolation using multi-version objects.

Visibility rules for observing a consistent snapshot

-TransactionIDs are used to decide visibility of objects. We have four rules:

1. Existing, active write-transactions are not visible to a new transaction

2. Writes by aborted transactions are ignored (because the new copy of the record containing the updated will be deleted as soon as the transaction is aborted)

3. Writes by newer transactions are ignored

4. All other writes (once that have been committed) are visible

MVCC, indexes and COW-B+-trees

-Should the index point to all versions of an object?

-PostgresSQL has optimalizations when the object versions are within the same page (as they typically are)

-Another solution (used by CouchDB) is copy-on-write B+-trees.

-With copy-on-write, all writes make copies of pages which are propagated up in the tree

-When the write commits, a new root tree is visible

Preventing lost updates

-Example of Lost Updates-problem: Two transactions are incrementing the same counter. They would both read the old value, and therefore, one of the increments would be lost

-One solution is having atomic update operations. This means that the reading of the old value. And the writing of the new value happens atomically. Therefore, you would not have a state where the old value has been read, but the new value has not been written.

-This is implemented by using exclusive locks

Explicit locking

-Explicit locking is a feature provided in SQL

-Its a way of preventing lost updates

-SELECT FOR UPDATE provides explicit locking. Example:

Example 7-1. Explicitly locking rows to prevent lost updates

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; (1)

-- Check whether move is valid, then update the position
-- of the piece that was returned by the previous SELECT.
UPDATE figures SET position = 'c4' WHERE id = 1234;

COMMIT;
```

-In the example above, all records in the figures-table that satisfies the WHERE-clause, would be locked for all other transactions until the transaction is committed.

-Some systems, such as ones using snapshot isolation, could solve Lost Update-problems automatically by detecting when a lost update has occured, and aborting the offending transaction.

Compare-and-set

-Used by some databases without transactions, often called lightweight transactions

-A way of atomically updating an area of your memory without having to use explicit lock-concepts

-works by first reading a value, and when when deciding to update it, using a compare-and-set (CAS)-function to check if the value you first read is still the current value of the data. If it is, then the value can be updated. If not, the CAS-function would return an error

Conflict resolution and Replication

-In multi-leader replication, you may have conflicts when parallell updates happen to the same record

-We have to use application code to resolve the conflicts

-Detecting the conflicts has to be provided by the database system in the form of vector clocks or version vectors.

Write scew and Phantoms

-Write scew is a concurrency problem that occurs when multple transactions is reading several records, and basing the decision of a write on those reads at the same time. Example:

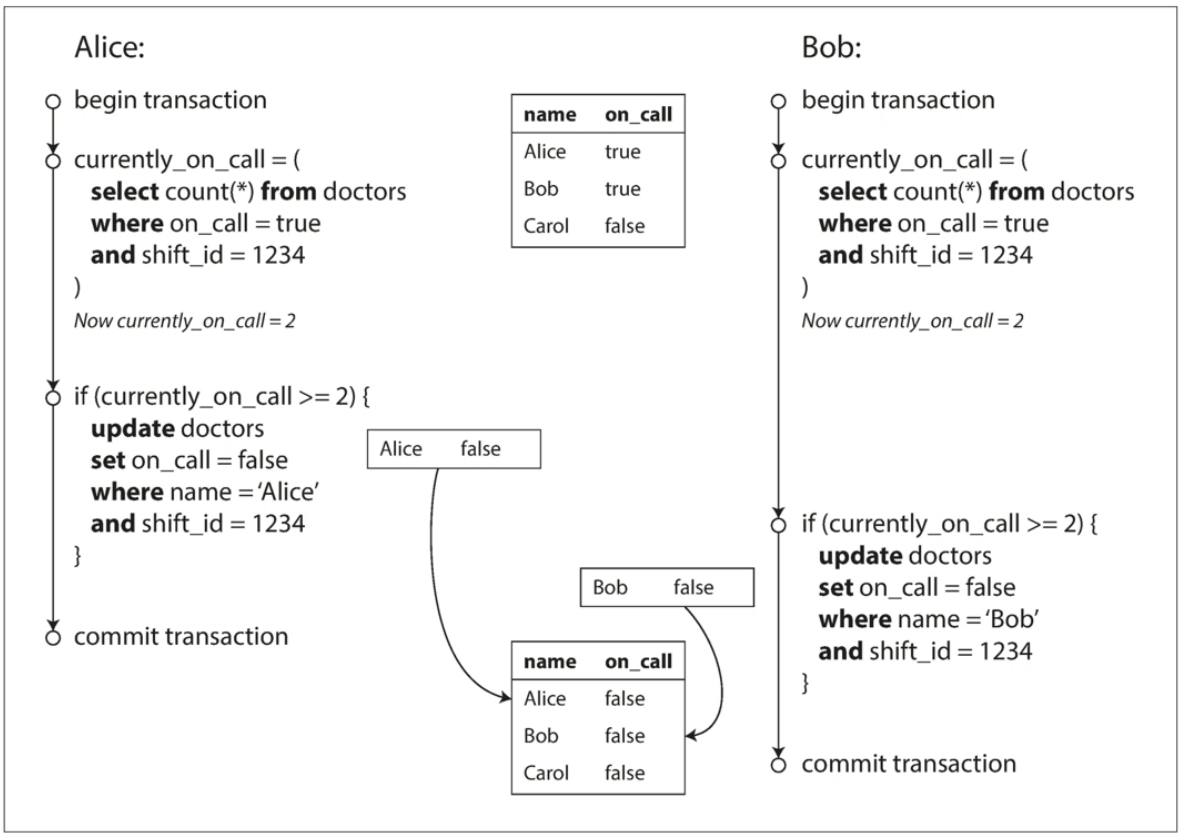


Figure 7-8. Example of write skew causing an application bug.

In the example above, there needs to be at least two doctors on call at all times. Both Alice and Bob sees that more than two doctors are on call, and therefore, they both check out. Now, no doctors are on call.

-This is solved by the serializable isolation level (which is the strongest degree of isolation)

-Could also be solved with explicit locking (with predicate locks) (SELECT FOR UPDATE as described earlier in the document)

Write Skew and Phantoms (3)

- Pattern for problem:

1. A SELECT query checks some conditions
2. Depending on the result of the first query, the application code decides how to continue
3. If the application decides to go ahead, it makes a write to the database and commits the transaction.

- The effect, where a write in one transaction changes the result of a search query in another transaction, is called a phantom
- E.g. insert a new row into a table being counted by another transaction

Serializability

-Strongest level of isolation in SQL databases

-Isolation levels are hard to understand, and inconsistently implemented in different databases

-Serializable isolation is the strongest isolation level, and guarantees that even though transactions may execute in parallell on the same data, the end result is the same as if they had executed one at a time, serially.

-How do databases provide serializablity?

1. literally executing transactions in a serial order
2. Two-phase locking (most common solution)
3. Optimistic concurrency control techniques

-Why run serially?

1. RAM is now cheap enough that in many cases, it is feasable to keep the entire active dataset in memory
2. OLTP transactions are usually short and only make a small number of reads and writes

Serializability (3)

- Each transaction is through one HTTP request.
- Executed by a stored procedure
- No interaction

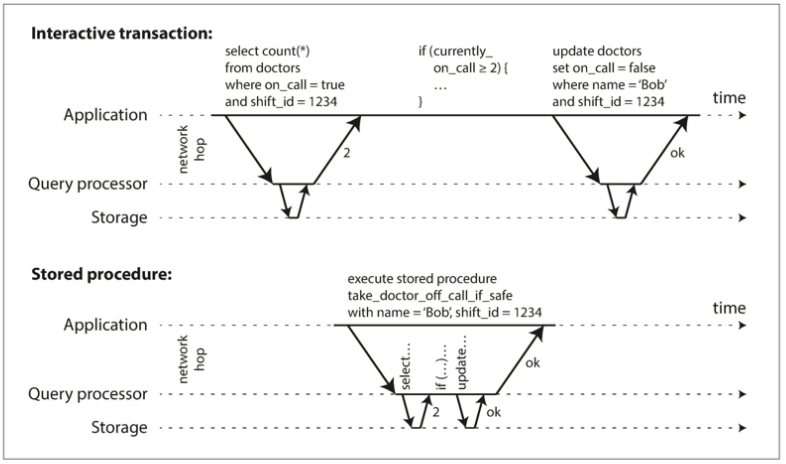


Figure 7-9. The difference between an interactive transaction and a stored procedure (using the example transaction of Figure 7-8).

Pros and cons of stored procedures

-Each database vendor has its own language for stored procedures

-Code running in a database is harder to manage than code running in your application bacause its a separate system

-A database is often much more performace-sensitive than an application server

-Modern implementations of stored procedures have abandoned SQL, and use existing general-purpose programming languages such as Java instead

Summary of serial execution

-Every transaction must be small and fast

-The active dataset must fit in memory

-Write throughput must be low enough to be handled on a single CPU-core

-Cross-partition transactions are possible, but there is a limit to the extent to which they can be used. So if you have a partitioned database, stored procedures is not the way to solve serializability

Two-phase locking

-Several transactions are allowed to concurrently read the same records as long as noone is writing to it

-If a transaction A has read a record, and transaction B wants to write to that record, then B has to wait for A to commit or abort before it can continue.

-Same the other way around: If transaction A has written to a record and transaction B wants to read that record, then B has to wait for A to commit or abort before it can continue

-2PL is used by the serializable isolation level in MySQL and SQL Server, and by the repeatable-read-isolation level in DB2

Two-Phase Locking (2PL) (2)

- T wants to read an object: T must acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously.
- T wants to write to an object: T must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time.
- T first reads and then writes an object, it may upgrade its shared lock to an exclusive lock.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort).

Performance of Two-phase locking

-Acquiring and releasing all the locks causes some overhead

-Reduced concurrency (which means more waiting)

-It may just take one slow transaction, or one transaction that accesses a lot of data and aqcuires a lot of locks to grind the system to a halt

-When using locks, we can get problems with deadlocks, which is when transactions are waiting for each others locks to be released

Predicate locks

-Instead of locking certain tuples or rows, predicate locks lock a predicate. This in practice means to lock all rows that satisfies the WHERE-clause.

-Used to solve write scew

Index-range locks

-Used in database systems where heap-files are used as main storage and b+-trees are used to access the heap-files

-Also called next-key locking

-Simplified approximation of predicate locking

-Instead of locking a predicate, you lock certain parts of your index (certain ranges of your index)

Serializable snapshot isolation

-Fairly new way of providing serializable isolation

-2PL is a pessimistic concurrency control meaning that you solve concurrency problem by using locks, which leads to more waiting

-SSI is an optimistic concurrency control, where instead of preventing problems, the problems are instead detected

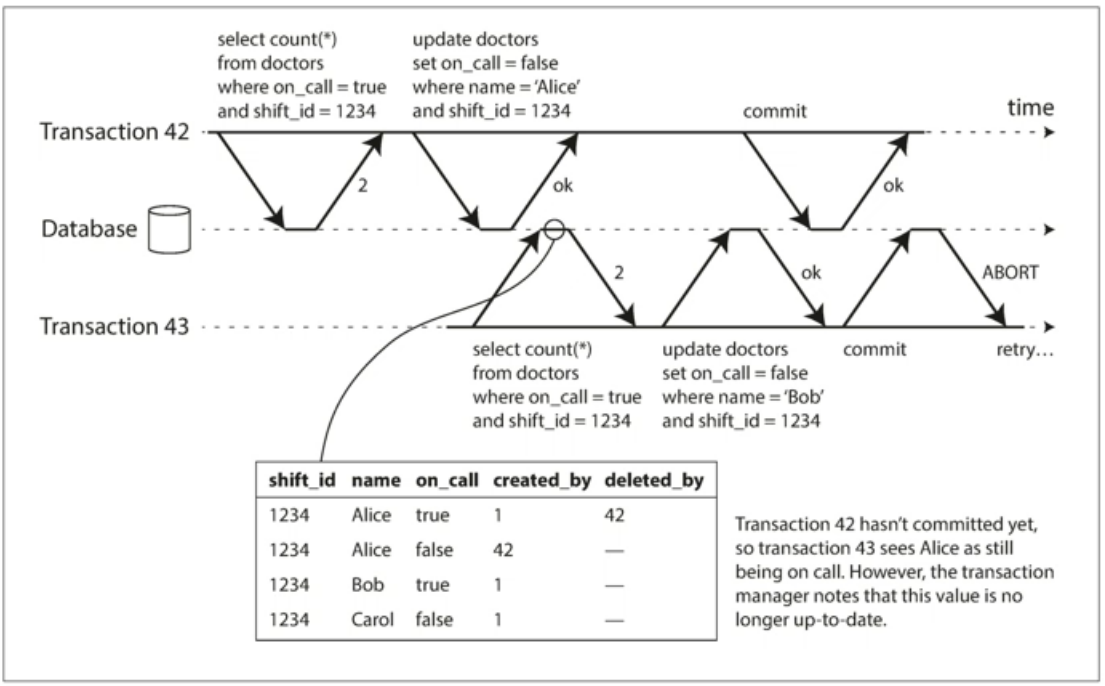


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

In the example above, when bob tries to commit the write-transaction, it is detected that he has not read the newest value that was committed by alice, and therefore, his write-transaction is aborted

-SSI also detects writes that affects prior reads

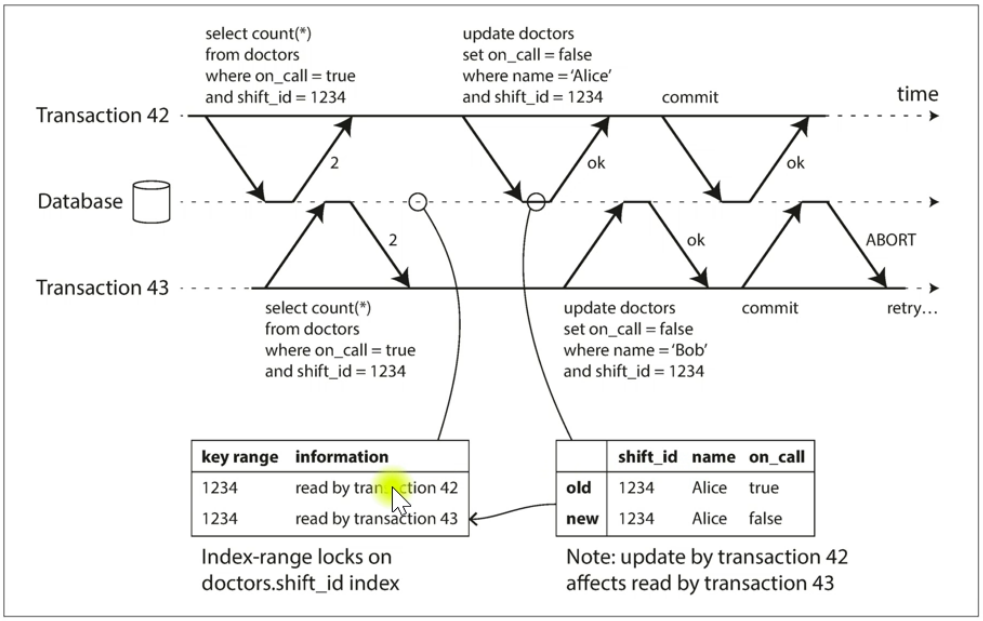


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

Advantages and performance of Serializable Snapshot Isolation (SSI)

-Compared to two-phase locking, the biggest advantage is that one transaction does not need to wait for locks from other transactions to be released

-Compared to serial execution, SSI is not limited to the throughput of a single CPU-core

Summary Chapter 7

-Transactions are abstractions used to simplify error handling

-Isolation levels: Read committed, snapshot isolation (repeatable read), and serializable isolation

-Concurrency problems:

-Dirty reads

-Dirty writes

-Read scew (non-repeatable reads)

-Lost updates

-Write scew

-Phantom reads

Chapter 8 – The trouble with distributed systems

Problems with distributed systems

-Fundementally different from writing software on a single computer

-Software running on a single computer has predictable behaviour unless there are bugs in the software, while the behaviour in distributed systems is more unpredictable because of ordering of events

-Some parts of the distributed system is ofter broken (partial failures)

Cloud computing and supercomputing

-High performance computing (HPC): Supercomputers with thousands of CPUs are typically used for computational-intensive scientific computing tasks

-Cloud computing: Multi-tenant datacenters, commodity computers connected with an IP network. Uses Elastic/on-demand resource allocation and metered billing, which means that you pay for exactly what you use

-In cloud computing, it is reasonable to assume that some parts of the distributed system is always broken

Asynchronous packet networks (the internet), no response?

-Your request may have been lost

-Your request may be waiting in a que and will be delivered later

-The remote node may have failed

-The remote node may have temporarily stopped responding

-The remote node may have processed your request, but the response itself is lost

-The remote node may have processed your request, but the response has been delayed

Network faults in practice

-Human misconfigurations is the biggest cause of faults

-Fault detection may be needed

1. A load balancer has to prevent sending requests to a dead node

2. In a distributed database with single-leader replication, one of the followers has to be promoted to leader if the current leader fails

Timouts and Unbounded delays in distributed systems

-A timeout is really the only way of detecting a fault. You define a timeout, and if you dont get a response before the time is up, then you assume that the node you are trying to communicate with is dead

-If you define a very long timeout, then you may have to wait a long time to find out that a node is dead, but the accuracy is better. Longer timeouts makes it harder to incorrectly declare a node dead, which is a good thing.

-If you define a very short timeout, then it detects faults faster, but you may incorrectly declare a node dead.

-When a node is declared dead, its responsibilities has to be transferred to other nodes, which places additional load on the network and the other nodes

Network congestion and queueing

-When recieving a network package, a CPU core needs to be available to instanciate an interrupt in the OS, and maybe decode the message. If all CPU cores are busy, the incoming network request is queued by the OS until the application is ready to handle it

-TCP/UDP (the two protocols used in the internet) may add delays at the sender node as there may come more data that should be sent in the same packet. This may delay data traffic

-To find a suitable timout, systems can continually measure the average response time of the nodes in the network, and adjust the timout accordingly. The problem with this method is that sometimes, the response time may vary wildly from the average response time. For example when the tickets for octoberfest are released, the server would be very busy, and using the timeout that is adjusted for the average response time would result in incorrect declarations of dead nodes.

Unreliable clocks

-Clocks are needed in distributed systems

-Two types:

1. Clocks used for duration (for measuring time)
2. Clocks used for point in time (for checking time, like a wrist watch)

-We know that the time when a message is recieved is always later than when it was sent, but due to variable delays in the network, we dont know how much later it is going to be

-This makes it hard to determine the order in which things happened when multiple machines are involved

-Network Time Protocol (NTP) is used to synchronize clocks

-Servers may use GPS to synchronize clocks (GPS is more accurate than NTP)

Monotonic vs Time-of-Day Clocks

-Time-of-Day clocks

-Wall-clock time

-System.currentTimeMillis()

-Monotonic clocks

-Suitable for measuring a duration

-System.nanoTime()

-Used for measurements

-Cannot be compared between computers

Clock synchronization and accuracy

-The quartz clock in a computer: can drift up to 17 seconds a day

-Local clock with difference from NTP-clock: causes refusion to synchronize, or applications need to accept clock adjustments

-NTP synchronization can only be as good as the network delay

-Some NTP servers are wrong or misconfigured

-Leap seconds, meaning extra seconds are inserted throughout a day

-In virtual machines, hardware clock is virtualized

-When running software on devices that you dont fully control, the clock cannot be trusted

Relying on synchronized clocks

-Clocks usually work well, but we still need robust software to deal with incorrectness

Timestamps for ordering events

-In multi-leader replication, Last Write Wins (LWW) is often used to determine which write should be replicated. This is a dangerous use of time-of-day clocks, because the clocks across all computers involved need to be perfectly synchronized. Example of using time-of-day clocks for LWW where the clocks are not synchronized:

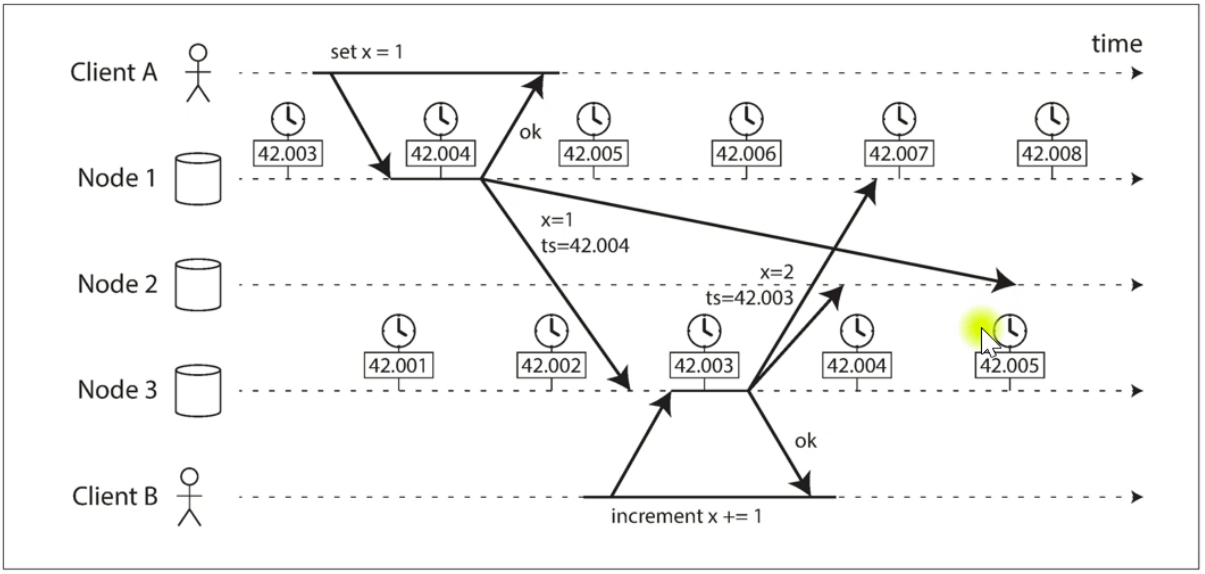


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

Accuracy of Last Write Wins

-NTP synchronization could propably not be made accurate enough so that such incorrect orderings as in the example above could be avoided at all times because the accuracy is limited by the networld round-trip time

-A safer alternative for ordering events is using logical clocks based on incrementing counters

Synchronized clocks for global snapshots in distributed systems and Google Spanner

-Snapshot isolation: allows read-only transactions to see the database in a consistent state at a particular point in time.

-Distribution: A global, monotonically increasing transactionID is difficult to generate

-With a lot of small, rapid transactions, creating transactionIDs in a distributed system becomes an untenable bottleneck

-Google Spanner implements snapshot isolation across datacenters using its TrueTime API

-Using overlapping intervals as undefined order

Another problem of distributed clocks: Process Pauses

-Nodes can obtain a lease with a timout that specifies how long the node can be a leader to execute some task

```
while (true) {
    request = getIncomingRequest();

    // Ensure that the lease always has at least 10 seconds remaining
    if (lease.expiryTimeMillis - System.currentTimeMillis() < 10000) {
        lease = lease.renew();
    }

    if (lease.isValid()) {
        process(request);
    }
}
```

The code above shows the code in a server recieving messages. The problem is that the lease has a time that exists on another computer, and the system.currentTimeMillis() is the clock on the server-computer. So if the clocks are not properly synchronized, we have a problem.

-Another possible problem is that the code assumes that little time passes between the point where it checks the time and the time when the request is processed. Sometimes, however, the process will be suspended for some time (for example for the OS to deal with an interrupt)

Response time guarantees

-Hard real-time systems (typically used in control systems that need hard real-time, such as in NASA)

-These hard real-time systems use special real-time operating systems (RTOS) that allows processes to be scheduled with a guaranteed allocation of CPU time in needed time intervals

-Typically written in C, because there is no automatic garbage allocation, but you can pre-allocate recources manually

Knowledge, truth, and lies

-It is hard to know what is the «truth» in distributed systems, as there is no shared memory, only message passing via unreliable network with variable delays, and also the systems may suffer from partial failures, unreliable clocks, and processing pauses

-What is the «truth» is, in distributed systems, decided by the majority by using concensus (often by quorums)

-A node cannot necessarily trust its own judgement of a situation.

-Quorum: voting among the nodes. Consensus algorithms

The leader and the lock

-In some cases in distributed system, it is required for only one node to be the leader. the leader can obtain a time-limited lease on a lock of a resource

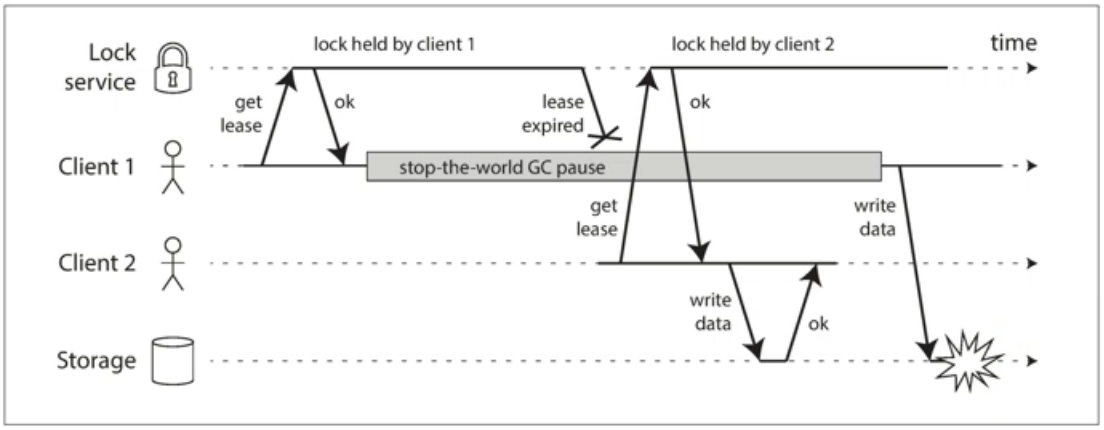


Figure 8-4. Incorrect implementation of a distributed lock: client 1 believes that it still has a valid lease, even though it has expired, and thus corrupts a file in storage.

In the example above, client 1 obtains a lease for a lock, but then his operating system pauses the process to deal with some garbage collection. While the process is still paused, his lease times out, and client 2 obtains the lease for the same resource. When client 1's process is unpaused by the OS, he believes that he still has the lock, and tries to write something to the recource. This leads to failure

-This problem can be solved with fencing tokens:

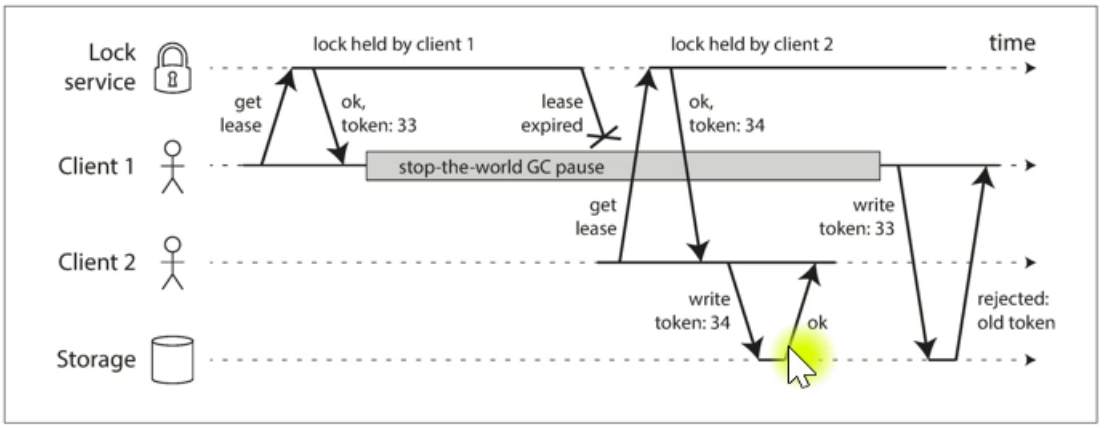


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

In the example above, client 1 gets a fencing token when he recieves confirmation of the obtained lease. When client 2 obtains the lease, he gets an incremented token. When client 1's process unpauses, het ries to write something, and simultaneously sends his fencing token, which the system recognizes is not the newest one, so therefore client 1's write-operation is rejected.

-Fencing token: A number that increases every time a lease is given

Byzantine faults

-We typically assume that nodes are unreliable, but honest, meaning they may be slow or never respond and their state might be outdated, but they never lie

-In distributed systems where you cannot trust all the nodes, everything becomes harder. Then you have to mitigate for byzantine faults

-For example blockchains need to tolerate byzantine faults, as people can try to fool other people for money

-A bug in the software could be regarded as a byzantine fault, but if the same bugged software is deployed to all nodes in the system, then byzantine fault-tolerant algorithms cannot save you. Therefore, you would need to have different versions of software of different nodes to ensure tolerancy against these problems

Handling «weak forms of lying»

-Errors due to software or hardware bugs

-One way of handling is using checksums on messages to see that everything in the message is valid

-Another way is to check input values

-Network Time Protocol (NTP)-clients has to connect to multiple servers to estimate the errors in time

System model and reality – timing

-Three different system models:

1. Synchronous model: Bounded network time, bounded clock error, bounded process pauses. This means that you can KNOW the max-time for message delays, the max time for process pauses and so on. This is a theoretical model and not used in practice, as the assumptions do not hold in real life.

2. Partially synchronous model: This is the most used model. They are usually synchronous, but they may experience and tolerate glitches

3. Asynchronous model: No timing assumptions

System model and node failures

-Crash-stop faults: Node fails by crashing

-Crash-recovery faults: Node fails, but recovers from persistent medium

-Byzantine faults: nodes may do anything

Correctness of an algorithm

- **Example**: Lock using fencing token. Properties:
- *Uniqueness*: No two requests for a fencing token return the same value.
- *Monotonic sequence*: If request **x** returned token **tx** , and request **y** returned token **ty** , and **x** completed before **y** began, then **tx < ty**
- *Availability*: A node that requests a fencing token and does not crash, eventually receives a response.

Chapter 9 – Consistency and Consensus

Consistensy guarantees

-Replicated systems will have inconsistent values

-Eventual consistency: convergence as we expect all replicas to eventually converge to the same value

-The edge cases of eventual consistency only become apparent when there is a fault in the system

-Stronger consistency guarantees may have worse performence, or worse availability than systems with weaker guarantees. There is a tradeoff between consistency guarantee and availibility

-Distributed consistency is mostly about coordinating the state of replicas in the face of faults and delays

Linearizability

-Also called atomic consistency, strong consistency, immediate consistency

-The basic idea of linearizability is to make it appear as if there is only one copy of the data (even with a lot of replicas), and as if all operations on it are atomic.

What Makes a System Linearizable?

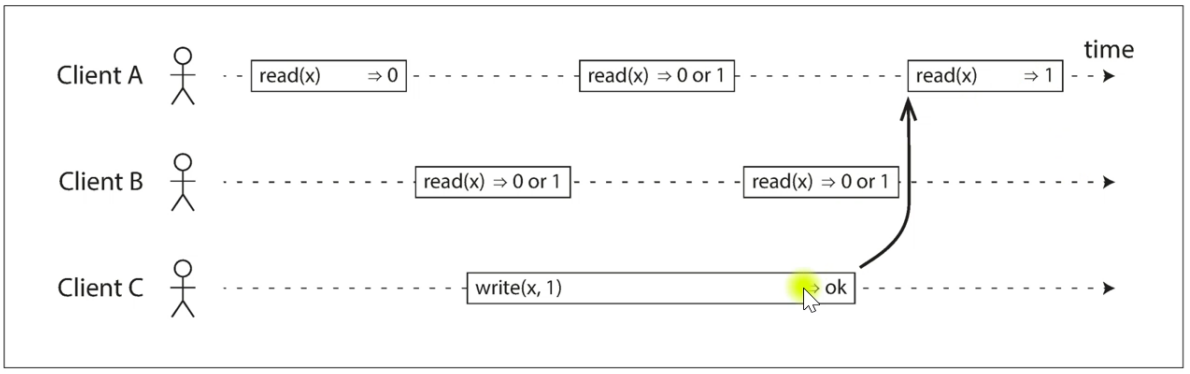


Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

Linearizable, Another Constraint

- An atomic point in time for the flip?

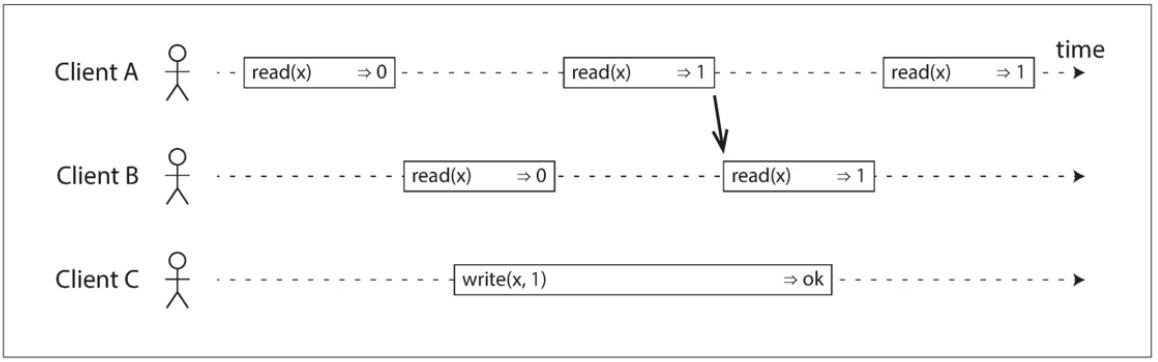


Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

Linearizable, CAS – Compare-and-Set

- cas(x, v_old , v_new) ⇒ r means the client requested an atomic compare-and-set
- An atomic compare-and-set (cas) operation can be used to check the value hasn't been concurrently changed by another client
- The requirement of linearizability is that the lines joining up the operation markers always move forward in time (from left to right), never backward.

Linearizability vs serializability

-Serializability: Guarantees that transactions behave as if they had executed in serial order

-Linearizability: Recency guarantee on reads and writes

-Implementations of serializability based on two-phase locking or actual serial execution, they are also linearizable

-Implementations of serializability based on snapshot isolation is by design not linearizable as it hides concurrent writes by only showing the last fully committed copy of the data to readers. Only when the entire write-transaction has been committed will the new value be visible to other transactions

When is linearizability needed?

-Locking and leader election: all nodes must agree which node owns the lock, and the most recent leader must be shown to all nodes

-Constraints and uniqueness guarantees: such as autoincrementing primary keys in relational databases. All nodes must agree on what should be the next value for such a key.

-Cross-channel timing dependencies:

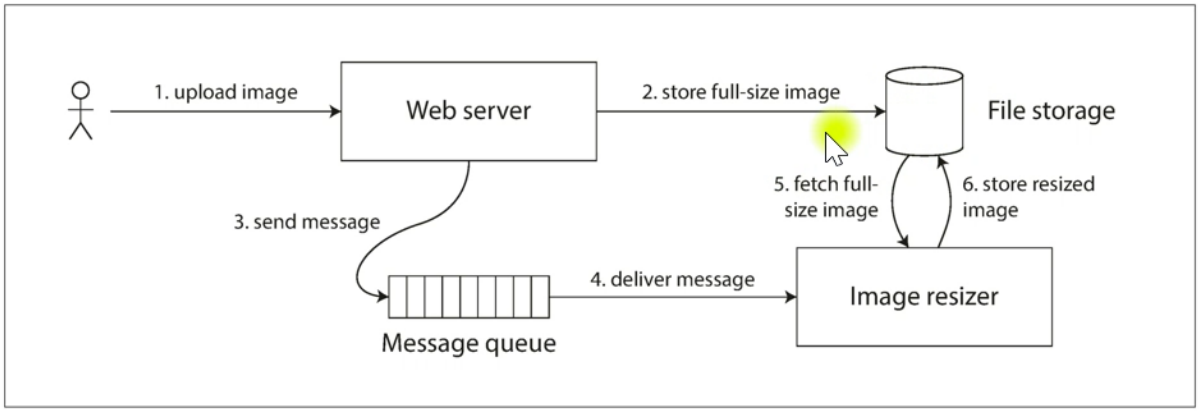


Figure 9-5. The web server and image resizer communicate both through file storage

Implementing linearizable systems

-Replication methods:

1. Single-leader replication (potentially linearizable, for example by using two-phase locking)
2. Consensus algorithms (linearizable)
3. Multi-leader replication (not linearizable, as multiple leaders can update the same value)
4. Leaderless replication (probably not linearizable)

Linearizability and quorums

- It seems as though strict quorum reads and writes should be linearizable, it may not when we have variable network delays:

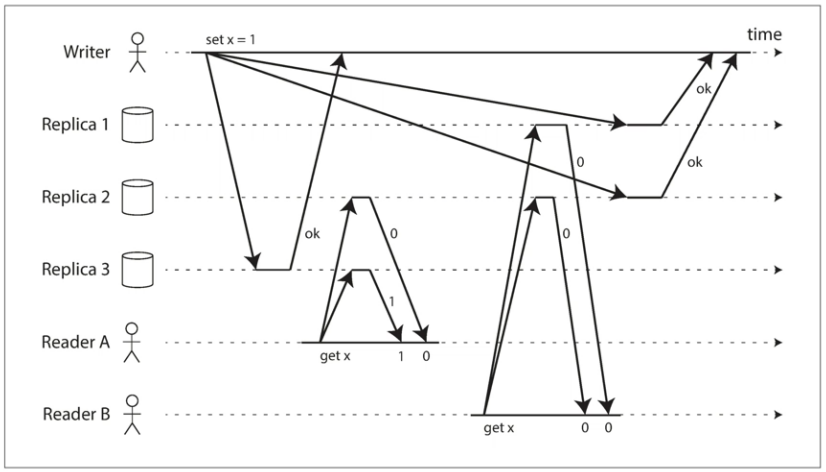


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

-Quorum systems are probably not linearizable

The cost of linearizablilty

The main cost of linearizability is reduced availibility.

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

The CAP theorem

The CAP theorem was proposed by Eric Brewer in 2000. It says that in a distributed system, you would often want Consistency, Availibility, and Partition tolerance, but you can only get two of them with the same system.

-Either consistent or available when partitioned

Ordering guarantees

Typical exam question: What is the connection between ordering, linearizability, and consensus?

-Ordering helps preserve causality, e.g. causal dependency between the question and the answer (A row must be created before it can be updated, meaning that the update is causally dependant on the creation)

-If a system obeys the ordering imposed by causality, it is **causally consistent**

-Linearizability: total order of operations.

-Causality: defines a partial order, but some operations are incomparable on non-dependant on each other

Linearizability is stronger than causal consistency

-Linearizability implies causality

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

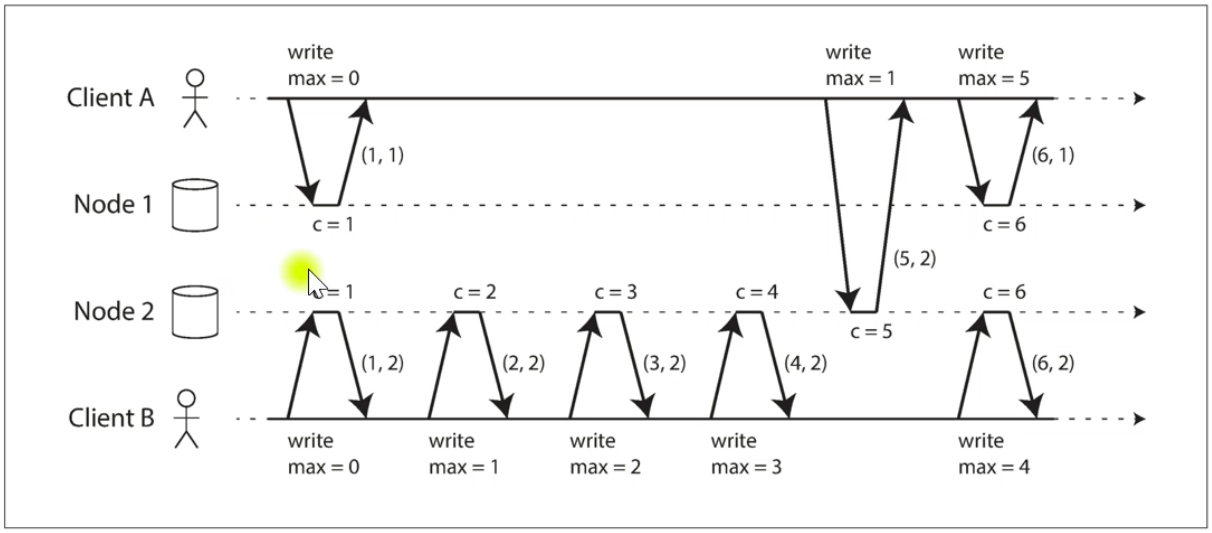
-We can use sequence numbers or timestamps to order the events

-Physical clocks cannot be used

-There are two main timestamp-methods: Lamport timestamps and vector clocks. Distributed systems use vector clocks, not Lamport timestamps

Lamport timestamps

-Every client keeps track of the highest counter value it has seen so far, and includes this value on every request. Every causal dependency results in an increased value



Timestamp ordering is not sufficient

-For instance to implement a uniqueness constraint for user names, it is not sufficient to have total ordering of operations, you also need to know when that order is finalized among all nodes

-A solution to this problem is **total order broadcasting**

-Total order broadcasting is a protocol for exchanging messages between nodes

-Reliable delivery (meaning you have some sort of resending if the message did not go through

-Totally ordered delivery (meaning every node recieves the messages in the same order)

-May be seen as a replication log where all nodes can read the same sequence of messages

Distributed transactions and consensus

-The goal is to get several nodes to agree on something

-Can be used for leader election

-Can be used for atomic commitment, meaning all participants in a transaction commits together

Two-phase commit (2PC)

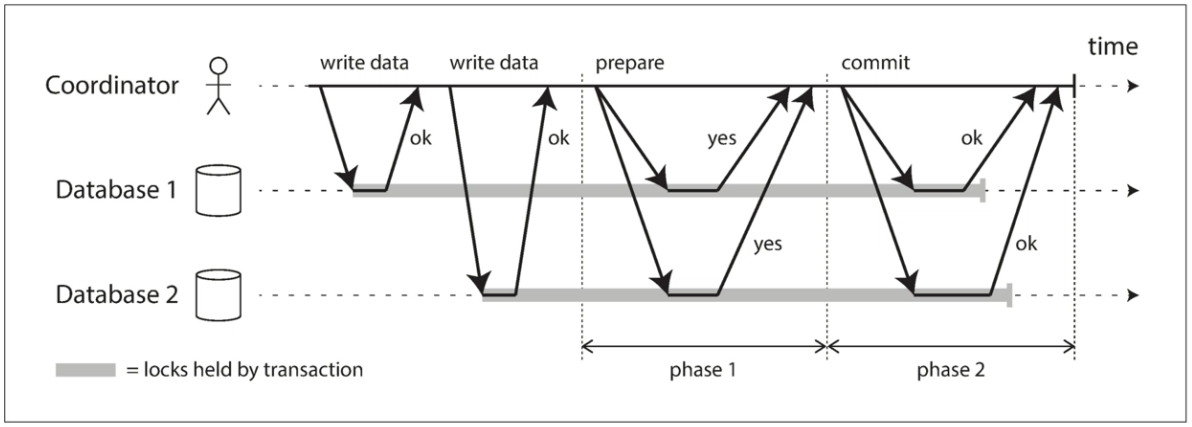


Figure 9-9. A successful execution of two-phase commit (2PC).

The example above shows a two-phase commit. The coordinator first writes some data to database 1, then to database 2, and then it asks both databases if they are ready to commit the write, and if both respond with yes, then it can commit.

Two-phase commit – how it works step-by-step

-There is one coordinator, and several participants

-The coordinator updates some data, and then asks all participants if they are ready to commit

-If all participants answer that they are ready to commit, the coordinator sends out a commit request to all participants in phase 2

-If any of the participants reply with «no», meaning they are not ready to commit, the coordinator sends out an abort request to all participants (nodes) in phase 2

More detailed step-by-step for two-phase commit

-First, the coordinator provides a transactionID for the transaction

-The transaction then starts executing on all nodes (participants). Any node may becide to abort in this step.

-Once the transaction has been executed on all nodes, the coordinator sends a prepare-to-commit message to all participants

-The participants then recieve the PTC, and replies with yes or no

-The coordinator recieves the replies, and decides whether to abort or commit, and logs the decision

-The commit or abort message is sent to all participants

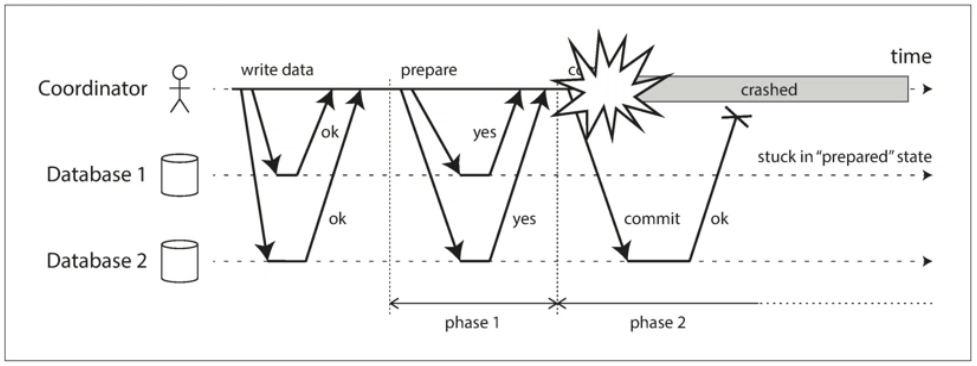
-When all participants have either aborted or committed, they send a message to the coordinator indicating that they are done

Coordinator failure

-if any of the prepare-to-commit-requests sent by the coordinator either fails or times out, the coordinator aborts the transaction

-If any of the commit or abort requests fail, the coordinator retries them indefinitely

-If the coordinator crashes after having sent the prepare-to-commit request, then all participants have to wait for the coordinator to recover. This can be costly, as we get transactions living on the participant nodes that may hold locks etc. but cannot be executed because the coordinator can not send the abort or commit request. One way of solving this is having standby coordinators that can take over if the main one crashes, but this is not standard 2PC. Another way is to have some garbage collection that removes the orphanaged transactions after some time



Performance of two-phase commit

-2PC can be costly, because if there are many participants, one of them will often be very slow, and not answer as quickly as the others. All other participants must then wait for that one slow participant

Fault-tolerant consensus

-One or more nodes may propose values, and the consensus algorithm decides on one of those values

-Uniform agreement: no two nodes decide differently once a consensus has been reached

-Integrity: no node decides twice

-Validity: If a node decides value v, then v must have been proposed by some node (could be the same node)

-Termination: every node that does not crash eventually decides some value

Epoch numbering and quorums

-When the consensus algorithms elect a new leader, they use an epoch number to guarantee that within each epoch, the leader is unique. So when a new leader is elected, the epoch number is incremented

-Anytime the current leader is thought to be dead, a vote is started among all nodes to elect a new leader

-The new leader collects votes from a quorum of nodes to see that it has the highest epoch number

-If a vote on a proposal succeeds, at least one of the nodes that voted for it must have also participated in the latest leader election.

Limitations of consensus

-Consensus algorithms are quite costly

-Consensus algorithms typically rely on timeouts to detect failed nodes. Variable network time may be a problem

Membership and coordination services

In distributed systems, we very often need some kind of coordination of services and membership. Which nodes are present, and what services do they provide?

- ZooKeeper and etcd are designed to hold small amounts of data that can fit entirely in memory
- ZooKeeper is modeled after Google's Chubby lock service
  - Linearizable atomic operations
  - Total ordering of operations
  - Failure detection
  - Change notification
- ZooKeeper runs on a fixed number of nodes (usually three or five) and performs its majority votes among those nodes while supporting a potentially large number of clients.
- Service discovery does not require consensus, leader election does.
- A membership service determines which nodes are currently active and live members of a cluster.

Summary chapter 9

- Consistency and consensus.
- Similar solutions / problems
  - Linearizable compare-and-set registers
  - Atomic transaction commit
  - Total order broadcast
  - Locks and leases
  - Membership / coordination service
  - Uniqueness constraint

Chapter 14 – Time and global states

Physical and logical time

-Physical time

-Timestamp of events

-Can be used to derive order of events, but this requires synchronized clocks

-Logical time

-Better for algorithms

-Ordering of events

-Focused on cause and effect

-typically a counter incremented for each event

-No need to synchronize

Physical time

-Can we trust physical clocks?

-"scew" – difference between clocks at a point in time

-"drift" – how the scew changes over time

UTC – Coordinated Universal Time

-A highly accurate international time standard

-Based on atomic clocks

-Todays computers usually get UTC-time by using GPS

-Leap seconds – because of the rotation of the earth, an extra second sometimes need to be inserted, however this is very rare. Last time was in 2016. Causes a lot of problems for software

-Time zones are relative to UTC – we are at UTC+1 in the winter and UTC+2 in the winter

Physical clock synchronization

1 – External synchronization (most used)

-Clocks synchronized against an external time source

-Christians algorithm

2 – Internal synchronization

-Used when we do not need to be synchronized with the rest of the world, only internally in a distributed system

-Not necessarily the correct time

-The Berkley algorithm

-Basic problem with physical clock synchronization: communication takes time

Christians algorithm

-Used to synchronize physical clocks against and external time server, S(UTC)

1 - p (a client) sends a message m_r to the server, S

2 – S replies with its time in a message m_t

3 – When p receives m_t, it sets its clock to t + half of the time passed since m_r, the first message, was sent

The Berkley algorithm

-More of a voting-algorithm

-Typically used in internal, high-speed networks, and internally in a distributed system

-One node is selected as master

1 – The master polls all other nodes (slaves)

2 – Slaves reply with their local time

3 – Master calculates average time (message latency is considered, and outliers are ignored)

4 – Master sends individual time differences to each slave, so that the slaves can use the difference to adjust their time

NTP: Network Time Protocol

-Protocol for synchronizing clocks that are connected to the internet

-Uses UTC

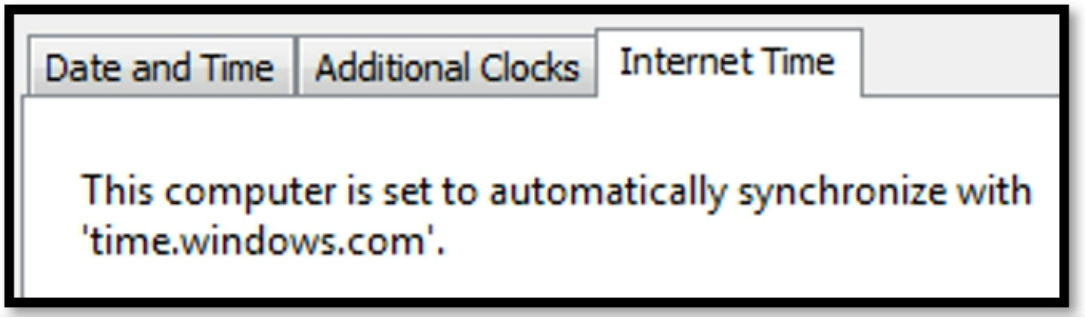
-Focus on:

-Scalability – hierarchy of servers

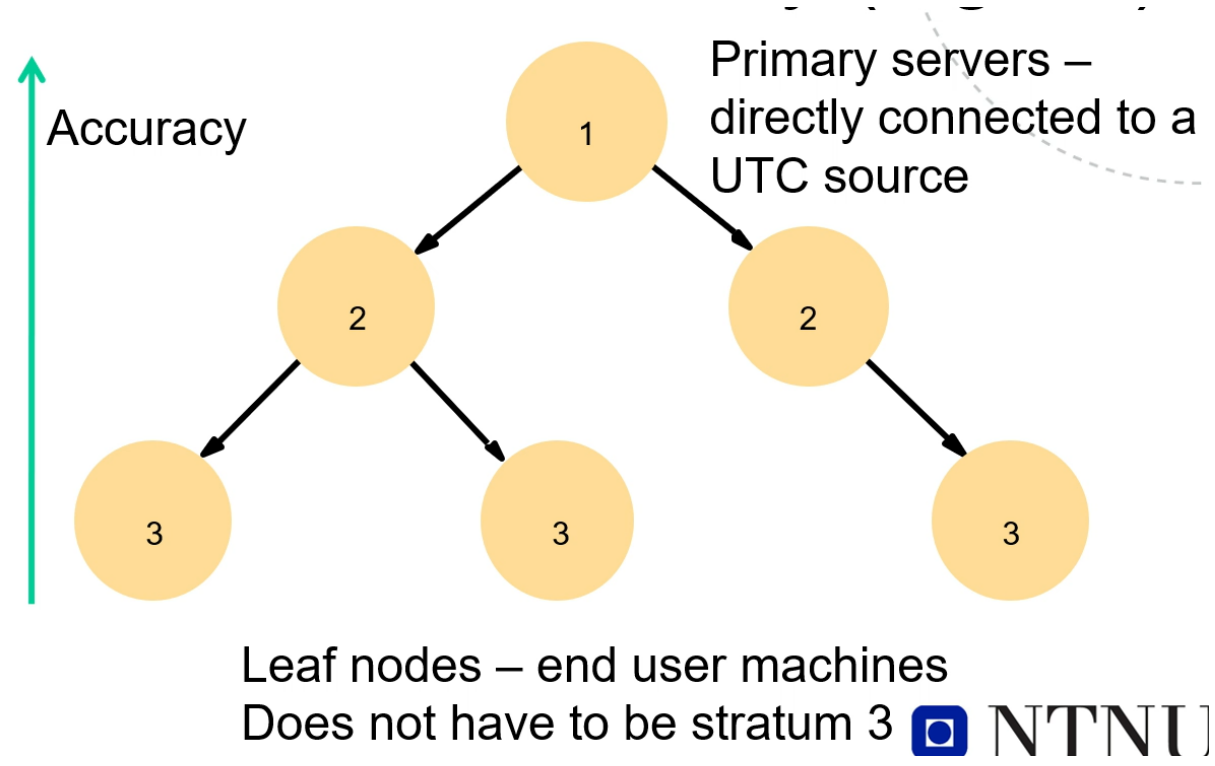
-Correctness – handling clock drift

-Reliability – dynamic reconfiguration

-Security – authentication etc.



NTP -Server hierarchy (logical)



NTP – Synchronization

-Three synchronization modes:

1 – Multicast (LAN) (meaning that a computer can send one message and distribute it to several other nodes defined by a list.

-Assumes fixed message latency (a good network)

-Periodic multicast (on demand)

2 – Procedure call (e.g. Christians algorithm) (most used)

3 – Symmetric mode (high accuracy)

-Servers communicate in pairs

-One server can be in multiple pairs

-Estimates offset and delay in time

Logical time

-If we only need to order events, physical time is not needed (overkill), and also might not be very accurate

-Perfect synchronization of physical clocks is impossible

-Local time – focus on event order

-Two logical events happens in the order observed by the process executing them

-A message must be sent before it can be received

-I.e. cause and effect.

-Increasing the eventnumber when you have some kind of event, and increasing the eventnumber when the receiver gets the message from the event. That way, it is visible that the event happened before the receiver got the message from the event.

Definitions (logical time)

-Collection of processes $p_i = 1, 2, \ldots N$

-Every process p_i has a state s_i (the state could be for example the value of all variables in the process)

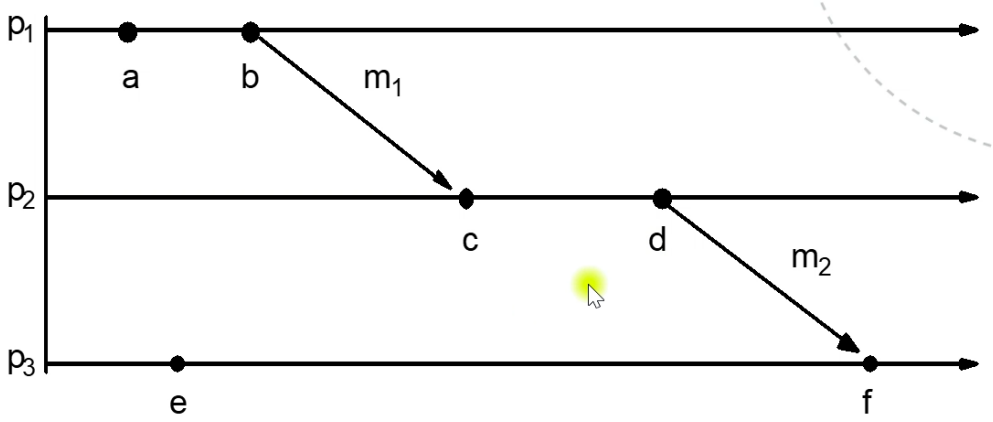
-Processes communicate using messages

-Events

-Local state changes

-Sending or receiving messages

Happened-before (→)



- Local events: a→b; c→d; e→f
- Messages: b→c; d→f
- Derived: a→c; a→f; b→d
- Concurrent: a || e; b || e; c || e; d || e

NTNU

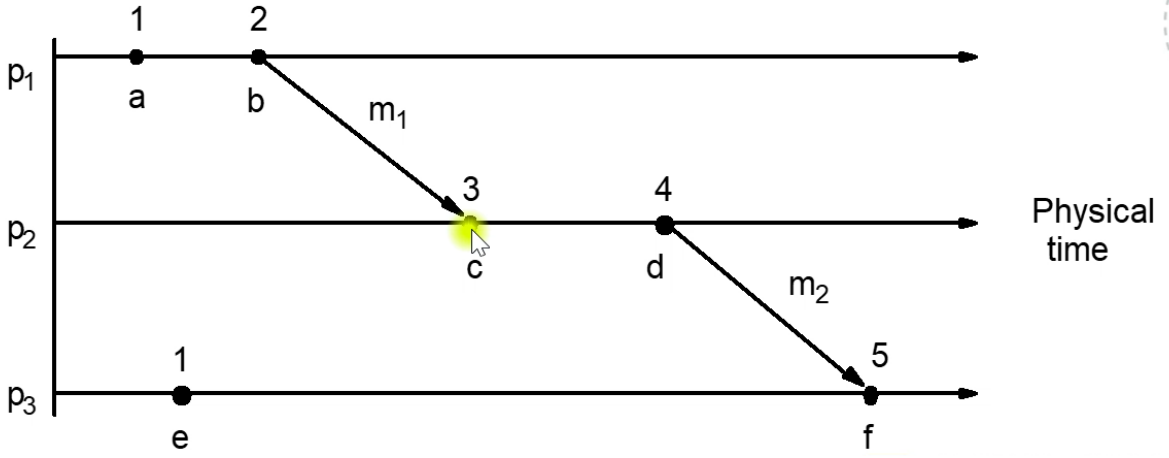
Logical clocks (Lamport)

-Every process has a logical clock (a counter) L_i

-Before every event: $L_i = L_i + 1$

-Clock value is attached to every message , $t = L_i$

-When receiving, $L_j = \max(L_j, t) + 1$



-if e happened before a, then L(e) < L(a)

-But the reverse is not true because L(e) can be less than L(a) without e having to have happened before a

-But if L(a) > L(e), then we can derive that a did not happen before e

Vector clocks

-Logical clocks only get you so far

-To figure out more about event order, we need to store/transfer more info

-Vector clock, given N processes

-Every process has a vector on N elements

-The vector contains the number of events form each process that the given process can have been affected by

Definitions (vector clocks)

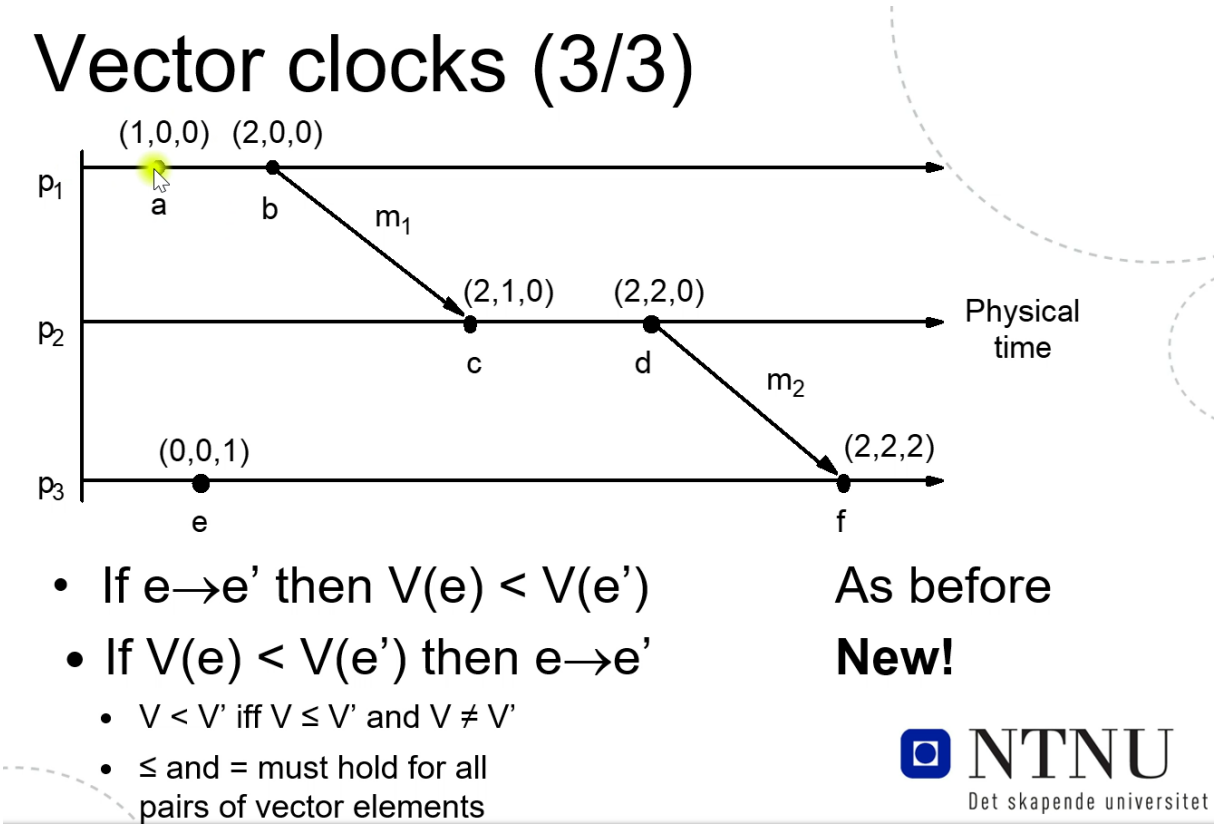
-V_i – vector at process i

-Initially, all vector elements are set to 0

-Before each event at p_i: $V_i[i] = V_i[i] + 1$

-p_i attaches $t = V_i$ to all messages

-when p_i receives a message from p_j: $V_i[j] = \max(V_i[j], t[j])$, for $j = 1, 2 \ldots N$.
(Also $V_i[i] = V_i[i] + 1$ because receiving a message is also an event)



In the example above, a is concurrent with e, because they both have one element of value 1, and the rest of value 0. But all the other events are happened-before-events, and we can tell from the vectors which events happened before which

Global states

-What and why?

-Distributed garbage colletion

-Distributed deadlock detection

-Distributed debugging

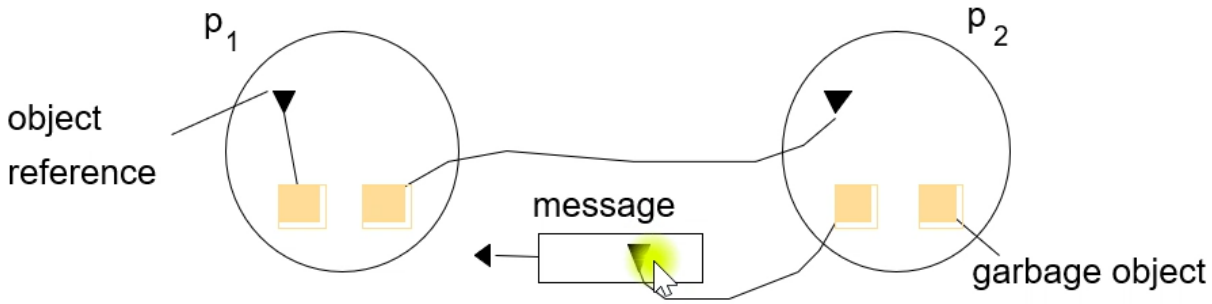
-How?

-Cuts and globally consistent states

Distributed garbage collection

-Garbage is defined as Objects without any active references

-in distributed systems, references can be both local, at other processes/nodes, and in messages



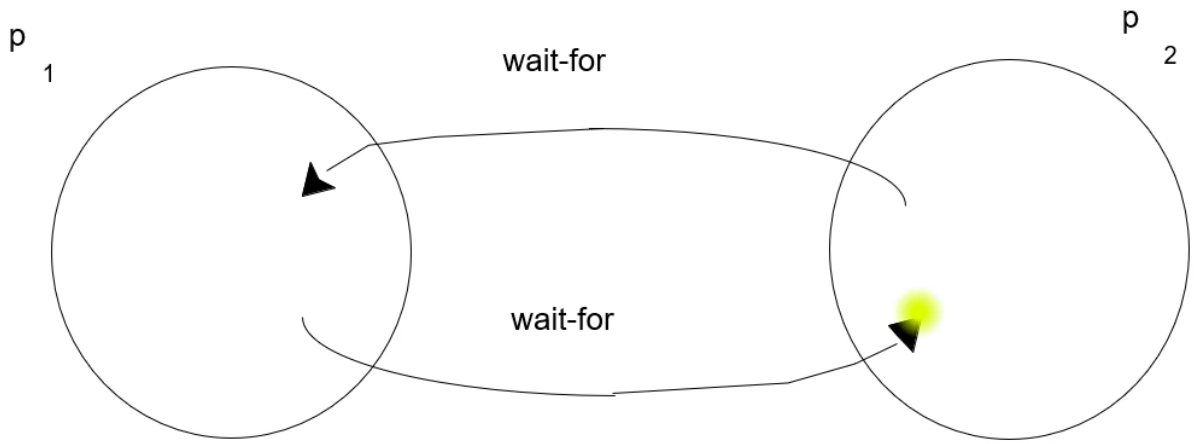
In the example above, we can see that all objects (the squares) have references, except for one which is labelled “garbage object”

-To do distributed garbage collection, we need a global state

Distributed deadlock detection

-Distributed wait-for cycle

-We need global state to detect this



Distributed debugging

-Comparing variables on different computers (nodes)

-How did variables change during runtime? (example: has $| x_1 - x_2 |$ ever been > 50?)

-Variables can be located at different processes/nodes

-Problem: Global consistent view of variable values

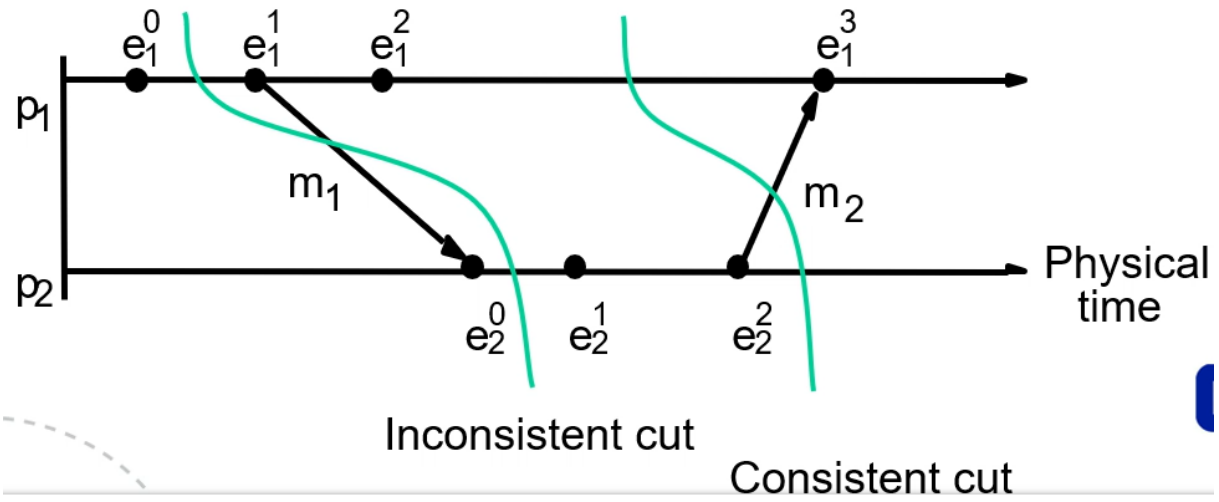
Cuts

-Local history: Events at one process

-Global history: Union of all local histories

-Cut: Subset of global history

-The problem is finding consistent cuts without global time



In the example above, the cut labelled "inconsistent cut" is inconsistent because it does not include e_1^1, which happened before e_2^0. So to be consistent, the cut should include all events that happened before the last event included in the cut

-A cut C is consistent if:

-For all events $e \in C$, if event f happened before e, then $f \in C$.

-The problem with an inconsistent cut is that the system never could have been in that state

-Consistent cut is a global consistent state (a state that might have occurred in the system)

-Run

-A global history where the order satisfies all local histories

-Consistent run

-All global states passed through are consistent

-Reachable

-B is **reachable** from A if there is a consistent run between them

Distributed debugging

-After the fact: Was a condition true during execution?

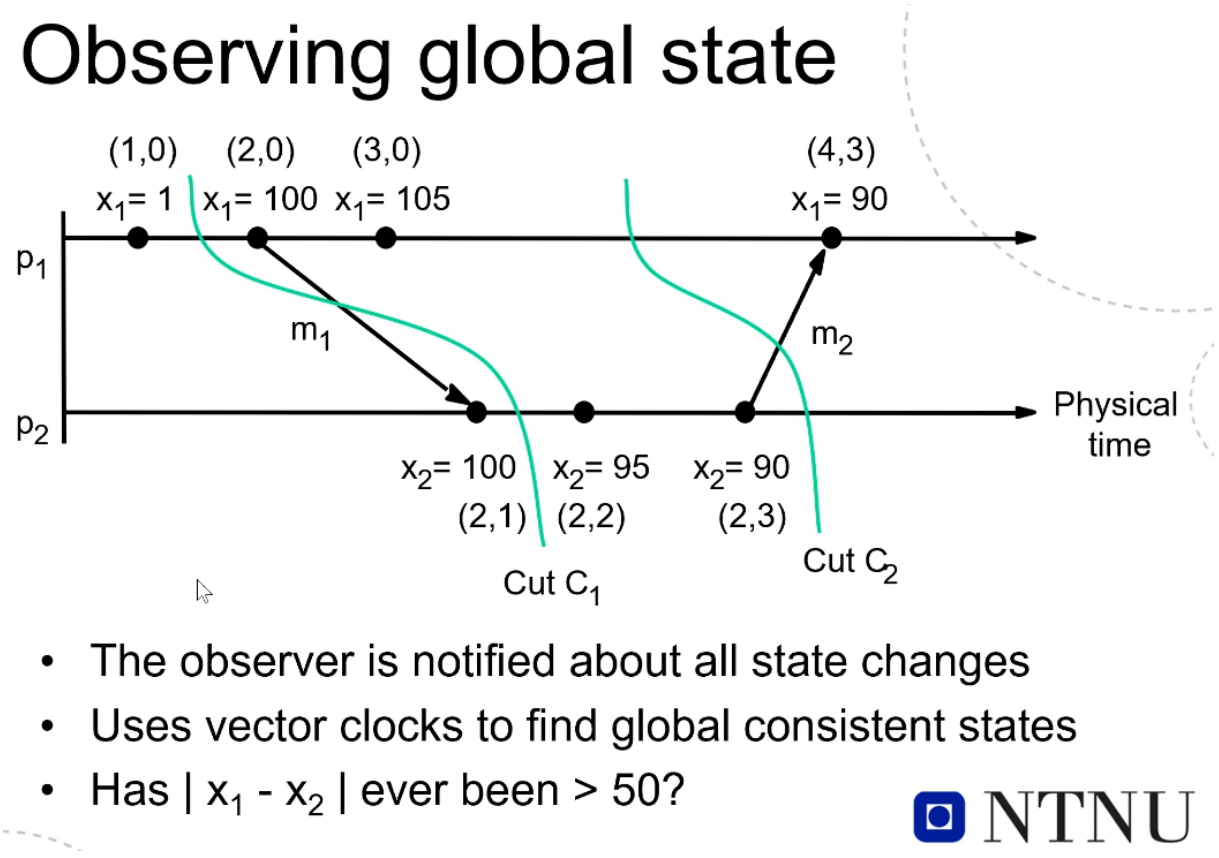
-All participants send information about state changes (vector clocks) to an external observer

-Global state predicate Φ (example: has $| x_1 - x_2 |$ ever been > 50?)

-Possibly Φ: at least one consistent run passes through a global consistent state where Φ = true

-Definitely Φ: All consistent runs pass through a global state where Φ = true

TYPICAL EXAM TASK: YOU GET SOME MESSAGES BETWEEN NODES, AND THE TASK IS TO SHOW THE LATTICE OF CONSISTENT STATES/RUNS (Må øve på dette. Ligger I andre forelesning fra 12.10.2021, slide "Observing global state"



-To find out if $| x_1 - x_2 |$ has ever been > 50, we need to check all possible states that the system could have been in

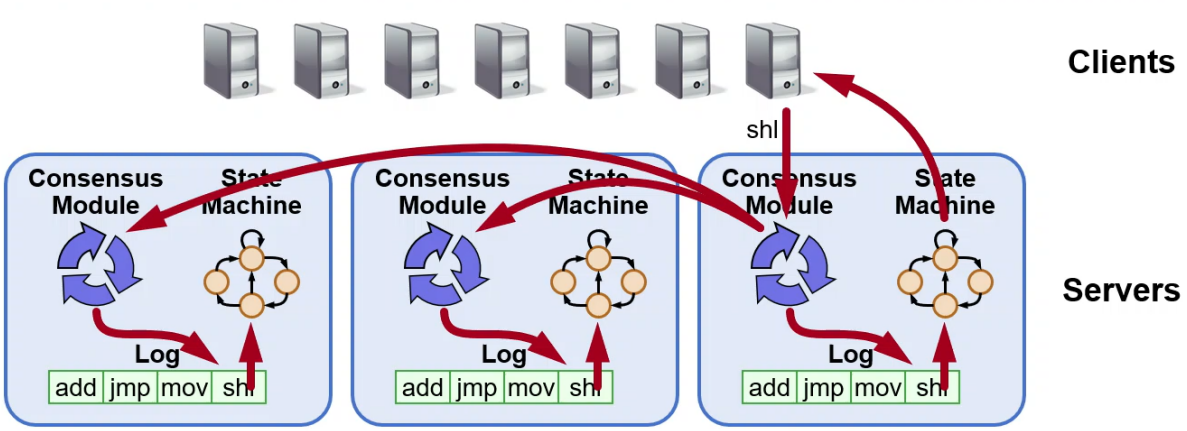
Raft: A consensus algorithm for replicated logs

Raft is regarded as the newest, simplest way of having multiple computers agree about something.

Goal: replicated log

-Raft is a state machine replication protocol, typically consisting of 5, 7, or 10 servers. It is used typically for directories in distributed systems containing important information about the system, such as which servers are up and down etc.

-The Raft system is a replicated log, where its trying to make state machines equal in a distributed setting



The figure above shows the high-level workings of Raft. First, a client contacts the leader-server (there is always just one leader). The leader then writes the operation to its own log, before replicating the operation to the other servers in the system. The other servers then writes the operation to their own respective logs, and when the majority of the servers respond to leader that they have put it in their logs, then all servers can update the state machine to include the operation.

-The consensus module ensures proper log replication

-The system makes progress as long as the majority of servers in the system (more than 50%) is up and running. If less than 50% is up and running, the servers will not be able to update the state machine and thereby commit the operation.

-The failure model in this system is fail-stop (not Byzantine), meaning that when something crashes, it goes down

Approaches to consensus

-Two general approaches:

1. Symmetric, leaderless (the traditional approach):

    -All serves have equal roles

    -Clients can contact any server, and the server will replicate the operation to the other servers

2. Asymmetric, leader-based:

    -At any time, one server is in charge as the leader, and the other servers accept the leaders decisions

    -Clients communicate with the leader only

-Raft uses the second approach. The main reasons is that having a leader decomposes the consensus problem into smaller subproblems (normal operations and leader changes). Since all updates go through one server (the leader), it simplifies the normal operations with regards to consistency etc (no conflicts).

Raft overview

-Leader election:

    -Selecting one server to act as the leader

    -Detecting crashes and selecting a new leader

-Normal operations (basic log replication)

-Safety and consistency after leader changes (the hardest aspect of Raft), how to synchronize the log when you have several crashes happening after each other

-Neutralizing old leaders, so that they do not believe that they are still the leader when they recover from the crash

-Client interactions

-Configuration changes: adding and removing servers

Server states

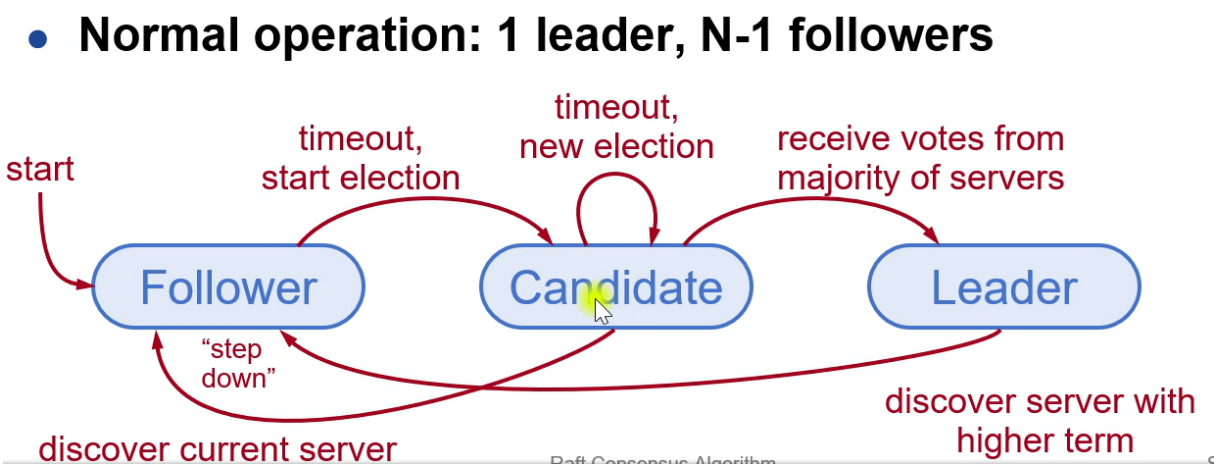
-At any given time, each server is either the leader, which handles all client interactions and does the log replication to the other servers, which are called the followers. The followers are completely passive and they do not issue any remote procedural calls, they only respond to

incoming ones coming from the leader. The last state a server can have is being a candidate. A server becomes a candidate when it has not received any messages from the leader, so it therefore assumes that the leader is down. It then becomes a candidate that can be elected as the new leader. So a server in Raft can be in three different states:

1. Leader

2. Follower

3. Candidate

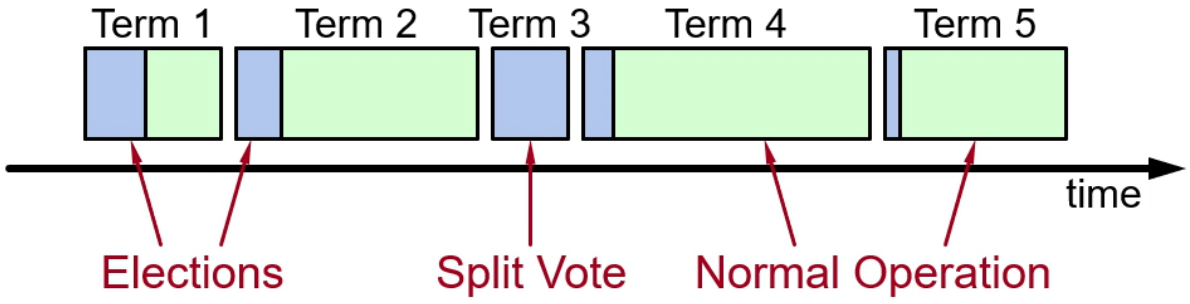


The figure above shows a state-diagram showing how a server can move through the different states

Terms

-Terms is a way of putting version-numbers on different periods in time

-So time is divided into terms. A term consists of an election, and normal operations. Any time there is a new election, there is also a new term



-There can be at most one leader per term

-Some terms, such as failed-election-terms like term 3 in the figure above, have no leader

-Each servers has a variable indicating what the current term is.

-The key role of terms is identifying servers that are using outdated information. When receiving a message with an older term-value than the current one, the message can be discarded as it is outdated

Raft summary (figure from lecture)

Followers

- Respond to RPCs from candidates and leaders.
- Convert to candidate if election timeout elapses without either:
  - Receiving valid AppendEntries RPC, or
  - Granting vote to candidate

Candidates

- Increment currentTerm, vote for self
- Reset election timeout
- Send RequestVote RPCs to all other servers, wait for either:
  - Votes received from majority of servers: become leader
  - AppendEntries RPC received from new leader: step down
  - Election timeout elapses without election resolution: increment term, start new election
  - Discover higher term: step down

Leaders

- Initialize nextIndex for each to last log index + 1
- Send initial empty AppendEntries RPCs (heartbeat) to each follower; repeat during idle periods to prevent election timeouts
- Accept commands from clients, append new entries to local log
- Whenever last log index ≥ nextIndex for a follower, send AppendEntries RPC with log entries starting at nextIndex, update nextIndex if successful
- If AppendEntries fails because of log inconsistency, decrement nextIndex and retry
- Mark log entries committed if stored on a majority of servers and at least one entry from current term is stored on a majority of servers
- Step down if currentTerm changes

Persistent State

Each server persists the following to stable storage synchronously before responding to RPCs:

currentTerm	latest term server has seen (initialized to 0 on first boot)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries

Log Entry

term	term when entry was received by leader
index	position of entry in the log
command	command for state machine

RequestVote RPC

Invoked by candidates to gather votes.

Arguments:

candidateId	candidate requesting vote
term	candidate's term
lastLogIndex	index of candidate's last log entry
lastLogTerm	term of candidate's last log entry

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Implementation:

1. If term > currentTerm, currentTerm ← term (step down if leader or candidate)
2. If term == currentTerm, votedFor is null or candidateId, and candidate's log is at least as complete as local log, grant vote and reset election timeout

AppendEntries RPC

Invoked by leader to replicate log entries and discover inconsistencies; also used as heartbeat .

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat)
commitIndex	last entry known to be committed

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Implementation:

1. Return if term < currentTerm
2. If term > currentTerm, currentTerm ← term
3. If candidate or leader, step down
4. Reset election timeout
5. Return failure if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm
6. If existing entries conflict with new entries, delete all existing entries starting with first conflicting entry
7. Append any new entries not already in the log
8. Advance state machine with newly committed entries

Heartbeats and timeouts

-The leader in a Raft system must send out heartbeats (which are empty AppendEntries remote procedural calls) to the followers to maintain authority.

-If the electionTimeout elapses and a follower has not received any heartbeats from the leader, the follower assumes that the leader has crashed.

-The electionTimeout is typically 100-500ms

Election basis

-When a follower starts a leader-election, the first thing it does is to increment the current term

-It then changes state from follower to candidate

-It votes for itself

-It then sends RequestVote RPCs (Remote Procedural Calls) to all other servers, and retries this request until one of three things happen:

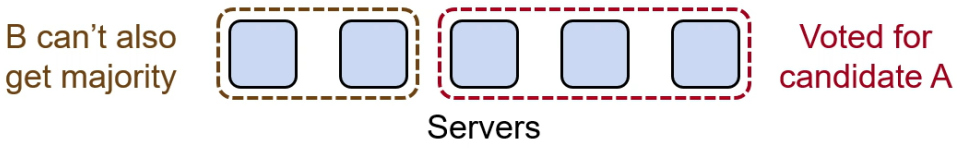
1. It receives votes from the majority of servers, in which case it becomes the leader, and sends AppendEntries-heartbeats to all other servers.

2. It receives AppendEntries-heartbeat from another server, meaning that the server that sent the RPC has gotten the majority of the votes and is therefore the new leader, in which case it returns to the follower-state

3. The election timeout elapses, and we get a no-one wins election. Then the server increments the term, and starts a new election again

Elections, cont'd

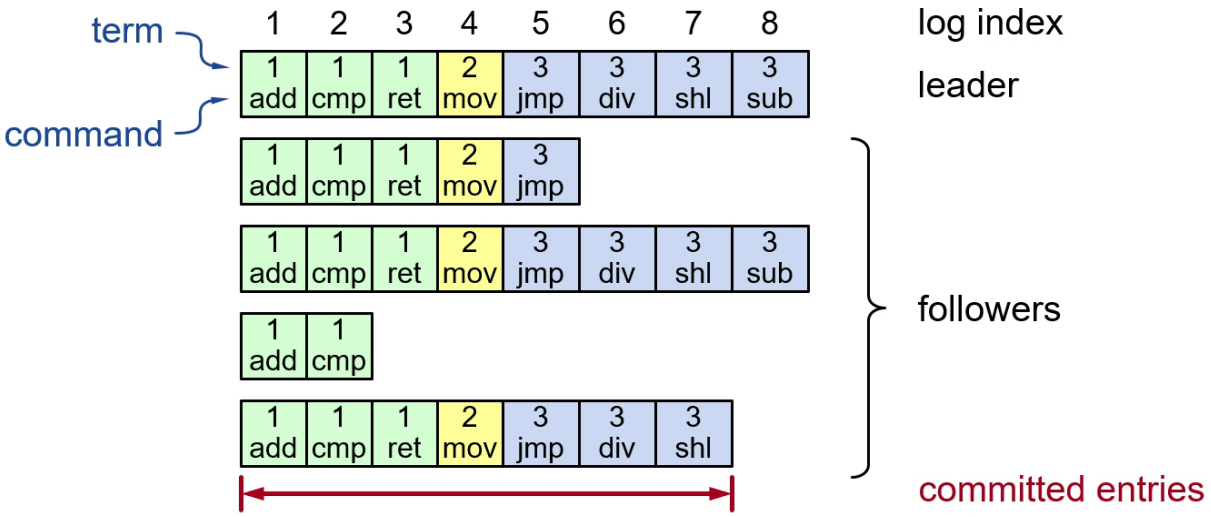
- **Safety**: allow at most one winner per term
  - Each server gives out only one vote per term (persist on disk)
  - Two different candidates can't accumulate majorities in same term



- **Liveness**: some candidate must eventually win
  - Choose election timeouts randomly in [T, 2T]
  - One server usually times out and wins election before others wake up
  - Works well if T >> broadcast time

Log structure in Raft

Below is the log structure for the log that is being replicated to the followers by the leader



- **Log entry = index, term, command**
- **Log stored on stable storage (disk); survives crashes**
- **Entry committed if known to be stored on majority of servers**
  - Durable, will eventually be executed by state machines

Normal Operation

- **Client sends command to leader**
- **Leader appends command to its log**
- **Leader sends AppendEntries RPCs to followers**
- **Once new entry committed:**
  - Leader passes command to its state machine, returns result to client
  - Leader notifies followers of committed entries in subsequent AppendEntries RPCs
  - Followers pass committed commands to their state machines
- **Crashed/slow followers?**
  - Leader retries RPCs until they succeed
- **Performance is optimal in common case:**
  - One successful RPC to any majority of servers

Log consistency (most complicated aspect of Raft)

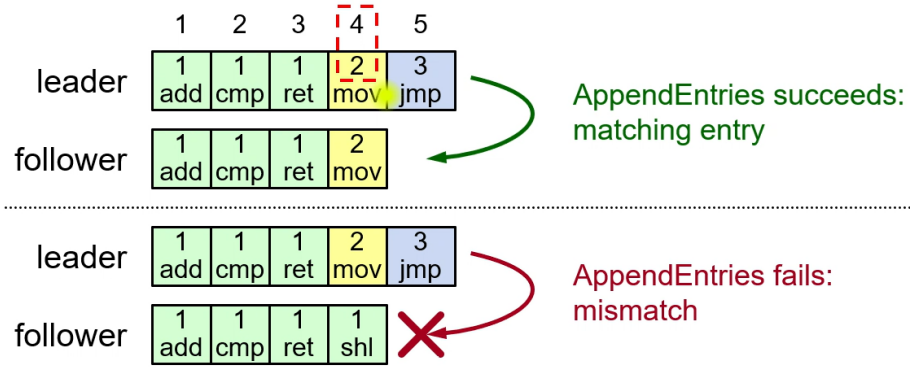
-In Raft, we know that if log entries on different servers have both the same index and term, then they store the same command, and we know that their logs are identical in all preceding entries

-If a given entry is committed, we know that all preceding entries are also committed.

AppendEntries consistency check

-Each AppendEntries-RPC contains indexnumber, term of entry for both the latest entry and all preceding entries

-The follower must contain the latest entry before the newest one sent in the RPC by the leader. If it does not contain that entry, then it rejects the request:



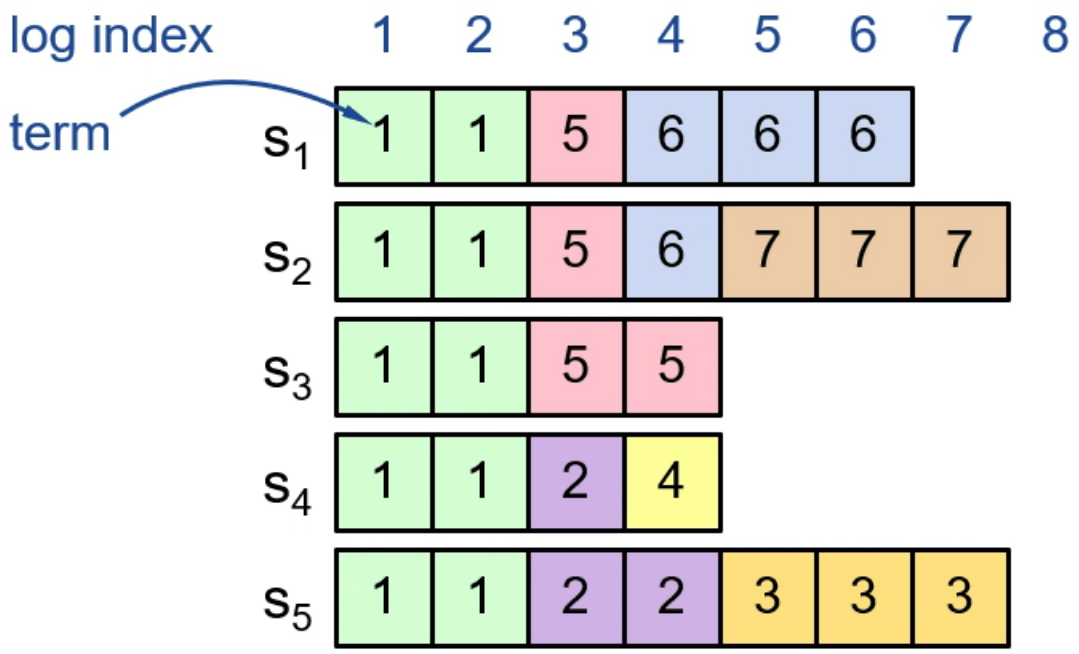
Leader changes

-At the beginning of new leaders term:

-the old leader may have left entries partially replicated because it didn't manage to replicate everything before it crashed.

-In such cases, there is no need for the new leader to take any special steps to mitigate the problem. It just starts a normal operation

-The new leaders log is the "truth", and as it replicates it to all of the followers, all followers will eventually be identical to the leaders.



From the example above, we can derive why there is no need for the new leader to take any special steps in the case of the old leader having left entries partially replicated. S_1 becomes the new leader, and we can see that operation 2 on index 3 for s4 and s5 is not replicated by the majority (only 2 out of 5), and has therefore not been committed, while operation 5 on index 3 has been replicated by the majority, and has therefore been committed.

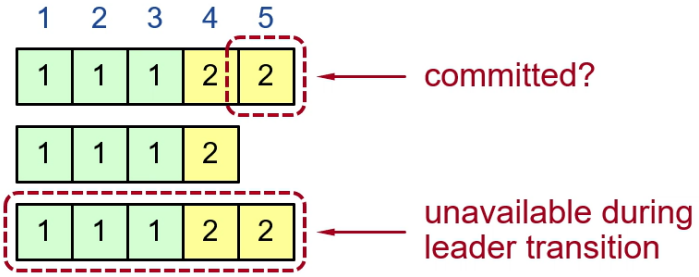
Safety requirement

-The main requirement is that once a log entry has been applied to a state machine, then no other state machines can apply a different value for that log entry

-Raft guarantees that this safety requirement is met because if a leader has decided that a log entry is committed, then that entry will be present in the logs of all future leaders

Picking the Best Leader

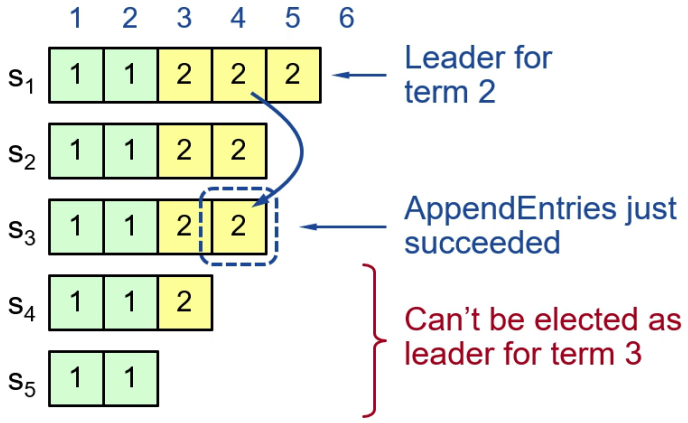
- **Can't tell which entries are committed!**



- **During elections, choose candidate with log most likely to contain all committed entries**
  - Candidates include log info in RequestVote RPCs (index & term of last log entry)
  - Voting server V denies vote if its log is "more complete":
    $(lastTerm_V > lastTerm_C)$ ||
    $(lastTerm_V == lastTerm_C)$ && $(lastIndex_V > lastIndex_C)$
  - Leader will have "most complete" log among electing majority

Committing Entry from Current Term

- **Case #1/2: Leader decides entry in current term is committed**



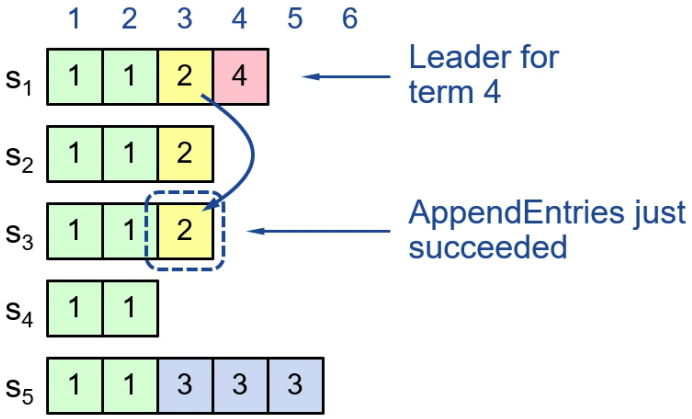
- **Safe: leader for term 3 must contain entry 4**

-The new leader would have had to have been either s2 or s3 because they have more log entries than the other ones.

-Since both index and log is included in the RequestVote RPCs, s2 and s3 would see that s4 and s5 is behind, and therefore reject their RequestVote.

Committing Entry from Earlier Term

- **Case #2/2: Leader is trying to finish committing entry from an earlier term**

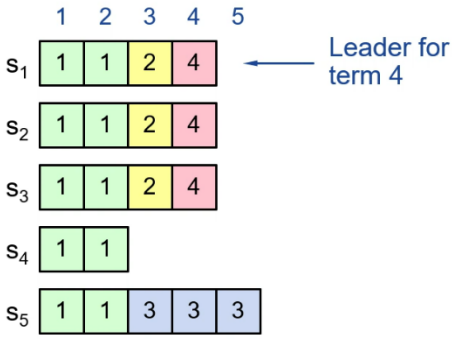


- **Entry 3 not safely committed:**
  - s_5 can be elected as leader for term 5
  - If elected, it will overwrite entry 3 on s_1, s_2, and s_3!

-Which is a problem. How to solve?:

New Commitment Rules

- **For a leader to decide an entry is committed:**
  - Must be stored on a majority of servers
  - At least one new entry from leader's term must also be stored on majority of servers
- **Once entry 4 committed:**
  - s_5 cannot be elected leader for term 5
  - Entries 3 and 4 both safe

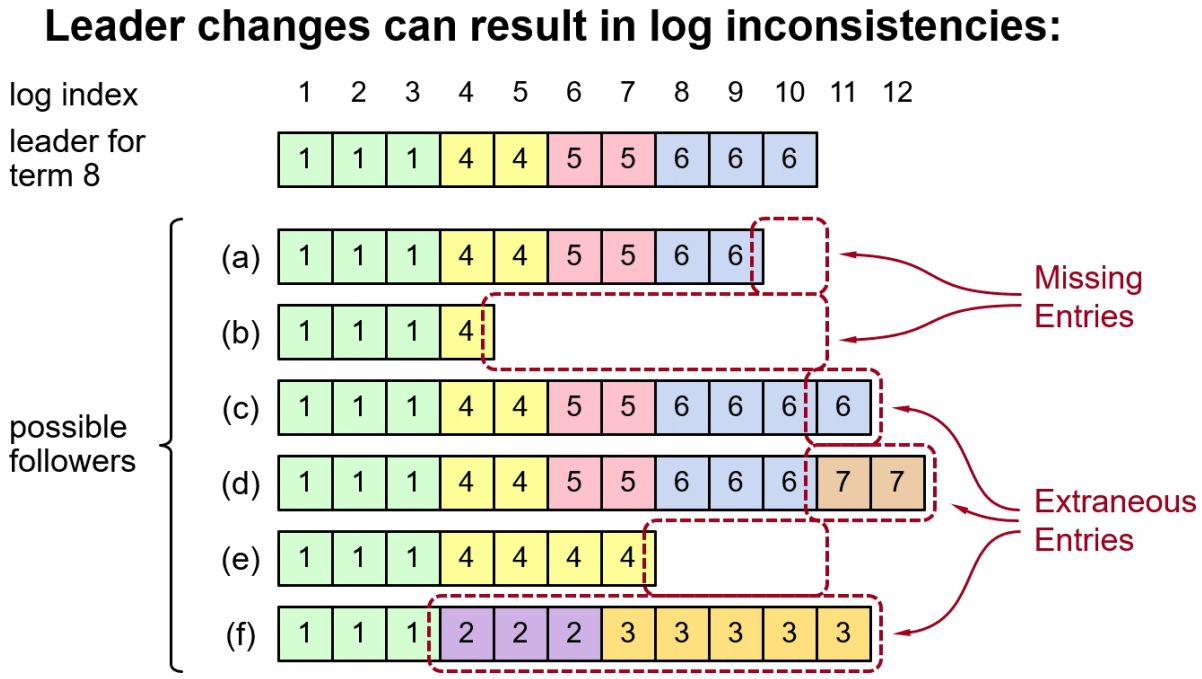


Combination of election rules and commitment rules makes Raft safe

-this is because you can only tell that an entry has been committed once you receive a new AppendEntries RPC for the next entry.

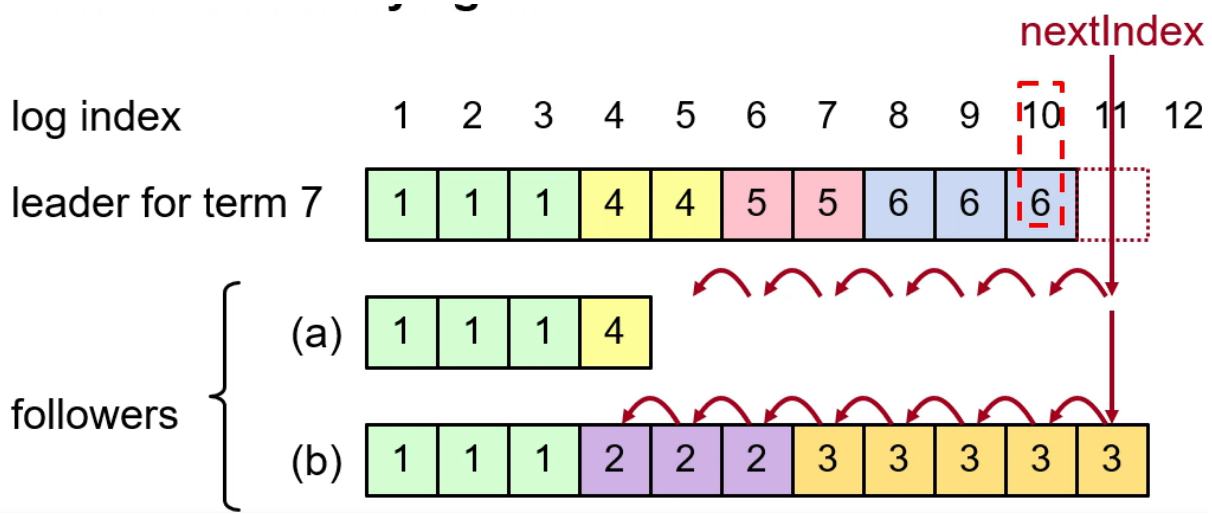
Repairing followers logs

In come cases, followers will have logs that deviate from the leaders log, either by having too many entries, or too few entries:

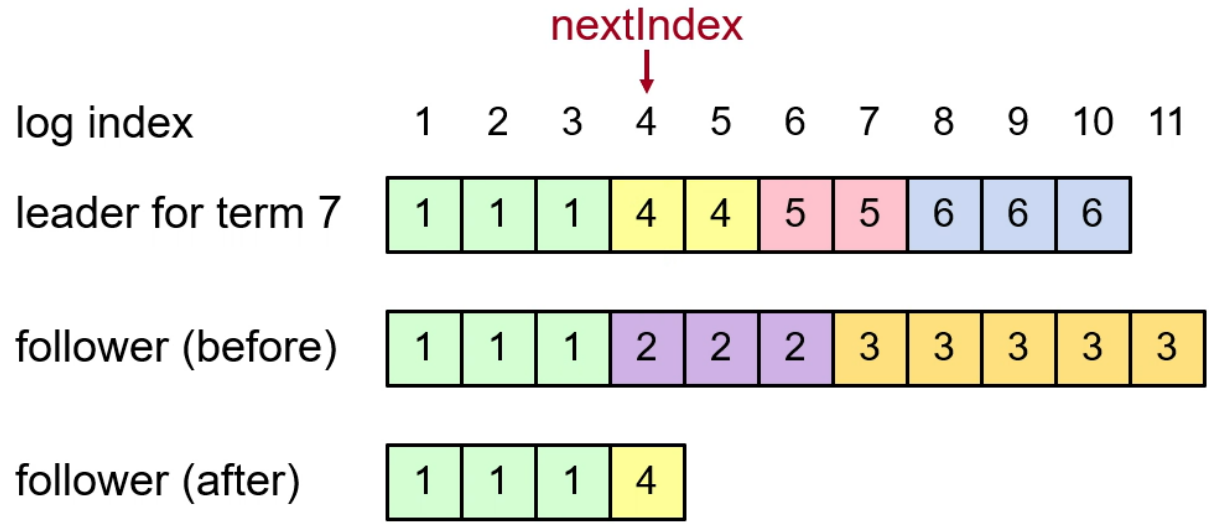


-To fix this, the leader keeps an array called nextIndex for each follower. This is an index of the next log entry to send to that follower. It is initialized to (1 + the leaders last index). When

AppendEntries consistency check fails, the leader will decrement the nextIndex for the server in question, and try again. It does this until the consistency check succeeds.



- When the consistency check finally succeeds, and the follower overwrites its first inconsistent entry, it will delete all of its subsequent entries (probably to make the subsequent consistency checks more efficient?):



Neutralizing old leaders

-A leader that is assumed dead by the followers and therefore discarded as the leader, may not really be dead. It can for example just be temporarily disconnected from the network.

-This means that when the followers have elected a new leader, the old leader may reconnect to the network and attempt to commit log entries because it believes that it is still the leader

-As mentioned earlier, every RPC contains the latest term of the sender. If the senders term is older than the follower receiving the RPC, then the RPC is rejected, and the sender reverts to follower-state and updates its term.

-Since terms are updated (incremented) every time there is an election, the old leader would have an outdated term when it is reconnected, and therefore revert to a follower-state, and not able to commit new log entries

Client protocol – How does the Raft system communicate with clients?

-The first step in client-Raft-communication, is that a client sends a command to the leader. If the client does not know which server is the leader, it can contact any server, and if that server is not the leader, it will redirect the client to the leader.

-The leader does not respond to the client until the command has been logged by the majority of servers, committed, and then executed by the leaders state machine.

-If the request times out before the leader has responded (can for example happen during leader crashes), the client sends the same command again to some other server, and will eventually be redirected to the new leader, which will retry the request.

-One problem that can occur is that the leader can crash after executing the command, but before responding to the client. It is important that the command is not executed twice.

-The solution to this in Raft, is that the client embeds a unique id for each command, and the server includes this id in its log entry. Before accepting a command, the leader checks its log for any entries with that id included in it. If the id is found in the leaders log, it will ignore the new command and send the response from the old command to the client.

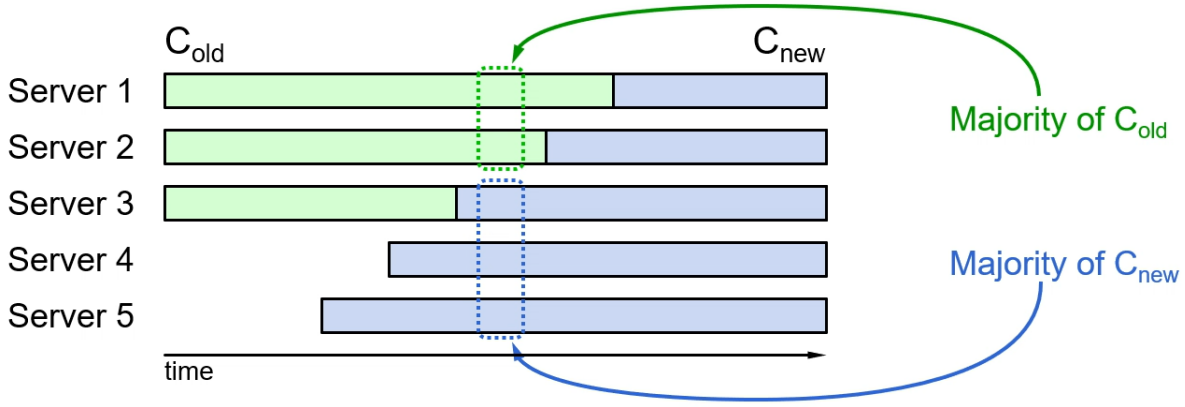
-The result is **exactly-once semantics** as long as the client does not crash

Configuration changes in Raft

-System configuration consists of IDs and addresses for each server in the Raft system. It must also be determined in the system configuration what constitutes a majority (does not necessarily have to be > 50%)

-The consensus mechanism must support changes in the configuration (for example if a failed machine is being replaced, or to change the degree of replication)

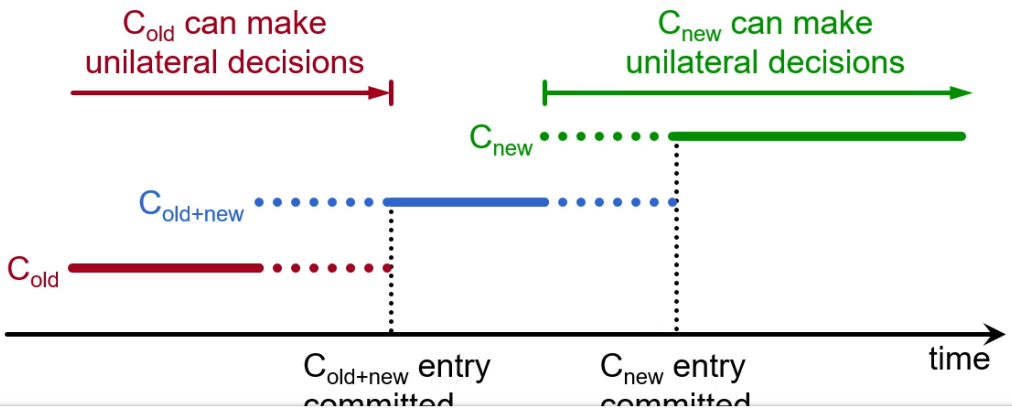
Cannot switch directly from one configuration to another: conflicting majorities could arise



Solution:

Joint Consensus

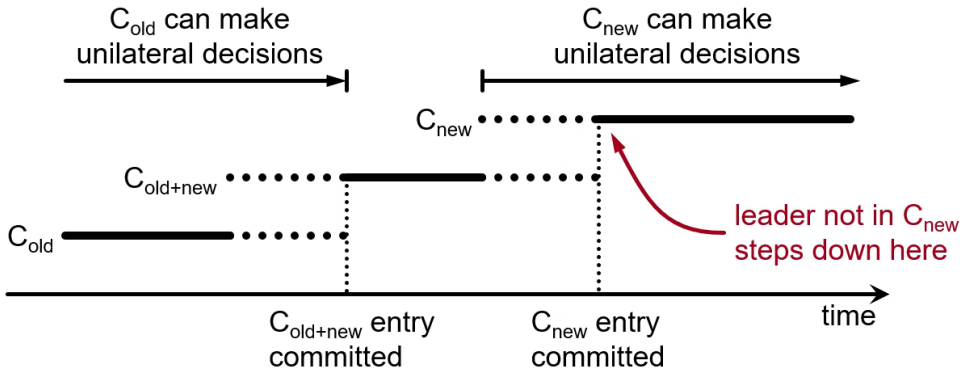
- **Raft uses a 2-phase approach:**
  - Intermediate phase uses joint consensus (need majority of both old and new configurations for elections, commitment)
  - Configuration change is just a log entry; applied immediately on receipt (committed or not)
  - Once joint consensus is committed, begin replicating log entry for final configuration



committed committed

Joint Consensus, cont'd

- **Additional details:**
  - Any server from either configuration can serve as leader
  - If current leader is not in C_{new}, must step down once C_{new} is committed.



Raft Summary

1. Leader election
2. Normal operation
3. Safety and consistency
4. Neutralize old leaders
5. Client protocol
6. Configuration changes

Dynamo: Amazons highly available key-value store

Dynamo is a highly available key-value storage system (simple NoSQL system) that was (came out in 2007) used by some of Amazons core services to provide an always-on experience. The Dynamo system focuses heavily on availability, and sacrifices consistency in order to increase availability. The reason why availability is so important is so that amazon could sell as much product as possible by being available to its customers. Even the slightest downtime would mean a big loss of revenue. Dynamo makes extensive use of object versioning. It uses vector clocks to merge differing replicas together.

Amazon's Services' Requirements

- Strict operational requirements on performance, reliability and efficiency
- Needs to be highly scalable

"Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust."

-Reliability and availability are regarded as synonyms in these slides

Why didn't Amazon use a traditional SQL database management system?

-Most of Amazons services only store and retrieve data by primary key (such as product number, customerID etc.). It retrieves objects, and does something with that object. It does not really require the complex querying (such as aggregations and grouping-operations) offered by SQL.

-Also, SQL systems typically focus heavily isolation and strong consistency, which is not really the main goal of amazons services. Their main goal is availability

System requirements and assumptions

- Simple read and write queries
- Data items uniquely identified by keys
- No operation span multiple data items

Dynamo targets applications that need to store objects that are relatively small (usually less than 1 MB).

Efficiency and Other Assumptions

- Commodity hardware infrastructure.
- Measures in the 99.9th percentile of the distribution.
- Services must be able to configure Dynamo to meet requirements in performance, cost efficiency, availability, and durability guarantees.
- Non-hostile environment: No security related requirements.
- Heterogeneous machines

Amazons Service Level Agreement (SLA)

-A single page request can include request to over 150 services, meaning that to show a single page on Amazon.com, you may need to contact over 150 different services.

-The traditional way of measuring performance is by the average, median, and expected variance, but for Amazon it is really important to perform acceptably for ALL customers, so they instead measure performance for the 99.9th percentile of customers and focus on this.

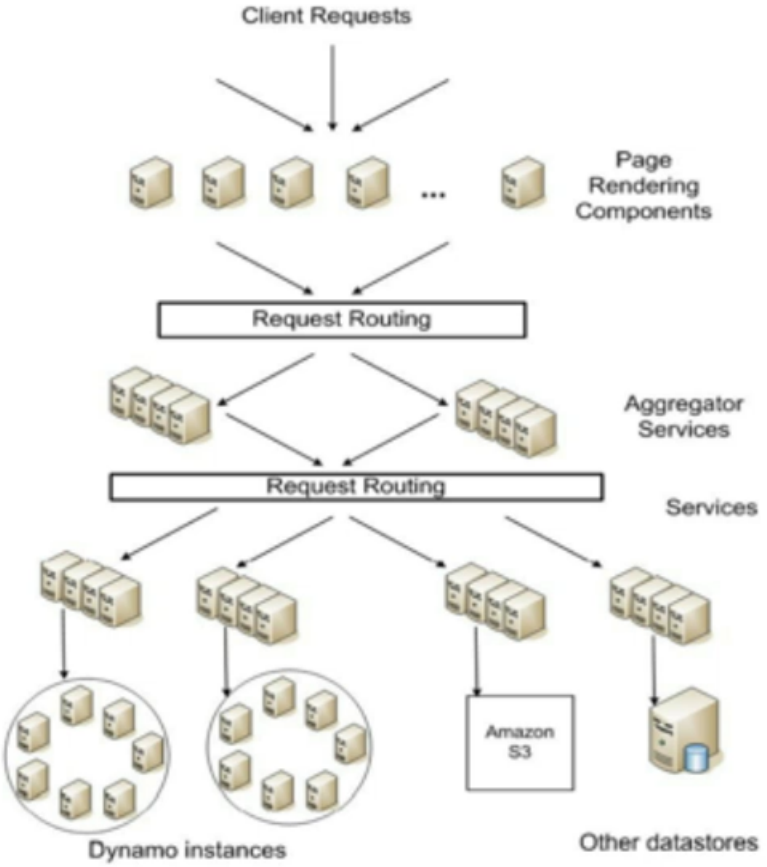


Figure 1: Service-oriented architecture of Amazon's platform

Design considerations

-Each service has control over their own system properties, such as durability and consistency, and each service can make their own tradeoffs between functionality, performance, and cost-effectiveness

-Dynamo is an **eventually consistent** data store, so it does not need to be consistent at all times.

-It is very important that it is always writeable. The conflict handling happens on read. This way, amazon can **always** process new orders.

-Both client application and the data store system can resolve conflicts.

-There are four main principles:

-Incremental scalability, meaning that Amazon should always be able to add new servers to the system as the load increases

-Symmetry, meaning that all servers provide the same services. There are no special nodes

-Decentralization

-Heterogeneity, meaning that different servers in the system can differ in power. This entails that you could for example run multiple dynamo servers on one computer if the computer is powerful enough

Key Concepts in Dynamo

Interface:

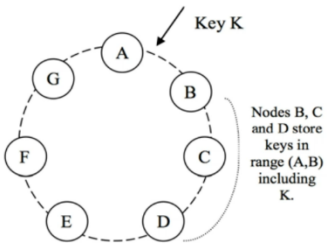
- get(key)
- put(context, key, object)

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Consistent hashing

-Invented in the 90s with the purpose of load balancing web servers

- Departure or arrival of a node only affects its immediate neighbors!
- Assign nodes a random responsible area of the hash ring (circular keyspace)
- Challenges:
  - non-uniform data and load distribution.
  - oblivious to the heterogeneity of hardware
- Solutions:
  - virtual nodes - more powerful computers take responsibility of more virtual nodes



Replication

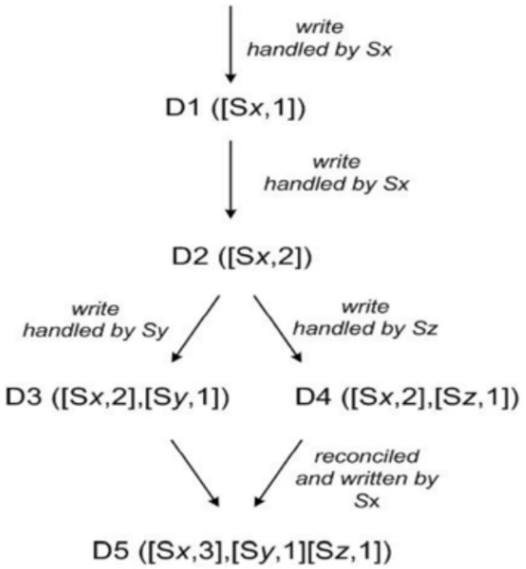
-Replication in Dynamo is done in the same hashing-ring as shown above.

-Replication happens along the ring, meaning that if some data is stored on node B, then it will also be replicated to C and D if the number of replicas required is three.

-It is the node that you are hashed to that is the coordinator, meaning that if you want to read some data with primary key K in the figure above, K is hashed to the range that node B is responsible for, and therefore, B is the coordinator. Node B then decides where to read the data from, either from just one node, which could be itself, or from several node in the form of quorums.

Data Versioning

- Eventual consistency
- Each modification is a new and immutable version of the data.
- Usually the system handles version branching, but in some cases multiple versions are returned for the client for reconciliation.
- Vector clocks to keep track between different versions



Execution of get() and put()

There are two different ways:

-Generic load balancer

-with this solution, client sends get or put request to a random node, which forwards the request to the first of the top N nodes in its preference list.

-with this solution, the client does not need to link any code specific to Dynamo in its application code.

-Partition aware client library

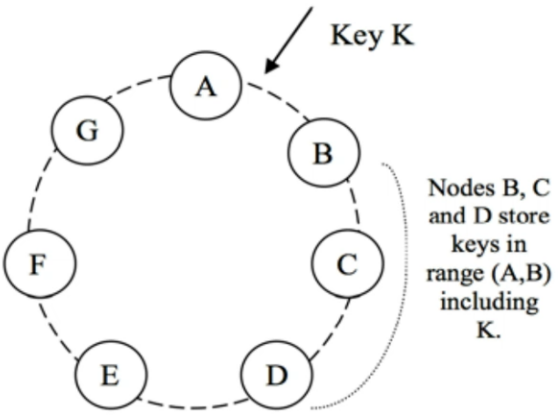
-With this solution, client sends request to one of the top N nodes in its preference list, meaning that you need Dynamo-specific code in the client application to get a preference list. This solution gives lower latency as there is one less step.

Durability Tuning

- Dynamo uses two key configurable values **R** and **W**.
- **R** is the minimum number of nodes that must participate in a successful read operation.
- **W** is the minimum number of nodes that must participate in a successful write operation. E.g.: if coordinator gets a write, it writes a new version locally then sends the new version to N healthy nodes. If W-1 nodes respond, the write is successful.
- R and W does not necessarily need to be the top N nodes, rather the top N healthy nodes.

Handling failure - Hinted handoff

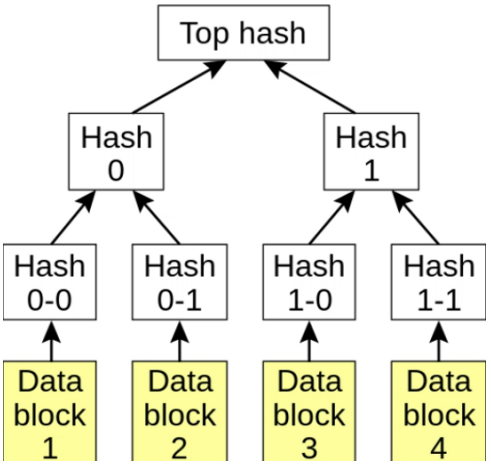
- Node A is temporary down during a write operation.
- Node D will receive the replica and a hint in its metadata that the object originally belongs to A.
- This will be kept in a separate local database and once A recovers, the object is transferred back to A and deleted.
  - Works best in low churn environments



Permanent failure – Replica reconstruction

-In some cases, replicas become unavailable

-Dynamo uses **Merkle trees** to detect inconsistencies between replicas.



-In the figure above, we can see that all data eventually hashes to the top hash. If Dynamo wants to check for inconsistencies between two replicas, it can simply just compare the root of both merkle trees, and if they differ, it uses tree reversal to find where the inconsistencies are, and fix them.

-The cost of this method is quite large, as the entire tree has to be updated if one data block is updated. The advantage of this method is that it becomes very cost-efficient to look for and fix inconsistencies, and as mentioned several times earlier, Dynamo focuses on availability instead of consistency, meaning that inconsistencies will occur quite often. It is therefore very important to make consistency-checks and repairs cost-efficient.

Membership and failure detection

A gossip-based protocol

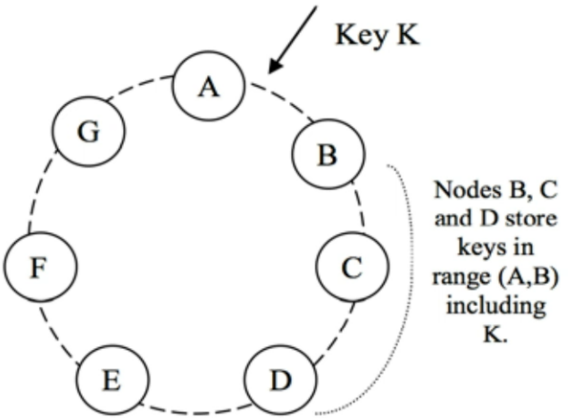
- Each node contacts a peer chosen at random every second
- The two nodes efficiently reconcile their persisted membership change histories.
- Seeds (nodes known to all nodes)
- Local notion of failure detection

Joining the ring:

- When a node starts for the first time, it chooses its set of tokens and maps nodes to their respective token sets.
- Reconciled using the same gossip-based protocol.
- Partitioning and placement information also propagates and each storage node is aware of the token ranges handled by its peers.

Adding and removing nodes

- Node X is added between A and B
- When X is added to the system, it is in charge of storing keys in the ranges (F, G], (G, A] and (A, X].
- This relieves Node B, C and D of some responsibility.
- These nodes will then offer to transfer the appropriate set of keys
- Removing a Node will lead to the process happening in reverse.



Experiences

Reconcilement done in different ways:

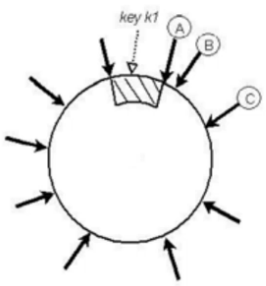
- Business logic specific
  - The client application handles divergent versions
- Timestamp based
  - Last write wins
- High performace read engine with good durability
  - R = 1, W = N

"The main advantage of Dynamo is that its client applications can tune the values of N, R and W to achieve their desired levels of performance, availability and durability."

"The common (N,R,W) configuration used by several instances of Dynamo is (3,2,2)."

-The main advantage of Dynamo is that client applications can tune the values of N, R, and W to achieve their desired levels of performance, availability, and durability.

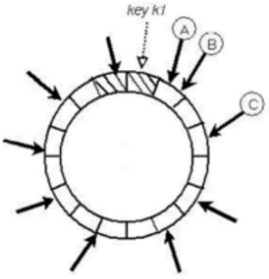
Partitioning strategies



Strategy 1

Random tokes and partitions by token value:

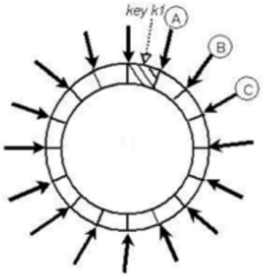
- Ranges vary
- Very resource intensive to add and remove nodes



Strategy 2

Random tokens but fixed partitions:

- Q equally sized partitions
- Consistent ranges
- Decoupling partitioning and partition placement



Strategy 3

Q/S tokens per node, fixed partitions:

- S is number of nodes in the system
- Decoupling partitioning and partition placement

Client- vs server-driven coordination

-It was mentioned earlier that when a client sends a request to a dynamo system, there is one coordinator that receives the request and decides where to read/write data from. This coordinator could also persist in the client application instead of on the dynamo servers.

-In this case, the client needs to poll membership information from a random node every 10 seconds. Since dynamo uses gossip-protocol to share information about its members, I would guess that the random node that is polled does not always have all the information about other members, as the gossip-protocol can take some time because the information has to spread through the system through messages first.

-This client-driven coordination scales better, but in the worst case, the client may try to poll a node that is dead, and therefore be stale for 10 seconds before polling again to another random node.

-I would guess that to client-driven coordination would be required for the “Partition aware client library”-way of executing gets and puts.

Dynamo's conclusion

- Dynamo provides a highly available single-key storage system for smaller sized objects.
- Provides easy developer customization by needs in regards to availability, performance and durability.
- Does not scale into tens of thousands!
  - each node has full list of routing info - lots of gossiping!
- May require application logic for data reconciliation
- Dynamo is the model for multiple NoSQL databases
- Available as a cloud service from Amazon