DDIA Reading Notes

Chapter 1: Reliable,Scalable, Maintainable Applications

Many applications today are data-intensive instead of compute-intensive. Limiting factor is not CPU power but amount of data, complexity of data and speed of data changing.

Main factor of data system design(non-functional requirement):
1.1 Reliability:
perform correct function even with hardware/software/human fault. ~fault tolerant/resilient

- Hardware Faults:
    - hard disk crash. main time to failure is about 10-50 years. On cluster with 10k disks, expect 1 disk to die per day
    - RAM faulty
    - power outage
    - internet connection

  first response is to add redundancy to each component on hardware level. but as data volume increase, moving toward on software level fault tolerance techniques.

- Software Errors
    - bug
    - runaway process
    - dependency service goes wrong

  Hardware faults are random, independent. Software faults are usually systematic and correlated.

- Human error:
    - wrong config
    - operational error

1.2 Scalability:
ability to handle growing traffic/load

1.2.1 Describe Load/Load parameter

- request per second to web server
- ratio of reads to writes to db
- fan out
- ...

1.2.2 Describe performance

- throughput: number of records can process per second(or total time to run jobs of certain size). running time ~ job size / throughput
- response time

performance has distribution, can use avg, or percentile to describe.
SLA: service level agreements

1.2.3 Maintain good performance when load param increase

- scale up(vertical): build powerful machine
- scale out(horizontal): distribute load to multiple machines

elastic: automatically add compute resource when load increase

There is no generic, one-fit-all scalable architecture. Each system is built around specific assumption of the load params.

1.3 Maintainability:

- Operability
- Simplicity
- Evolvability

Chapter 2: Data Models and Query Languages

Relational Model: data is organized into relations=tables(unordered collection of tuples/rows).
Provide good support for JOIN, and many-to-one, many-to-many relations.(data normalization: put one side of the relationship into a separate table and give index. refer in other tables by this index/foreign key)

NoSQL: not only SQL
Motivation: need greater scalability, special query operation, more dynamic expressive data model

- Document Model: data comes in self-contained doc, like JSON. Relation between docs are rare.
  - better data locality(one-to-many/tree relation), schema flexibility, closer to data structure used by app. weak support for JOIN.
  - performance advantage when often access entire doc, wasteful if only access small part of the doc each time.
- Graph Model: vertices(nodes/entities)+edges(relations/arcs)
- ...

Query Languages
...

Chapter 3: Storage and Retrieval

Data model is a logical layer, this chapter describe how they're implemented under the hood.

3.1 DATA STRUCTURES

3.1.1 Log-structured
An append-only sequence of records.

Also need to keep some additional metadata to efficiently find a particular value in db: *index*.
trade-off: well chosen indexes speed up read queries, but every index slows down writes

3.1.1.1 Hash Indexes (for key-value data)
Keep an in-memory hashmap where each key is mapped to the byte-offset in the data file. Whenever appending a new line to the log, just update the hashmap for both inserting new key or updating existing key.

Well suited for system where each key is updated frequently. lots of writes, but not toot many distinct keys. so the the size of index is much smaller than the actual data file, therefore can fit into memory.

Also to avoid running out of disk, we can break the log into segments of a certain size. whenever the current log file reach the limit, start a new file. Then perform compaction on these segments by deduping keys, and then merging small segment files into one file. Original file not modified during compaction, new merged file will be written into a new segment. So the compaction process can be done in background thread. each segment will have its own in-memory hash table. read query will check from the most recent index and so on.

Other implementation details:

- file format: can be just binary format, first few bytes encode the length of current record, followed by the raw string of the data.
- delete record: can append a deletion record to the log, will discard all previous keys during compaction.
- crash recovery: in-memory index will be lost, but log files on disk are still there, can just rebuild the index
- partial written record: can be detected by file checksum and ignore.
- concurrency control: only have one writer, and multiple readers

pros:

- high write throughput, append-only log uses sequential write operation, which is much faster than random writes. (especially on magnetic spinning disk hard drive, also preferable on ssd)
- concurrency and crash recovery is very easy since the file is append only and immutable(no overwrite operations which can leave files partially old partially new when crash happen during overwriting)

cons:

- number of distinct keys must be limited to make sure index can fit in memory
- range query is not efficient. need to look up each key individually in the index.

3.1.1.2 SSTables and LSM-Trees

SSTable(Sorted String Table): the key-value pairs in each segment log file are kept sorted by key, also each key will only occur once in each merged segment file(compaction already ensures this).

Constructing and maintaining SSTables: LSM-Tree(Log-structures merge-tree)

- maintain a sorted structure in memory (using red-black tree or AVL tree) as buffer, also called memtable
- new record will be inserted into memtable
- when memtable reaches a certain size, write to disk as a SSTable file. start a new memtable to handle new request while writing SSTable file.
- read query will go to memtable first, and then the on disk SSTable file
- run compaction in background regularly
- also to handle memtable lost during crash, can maintain another unsorted log file on disk for recovery usage. each log file will be deleted after its memtable is written to SSTable.

Performance optimization:

- use bloom filter to speed up non-exist key query

-

pros:

- merging segments will be more efficient. can use merge-sort, therefore the segment file can be larger than the available memory.
- index can be sparse to reduce index size. e.g. one key for every kilobytes is sufficient.
- efficient range query since key is sorted in SSTables
- still support high throughput since the disk write is still sequential

3.1.2 Update-in-place structured: B-Trees
Used by most of the relational databases, and also may norelational db.

Main idea is instead of break sorted structure into segments and maintain the header in memory, B-Tree directly maintain the whole sorted structure on disk.

Basic unit of B-Tree is a fixed-sized blocks/pages on disk, traditionally 4kb, read/write the whole page at one time. this is also similar to the underlying hardware, disk is also arranged in fixed sized blocks and read/write by blocks. Each page can be identified by an address/location on disk.

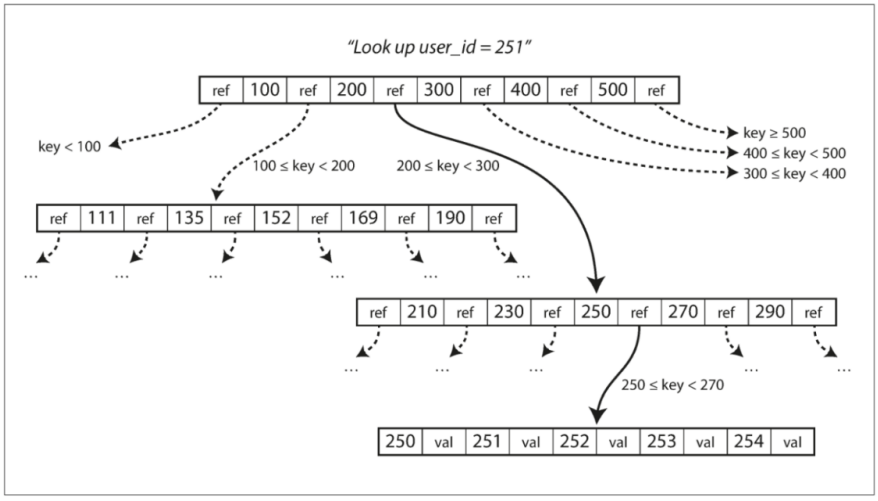


Figure 3-6. Looking up a key using a B-tree index.

- branching factor: number of refs to child in one page, typically several hundreds. e.g. for the figure above, branching factor is 6.

Updating existing key: just overwrite corresponding leaf page

Adding new key: update leaf page if it has enough free space, otherwise split it into two leaf pages, and add a new key in parent page.

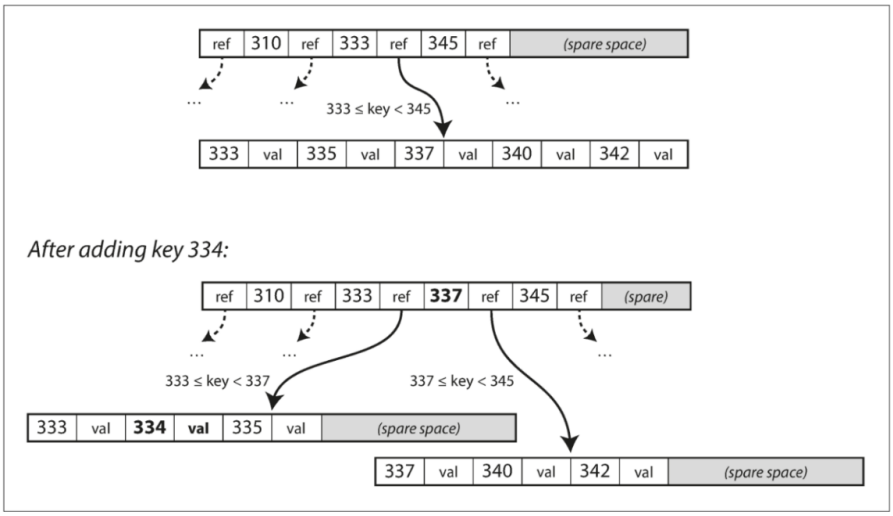


Figure 3-7. Growing a B-tree by splitting a page.

The algorithm ensures the tree is balanced: a B-tree with n keys always has a depth of O(log n). Most db can fit into a B-Tree with 3-4 layers. (4 layer + 4KB page size + 500 branching factor => 256TB ~ 500^4 * 4KB)

- Reliability issue:
  - basic operation in B-Tree is overwrite a page on disk with new data, the location of the page is usually unchanged.
  - crash during overwriting or during page splitting may corrupt the data.
  - therefore usually need to maintain another write-ahead-log(WAL) on disk to log each write operation before it's applied to B-Tree. used to recover after crash.
- Concurrency control:
  - multiple threads access B-Tree may cause inconsistent state, need to protect by locks.
  - log structure doesn't have this issue since its file is immutable
- Optimization
  - instead of update-in-place and maintain a WAL, just do copy-on-write. can solve reliability and concurrency issue.
  - can add extra pointers , e.g. leaf node can have pointer to its sibling leaf nodes to speed up range query
- it's hard to maintain pages to be sequential on disks, usually located at different locations. so large range query is not efficient as LSM-trees.

LSM-tree vs B-Tree

- LSM-tree has faster write, higher write throughput
  - LSM-tree is sequential write on disk and B-Tree is random write. (although many SSD may use a log-structured buffer to turn some random write into sequential)
  - B-Tree has high write amplification:
    - same data needs to be written to bot WAL and page
    - need to write the entire page at a time even only a few bytes changed (?)
  - LSM-tree file is more compacted, and less fragmented. Compaction on SSTables can remove fragmentation and have lower storage overhead.
- B-tree has faster read

- Compaction in LSM-tree needs to be carefully configed, compaction also takes disk bandwidth, can happen that compaction cannot catch up with incoming write and unmerged segments keep growing.

3.1.3 Other structures

- can have a secondary index on other key fields besides the primary index.
- for B-tree, the index can store a ref to the row, and the actual row is stored in an append-only heap file.
- multi-column indexes
- ...
- In-memory database:
    - for cache usage, ok to lost after crash
    - use battery powered RAM
    - maintain a log file on disk/snapshot for recovery
    - better performance since no overhead to serde, also support more flexible data model.

3.2 OLTP, OLAP

OLTP(online transaction processing): read/write to log new event taking place. Doesn't need to be involve money change, and ACID(atomic, consistency, isolation and durability).

- small number of records per query, fetched by key
- random access, low latency
- primarily used by application, user facing
- indicating latest status

OLAP(online analytic processing): batched, periodically run query, usually need to scan large huge number of records.

- Aggregate over large number of records
- used for internal analysis or decision making
- historical data

It's hard to run both OLTP and OLAP in the same system(very small scale company may still do this), so usually will maintain a separate database for OLAP: *Data Warehouse*

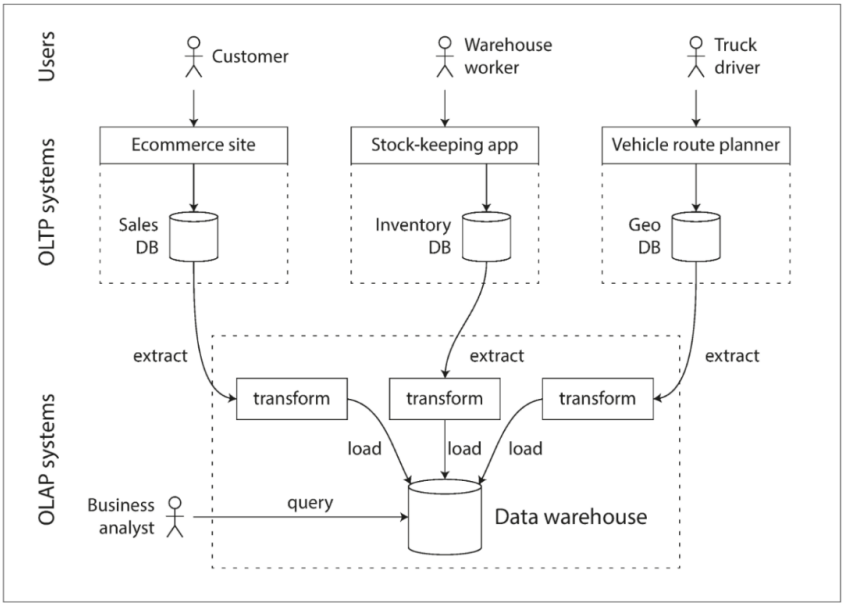


Figure 3-8. Simplified outline of ETL into a data warehouse.

Data warehouse contains a read-only copy of the data in all various of OLTPs in the company. Data is extracted from OLTP DBs periodically or continuously, transformed into analysis-friendly schema, and loaded into data warehouse. This process is known as *ETL(Extract-Transform-Load)*.

Questions

- (?) In FB:
  - OLTP: Tao
  - ETL: ?
  - OLAP: spark/presto + Hive table + warm storage ?
  - ? https://www.internalfb.com/intern/wiki/DataInfra/XLDB/XLDB_Project/
- (?) Data lake: storage for untransformed, raw data

Data Model

There are many different data models used by OLTP system based on logic in apps. But most data warehouses all use relational data model since SQL is a good fit for data analysis. Its interface looks very similar to relational OLTP db, but they are optimized for very different query patterns internally.

Specifically, most data warehouse are used in a very similar way, called: *star schema/dimensional modeling*

- one gigantic *fact table* contains all raw events
- each col in fact table represent an attribute of the event, or if there are many-to-one mapping for this attribute, it can be describes in another *dimension table*, and referenced in fact table by foreign key.
- star schema: one big fact table linked to many smaller dimension tables contains information on who,what,where,when,how,why of the event
  - e.g. ad_metrics:ad_clks_annotated + (dim_all_users, dim_adid_to_api_creative_type, ...)

- snowflake schema: dim table can have sub-dim table

Column-oriented/Columnar Storage
Most OLTP DB is in row-oriented storage, all columns in each table row are stored together, or each entire document is stored as one contiguous of bytes.

In OLAP, both fact and dimension tables can be very wide, with hundreds/thousands of columns, but most of the analysis queries only need to access a few of them at one time("select * from" is rarely needed), which makes row-oriented storage very inefficient. Therefore we use column-oriented storage: each column for all rows are stored together in a separate file. Each column file need to have the same row order, so one can easily reassemble each row by its row index.

- Column Compression: the column values in all rows are quite repetitive, many rows have identical values. Therefore can apply compression to reduce demand of disk IO throughput.
    - bitmap encoding
    - run-length encoding

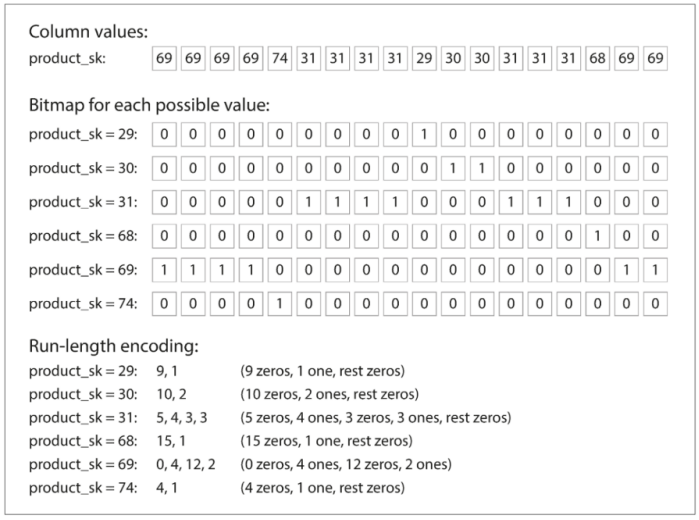


Figure 3-11. Compressed, bitmap-indexed storage of a single column.

- (?) column families != column oriented. e.g.: Cassandra, HBase are inherited from BigTable
- Vectorized Process: main bottleneck of OLAP is disk IO throughput(disk → memory), but also there is other bottlenecks like memory→CPU cache throughput. Modern CPU provide SIMD(single-instruction-multi-data) instructions, to load compressed column data to CPU's L1 cache, and iterate through a tight loop(with no function calls), which is much faster than writing for-loops in high level programming languages. e.g. Velox
- also we can sort the rows inside each column file, since all column files need to have the same order, we can only choose one column as the sort key(can also have secondary sort key when primary sort key have same values). sorting will also help the column compression, and has strongest effect on primary sort key column.
- Column oriented storage makes read queries faster, but write more difficult. Insert a new row needs to update all column files. can use similar approach as LSM-Tree to have an in-memory buffer for new write and bulk write to disk.
- Index is not very important here, since most OLAP queries require sequentially scan across large number of rows,

instead compression, minimize amount of data need to read in the query is more important.

Chapter 4: Encoding and Evolution

Data in system usually at least have two different representations:

- In-memory version, data is kept in objects in some programming language, optimized for efficient access and manipulation by CPU.
- Encoded version, when writing data to disk file or send over network. Data is kept in self-contained sequence of bytes.

Conversion between these two format is called *encoding/serialization/marshalling, decoding/parsing/deserialization/unmarshalling.*

LANGUAGE SPECIFIC ENCODING FORMATS

Many languages have builtin support for encoding im-memory objects into byte sequences, like java.io.Serializable in Jave or pickle in Python.

- pros: convenient to use, require little code, aimed for quick and easy usage
- cons:
    - tied to only one language, no trans-language support
    - decoding process need to be able to construct arbitrary classes, have security issues
    - versioning support is afterthought
    - efficiency is afterthought

JSON, XML, CSV...

- pros: textual format, human readable
- cons:
    - ambiguity around encoding numbers: not distinguish int and float, not specify precision
    - doesn't support binary data(usually need to use Base64, but data size is 33% larger)
    - no good schema support

THRIFT, PROTOCOL BUFFER

Binary encoding library, have their own API to define data schema and use a codegen tool to produce the class that implement the schema in various languages. Application code can call the generated code to serde.

Thrift

```
struct Person {
    1: required string userName,
    2: optional i64 favoriteNumber,
    3: optional list<string> interests
}
```

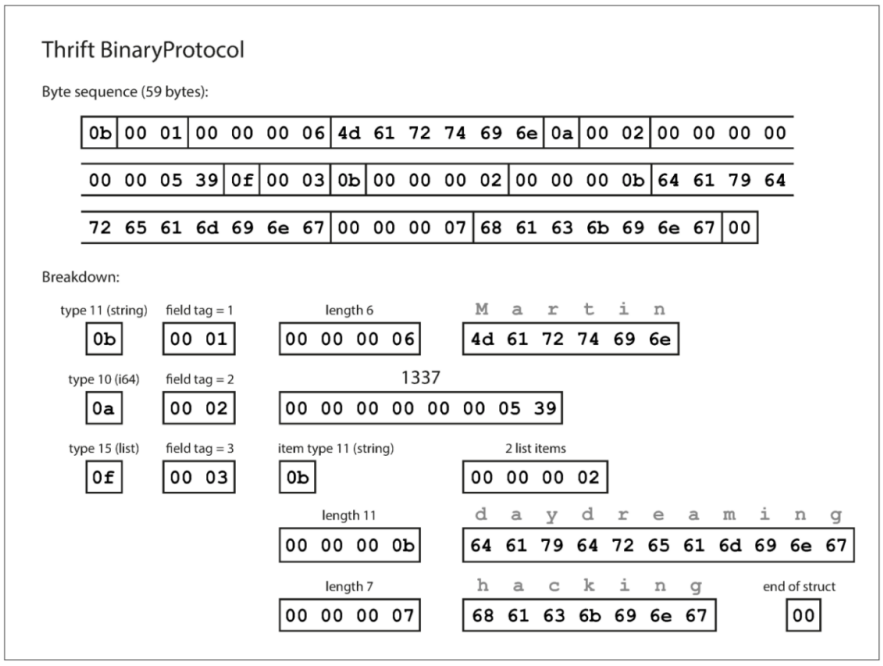


Figure 4-2. Example record encoded using Thrift's BinaryProtocol.

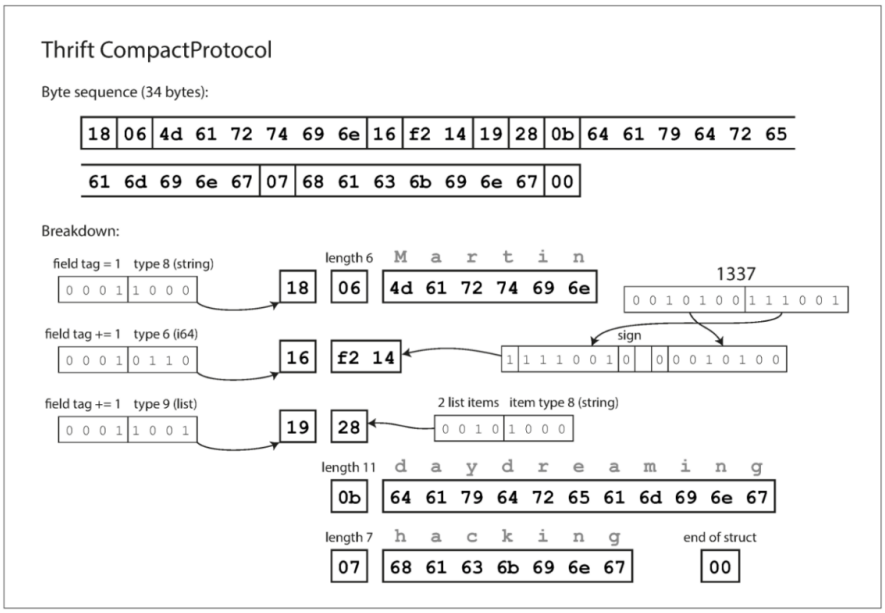


Figure 4-3. Example record encoded using Thrift's CompactProtocol.

Protobuf

```
message Person {
 required string user_name =1;
 optional int64 favorite_number =2;
 repeated string interests = 3;
}
```

encoding similar to Thrift CompactProtocol.

- required/optional annotation doesn't affect the encoding. only difference is required field will have a runtime check during serde code
- fields are only refered by the field index number in the encoding. So field name change is allowed.
    - forward compatibility: old code can read new version data: ignore unrecognized new fields index
    - backward compatibility: new code can read old version data: field added after the initial deployment can only be optional or have default value. also can only remove optional field

AVRO

Developed for Hadoop. Main difference with Thrift,protobuf is it doesn't have field index number in schema definition, encoding just concatenate each fields sequentially. So format is more compact, and also support dynamically generated schema. Avro provide codegen API, but it's optional. The file is self-describing, can directly open and parse without a schema file. Main use cases in Hadoop:

- large file with millions of records, all encoded with same schema
- intermediate files generated inside the DB

MODES OF DATAFLOW

- via database
- via service call
- via asynchronous message passing
    - message broker/message queue/message-oriented middleware
        - buffer to decouple sender and recipient, thus improve reliability
        - auto-redeliver after crash
        - support multiple recipients
        - asynchronous: sender doesn't wait
        - e.g. Kafka

[TODO]: list of common DBs and their property?

Chapter 5: Replication

Replication: Keeping a copy of the same data on multiple machines

- keep data geographically close to users to reduce latency
- increase availability
- increase read throughput

Difficulty: handle changes to replicated data

Three common solutions: single-leader, multi-leader, leaderless

SINGLE-LEADER

leader-based-replication/ active-passive/ master-slave replication: write request can only go to leader node, read request can go to any node.

follower node can be synchronous or asynchronous:

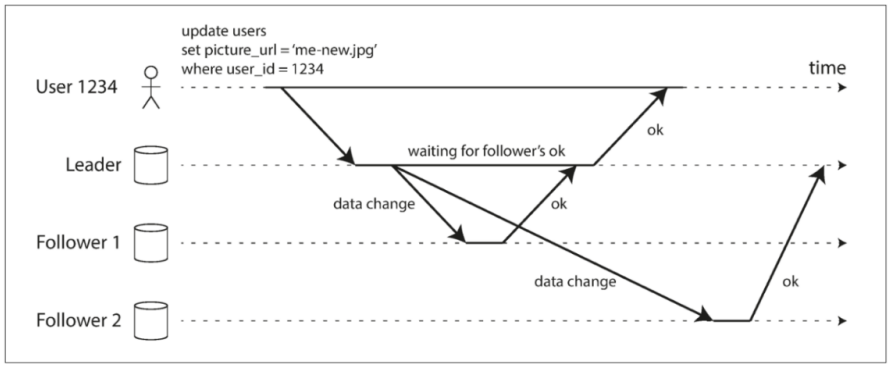


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

- Setting up new follower node:
  - take a consistent snapshot of the leader's data + leader need to maintain log for every operation
  - copy snapshot to new follower
  - new follower request data changes happened since the snapshot was taken, until it caught up.
- Handling node outage
  - Follower failure
    - follower also maintains a log of data changes received from leader
    - same as setting up new followers
  - Leader failure/Failover
    - Manually failover: select a follower as new leader and reconfig clients to send writes to new leader
    - Automatic failover:
      - Determine that the leader has failed:
        - many things can cause node failure, most system use timeout to determine: nodes frequently send messages to each other, if a node doesn't respond for some time, it's assumed to be dead.
      - Choose a new leader:
        - need to have en election process, or use previously elected backup node
        - best candidate is the replica with the most up-to-date data changes to minimize data loss
      - Reconfig system to use new leader
        - client send to new leader
        - old leader becomes follower after recover
    - Issues:
      - If asynchronous replication is used, new leader may haven't received all writes from old leader, then when the old leader is recovered, usually those writes are discarded. which may violate clients's durability expectations, also cause errors in downstream systems.

        - Could happen that two nodes both believe they are the leader(split brain), may have another process to shit down one of them, but then will have risk to shit down both...
        - How to decide the right timeout? Longer timeout means a longer time to recover. Too short timeout could cause unnecessary failovers. e.g. a tmp load spike/network lag caused node's response time to increase, system is already struggling with high load, so unnecessary failover will make it worse.
        - no easy solutions to those issues, so sometime people prefer to perform failover manually.
- Replication Implementation
    - Statement-based log Replication: Leader log each write request statement(e.g. SQL query), and send to followers.
        - pros: statement log is compact
        - cons: statement can contain nondeterministic functions(e.g. now(), autoincreasing column), or have other side effect
    - Write-ahead log(WAL) replication: For log-structured storage engine(SSTable), data file itself is the log; for update-in-place structures storage(B-Tree), log each modification on the page block, it's also needed for crash recovery.
        - pros: log is on bytes level, so it's deterministic
        - cons: log is too low level, closely coupled with the storage engine, cannot run different versions of the software on leader and followers. so hard to perform zero-downtime software upgrade.
    - Logical(row-based) log replication:
        - decouple from storage engine and physical data representation.
        - easier for external app to parse

Replication Lag
single-leader works very well on scaling read request, but all writes need to go to the leader node, therefore usually use asynchronous way to replicate to followers. Eventually will reach consistent state but there will be lag between leader commit the write to client and all replica receive the write. This effect is called *eventual consistency.* This may cause some issues in real world:

- Reading your own writes: same user read the value just write
    - need *read-after-write/read-your-write consistency*. (no promise on other user's write)
    - implementation:
        - when infer the read contain data that user may modified, direct to leader node only
        - track the timestamp of last update on some data, direct to leader node if read comes within certain amount of time after the latest write
        - or track a logical timestamp, like a counter, for each write.
        - difficult to do when user read from a different device, also when replica is distributed in multiple data centers.
- Monotonic reads:
    - user read the same data multiple times. request may direct to different replica and may see inconsistent result.
    - solution: can ensure one user always read from same replica. like to route read request based on hash of user id
- Consistent prefix reads:
    - two sequential writes may arrive at a replica in different order. so the "happens-before" relationship may be messed up.

Replication lag are very hard to handle. Usually need to have very complex logic on the client side which is easy to get wrong. So it will be better if the database itself can provide strong guarantee on its behavior so client don't need to worry about the lag. Then can decouple the replication logic with client logic. Will discuss this in later chapter on "transaction".

MULTI-LEADER REPLICATION

One main constrain on single-leader mode is low write throughput, so natural way to extend is add more leaders to handle more write requests. It's usually used in several scenarios:

- Multi-datacenter
    - db is replicated in multiple datacenters to tolerate datacenter failure, also close to user.
    - therefore can have one leader in each datacenter
- Collaborative editing
    - e.g. quip, gdoc. multiple people can modify same file concurrently. So write applies to their local replica first, and then replicated to other people's machine asynchronously.

Biggest issue for multi-leader is to handle write conflicts:

- simplest way is to make sure all writes for a particular record go through the same leader node to avoid conflict.
- Give each write a unique ID or timestamp, later write can overwrite old write.known as last write wins(LWW).
- somehow merge different data together by some special logic or use special mergable data structure.
- provide atomic write operations

Topologies:

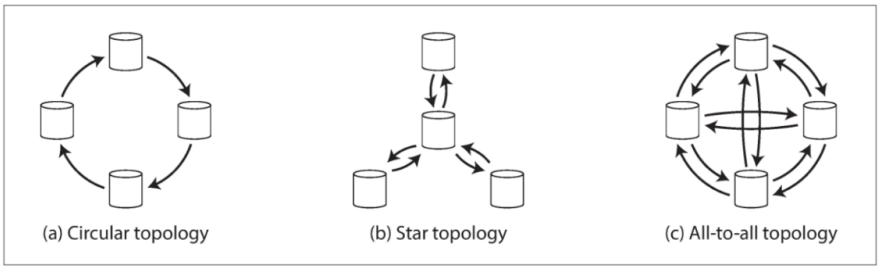


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

- In circular and all-to-all schema, to prevent infinite loop, we can give each replica a unique ID and append it in the replication request to make sure we don't go through duplicated nodes.
- In circular and start topology, one node failure may fail the overall flow, and need to be reconfiged. so usually more densely connected topology can have better fault tolerance.

LEADERLESS REPLICATION

Both write request and read request are send to multiple replicas, also known as dynamo-style.

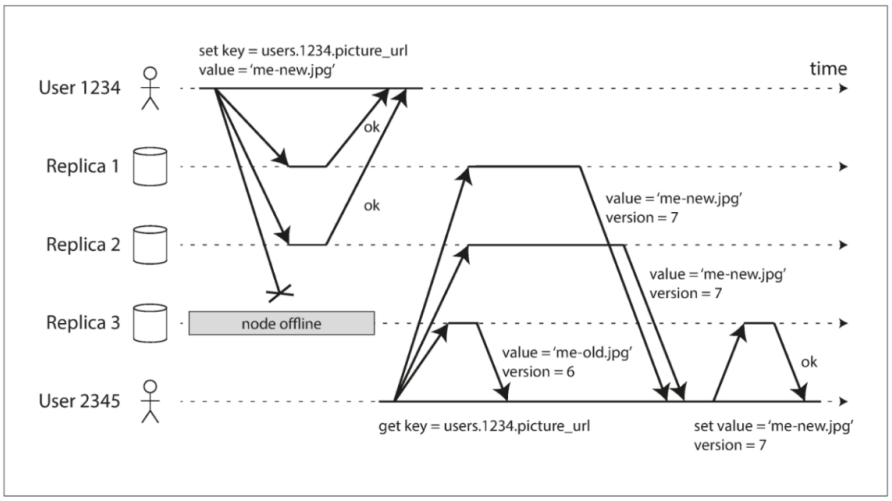


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

- Node failure handling:
  - Read repair: when client send a read to multiple replicas, it will figure out which node has stale value and write the new value back.
  - Anti-entropy process: background process run in DB to check difference between replicas and fix.
- Quorum for reading and writing
  - Write request is send to multiple nodes, how to decide if the write is success or not:
    - In the example above, write is considered to be successful when two out of three replica are successful
    - more general, the write request should be good as long as the later read request can get at least one replica with latest result.

> More generally, if there are n replicas, every write must be confirmed by w nodes to be considered successful, and we must query at least r nodes for each read. (In our example, $n = 3$, $w = 2$, $r = 2$.) As long as $w + r > n$, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date. Reads and writes that obey these r and w values are called *quorum* reads and writes [44].[vii] You can think of r and w as the minimum number of votes required for the read or write to be valid.

  - w,r,n are configurable in the DB, common choice is to make n an odd number, and w=r=(n+1)/2
  - In read heavy system, can also make w=n, r=1 to make read faster.
  - Usually w and r are smaller than n, which means write/read can tolerate failure on some nodes in the system.
  - Normally read and write are send to all n replicas, but will only wait for w/r successful node response to commit to client.
  - Quorum consistency may still have issues:
    - concurrent writes
    - when write happens concurrently with a read, result of the read is undetermined.
    - write succeed on some nodes but failed on others, and is decided as failed overall, those successful writes are not rolled back.
    - all the issues on replication lag in single-leader case.
  - Dynamo-style DB are generally optimized for use cases can tolerate eventual consistency. w,r,n can adjust the

probability, cannot provide absolute guarantees.

- Sloppy quorums and hinted handoff: there might be cases that some nodes within n are down, so the number of remaining living nodes are less than w. We can then reach out some other extra nodes outside of n(thinking in a large cluster that number of total nodes >> n), and write the value there. later when nodes within n are recovered, the extra node will write the value back to them.
  - so as long as any w nodes are available no matter if it's within n or not, we can consider the write to be successful.
  - increase write availability

- Detecting concurrent writes
  - in leader based DB, order of writes are well defined. but in leaderless system, there is no clear order for concurrent writes. different replica may have different orders.
  - Usually need to add timestamp or version number for every key inside the storage to detect conflict writes and casual dependency. Each replica will need to maintain their own versions, so overall will have a version vector.

Each approach has advantage and disadvantages. single-leader has no consistency issue, multi-leader and leaderless can be more robust to node failure and network lag and scalable, at cost of weak consistency guarantees.

Chapter 6: Partitioning

To improve system scalability, we can split data into partitions, and distribute on multiple machines. Partitioning usually is combined with replication, so each partition is also replicated on multiple machines.

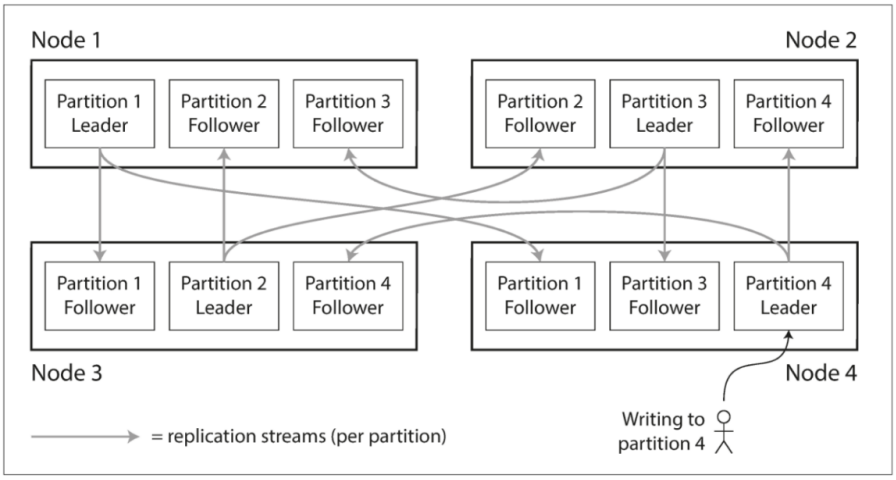


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

Choice of partition schema is mostly independent of the choice of replication schema, so we can just ignore replication in this chapter.

- Partition by key range
  - pros: range query is efficient
  - cons: certain query pattern may cause hot spot

- Partition by hash of the key, then by hash range
    - pros: make all keys uniformly distributed on partitions, less skewness
    - cons: lost key range sort property. range query need to send to all partitions
- (Cassandra supports a compromised way: can declare a compound primary key consisting of multiple columns, first col is hashed to determine partition, then within one partition, SSTable is sorted by the concatenated key of other secondary cols)

[TODO] consistent hashing

- Even with hash key partition, workload can still be skewed. e.g. celebrity user with millions of followers. Cannot be handled automatically. Usually need to manually split it further, like add a random number to its key to distribute to multiple partitions, also read request will be query multiple partitions to collect data. So can only apply on small amount of really hot keys.

REBALANCING PARTITIONS

Very often we need to adjust the partition setup, like add more nodes to handle larger traffic, or replace the failed machine. Therefore we need to move the data around and also reroute request to new partitions, this is called rebalancing.

- rebalancing should keep the load(data size, request throughput) shared fairly between nodes.
- system should still be able to handle new request while rebalancing.
- minimize disk/network IO as much as possible

A few strategies for rebalancing:

- hash mod N
    - simplest way is to use `hash mod N` to assign partitions. N is total number of nodes.
    - Not recommended since most of keys will be moved when N changed.
- Fixed number of partitions
    - create large amount of logical/virtual partitions that is much more than the number of nodes. and then assign logical partitions to physical nodes.
    - one physical node contains many logical partitions. when new node added, just steal one partition from each node and give to the new node:

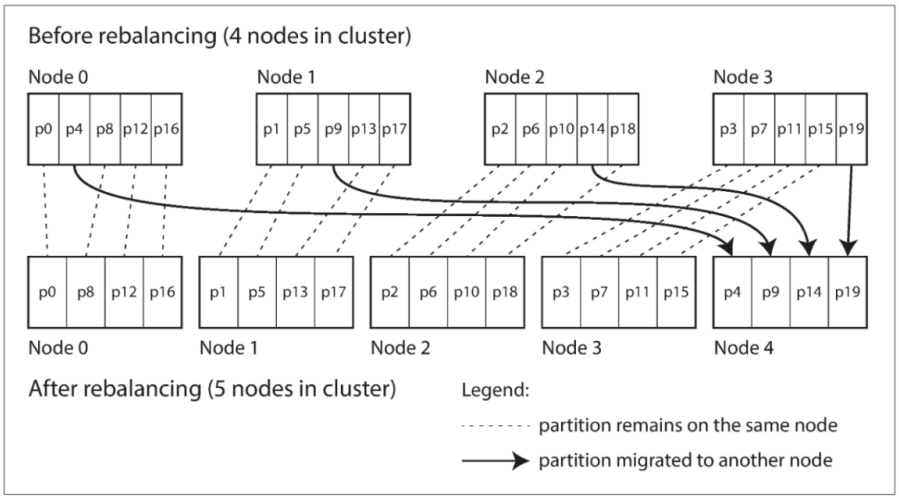


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

    - key→partition mapping is not changed, total partition number is not changed.
    - size of each partition grows proportionally to the total amount of data in the system.
    - choose right number of partition at the beginning is difficult.
- Dynamic partitioning
    - key→partition is not changed in fixed partition number, then some hot partition will always be hot, or size of some partitions may be much larger.
    - therefore can allow split large partition into two(when its size hit preconfiged limit), similar as B-Tree. Also when lots of data is deleted from one partition, it can be merged with nearby partitions.
    - number of partitions grows proportionally to total data size
- Partitioning proportionally to nodes
    - have a fixed number of partitions per node. so number of partitions is proportional to number of nodes
    - when a new node joining, randomly choose a fixed number of existing partitions to split, and take one half of each split partitions.

Rebalancing is an expensive operation, requires to reroute request and move large amount of data between nodes. Similar to the failover handling for replication, partition rebalancing could be tricky to be handled automatically. Especially combined with auto node failure detection: when a node is overloaded and becomes slow. If the system determines it's dead, and start auto rebalancing, will introduce extra load. So many system choose to handle rebalancing manually.

REQUEST REROUTING

More general topic: service discovery/routing. a few ways on high level:

- clients doesn't know partition configuration, client request will be send any arbitrary node(e.g. through a round-robin load balancer). The node will direct the request to appropriate nodes.
- have a routing tier holding the partition schema, and all request will go to the routing tier first. partition-aware load balancer
- client be aware of the partition schema, and contact node directly.

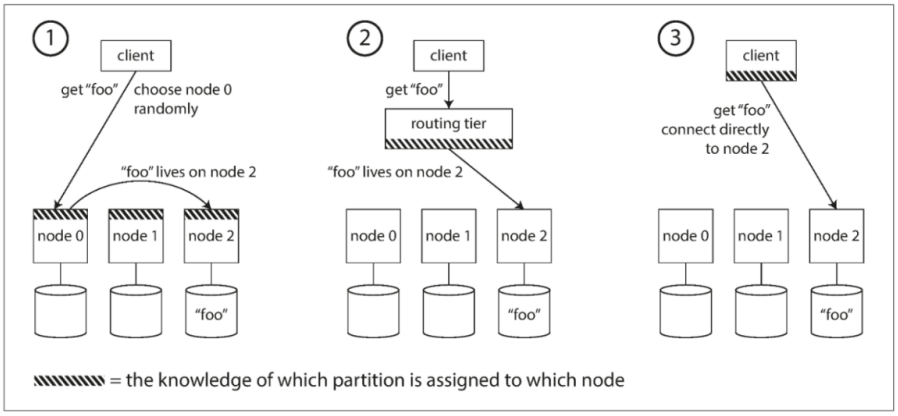
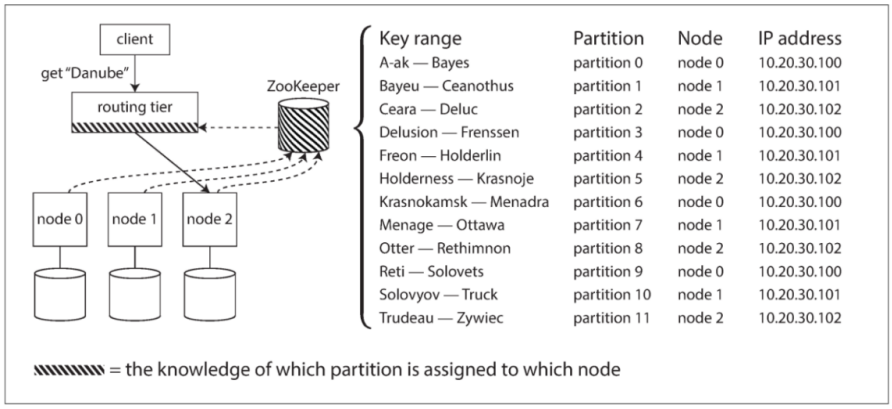


Figure 6-7. Three different ways of routing a request to the right node.

Whichever approach, rerouting request when rebalancing is a challenging problem:

- many system rely on a separate coordination service, like Zookeeper to track those partition metadata.



Key range	Partition	Node	IP address
A-ak — Bayes	partition 0	node 0	10.20.30.100
Bayeu — Ceanothus	partition 1	node 1	10.20.30.101
Ceara — Deluc	partition 2	node 2	10.20.30.102
Delusion — Frenssen	partition 3	node 0	10.20.30.100
Freon — Holderlin	partition 4	node 1	10.20.30.101
Holderness — Krasnoje	partition 5	node 2	10.20.30.102
Krasnokamsk — Menadra	partition 6	node 0	10.20.30.100
Menage — Ottawa	partition 7	node 1	10.20.30.101
Otter — Rethimnon	partition 8	node 2	10.20.30.102
Reti — Solovets	partition 9	node 0	10.20.30.100
Solovyov — Truck	partition 10	node 1	10.20.30.101
Trudeau — Zywiec	partition 11	node 2	10.20.30.102

Figure 6-8. Using ZooKeeper to keep track of assignment of partitions to nodes.

- gossip protocol among nodes. similar to the approach 1.

Chapter 7: Transactions

As we've seen, many things can go wrong in distributed system, and applications need to implement lots of complex logic to achieve fault tolerant. It would be great if we can abstract out those logic and hide inside the db implementation so db have these safety guarantees and applications don't need to take care of concurrency issues. Therefore we can simplify the programming model for applications accessing a db.

Transaction is proposed as a strong consistency guarantee. It's supported in most of the relational database and some NoSQL databases.

TRANSACTION

The safety guarantee provides by transaction are often known as ACID: atomicity, consistency, isolation, durability. But these properties may have different representations in different system. So it's actually not very clear what guarantees you can expect when a system claims to be "ACID compliant".

- Atomicity:
  - In multi-thread programming, atomicity means operation on one thread cannot be interrupted by other threads. so other thread can only see the state before and after the operation, not intermediate state. Basically the guarantee for concurrent operations. this is actually the isolation in ACID, not atomicity.
  - In ACID, atomicity means if client want to make several changes in one commit, then this transaction is either successful committed or aborted when there is failure so can be safely retried. So atomicity is more like abortability.
- Consistency:
  - In replication, consistency means same data for all replicas.
  - In ACID, means db in a "good state", some certain statement(invariant) should always hold. It's actually a property of the application which rely on atomicity and isolation, not really belong to db and ACID.
- Isolation:
  - Isolation between concurrent operations. If one transaction makes several writes, other transaction should either see all of them or none of them.
- Durability:
  - once committed, the data won't be lost
  - Historically implemented by writing data to disk, but more recently has been adapted to replication. both approaches have pros and cons....

ISOLATION LEVELS

Here the most important property is probably isolation which is mainly for concurrency handling. But isolation is just a very high level concept. As we just mentioned, different system may have different representation on this, and provide isolation on different levels. Strong isolation level can provide better guarantee for concurrency issues(race conditions), but also with large performance cost. so different system need to select proper isolation level depends on trade-offs.

Read committed:
The most basic level of transaction isolation:

- no dirty read: read request can only see data that has been committed

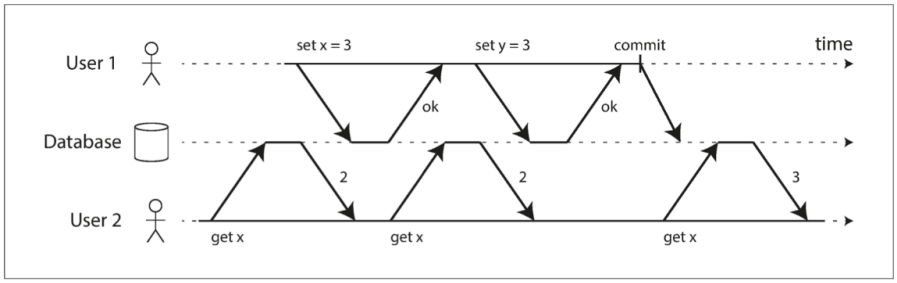


Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.

- no dirty write: write request can only overwrite data that has been committed. if there is an ongoing transaction that modified a record. other transactions writing on this record need to wait until the previous one is committed or aborted.

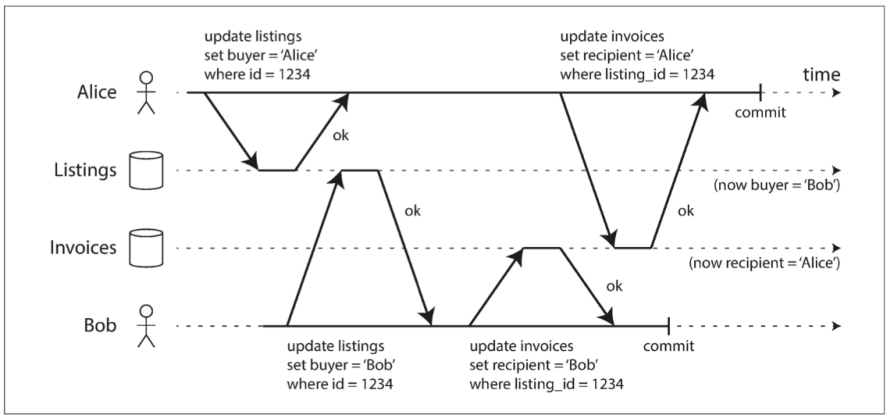


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

Implementing read committed:

- preventing dirty write is implemented with a row level lock. the lock will only be released when the current transaction is committed or aborted.
- preventing dirty read:
  - can do the same as dirty write handling with a lock, but will largely harm the read throughput
  - so usually can preserve the old value when new write coming, will return the old value when the new write is not committed.

Snapshot isolation

There are situations that read committed isolation can go wrong. e.g. non-repeatable read/read skew:

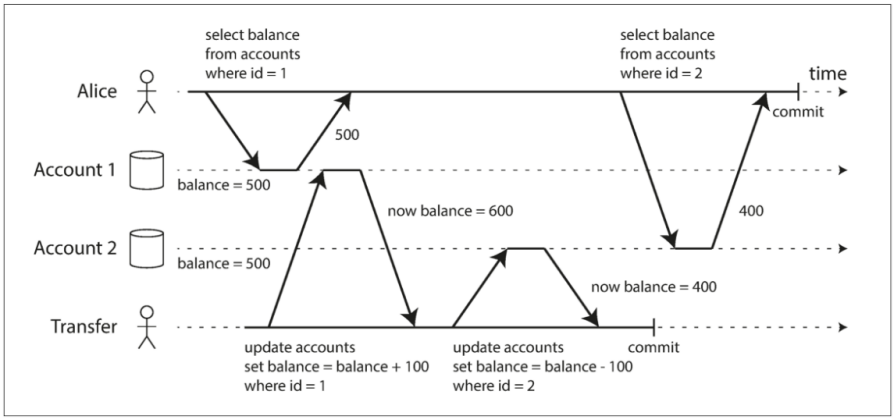


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

This temporary inconsistency might be acceptable for many systems, but may cause trouble in some cases, like in some one time analysis queries or integrity checks, also during taking snapshots for the db.

- Therefore snapshot isolation/repeatable read is proposed to prevent this issue. Idea is that one transaction should read

from a consistent snapshot of the db. if there is new changes committed during a transaction, it should still get the value from the snapshot of the db when the transaction started.

- also for performance, a key principle for snapshot isolation is readers never block writers, and writers never block readers.
- very useful for read-only transactions.
- Implementation: similar as implementation of read committed, we can implement snapshot isolation with keeping multiple versions of the data. But instead of only two versions, we should keep have a unique version number/transaction id for each write operation. Basically we never update values in place but instead creating a new version for the data. and always return a certain version of the data for read requests. This technique is called multi-version concurrency control(MVCC).

Preventing lost updates

Even with read committed and snapshot isolation, there are still cases known as lost updates problem, e.g.:

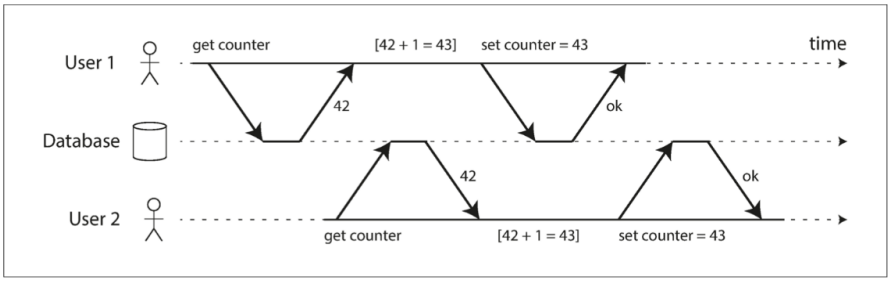


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

Usually occurs in read-modify-write operations. So second write doesn't include the change from first write. e.g. incrementing a counter/bank account, make local change on a complex value like json doc or wikipage.

Some solutions to prevent:

- Atomic read-modify-write operation: db can provide atomic write operation. implemented by exclusive lock on each object for read operations.
- Explicit locking: application can set lock when db doesn't provide atomic operation.
- Basically just to force all read-modify-write operations execute sequentially
- another way is to let them run in parallel, and have another thread detecting lost updates, and abort the offending transaction and force retry.
- also some db may provide compare-and-set operation, can use it to prevent lost update as well.
- When db is replicated, explicit lock and compare-and-set won't work since they assume there is only one copy of the data. atomic read-modify-write can work for commutative/mergable operations(like increase counter), so each replica will execute them sequentially but maybe in different order, in the end can get the same result.

Write skew and phantoms

A more generalized pattern for lost update, two transactions may update two different objects so no conflict from db side, but may cause conflict on some customized logic inside applications, like creating two meetings in the same room, creating two accounts with same user name. Root cause is you have some logical constraints in application that is not in db. Usually need to address by explicitly represent the conflicts on some concrete rows in the db, like create an object for each username/meeting room, called materializing conflicts.

Serializability
The strongest isolation level should be serializable isolation, which means each transaction can pretend that it's the only transaction running on the entire db. Result of concurrent operations should be same as if they had run serially. This should prevent all concurrency issues. Other weaker non-serializable isolations may only protect against some of the issues for better performance.

Implementation:

- Actual serial execution: literally run transactions in a serial order
    - Stored procedures: instead of interactively operations, each transaction need to submit all steps it wants to execute in the request, which will be queued in system and run sequentially.
    - Partitioning: If the db can be partitioned, then can execute transactions on different partitions in parallel. but will need lots of handlings if a transaction needs to involve multiple partitions.
- Two-phase locking(2PL)
    - In read committed, lock is only exclusive for writes, reads won't be block.
    - 2PL basically just makes it stronger:
        - ongoing transaction with write will block read
        - ongoing transaction with read will block write
    - Implementation: lock will have two mode: shared mode(can be held by multiple transactions) and exclusive mode(can only be held by one transaction)
        - read request needs to acquire the lock in shared mode, so concurrent read is allowed.
        - write request needs to acquire the lock in exclusive mode, so need to wait for all shared mode locks or exclusive mode lock are released.
    - there are many overhead in 2PL, so its throughput and response time are very bad. also have much higher frequency to have deadlock and need to abort transactions and retry.

Serializable snapshot isolation(SSI)
New, still developing approach, but looks promising. main idea is not to block concurrent operations, just let everyone run in parallel, and actively detect conflict and abort. Provide full serializable with only little performance penalty compared to snapshot isolation.

Chapter 8: The Trouble of Distributed Systems

If we want to make distributed system work, we must accept the possibility of partial failure and build fault-tolerance mechanism into software. In other words, we need to build a reliable system from unreliable components.

UNRELIABLE NETWORKS

For share-nothing systems, network is the only way those machines can communicate.

- sent request may get lost, delayed
- remote node may failed and getting slow.
- response may get lost delayed.

Medium-sized datacenter can have 12 network faults per month, half of them disconnect a single machine, half of them disconnect a whole rack. *Network partition/netsplit*: one part of the network is cut off from the rest due to network fault.

There is no way to tell what's going on, so usually just use timeout to determine if a machine is dead or not. But hard to decide timeout value. Network have unbounded delays due to all traffics are sharing same bandwidth instead of telephone network where each line has its reserved bandwidth.

UNRELIABLE CLOCKS

Each machine, each CPU in one machine can have their own clocks. Hard to synchronize them, yet many things are relying on synchronized clocks, such as: ordering events by timestamp, creating version number/transaction ID, lock timeout.

Chapter 9: Consistency and Consensus

How to build fault tolerant distributed system with all the troubles described in chapter 8. The best way is to find some general-purpose abstractions with useful guarantees, implement them once, and let application relying on those guarantees, same as transaction. One fundamental abstraction here is consensus, which can be used for leader selection.

CONSISTENCY GUARANTEES

Consistency is mainly caused by replication and replication lag. Usually we can have eventual consistency which is not enough for many use cases.

Linearizability
Strongest consistency guarantee in distributed system is called Linearizability/atomic consistency/strong consistency/immediate consistency/external consistency: make a system appear as if there were only one copy of the data and all operations on it are atomic. No parallel universe, only one single timeline. As soon as one client completes a write, all clients should read this new result. All read results are most recent, up-to-date value. In other words, linearizability is a recency guarantee.

No linearizability:

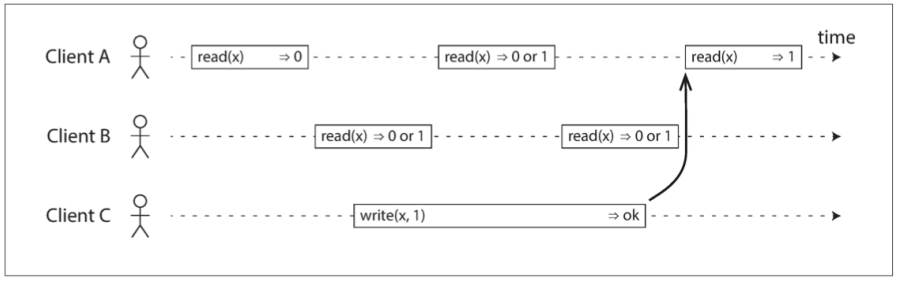


Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

With linearizability:

We can imagine for the write operation, there must be some point in time where the value of x atomically flips from 0 to 1.

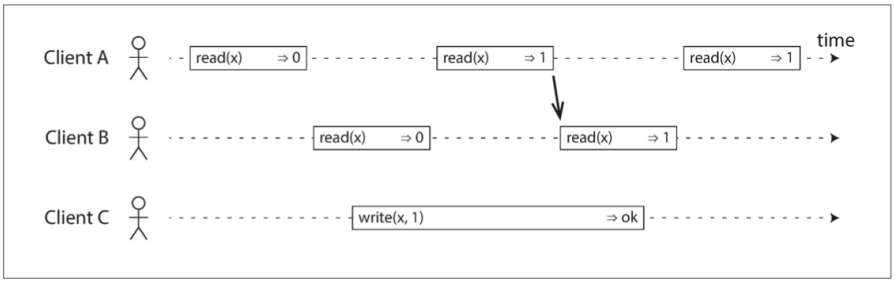


Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

Basically each operation should take effective at a single point in the timeline. A more complex case:

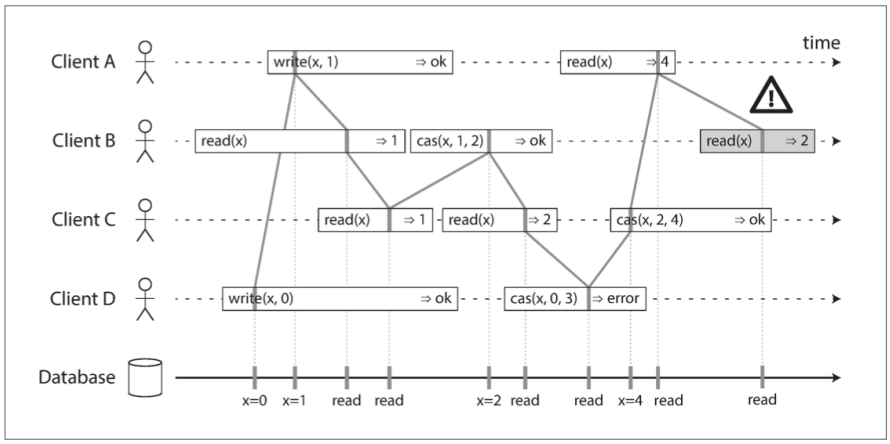


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

- Linearizability vs Serializability
  - Serializability is more about isolation for concurrent operations, so that “can be arranged in a sequential order”. Can also be applied to non-distributed systems.
  - Linearizability is more about consistency guarantee for distributed/replicated system, cannot prevent issues in transaction like write skew.
- Usage of Linearizability storage service
  - Distributed lock, leader election: once a node acquire the lock/elected as the leader, all other nodes should agree on this. otherwise it’s useless.
  - Constraints and uniqueness guarantee: e.g. unique username, email address, flight booking, atomic compare-and-set..... serializable may also needed here, but linearizability emphasize on that all nodes need to agree on the result.
  - ....
  - basically to avoid race conditions
- Implementing Linearizability systems
  - Single-leader replication (potentially linearizable): synchronous replication is linearizable, asynchronous is not.
  - Multi-leader replication (not linearizable)

- Leaderless replication (probably not linearizable): we can use quorum(w+r>n), but it actually cannot guarantee linearizability with replication lag:

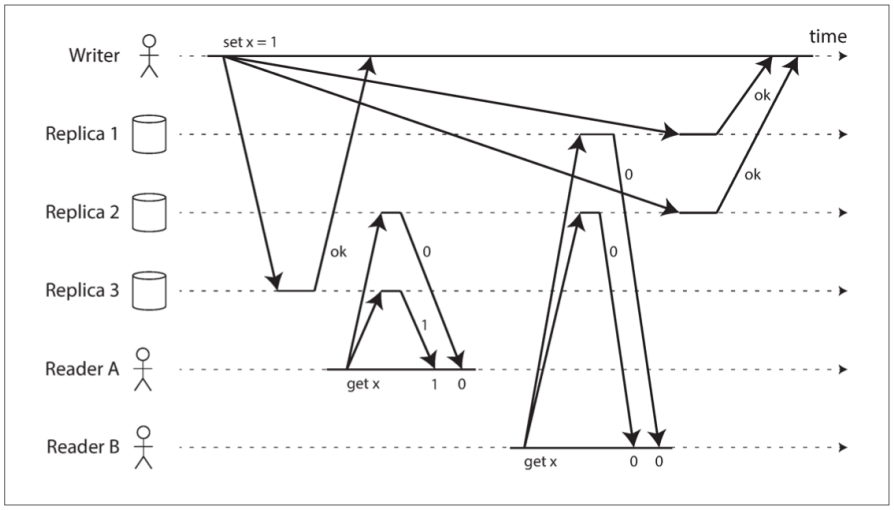


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

  - one fix for this is to enforce read repair synchronously
- Consensus algorithms (linearizable)

- Cost of linearizability
  - will definitely hurt system availability: for multi data center setup, network interrupt between data center will make the linearizable system unavailable. But with multi-leader replication, single data center can still precess request.
  - The CAP theorem:
    - Original definition: consistency, availability, network partition tolerance. Pick 2.
    - but since network partition is inevitable, now it mainly means: trade-off between consistency and availability when there is network partition.
    - There are many other trade-off scenarios , CAP theorem only discuss on one of them. Like performance-consistency.
      - e.g. In multi-core CPU, RAM is shared by all the cores, and each core has its own cache which also holds one copy of the data, kind of like a replicated storage. Memory access will go through cache first by default to get better performance, but may have consistency issue.
  - Linearizability is slow. It's true all the time not only during network fault. Response time is at least proportional to the uncertainty of delays in network. Faster algorithm of linearizability doesn't exist, but weaker consistency model can be much faster.

ORDERING GUARANTEES

Similar to serializability, main benefit of linearizability is the ability to have a well defined order for all operations on distributed system. We can explore some other consistency level with weaker guarantee on ordering.

- Causal consistency
  - Instead of maintaining a total order, only keep the order when there is causal dependency between operations. Other operations are still concurrent which cause timeline branches and need to be merged.
  - causal consistency is the strongest consistency model that doesn't slow down due to network delays and remains availability when network failures.

    - implementation described in replication chapter.
- Sequence number ordering
    - Tracking all causal dependencies can be infeasible, we can use a sequence number/timestamp to order all events.
    - If A causally happened before B, then sequence number of A must be smaller than B. Concurrent events may be ordered arbitrarily.
    - e.g. for single-leader replication, the replication log on leader node defines the total order, followers will apply operations in the same order.
    - Implementation:
        - simply generate a timestamp for each event will not work since it's usually generated on different nodes or client machines. And their clocks are not synchronized. So is not consistent with causality.
        - Lamport timestamp:
            - each node has a unique node id, and keeps a counter of the number of operations it has processed.
            - then the Lamport timestamp is (counter, node id) pair. to order two events, first compare counter value, then compare node id when counter value are the same.
            - Every node and client keeps track of the max counter value it has seen, and send this max value along with each request/response as well. When a node receives a request/response with max value larger than its local one, immediately increases its own counter to this max value. Therefore each causal dependency results in an increased timestamp.
            - causal dependency is mainly maintained by the counter value and tracked max value, node id is just to have a way to order concurrent events.
            - Lamport timestamp vs version vector: version vector can distinguish whether two operations are concurrent or causally dependent. Lamport timestamp is to enforce a total ordering while preserving causality, cannot tell whether it's concurrent or causal dependent, but is more compact compared with version vector.
- Total order broadcast/atomic broadcast
    - Lamport timestamp is able to maintain total order. But it's actually not quite useful, since you can only get this total order through god's eye view. When a request comes to a node, it cannot tell whether another node is concurrently operating on the same data, unless check with every other node to see what it's doing. Total order only emerges after you have collected information from all nodes.
    - Therefore we need to broadcast this total ordering to make it useful. Need to let all nodes agree on this total ordering. e.g. for single leader replication, write request order can be broadcasted by leader node, but leader election still need a total order broadcast.
    - Usually described as a protocol for exchanging messages:
        - Reliable delivery: no msg lost. If a msg is delivered to one node, it's delivered to all nodes.
        - Totally order delivery: msg is delivered to all nodes in same order.
    - When there is network fault, msg may not be able to delivered to some nodes, but the protocol/algorithm will keep retrying.
    - Usages:
        - db replication: total order broadcast is exactly what we need for replication. each messgae represent a write, and every replica process them in same order, then they will remain consistent with each other. This principle is known as state machine replication.
        - Serializable transaction
        - total order broadcast can be viewed as a way to create a log(distributed): replication log, transaction log, WAL. Deliver a msg is like append a new record to the log.

        - Lock: request to acquire a lock will append a record to log.
    - It can be proved that linearizable compare-and-set register is equivalent to total order broadcasting:
        - Implementing linearizable storage with total order broadcasting:
            - ...
        - Implementing total order broadcasting with linearizable storage:
            - ...

CONSENSUS

Get several nodes to agree on something. Or more formally: one or more nodes may propose values, and the consensus algorithm decides on one of those values. It can also be proved that total order broadcasting is equivalent to consensus: making several rounds of decisions <=> total order broadcasting.

First talk about one of the implementation of consensus in distributed transaction:

Atomic commit

We talked about atomicity in transaction. It basically means abortability. but a transaction may succeed on some nodes but fail on others. We need to get all nodes to agree on the outcome, either all commit or all abort. This is called atomic commit .

- For single-node system: storage engine will first append a record with new value to disk, and then append a commit record to mark as success. Before the commit record, operation can be aborted due to crash.
- For multi-node system: cannot simply send commit message to all nodes, since the commit request may succeed in some node and fail in others.
    - Two-phase commit(2PC):
        - requires a new component called coordinator/transaction mgr
        - A 2PC transaction begins with the application reading and writing data on multiple database nodes, as normal. We call these database nodes *participants* in the transaction. When the application is ready to commit, the coordinator begins phase 1: it sends a *prepare* request to each of the nodes, asking them whether they are able to commit. The coordinator then tracks the responses from the participants:

            - If all participants reply "yes," indicating they are ready to commit, then the coordinator sends out a *commit* request in phase 2, and the commit actually takes place.
            - If any of the participants replies "no," the coordinator sends an *abort* request to all nodes in phase 2.

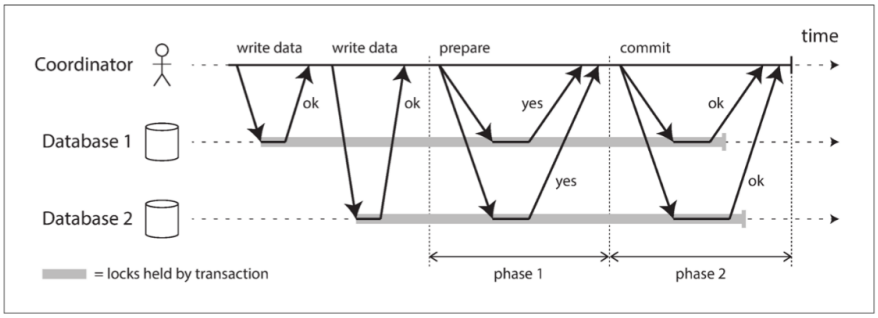


Figure 9-9. A successful execution of two-phase commit (2PC).

        - When a participant receives a prepare request from coordinator, it need to check to make sure it can definitely commit the transaction before replying YES msg. This includes writing write value to disk, check conflict and constraints. By replying YES, it surrenders the right to abort the transaction to coordinator, but without actually committing it.

        - After getting back from all prepare requests, coordinator will either decide to abort or write commit record to its own transaction log. This is called commit point. Once done, there is no way back. it will send abort/commit msg to all nodes and retry forever until it succeeds.
        - If coordinator crashed before sending the prepare request, participant can safely abort transaction. But if crashes after participant reply YES, participant need to wait for coordinator to recover and let it decide.
        - 2PC has a mixed reputation, it can provide safety guarantee but can cause many operational issues and kill performance since it has addition communication overhead and force disk writing.
        - But most importantly, 2PC is not fault tolerant, any node failure(coordinator or participant) will block all transactions.

Fault-tolerant consensus
Must satisfy following properties:

- uniform agreement: no two nodes decide differently
- integrity: no node decides twice. once decided cannot change mind
- validity: if a node decide value v, then v was proposed by some node. to rule out trivial solutions like always decide on null.
- termination: every node that doesn't crash eventually decides some value. For fault tolerant, if one node fails, other nodes should still be able to make decision, don't need to wait for recovery. But still need to have majority of the nodes stay alive to be able to form a quorum.

Best-known fault-tolerant consensus algorithms are Viewstamped Replication(VSR), Paxos, Raft and Zab. Most of them directly implement total order broadcasting to decide on a sequence of values.

- high level idea:
    - using single-leader pattern to achieve consensus is not appropriate since leader election itself requires consensus.
    - instead most algorithms allow multiple leaders with an attached epoch/ballot/view/term number, and guarantee within each epoch the leader is unique.
    - every time a leader is considered to be dead, need to elect new leader with an incremented epoch number, so epoch number are ordered and monotonically increasing.
    - when there is conflict between two leaders, one with higher epoch number wins.
    - for every decision that a leader wants to make, it must send the proposed value to other nodes and wait for a quorum of nodes to response.
    - we have two rounds of voting: once to choose a leader, the second time to vote on a leader's proposal. key idea is quorum of these two votes must overlap.
    - this looks very similar to 2PC, main difference is coordinator is not auto-elected in 2PC, also it requires all nodes to vote on the decision instead of just a majority of them, which is not fault tolerant.
- limitations:
    - voting on proposal is kind of synchronous replication which is slow.
    - usually assume a fixed set of nodes that participate the voting, can't easily add/remove nodes.
    - relying on timeouts to detect failed node, so very sensitive to network issues. can be flaky when there is network interruptions and results in lots of unnecessary leader elections.
- Usages: Zookeeper or etcd are built on top of fault tolerant consensus, they are often described as coordination/configuration services or distributed key-value stores. the managed data should be slow changing(on timescale of minutes or hours) not intended for storing some runtime state which may change thousands/millions of times per second. Their APIs look very similar to a database, like read and write based on key. But they are designed to only hold small amounts of data that can fit entirely in memory(although will write to disk for durability). Also So you

won't directly store application's data here, but instead relying on it indirectly through other projects to achieve lots of things:

- Linearizable atomic operations: use atomic compare-and-set to implement a distributed lock system/
- Total ordering of operations: generate monotonically increasing version number/transaction id.
- Node failure decision
- Change notifications
- Allocation work to nodes: for partitioned system, the partition mapping can be stored and configed in coordination system
- service discovery

[TODO] Paper for Raft, Paxos:

- https://raft.github.io/
- https://medium.com/coccoc-engineering-blog/raft-consensus-algorithms-b48bb88afb17

Chapter 10: Batch Processing

Three ways of building systems:

- Services(online system)
  - request->response
  - response time is usually the primary measure of performance
  - availability is often very important
- Batch processing systems (offline system)
  - input->output
  - throughput is the primary measure of perf
- Stream processing system (near-real-time system)
  - between online and offline processing

DISTRIBUTED FILESYSTEM

Batch processing with Unix Tools

- make each program do one thing well. For a new job, build a new program rather than complicate old program by adding new features
- having a uniform interface for each program's input/output(in Unix is file, file descriptor), and be able to chain multiple programs together.

MapReduce and distributed filesystems
while unix use stdin and stdout as input and output, MapReduce use a distributed filesystem(e.g. HDFS, an open source implementation of GFS).

- based on shared-nothing principle. so no special hardware requirements
- scaled very well. current biggest HDFS deployment run on tens of thousands of machines with storage capacity of hundreds of peta-bytes

MAPREDUCE JOB EXECUTION

MapReduce is a programming framework that can parallelize a computation across many machines without you having to write code to explicitly handle the parallelism. User only need to define two callback functions:

- mapper: called once for each input record. its job is to extract the key and value from each input independently. stateless
- reducer: iterate over mapper outputs with same key, produce result. MR use has of the key to determine which reducer

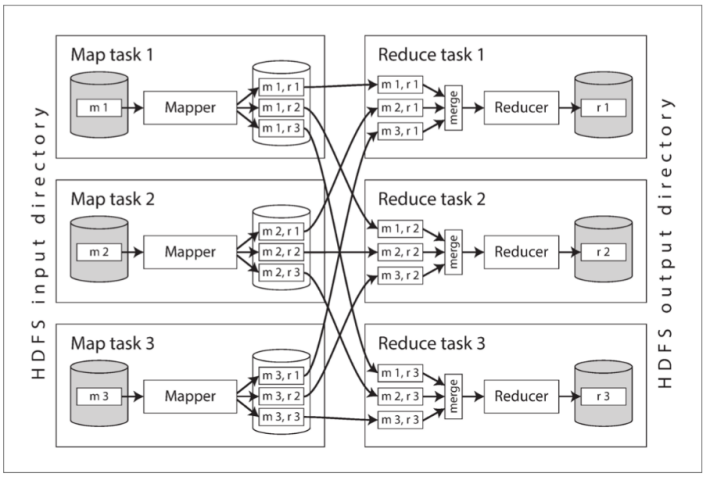


Figure 10-1. A MapReduce job with three mappers and three reducers.

- MR is a framework, can be implemented by standard Unix tools or any languages. In Hadoop it's Java.
- Input to a job is typically a directory in HDFS, and each file within it is considered to be a separate partition that can be processed by a mapper task.
- Each input file is typically hundreds of megabytes in size. The scheduler tries to run each mapper on one of the machines that stores a replica of the input file also with enough spare RAM and CPU resources to increase locality and reduce IO.
    - distributed file system and other processing systems are deployed to same machines
- ? mapper output file must be sorted
- shuffle: the process of partitioning by reducer, sorting, and copying data partitions from mapper to reducer.
- can chain multiple MR jobs into a workflow. chaining is done implicitly by directory names.

JOINS

https://www.dropbox.com/s/n8uzk95xsba7x0f/Spark-201%20Aug-2021.pdf?dl=0

- broadcast hash join
- shuffle hash join
- sort merge join

Reduce-side joins

- mapper takes the role of preparing reducer input data: extracting key,value pair, partition by reducer, and sort by key. No

need to make any assumption about the input data.

- output is partitioned and sorted by join key
- Downside is all the sorting, copying merging can be quite expensive.

Map-side joins

- when can make assumption on input data, we can cut-down the reducer job and join on mapper to avoid shuffle.
- output is partitioned and sorted as the way of the input
- broadcast hash join can be performed on map-side
- shuffle hash join and sort merge join can be performed on map-side when both sides are partitioned by the join key with the same way(bucketing)

Handling skew
Some partitions are much larger than others due to hot key. Can be detected by a separate sampling job before join or specified by user explicitly.

- One way to handle is to break down them into smaller partitions to be handled by multiple reducers, and replicate the other side table input to them.
- When specified by user, can store hot partitions in separate files and join on map side.

OUTPUT OF BATCH WORKFLOWS

- treat input as immutable and avoid side effect. only output new data instead of modifying old data, can roll back to previous version
- also can easily retry the flow or a single job
- same file can be used as input for multiple jobs

COMPARING HADOOP TO DISTRIBUTED DATABASES

Parallel execution can also be done in massively parallel processing(MPP) databases, where the queries are executed on a cluster of machines within the db. It's more focused on SQL queries, while MR + distributed filesystem are more like a general-purpose operating system that can run arbitrary programs.

- diversity of storage:
    - db requires to use structured data according to the data model, whereas distributed filesystem allows you use arbitrary data and just encode into byte sequences.
    - db requires careful up-front data modeling and query patterns before importing the data which slows down the centralized data collection, whereas Hadoop allows to dump data quickly with arbitrary format and figure out how to process it later, making data available quickly. sushi principle: raw data is better.
    - thus Hadoop has often been used for ETL process. data from transaction is dumped into distributed filesystem in raw format. then use MR flow to clean up and transform into a relational format and import into MPP data warehouse for analytic purpose. decouple the data collection and data modeling.
- diversity of processing models
    - MPP system is tightly integrated with certain database itself, including storage layout on disk, query planning, scheduling and execution, is also tuned and optimized for specific needs of the db. so it can achieve very good performance on certain designed query types.
    - but not all processes can be expressed as SQL, like ML pipelines, recommendation systems...

    - MR is more flexible which allow people to easily run their own code over large data. SQL query is only one of the usage(Hive).
    - Data on distributed filesystem can be shared by different flows . so no need to move/import data to different systems.
- fault tolerance
    - MPP db usually keep as much data as possible in memory for good performance. needs to abort the entire query and retry if one node is down(e.g. Presto). while MR eager to write data to disk, so can only retry the failed job along(also for handle large data which cannot fit into memory).
    - so MR is more appropriate for larger jobs that process more data and need to run for a long time and run on unreliable multi-tenant system, which large likely will experience some failure along the way.
    - MR doesn't guarantee performance.

BEYOND MAPREDUCE

MR will fully materialize all intermediate states. this can decouple jobs and easily retry. This is expensive and sometimes not necessary. For most cases the intermediate state is only used by one flow, won't be shared by others, so decoupling each job in a flow doesn't help too much.

Therefore Spark, Flink,... are developed, known as dataflow engine which handle an entire workflow as one job, rather than breaking it up into independent subjobs. subjobs can be assembled in more flexible ways. no clear boundary between mapper and reducer, but all called operator. and more flexible to connect two operators, can do as regular shuffle to repartition and sort the data, or only repartition without sort, or broadcast.

- avoid unnecessary sort between every map reduce stage
- remove unnecessary map jobs
- scheduler have global view of the whole flow, can make locality optimizations, like put task that consumes some data on the same machine as the task that produces it. so can just stream the data through a shared memory buffer instead of copy over network.
- intermediate state can be kept in memory or local disk instead of on distributed filesystem where the data need to be replicated to multiple machines.
- since each job is decoupled in MR, so it need to wait for all its upstream jobs finished. but in workflow engine, each job can start execution once its own input is ready.
- for fault tolerance, if one node is lost, workflow engine will recompute that node from the latest available data. therefore it needs to track how each node's data is computed on partition level. e.g. RDD(resilient distributed dataset) in Spark, Flink will checkpoints operator state to allow resume running.
- only apply to intermediate state, final output will still go to HDFS for durable storage.
- workflow engine usually allow user to use high level declarative APIs like SQL to author pipelines, instead of writing the actual function code like MR. this allows query optimizer to optimize the physical query plan, like take advantage of the column oriented storage to only read required columns, or use vectorized execution. By incorporating declarative APIs and having query optimizer, batch processing frameworks begin to look more like MPP databases that can achieve comparable performance where having the flexibility of MR.

Chapter 11: Stream Processing

Input for batch processing is bounded with a finite size, but in reality lots of data is unbounded since it arrives gradually over

time, and it never complete.

MESSAGING SYSTEM

input of streaming system is an event stream, common approach is all the consumers of the event will subscribed to the event producer, and when new event comes, the producer will notify consumers using a messaging system.

Basic problem for messaging system:

- what happens if the producer send message faster than the consumer can process?
    - drop message?
    - save into buffer? what is buffer is full? write to disk?
    - backpressure/flow control: block producer
- what happens if nodes crash? will there be message lost?

Direct messaging
e.g. using UDP multicast or other protocols.

- low latency
- unreliable. need to handle message lost on application level
- tmp offline consumer will lost all data during down time.

Message brokers/queue
durability issue is moved to the broker, and consumers are asynchronous.

message broker is kind of like a database optimized for message stream:

- db usually keep data until it's explicitly deleted whereas message broker auto delete a msg when it's successfully delivered to all consumers, not for long term storage.
- most msg broker assume the data is fairly small since old data will be deleted quickly, and the queue is not for large scale data
- db can support secondary index and search through data, msg broker can only support subscribe to some certain subscription pattern

Log-based broker
Instead of holding msg in a queue and delete quickly. we can implement broker using append-only log. New msg is append to the log with an offset/sequence number, and broker notify consumer with the offset of the event in the log. Consumer can read the event value by tailing the log file.

- pros:
    - durable storage
    - log file can be partitioned to scale to higher throughput + replication for fault tolerance
    - easily fan out to multiple consumers compared with traditional messaging broker who needs to send independent msg to each consumer.
    - msg deliver order within one partition is guaranteed. traditional broker cannot due to pkg lost and retry.
    - can recover lost msg after node failure based on historical offset
    - can replay historical data for backfill
- cons:

    - consumer can only subscribe based on partition and max parallelism to process the input event will be the number of partitions, whereas traditional broker allows more flexible fine grained patterns and allow more consumers to process in parallel.
    -
- for a 6TB disk with 150MB/s throughput input stream, the disk will be full after 11hours, and have to start overwriting old data, usually real data has lower average throughput so the buffer can hold for days or even weeks.

DATABASES AND STREAMS

Each transaction in db can also be considered as an event. Single-leader replication is kind of like the producer-consumer pattern, new event first write to leader node(replication log is like the broker log), and further send to all followers.

Instead of replicating data within a db, streaming process is more like syncing data within multiple dbs. Thinking the ETL process to sync data from OLTP to OLAP, usually it use a batch processing to take a full copy of the data and bulk loading to warehouse. But if we want to better freshness, we may need stream processing.

One simple way is when new event comes, we write it twice, one to OLTP, the other to warehouse. but since different dbs are independent and have their own leader nodes, have their own replication logic and log files. this will have lots of inconsistency and fault tolerance issue between two dbs when there is delay or node failure in the system. It's hard to keep all the dbs in sync.

Therefore people proposed change data capture(CDC), which is the process of observing all data changes written to a db and extracting them in a form that can be replicated to other dbs. Basically exposed the internal replication log to other dbs as well, stream the changes to allow them apply in same order, make one db as leader, and others as followers. (a mechanism for ensuring all changes made to the system of record are also reflected in the derived data system with accurate copy of the data). example: FB's Wormhole.

Usually CDC also support take a snapshot of the DB to a known position offset, so can reconstruct the state from it without replaying whole historical data. Also same for log compaction.

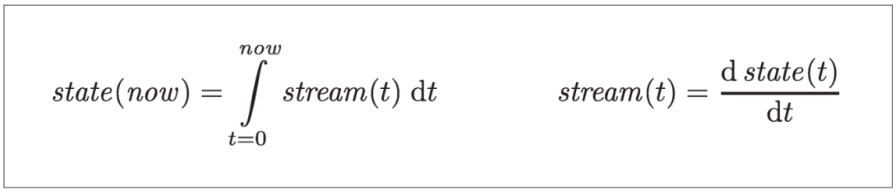
EVENT SOURCING

Similar to CDC, event sourcing is a technique developed in domain-driven design(DDD), but on a different level of abstraction:

- in CDC, db is mutable, user can modify or delete a record, and CDC can also perform log compaction. the log change is at low level of the db state.
- in event sourcing, the log change is at application level. we treat all event as immutable, don't care the event is to modify or delete a record, we just treat it as an event from user, and log all user actions. this full history of the raw events will contain more information than the latest value/state in db.

STATE, STREAMS, IMMUTABILITY

We usually think db as storing the current state of the application, so it supports update, delete operation. this is optimized for reads. And the append only log stores all the raw event in immutable way. mutable state of the db is the accumulation result of all historical immutable events.



$$state(now) = \int_{t=0}^{now} stream(t)\ \mathrm{d}t \qquad stream(t) = \frac{\mathrm{d}\ state(t)}{\mathrm{d}t}$$

Figure 11-6. The relationship between the current application state and an event stream.

Log compaction can be viewed as the integral operation.

Immutability has many advantages:

- contains user history which can reveal more information than latest state
- durable storage, easy to recover and replay
- in batch processing, input files are treated as immutable, which allows us to retry each jobs and share across multiple flows
- similar for streaming, allow to derive multiple different views from the same log of events
- writing the raw event and query from the derived data, allows to separate the form in which data is written from the form it is read. can transform from write-optimized event log to read-optimized app state. easier data modeling than only have one db.

PROCESSING STREAMS

Besides just passing the msg, the system can also process the input streams and produce transformed data. Similar to the MR flows, partitioning and parallelization can also be applied here as well. But main difference is data is unbounded, so operations like sorting doesn't make sense here, rerunning a job from scratch is also not straight forward since the the state may be accumulated from all the events in past few years.

- Reasoning about times: ...
- Aggregation window: tumbling window, hopping window, sliding window, session window
- Stream Joins
    - stream-stream join
        - e.g. join user impression with click on the same session ID to calculate click through rate
        - since both sources are unbounded, cannot wait forever for the join, so need to define a time window for the join: join a click with impressions in the past hour.
        - so the processor needs to maintain a state: e.g. all events happened in last hour, indexed by session ID. so every time a new event(impression or click) comes, join with the stored events.
    - stream-table join
        - enriching the activity events with the information from the db.
        - if the db is large may need to query in the db for every event. if it's small enough, can just load it into processor, like the map-side broadcast join in batch.
        - unlike batch, streaming job is long-running, so the state in db is changing over time. so will make sure the processor's local copy is up to date. can be solved by using CDC and subscribe to the change log of the db.
    - table-table join
        - joining two tables in batch is just one time calculation, but in streaming, both tables are keep changing by its own event streams. so need to maintain the result and get latest value when new events comes.

    - not joining on single event, but on the accumulated state and maintaining a materialized view.
    - e.g. stream 1: user send new posts, stream 2: user follows new people. To get the cache of latest feed for a user, we need to maintain a materialized view which is the result of joining all latest user new posts with user's latest following list.
    - need to maintain state for both streams in processor.

FAULT TOLERANCE

streaming input is unbounded so cannot simply retry as batch.

- micro-batching: e.g. Spark Streaming, implicitly implementing a tumbling window as the batch size.
- checkpointing: e.g. Flink, periodically write checkpoint to durable storage so failed job can restart from it.

Chapter 12: Future of Data System

...