Table Of Contents

Chapter 1 Reliable, Scalable, and Maintainable Applications

Reliability

- Simply means Continuing to work correctly, even when things go wrong.
- A fault is usually defined as one component of the system deviating from its spec, whereas a failure is when the system as a whole stops providing the required service to the user. It is impossible to reduce the probability of a fault to zero; therefore it is usually best to design fault-tolerance mechanisms that prevent faults from causing failures.
- Although we generally prefer tolerating faults over preventing faults, there are cases where prevention is better than cure (e.g., because no cure exists). This is the case with security matters

Hardware Faults

Software Faults :

- The bugs that cause these kinds of software faults often lie dormant for a long time until they are triggered by an unusual set of circumstances. In those circumstances, it is revealed that the software is making some kind of assumption about its environment—and while that assumption is usually true, it eventually stops being true for some reason.

Human Errors :

To make systems reliable in spite of unreliable humans :

- Design systems in a way that minimizes opportunities for error.
- Decouple the places where people make the most mistakes from the places where they can cause failures.
- Test thoroughly at all levels.
- Allow quick and easy recovery.
- Set up detailed and clear monitoring.

Scalability

- Scalability is the term we use to describe a system's ability to cope with increased load.
- However it is important to note that it's not a one-dimensional label.
- Scalability means considering questions like "If the system grows in a particular way, what are our options for coping with the growth?

Describing Load

- Load can be described with the help of load parameters.
- The best choice of parameters depends on the architecture of your system: it may be requests per second to a web server, the ratio of reads to writes in a database, the number of simultaneously active users in a chat room, the hit rate on a cache, or something else.

- Eg : Twitter's scaling challenge is not primarily due to tweet volume, but due to fan-out—each user follows many people, and each user is followed by many people.

Describing Performance

- Once you have described the load on your system ,you can investigate what happens when the load increases :
    - When you increase a load parameter and keep the system resources (CPU, memory, network bandwidth, etc.) unchanged, how is the performance of your system affected?
    - When you increase a load parameter, how much do you need to increase the resources if you want to keep performance unchanged?
- Latency and response time : Latency and response time are often used synonymously, but they are not the same. The response time is what the client sees: besides the actual time to process the request (the service time), it includes network delays and queueing delays. Latency is the duration that a request is waiting to be handled—during which it is latent, awaiting service.
- In practice, in a system handling a variety of requests, the response time can vary a lot. We therefore need to think of response time not as a single number, but as a distribution of values that you can measure.
- The mean is not a very good metric if you want to know your "typical" response time, because it doesn't tell you how many users actually experienced that delay.
- Usually it is better to use percentiles.Eg : median(p50 or 50th percentile) response time is 200 ms, that means half your requests return in less than 200 ms, and half your requests take longer than that.
- High percentiles of response times, also known as tail latencies, are important because they directly affect users' experience of the service.
- Reducing response times at very high percentiles is difficult because they are easily affected by random events outside of your control, and the benefits are diminishing.
- It takes just one slow call to make the entire end-user request slow.This is known as tail latency amplification.

Maintainability

Operability

- Making Life Easy for Operation.Good operability means making routine tasks easy, allowing the operations team to focus their efforts on high-value activities.

Simplicity

- Managing Complexity.Reducing complexity greatly improves the maintainability of software, and thus simplicity should be a key goal for the system we build.
- One of the best tools we have for removing accidental complexity is abstraction.

Evolvability

- Making Change Easy.The ease with which you can modify a data system, and adapt it to changing requirements, is closely linked to its simplicity and its abstractions: simple and easy-to-understand systems are usually easier to modify than complex ones.

Chapter 2 Data Models and Query Languages

Several driving forces behind the adoption of NoSQL databases :

- Greater scalability
- Free and open source software
- Specialized query operations
- Dynamic and expressive data model

The Object-Relational Mismatch

- Object-relational mapping (ORM) frameworks like ActiveRecord and Hibernate reduce the amount of boilerplate code required for this translation layer, but they can't completely hide the differences between the two models.
- If you want to fetch a profile in the relational example, you need to either perform multiple queries (query each table by user_id) or perform a messy multi-way join between the users table and its subordinate tables. In the JSON representation, all the relevant information is in one place, and one query is sufficient.
- The one-to-many relationships from the user profile to the user's positions, educational history, and contact information imply a tree structure in the data, and the JSON representation makes this tree structure explicit.
- In document databases, joins are not needed for one-to-many tree structures, and support for joins is often weak.If the database itself does not support joins, you have to emulate a join in application code by making multiple queries to the database.

Relational Versus Document Databases Today

- When it comes to representing many-to-one and many-to-many relationships, relational and document databases are not fundamentally different: in both cases, the related item is referenced by a unique identifier, which is called a foreign key in the relational model and a document reference in the document model.That identifier is resolved at read time by using a join or follow-up queries.
- The main arguments in favor of the document data model are schema flexibility, better performance due to locality, and that for some applications it is closer to the data structures used

by the application. The relational model counters by providing better support for joins, and many-to-one and many-to-many relationships.

Schema flexibility in the document model

- Document databases are sometimes called schemaless, but that's misleading, as the code that reads the data usually assumes some kind of structure—i.e., there is an implicit schema, but it is not enforced by the database.A more accurate term is schema-on-read (the structure of the data is implicit, and only interpreted when the data is read), in contrast with schema-on-write (the traditional approach of relational databases, where the schema is explicit and the database ensures all written data conforms to it)
- Schema-on-read is similar to dynamic (runtime) type checking in programming languages, whereas schema-on-write is similar to static (compile-time) type checking.

Data locality for queries

- If data is split across multiple tables multiple index lookups are required to retrieve it all, which may require more disk seeks and take more time.
- The locality advantage only applies if you need large parts of the document at the same time. The database typically needs to load the entire document, even if you access only a small portion of it, which can be wasteful on large documents. On updates to a document, the entire document usually needs to be rewritten—only modifications that don't change the encoded size of a document can easily be performed in place. For these reasons, it is generally recommended that you keep documents fairly small and avoid writes that increase the size of a document.

Query Languages for Data

- An imperative language tells the computer to perform certain operations in a certain order.You can imagine stepping through the code line by line, evaluating conditions, updating variables, and deciding whether to go around the loop one more time.
- In a declarative query language, like SQL or relational algebra, you just specify the pattern of the data you want—what conditions the results must meet, and how you want the data to be transformed (e.g., sorted, grouped, and aggregated)—but not how to achieve that goal.
- A declarative query language is attractive because it is typically more concise and easier to work with than an imperative API. But more importantly, it also hides implementation details of the database engine.Also Declarative languages have a better chance of getting faster in parallel execution

MapReduce Querying

- MapReduce is neither a declarative query language nor a fully imperative query API, but somewhere in between: the logic of the query is expressed with snippets of code, which are called repeatedly by the processing framework.

- The map and reduce functions are somewhat restricted in what they are allowed to do. They must be pure functions, which means they only use the data that is passed to them as input, they cannot perform additional database queries, and they must not have any side effects.

Graph-Like Data Models

- Graphs are not limited to such homogeneous data: an equally powerful use of graphs is to provide a consistent way of storing completely different types of objects in a single datastore.
- Graphs are good for evolvability: as you add features to your application, a graph can easily be extended to accommodate changes in your application's data structures.

Property Graphs

In the property graph model, each vertex consists of:

- A unique identifier
- A set of outgoing edges
- A set of incoming edges
- A collection of properties (key-value pairs)

Each edge consists of:

- A unique identifier
- The vertex at which the edge starts (the tail vertex)
- The vertex at which the edge ends (the head vertex)
- A label to describe the kind of relationship between the two vertices
- A collection of properties (key-value pairs)

Triple-Stores and SPARQL

- The triple-store model is mostly equivalent to the property graph model, using different words to describe the same ideas.
- In a triple-store, all information is stored in the form of very simple three-part statements: (subject, predicate, object). For example, in the triple (Jim, likes, bananas).
- The subject of a triple is equivalent to a vertex in a graph. The object is one of two things:A value in primitive datatype or another vertex in the graph.

Semantic Web

- The semantic web is fundamentally a simple and reasonable idea: websites already publish information as text and pictures for humans to read, so why don't they also publish information as machine-readable data for computers to read?

Chapter 3 Storage and Retrieval

- There is an important trade-off in storage systems: well-chosen indexes speed up read queries, but every index slows down writes.
- Log is used in the more general sense: an append-only sequence of records.

Hash Indexes

- Keep an in-memory hash map where every key is mapped to a byte offset in the data file—the location at which the value can be found. Whenever you append a new key-value pair to the file, you also update the hash map to reflect the offset of the data you just wrote.
- As described so far, we only ever append to a file—so how do we avoid eventually running out of disk space? A good solution is to break the log into segments of a certain size by closing a segment file when it reaches a certain size, and making subsequent writes to a new segment file. We can then perform compaction on these segments. Compaction means throwing away duplicate keys in the log, and keeping only the most recent update for each key.
- We can also merge several segments together at the same time as performing the compaction.Segments are never modified after they have been written.When the merging and compaction process can be done in background thread and we can serve requests with the help of older segments.
- Each segment has its own in-memory hash table.

Why append only log?

An append only log seems wasteful(why not to update in place?) but an append-only design turns out to be good because :

- Appending and segment merging are sequential write operations, which are generally much faster than random writes, especially on magnetic spinning-disk hard drives.
- Concurrency and crash recovery are much simpler if segment files are append-only or immutable.
- Merging old segments avoids the problem of data files getting fragmented over time.

Limitations of Hash Index :

- The hash table must fit in memory.
- Range queries are not efficient.

SSTables(Sorted String Table) and LSM-Trees

- In SSTable we require that the sequence of key-value pairs is sorted by key.
- SSTables have several big advantages over log segments with hash indexes:
    - Merging segments is simple and efficient.

- In order to find a particular key in the file, you no longer need to keep an index of all the keys in memory.
- Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk (indicated by the shaded area in Figure 3-5).

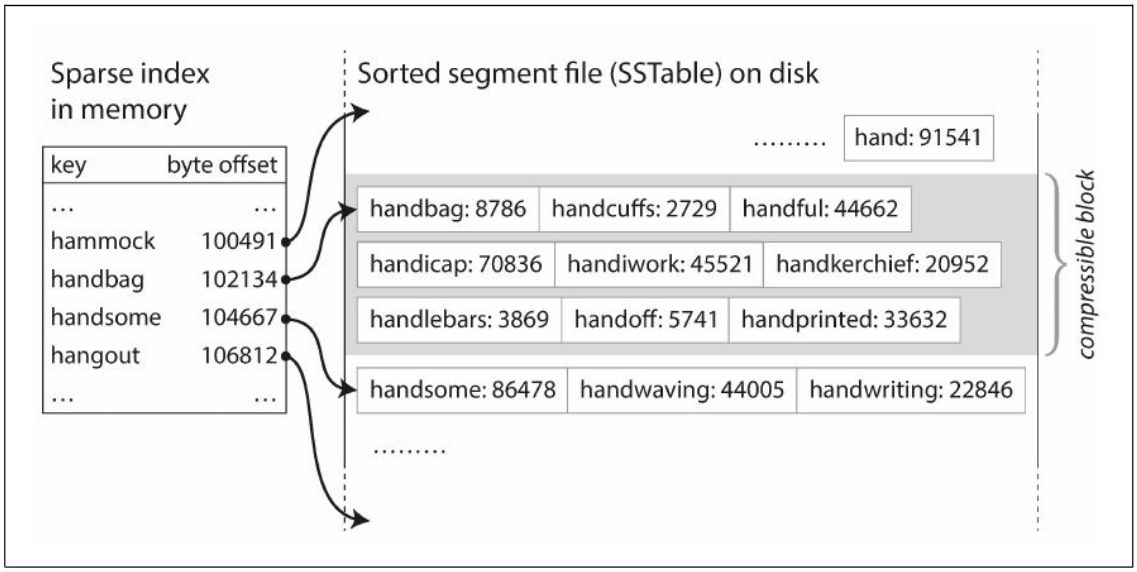


Figure 3-5. An SSTable with an in-memory index.

Constructing and maintaining SSTables

We can now make our storage engine work as follows:

- When a write comes in add it to an in-memory balanced tree data structure(called memtable).
- When the memtable gets bigger than some threshold—typically a few megabytes —write it out to disk as an SSTable file.The new SSTable file becomes the most recent segment of the database.
- In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
- From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

In this,if the database crashes all recent writes(in memtable and not updated in disk) are lost.
In order to avoid that problem, we can keep a separate log on disk to which every write is immediately appended.That log is not in sorted order, but that doesn't matter, because its only purpose is to restore the memtable after a crash. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

Making an LSM-tree out of SSTables

- Storage engines that are based on the principle of merging and compacting sorted files are often called LSM(log-structured merge-tree) storage engines.

- LSM-tree consists of some memory components and some disk components. Basically SSTable is just one implementation of disk component for LSM-tree.

Performance Optimizations

- The LSM-tree algorithm can be slow when looking up keys that do not exist in the database as you have to check the memtable, then the segments all the way back to the oldest. In order to optimize this kind of access, storage engines often use additional Bloom filters.
- Even though there are many subtleties, the basic idea of LSM-trees—keeping a cascade of SSTables that are merged in the background—is simple and effective.

B-Trees

- The log-structured indexes we saw earlier break the database down into variable-size segments, typically several megabytes or more in size, and always write a segment sequentially. By contrast, B-trees break the database down into fixed-size blocks or pages, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time.
- If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, change the value in that page, and write the page back to disk.
- Most databases can fit into a B-tree that is three or four levels deep, so you don't need to follow many page references to find the page you are looking for. (A four-level tree of 4 KB pages with a branching factor of 500 can store up to 256 TB.)

Making B-trees reliable

- In order to make the database resilient to crashes, it is common for B-tree implementations to include an additional data structure on disk: a write-ahead log (WAL, also known as a redo log). This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself.
- An additional complication of updating pages in place is that careful concurrency control is required if multiple threads are going to access the B-tree at the same time —otherwise a thread may see the tree in an inconsistent state. This is typically done by protecting the tree's data structures with latches (lightweight locks).

B-tree optimizations

- Instead of overwriting pages and maintaining a WAL(Write Ahead Log) for crash recovery, some databases (like LMDB) use a copy-on-write scheme. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location.
- We can save space in pages by not storing the entire key, but abbreviating it. Especially in pages on the interior of the tree, keys only need to provide enough information to act as boundaries between key ranges. Packing more keys into a page allows the tree to have a higher branching factor, and thus fewer levels.

Comparing B-Trees and LSM-Trees

- As a rule of thumb, LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads.Reads are typically slower on LSM-trees because they have to check several different data structures and SSTables at different stages of compaction.

Advantages of LSM-trees

- Write-amplification : one write to the database resulting in multiple writes to the disk over the course of the database's lifetime—is known as write amplification. It is of particular concern on SSDs, which can only overwrite blocks a limited number of times before wearing out.
- LSM-trees are typically able to sustain higher write throughput than B-trees, partly because they sometimes have lower write amplification and partly because they sequentially write compact SSTable files rather than having to overwrite several pages in the tree.
- Since LSM-trees are not page-oriented and periodically rewrite SSTables to remove fragmentation, they have lower storage overheads, especially when using leveled compaction.

Downsides of LSM-trees

- A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes.
- If write throughput is high and compaction is not configured carefully, it can happen that compaction cannot keep up with the rate of incoming writes. In this case, the number of unmerged segments on disk keeps growing.
- An advantage of B-trees is that each key exists in exactly one place in the index, whereas a log-structured storage engine may have multiple copies of the same key in different segments. This aspect makes B-trees attractive in databases that want to offer strong transactional semantics.

Other Indexing Structures

Storing values within the index

- The key in an index is the thing that queries search for, but the value can be one of two things: it could be the actual row (document, vertex) in question, or it could be a reference to the row stored elsewhere. In the latter case, the place where rows are stored is known as a heap file, and it stores data in no particular order.
- In some situations, the extra hop from the index to the heap file is too much of a performance penalty for reads, so it can be desirable to store the indexed row directly within an index. This is known as a clustered index. For example, in MySQL's InnoDB storage engine, the primary key of a table is always a clustered index, and secondary indexes refer to the primary key (rather than a heap file location).

- A compromise between a clustered index (storing all row data within the index) and a nonclustered index (storing only references to the data within the index) is known as a covering index or index with included columns, which stores some of a table's columns within the index.

Multi-column indexes

- The most common type of multi-column index is called a concatenated index, which simply combines several fields into one key by appending one column to another
- Multi-dimensional indexes are a more general way of querying several columns at once, which is particularly important for geospatial data.Or in a database of weather observations you could have a two-dimensional index on (date, temperature) in order to efficiently search for all the observations during the year 2013 where the temperature was between 25 and 30°C.

Full-text search and fuzzy indexes

- All the indexes discussed so far assume that you have exact data and allow you to query for exact values of a key, or a range of values of a key with a sort order. What they don't allow you to do is search for similar keys, such as misspelled words.
- Lucene uses a SSTable-like structure for its term dictionary.
- In LevelDB, this in-memory index is a sparse collection of some of the keys, but in Lucene, the in-memory index is a finite state automaton over the characters in the keys, similar to a trie.

Transaction Processing or Analytics?

- A transaction needn't necessarily have ACID (atomicity, consistency, isolation, and durability) properties. Transaction processing just means allowing clients to make low-latency reads and writes— as opposed to batch processing jobs, which only run periodically (for example, once per day).
-

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

- The data warehouse contains a read-only copy of the data in all the various OLTP systems in the company.

- This process of getting data into the warehouse is known as Extract–Transform–Load (ETL).
- The data model of a data warehouse is most commonly relational, because SQL is generally a good fit for analytic queries.

Stars and Snowflakes: Schemas for Analytics

- Many data warehouses are used in a fairly formulaic style, known as a star schema (also known as dimensional modeling).
- At the center of the schema is a so-called fact table.Other columns in the fact table are foreign key references to other tables, called dimension tables.
- The dimensions represent the who, what, where, when, how, and why of the event.
- A variation of this template is known as the snowflake schema, where dimensions are further broken down into subdimensions.
- Snowflake schemas are more normalized than star schemas,but star schemas are often preferred because they are simpler for analysts to work with.

Column-Oriented Storage

- The idea behind column-oriented storage is simple: don't store all the values from one row together, but store all the values from each column together instead.

fact_sales table

date_key	product_sk	store_sk	promotion_sk	customer_sk	quantity	net_price	discount_price
140102	69	4	NULL	NULL	1	13.99	13.99
140102	69	5	19	NULL	3	14.99	9.99
140102	69	5	NULL	191	1	14.99	14.99
140102	74	3	23	202	5	0.99	0.89
140103	31	2	NULL	NULL	1	2.49	2.49
140103	31	3	NULL	NULL	3	14.99	9.99
140103	31	3	21	123	1	49.99	39.99
140103	31	8	NULL	233	1	0.99	0.99

Columnar storage layout:

date_key file contents:	140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents:	69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents:	4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents:	NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents:	NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents:	1, 3, 1, 5, 1, 3, 1, 1
net_price file contents:	13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents:	13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

Figure 3-10. Storing relational data by column, rather than by row.

Column Compression

- Besides only loading those columns from disk that are required for a query, we can further reduce the demands on disk throughput by compressing data. Fortunately, column-oriented storage often lends itself very well to compression(Because of repetitive data).

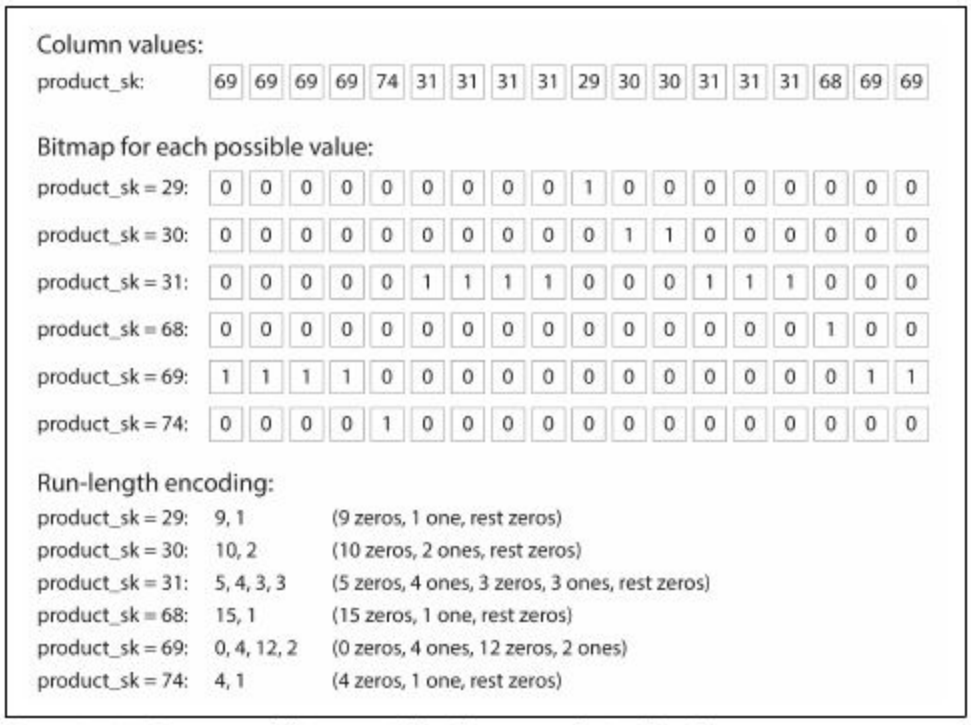


Figure 3-11. Compressed, bitmap-indexed storage of a single column.

Cassandra and HBase have a concept of column families, which they inherited from Bigtable. However, it is very misleading to call them column-oriented: within each column family, they store all columns from a row together, along with a row key, and they do not use column compression. Thus, the Bigtable model is still mostly row-oriented.

Memory bandwidth and vectorized processing

- Besides reducing the volume of data that needs to be loaded from disk, column-oriented storage layouts are also good for making efficient use of CPU cycles.
- Operators, such as the bitwise AND and OR described previously, can be designed to operate on such chunks of compressed column data directly. This technique is known as vectorized processing.

Sort Order in Column Storage

- It wouldn't make sense to sort each column independently, because then we would no longer know which items in the columns belong to the same row.

- The administrator of the database can choose the columns by which the table should be sorted, using their knowledge of common queries(eg : date ranges).
- Having multiple sort orders in a column-oriented store is a bit similar to having multiple secondary indexes in a row-oriented store. But the big difference is that the row-oriented store keeps every row in one place (in the heap file or a clustered index), and secondary indexes just contain pointers to the matching rows. In a column store, there normally aren't any pointers to data elsewhere, only columns containing values.

Writing to Column-Oriented Storage

- Column-oriented storage, compression, and sorting all help to make those read queries faster.However, they have the downside of making writes more difficult.
- All writes first go to an in-memory store,where they are added to a sorted structure and prepared for writing to disk. It doesn't matter whether the in-memory store is row-oriented or column-oriented. When enough writes have accumulated, they are merged with the column files on disk and written to new files in bulk.

Aggregation: Data Cubes and Materialized Views

- Columnar storage can be significantly faster for ad hoc analytical queries, so it is rapidly gaining popularity.
- A materialized view is an actual copy of the query results, written to disk, whereas a virtual view is just a shortcut for writing queries. When you read from a virtual view, the SQL engine expands it into the view's underlying query on the fly and then processes the expanded query.
- In read-heavy data warehouses materialized views can make more sense.

Chapter 4 Encoding and Evolution

- With server-side applications you may want to perform a rolling upgrade (also known as a staged rollout), deploying the new version to a few nodes at a time, checking whether the new version is running smoothly, and gradually working your way through all the nodes. This allows new versions to be deployed without service downtime, and thus encourages more frequent releases and better evolvability.
- With client-side applications you're at the mercy of the user, who may not install the update for some time.
- This means that old and new versions of the code, and old and new data formats, may potentially all coexist in the system at the same time.
- In order for the system to continue running smoothly, we need to maintain compatibility in both directions:
    - Backward compatibility : Newer code can read data that was written by older code.
    - Forward compatibility : Older code can read data that was written by newer code.

- The translation from the in-memory representation to a byte sequence is called encoding (also known as serialization or marshalling), and the reverse is called decoding (parsing,deserialization, unmarshalling).
- The encoding is often tied to a particular programming language, and reading the data in another language is very difficult.
- JSON, XML, and CSV are textual formats, and thus somewhat human-readable.
- There is a lot of ambiguity around the encoding of numbers. In XML and CSV, you cannot distinguish between a number and a string that happens to consist of digits (except by referring to an external schema). JSON distinguishes strings and numbers, but it doesn't distinguish integers and floating-point numbers, and it doesn't specify a precision.
- JSON and XML have good support for Unicode character strings (i.e., human-readable text), but they don't support binary strings (sequences of bytes without a character encoding). Binary strings are a useful feature, so people get around this limitation by encoding the binary data as text using Base64.

Binary Encoding

- For data that is used only internally within your organization, there is less pressure to use a lowest-common-denominator encoding format. For example, you could choose a format that is more compact or faster to parse.
- JSON is less verbose than XML, but both still use a lot of space compared to binary formats. This observation led to the development of a profusion of binary encodings for JSON (MessagePack, BSON, BJSON, UBJSON, BISON).
  Eg JSON record

```
{
    "userName": "Martin",
    "favoriteNumber": 1337,
    "interests": ["daydreaming", "hacking"]
}
```

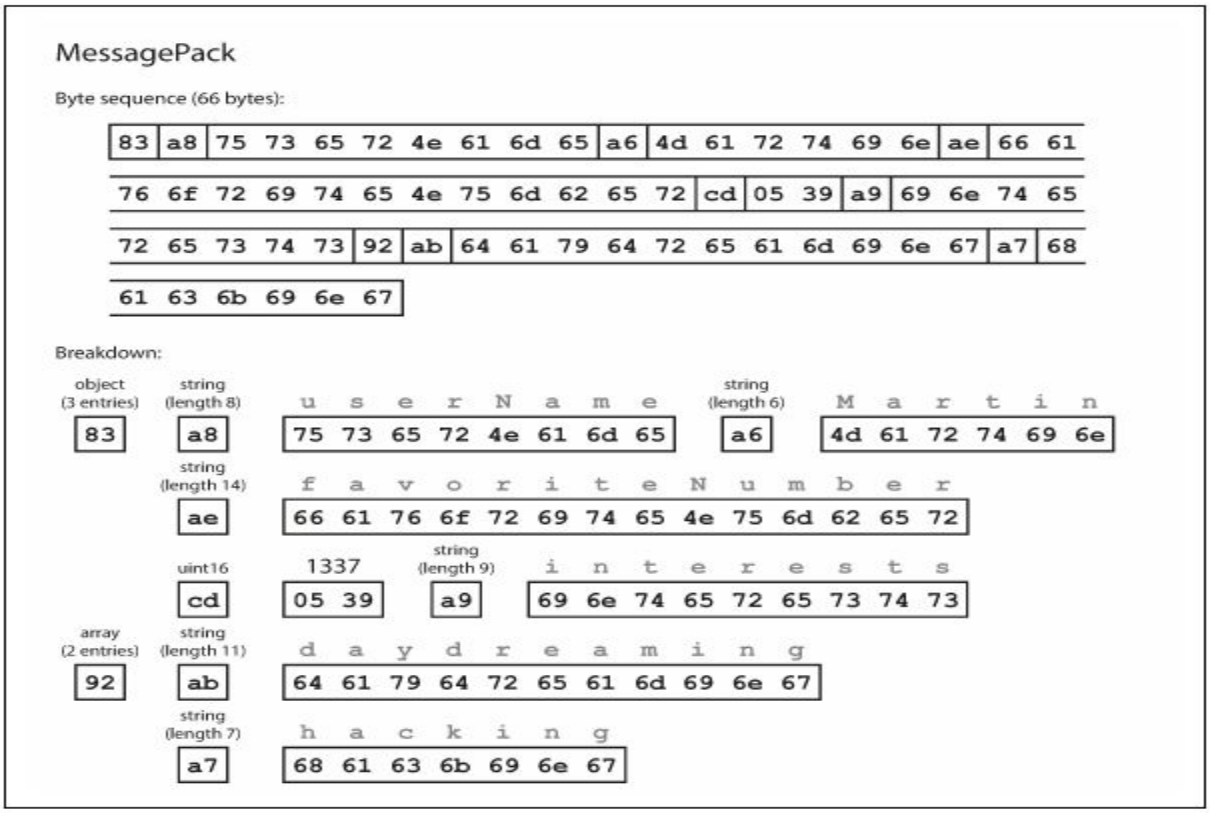


Figure 4-1. Example record (Example 4-1) encoded using MessagePack.

Thrift and Protocol Buffers

- Both Thrift and Protocol Buffers require a schema for any data that is encoded.

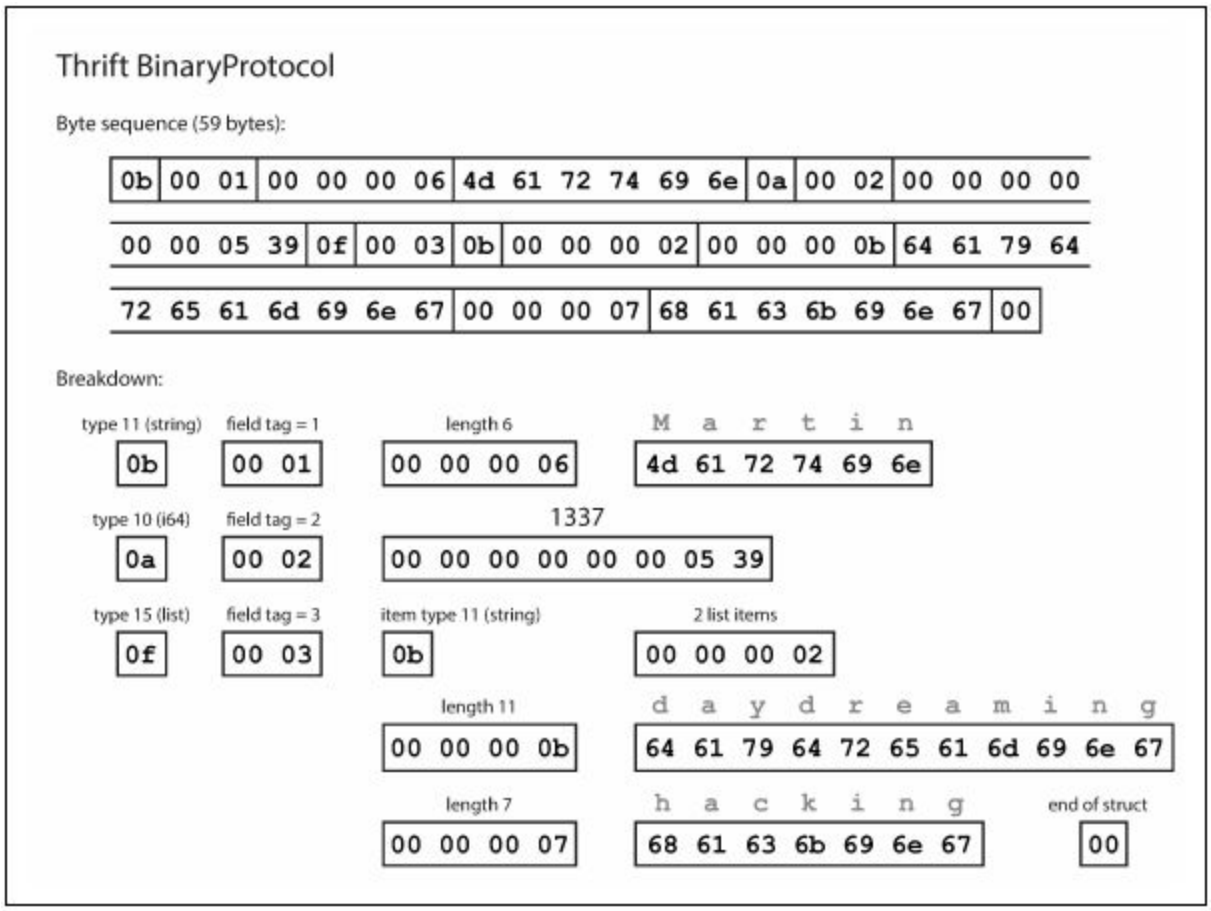


Figure 4-2. Example record encoded using Thrift's BinaryProtocol.

- The big difference compared to Figure 4-1 is that there are no field names (userName, favoriteNumber, interests). Instead, the encoded data contains field tags, which are numbers (1, 2, and 3).
- The Thrift CompactProtocol encoding is semantically equivalent to BinaryProtocol.It packs the same information into only 34 bytes. It does this by packing the field type and tag number into a single byte, and by using variable-length integers.
- You can add new fields to the schema, provided that you give each field a new tag number.If you add a new field, you cannot make it required.

Avro

- Avro also uses a schema to specify the structure of the data being encoded. It has two schema languages: one (Avro IDL) intended for human editing, and one (based on JSON) that is more easily machine-readable.

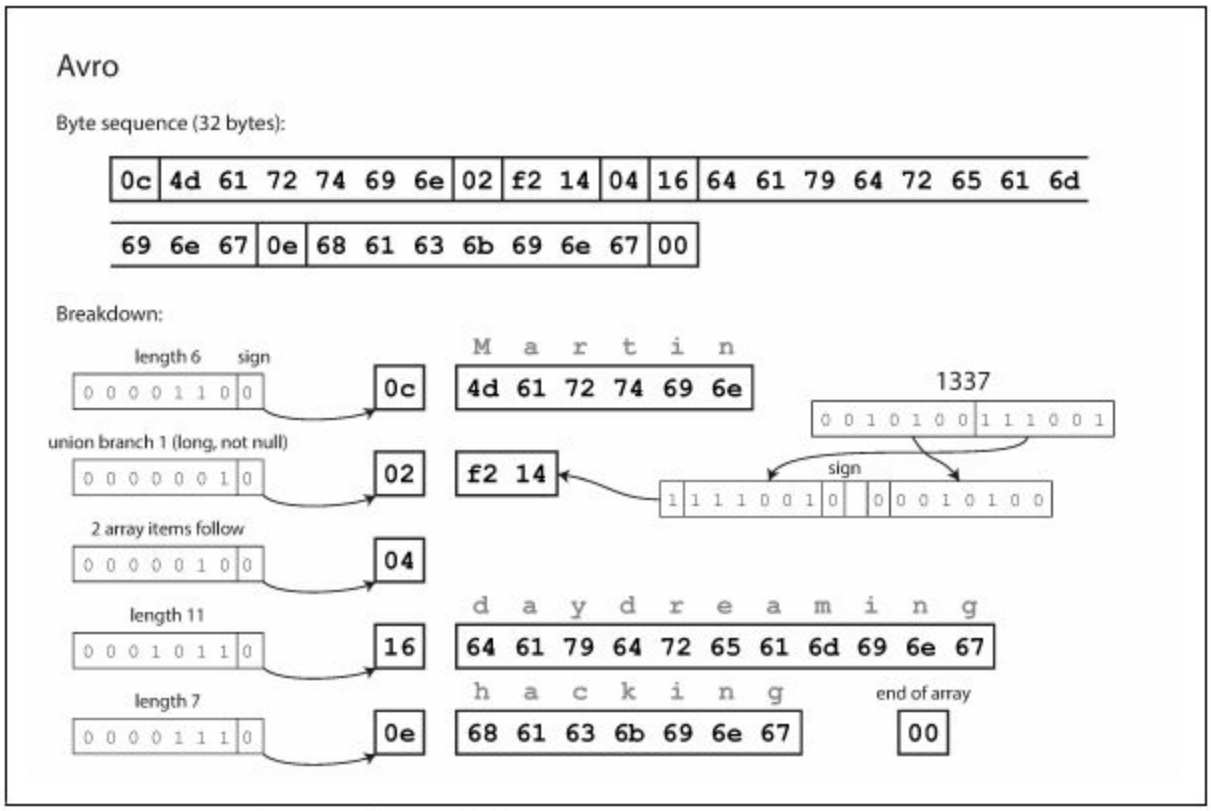


Figure 4-5. Example record encoded using Avro.

- Any mismatch in the schema between the reader and the writer would mean incorrectly decoded data.

The writer's schema and the reader's schema

- With Avro, when an application wants to encode some data it encodes the data using whatever version of the schema it knows about.This is known as the writer's schema.
- When an application wants to decode some data it is expecting the data to be in some schema, which is known as the reader's schema.
- The key idea with Avro is that the writer's schema and the reader's schema don't have to be the same—they only need to be compatible.

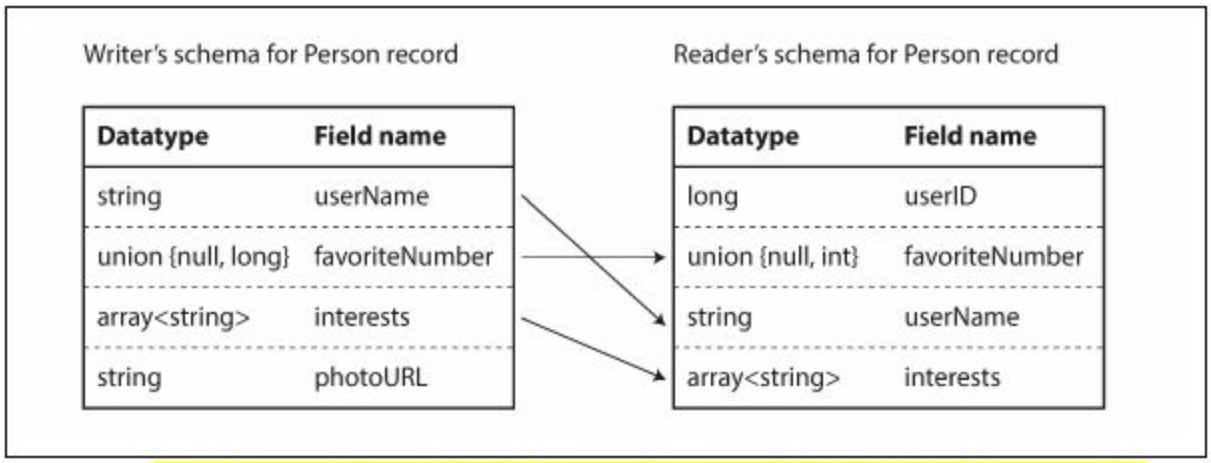


Figure 4-6. An Avro reader resolves differences between the writer's schema and the reader's schema.

- If you want to allow a field to be null, you have to use a union type. For example, union { null, long, string } field; indicates that field can be a number, or a string, or null.

But what is the writer's schema?

- How does the reader know the writer's schema with which a particular piece of data was encoded? We can't just include the entire schema with every record, because the schema would likely be much bigger than the encoded data, making all the space savings from the binary encoding futile.
- It depends on the context in which Avro is being used. Eg :
    - Large file with lots of records(Hadoop) - In this case, the writer of that file can just include the writer's schema once at the beginning of the file.
    - Database with individually written records -
        - In a database,different records are inserted at different times with different writer schemas.
        - The simplest solution is to include a version number at the beginning of every encoded record, and to keep a list of schema versions in your database.
        - A reader can fetch a record, extract the version number, and then fetch the writer's schema for that version number from the database.
    - Sending records over a network connection - They can negotiate the schema version on connection setup and then use that schema for the lifetime of the connection.

Dynamically generated schemas

- Avro is friendlier to dynamically generated schemas because its schema doesn't use tag numbers.

The Merits of Schemas

We can see that although textual data formats such as JSON, XML, and CSV are widespread, binary encodings based on schemas are also a viable option. They have a number of nice properties:

- Compact
- Schema is a valuable form of documentation.
- Keeping a database of schemas allows you to check forward and backward compatibility of schema changes, before anything is deployed.
- Ability to generate code from schema.

Modes of Dataflow

Compatibility is a relationship between one process that encodes the data, and another process that decodes it.

Dataflow Through Databases

- In a database, the process that writes to the database encodes the data, and the process that reads from the database decodes it.
- Backward compatibility is clearly necessary here; otherwise your future self won't be able to decode what you previously wrote.
- This means that a value in the database may be written by a newer version of the code, and subsequently read by an older version of the code that is still running. Thus, forward compatibility is also often required for databases.
- Rewriting (migrating) data into a new schema is certainly possible, but it's an expensive thing to do on a large dataset, so most databases avoid it if possible.

Dataflow Through Services: REST and RPC

- The API exposed by the server is known as a service.A key design goal of a service-oriented/microservices architecture is to make the application easier to change and maintain by making services independently deployable and evolvable.
- We should expect old and new versions of servers and clients to be running at the same time, and so the data encoding used by servers and clients must be compatible across versions of the service API.
- When HTTP is used as the underlying protocol for talking to the service, it is called a web service.There are two popular approaches to web services: REST and SOAP.REST is not a protocol, but rather a design philosophy that builds upon the principles of HTTP. SOAP is an XML-based protocol for making network API requests.

The problems with remote procedure calls (RPCs)

- The RPC model tries to make a request to a remote network service look the same as calling a function or method in your programming language, within the same process.
- A network request is very different from a local function call:
    - A local function call is predictable and either succeeds or fails, depending only on parameters that are under your control. A network request is unpredictable.
    - A local function call can either return a result or an exception or run forever in a loop.Network request has another possible outcome : timeout.
- Every time you call a local function, it normally takes about the same time to execute. A network request is much slower than a function call, and its latency is also wildly variable
- The client and the service may be implemented in different programming languages, so the RPC framework must translate datatypes from one language into another.
- All of these factors mean that there's no point trying to make a remote service look too much like a local object in your programming language, because it's a fundamentally different thing. Part of the appeal of REST is that it doesn't try to hide the fact that it's a network protocol.

Message-Passing Dataflow

- Asynchronous message-passing systems are somewhere between RPC and databases. They are similar to RPC in that a client's request (usually called a message) is delivered to another process with low latency. They are similar to databases in that the message is not sent via a direct network connection, but goes via an intermediary called a message broker (also called a message queue or message-oriented middleware), which stores the message temporarily.
- Using a message broker has several advantages compared to direct RPC:
    - It can act as a buffer.
    - Automatically redeliver messages.
    - Avoids the sender needing to know the IP address and port number of the recipient.
    - Logically decouples the sender from the recipient

Chapter 5 Replication

Replication means keeping a copy of the same data on multiple machines that are connected via a network.

Leader-Based Replication

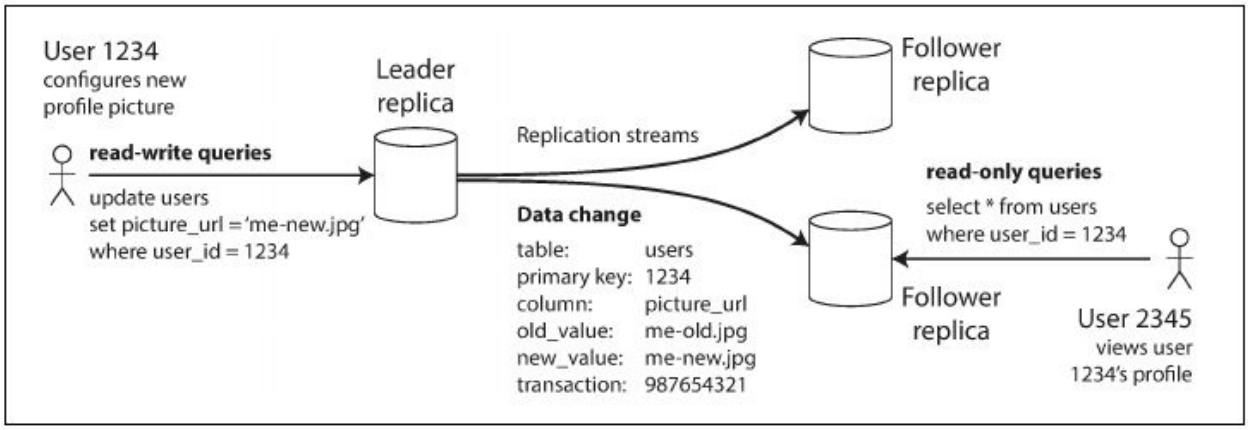


Figure 5-1. Leader-based (master–slave) replication.

Synchronous Versus Asynchronous Replication

- The advantage of synchronous replication is that the follower is guaranteed to have an up-to-date copy of the data that is consistent with the leader. If the leader suddenly fails, we can be sure that the data is still available on the follower. The disadvantage is that if the synchronous follower doesn't respond (because it has crashed, or there is a network fault, or for any other reason), the write cannot be processed.
- For that reason, it is impractical for all followers to be synchronous: any one node outage would cause the whole system to grind to a halt.
- If the synchronous follower becomes unavailable or slow, one of the asynchronous followers is made synchronous.This configuration is sometimes also called semi-synchronous

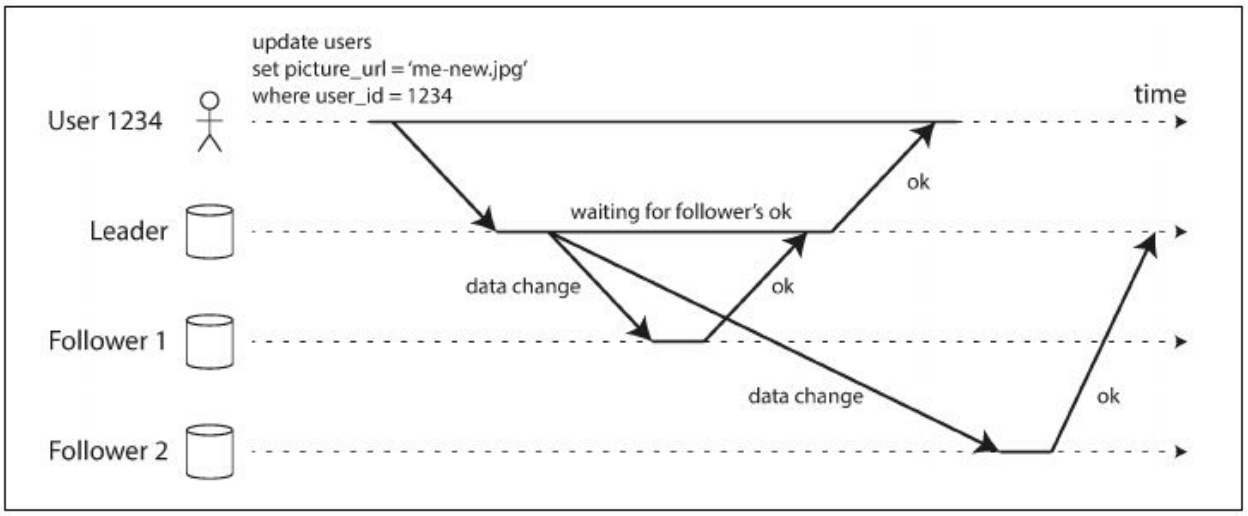


Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.

Setting up new Followers

Conceptually the process is :

1. Take a consistent snapshot of the leader's database at some point in time(if possible without taking a lock on the entire database).
2. Copy the snapshot to the new follower node.
3. The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken.This requires that the snapshot is associated with an exact position in the leader's replication log.
4. When the follower has processed the backlog i.e. it has caught up it is ready.

Handling Node Outages

Follower failure: Catch-up recovery

The follower can connect to the leader and request all the data changes that occurred during the time when the follower was disconnected.

Leader failure: Failover

An automatic failover process usually consists of the following steps:

1. Determining that the leader has failed - If node doesn't respond for a certain period of time then it is assumed to be dead.
2. Choosing a new leader - Getting all the nodes to agree on a new leader is a consensus problem
3. Reconfiguring the system to use the new leader - The system needs to ensure that the old leader becomes a follower and recognizes the new leader.

Failover is fraught with things that can go wrong:

- If asynchronous replication is used, the new leader may not have received all the writes from the old leader before it failed. If the former leader rejoins the cluster after a new leader has been chosen, what should happen to those writes?
- It could happen that two nodes both believe that they are the leader. This situation is called split brain.
- What is the right timeout before the leader is declared dead?

Implementation of Replication Logs

Several different replication methods are used in practice:

Statement-based replication

In the simplest case, the leader logs every write request (statement) that it executes and sends that statement log to its followers.Although this may sound reasonable, there are various ways in which this approach to replication can break down:

- Any statement that calls a nondeterministic function.(To solve this the leader can replace calls to non-deterministic functions with a fixed value).
- The statements must be executed in the same order on each replica or else they may have a different effect.
- Statements can result in different side effects on each replica.

However, because there are so many edge cases, other replication methods are now generally preferred.

Write-ahead log (WAL) shipping

- The log is an append-only sequence of bytes containing all writes to the database.
- The main disadvantage is that the log describes the data on a very low level: a WAL contains details of which bytes were changed in which disk blocks. This makes replication closely coupled to the storage engine. If the database changes its storage format from one version to another, it is typically not possible to run different versions of the database software on the leader and the followers.

Logical (row-based) log replication

- A logical log is decoupled from the storage engine internals, it can more easily be kept backward compatible, allowing the leader and the follower to run different versions of the database software, or even different storage engines.
- A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row.Eg : For a deleted row, the log contains enough information to uniquely identify the row that was deleted.
- A logical log format is also easier for external applications to parse.

Trigger-based replication

- A trigger lets you register custom application code that is automatically executed when a data change (write transaction) occurs in a database system.

- The trigger has the opportunity to log this change into a separate table, from which it can be read by an external process.
- Trigger-based replication typically has greater overheads than other replication methods, and is more prone to bugs and limitations than the database's built-in replication.

Problems with Replication Lag

- In normal operation, the delay between a write happening on the leader and being reflected on a follower—the replication lag—may be only a fraction of a second, and not noticeable in practice. However, if the system is operating near capacity or if there is a problem in the network, the lag can easily increase to several seconds or even minutes.
- The inconsistency is just a temporary state—if you stop writing to the database and wait a while, the followers will eventually catch up and become consistent with the leader. For that reason, this effect is known as eventual consistency.

Examples of problems that are likely to occur when there is considerable replication lag :

Reading Your Own Writes

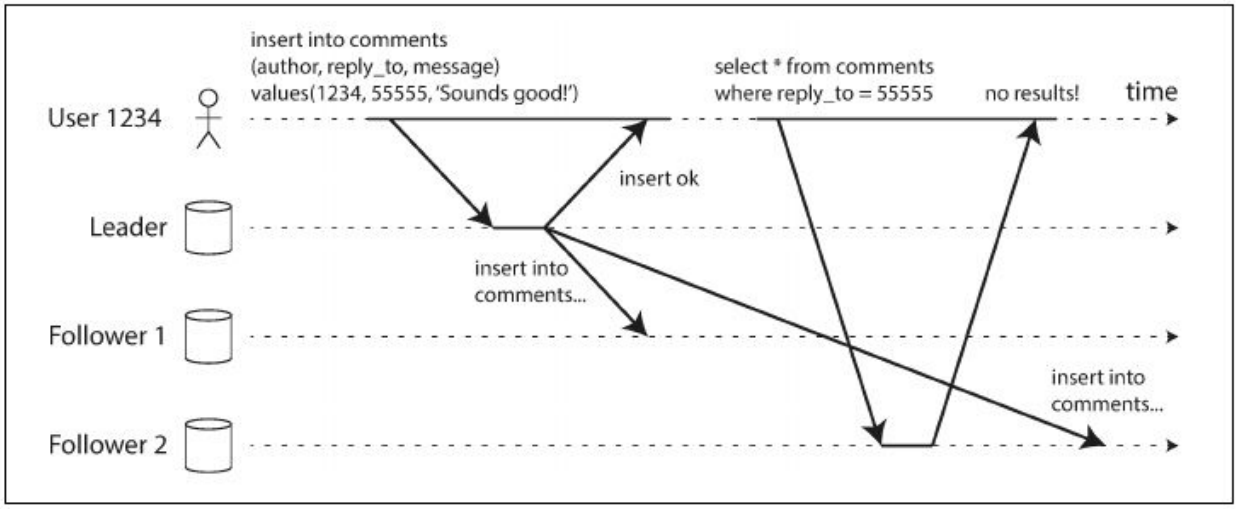


Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.

In this situation, we need read-after-write consistency, also known as read-your-writes consistency. This is a guarantee that if the user reloads the page, they will always see any updates they submitted themselves. It makes no promises about other users.

There are various techniques to achieve this :

- When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower.Eg:always read the user's own profile from the leader, and any other users' profiles from a follower.
- You could track the time of the last update and, for one minute after the last update, make all reads from the leader. You could also monitor the replication lag on followers and prevent queries on any follower that is more than one minute behind the leader.

Another complication arises when the same user is accessing your service from multiple devices, for example a desktop web browser and a mobile app. In this case you may want to provide cross-device read-after-write consistency: if the user enters some information on one device and then views it on another device, they should see the information they just entered.
In this case there are some additional issues to consider :

- Approaches that require remembering the timestamp of the user's last update become more difficult(metadata has to be centralized).
- If your replicas are distributed across different datacenters, there is no guarantee that connections from different devices will be routed to the same datacenter.If your approach requires reading from leader,you may first need to route requests from all of a user's devices to the same datacenter.

Monotonic Reads

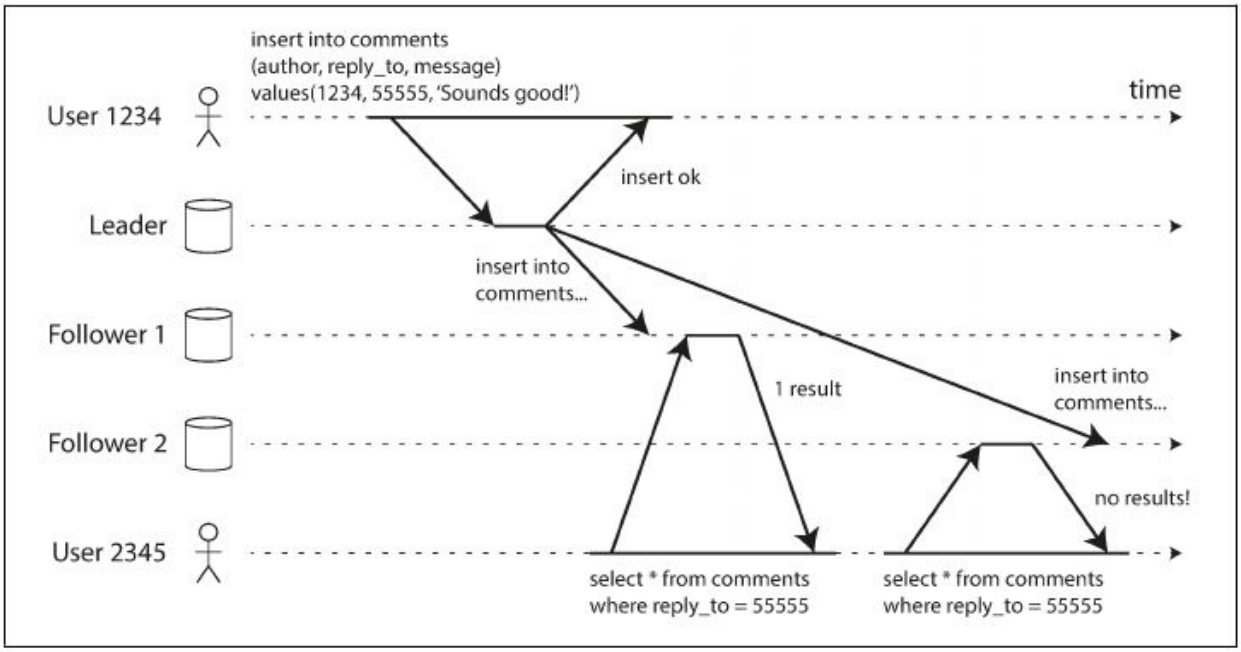


Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.

- Monotonic reads only means that if one user makes several reads in sequence, they will not see time go backward—i.e., they will not read older data after having previously read newer data.
- Monotonic reads is a guarantee that this kind of anomaly does not happen. It's a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency.
- One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica

Consistent Prefix Reads

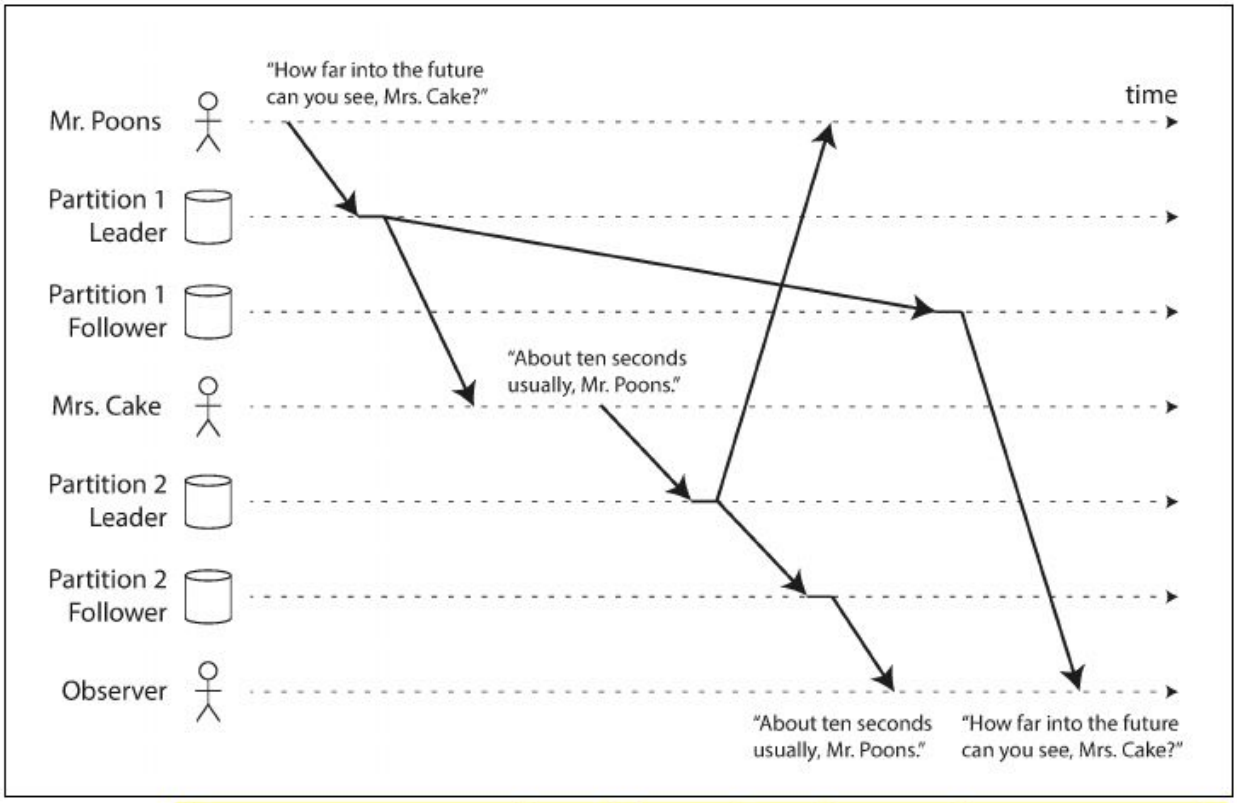


Figure 5-5. If some partitions are replicated slower than others, an observer may see the answer before they see the question.

- Consistent prefix reads guarantee says that if a sequence of writes happens in a certain order, then anyone reading those writes will see them appear in the same order.
- One solution is to make sure that any writes that are causally related to each other are written to the same partition—but in some applications that cannot be done efficiently.

Multi-Leader Replication

In a multi-leader configuration, you can have a leader in each datacenter.

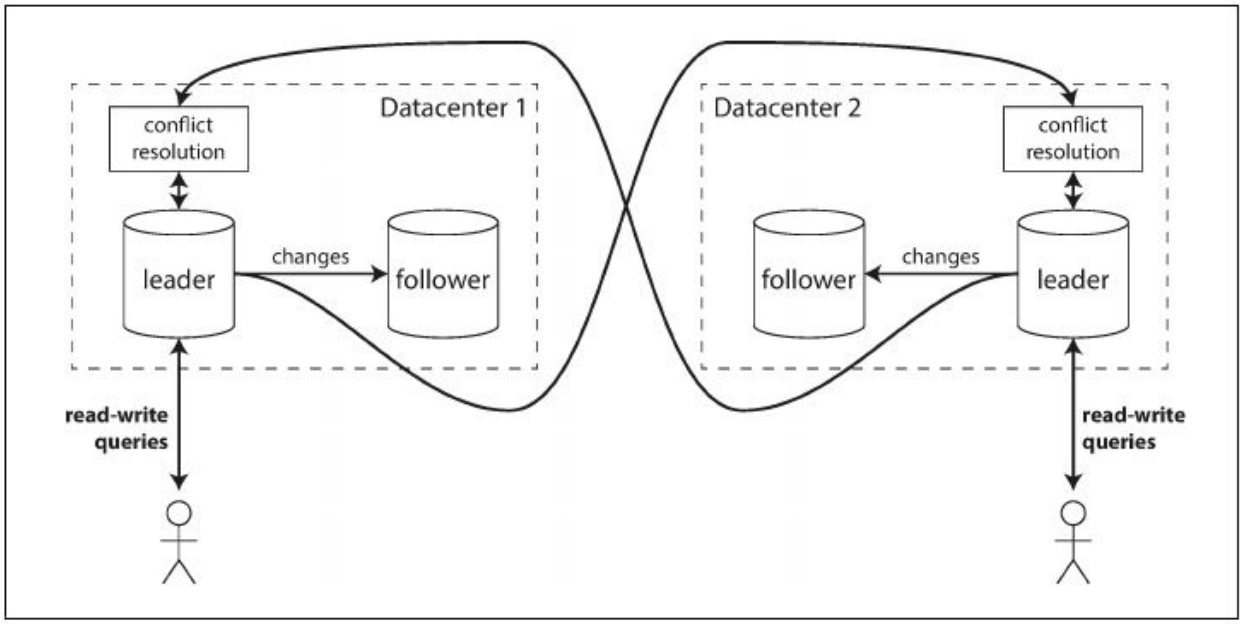


Figure 5-6. Multi-leader replication across multiple datacenters.

Comparison of Multi-Leader and Single-Leader

- Performance -
    - In a single-leader configuration, every write must go over the internet to the datacenter with the leader,this can add significant latency to writes.
    - In multi-leader configuration, every write can be processed in the local datacenter and is replicated asynchronously to the other datacenters.Hence perceived performance may be better.
- Tolerance of datacenter outages -
    - In multi-leader configuration,each datacenter can continue operating independently of the others unlike single-leader where another is promoted.
- Tolerance of network problems -
    - A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes being processed.
- The biggest downside of Multi-Leader is that the same data may be concurrently modified in two different datacenters, and those write conflicts must be resolved.
- Multi-leader replication is often considered dangerous territory that should be avoided if possible.

Clients with offline operation

- Another situation in which multi-leader replication is appropriate is if you have an application that needs to continue to work while it is disconnected from the internet.
- In this case, every device has a local database that acts as a leader (it accepts write requests), and there is an asynchronous multi-leader replication process (sync) between the replicas of your

calendar on all of your devices. The replication lag may be hours or even days, depending on when you have internet access available.

Collaborative editing

- We don't usually think of collaborative editing as a database replication problem, but it has a lot in common with the previously mentioned offline editing use case.
- When one user edits a document, the changes are instantly applied to their local replica and asynchronously replicated to the server and any other users who are editing the same document.
- If you want to guarantee that there will be no editing conflicts, the application must obtain a lock on the document before a user can edit it.(Similar to single-leader replication).
- For faster collaboration, you may want to make the unit of change very small (e.g., a single keystroke) and avoid locking. This approach allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication.

Handling Write Conflicts

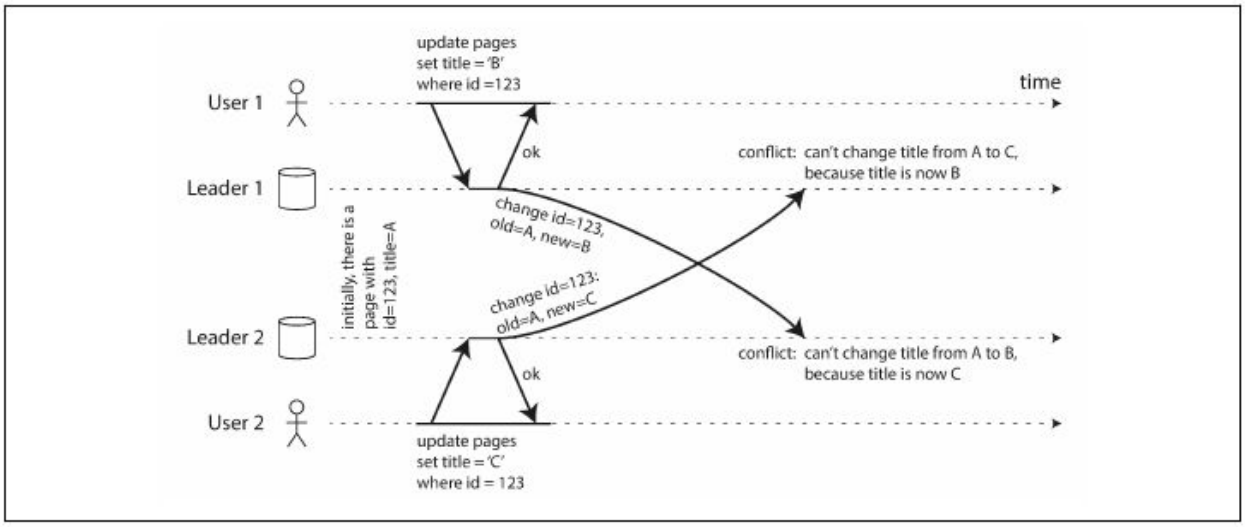


Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.

Synchronous versus asynchronous conflict detection

- If both writes are successful and conflict is detected asynchronously later,it may be too late to ask the user to resolve the conflict.
- If you want synchronous conflict detection, you might as well just use single-leader replication.

Conflict avoidance

- The simplest strategy for dealing with conflicts is to avoid them: if the application can ensure that all writes for a particular record go through the same leader, then conflicts cannot occur.

- However conflict avoidance breaks down when the designated leader breaks down and you have to reroute to another leader.

Converging toward a consistent state

In a multi-leader configuration, there is no defined ordering of writes, so it's not clear what the final value should be.
The various of achieving convergent conflict resolution :

- Pick the write with the highest ID as the winner, and throw away the other writes. If a timestamp is used, this technique is known as last write wins(LWW).
- Let the replica with higher ID take precedence.
- Somehow merge the values together.
- Record the conflict in an explicit data structure and write application code that resolves the conflict at some later time.

Custom conflict resolution logic

The conflict resolution can be executed on write or read :

- On write - As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler.
- On read - When a conflict is detected, all the conflicting writes are stored. The next time the data is read, these multiple versions of the data are returned to the application.

Automatic Conflict Resolution

- Conflict-free replicated datatypes (CRDTs) are a family of data structures for sets, maps, ordered lists, counters, etc. that can be concurrently edited by multiple users, and which automatically resolve conflicts in sensible ways.
- Mergeable persistent data structures track history explicitly, similarly to the Git version control system, and use a three-way merge function (whereas CRDTs use two-way merges).
- Operational transformation is the conflict resolution algorithm behind collaborative editing applications such as Google Docs.
- Automatic conflict resolution could make multi-leader data synchronization much simpler for applications to deal with.

Multi-Leader Replication Topologies

- A replication topology describes the communication paths along which writes are propagated from one node to another.

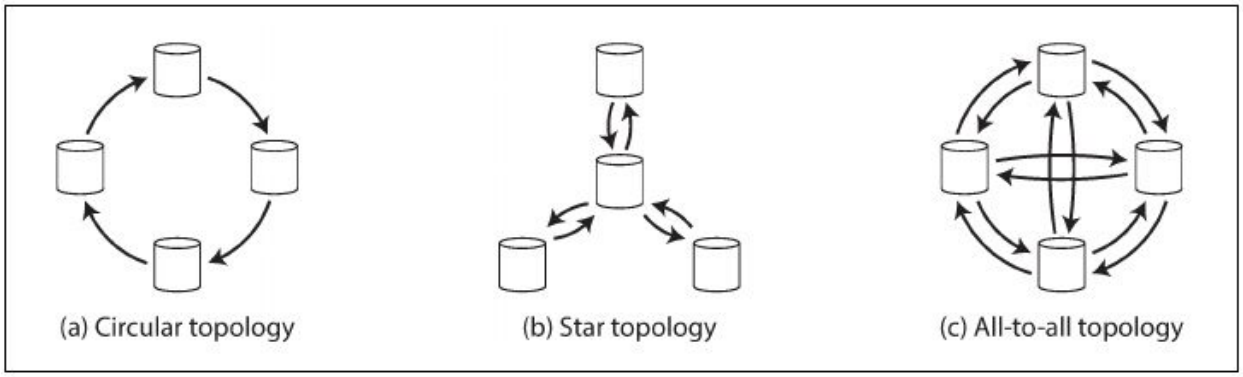


Figure 5-8. Three example topologies in which multi-leader replication can be set up.

- The most general topology is all-to-all in which every leader sends its writes to every other leader.
- all-to-all topologies can have issues too. In particular, some network links may be faster than others (e.g., due to network congestion), with the result that some replication messages may "overtke others".(basically in wrong order).

Leaderless Replication

In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does this on behalf of the client. However,unlike a leader database, that coordinator does not enforce a particular ordering of writes.

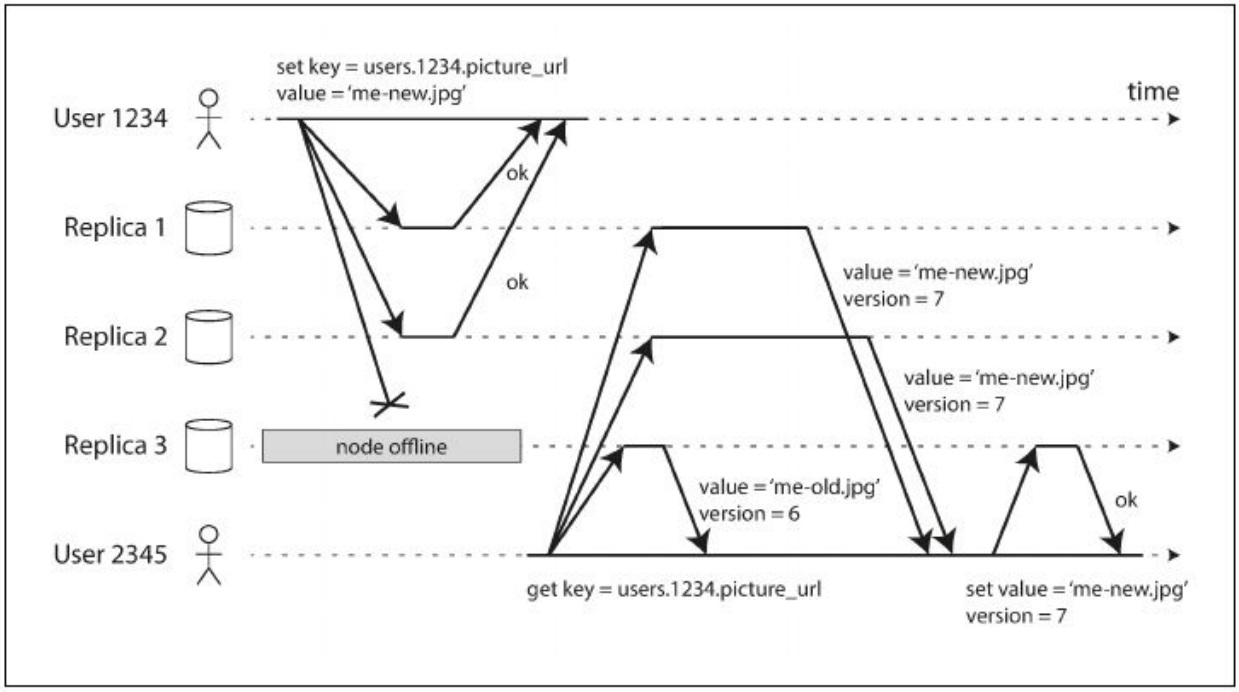


Figure 5-10. A quorum write, quorum read, and read repair after a node outage.

Writing to the Database When a Node Is Down

- When a client reads from the database, it doesn't just send its request to one replica: read requests are also sent to several nodes in parallel.Version numbers are used to determine which value is newer

Read repair and anti-entropy

After an unavailable node comes back online how does it catch up ?Two mechanisms:

- Read repair -
    - When a client makes a read from several nodes in parallel, it can detect any stale responses.
    - The client sees that replica 3 has a stale value and writes the newer value back to that replica.
    - This approach works well for values that are frequently read.
- Anti-entropy process
    - In addition, some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another.

Quorums for reading and writing

- 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 as long as w+r>n.
- Reads and writes that obey these r and w values are called quorum reads and writes(strict quorums).
- A common choice is to make n an odd number (typically 3 or 5) and to set
  $w = r = (n + 1) / 2$ (rounded up).

Limitations of Quorum Consistency

- You may also set w and r to smaller numbers, so that $w + r \leq n$. In this case, reads and writes will still be sent to n nodes, but a smaller number of successful responses is required for the operation to succeed.
- With a smaller w and r you are more likely to read stale values, because it's more likely that your read didn't include the node with the latest value.But allows lower latency and higher availability.
- However, even with $w + r > n$, there are likely to be edge cases where stale values are returned.
- Dynamo-style databases are generally optimized for use cases that can tolerate eventual consistency.

Monitoring staleness

- In systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult.
- It would be good to include staleness measurements in the standard set of metrics for databases.

Sloppy Quorums and Hinted Handoff

- Sloppy quorum : writes and reads still require w and r successful responses, but those may include nodes that are not among the designated n “home” nodes for a value.
- Any writes that one node temporarily accepted on behalf of another node are sent to the appropriate “home” nodes. This is hinted handoff.
- There is no guarantee that a read of r nodes will see it until the hinted handoff has completed.
- Leaderless replication is also suitable for multi-datacenter operation, since it is designed to tolerate conflicting concurrent writes, network interruptions, and latency spikes.

Detecting Concurrent Writes

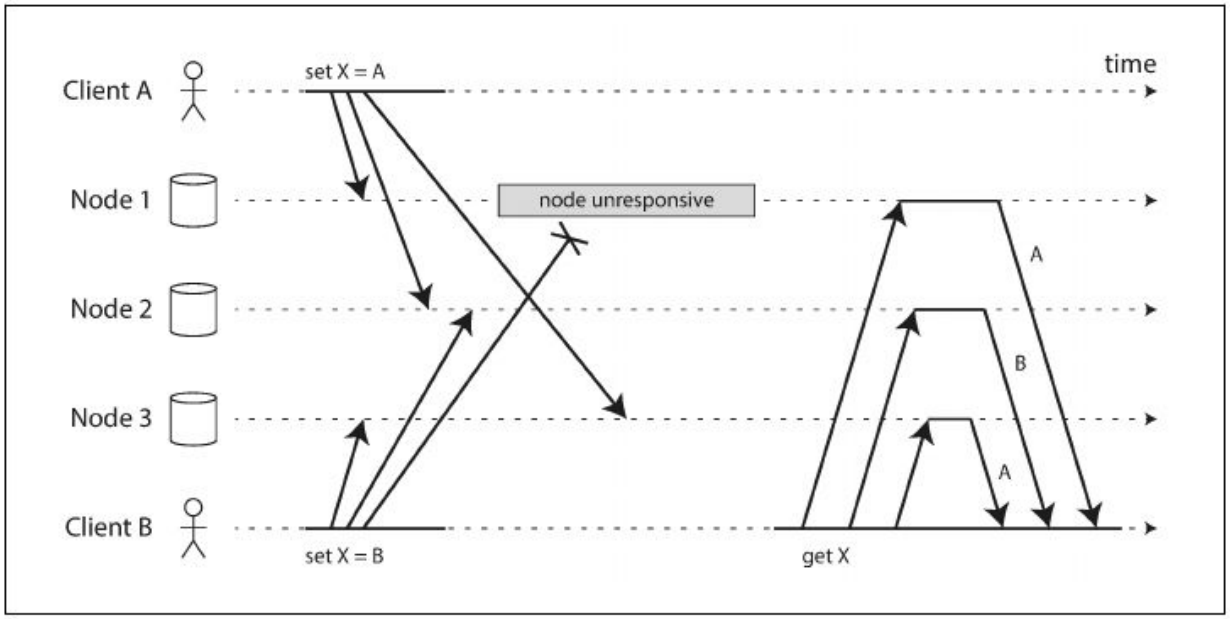


Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.

Last write wins (discarding concurrent writes)

- One approach for achieving eventual convergence is to declare that each replica need only store the most “recent” value and allow “older” values to be overwritten and discarded.
- If losing data is not acceptable, LWW is a poor choice for conflict resolution.
- The only safe way of using a database with LWW is to ensure that a key is only written once and thereafter treated as immutable.

The “happens-before” relationship and concurrency

- The two writes in Figure 5-12 are concurrent: when each client starts the operation, it does not know that another client is also performing an operation on the same key. Thus, there is no causal

dependency between the operations. We can simply say that two operations are concurrent if neither happens before the other(exact time doesn't matter).

Capturing the happens-before relationship :

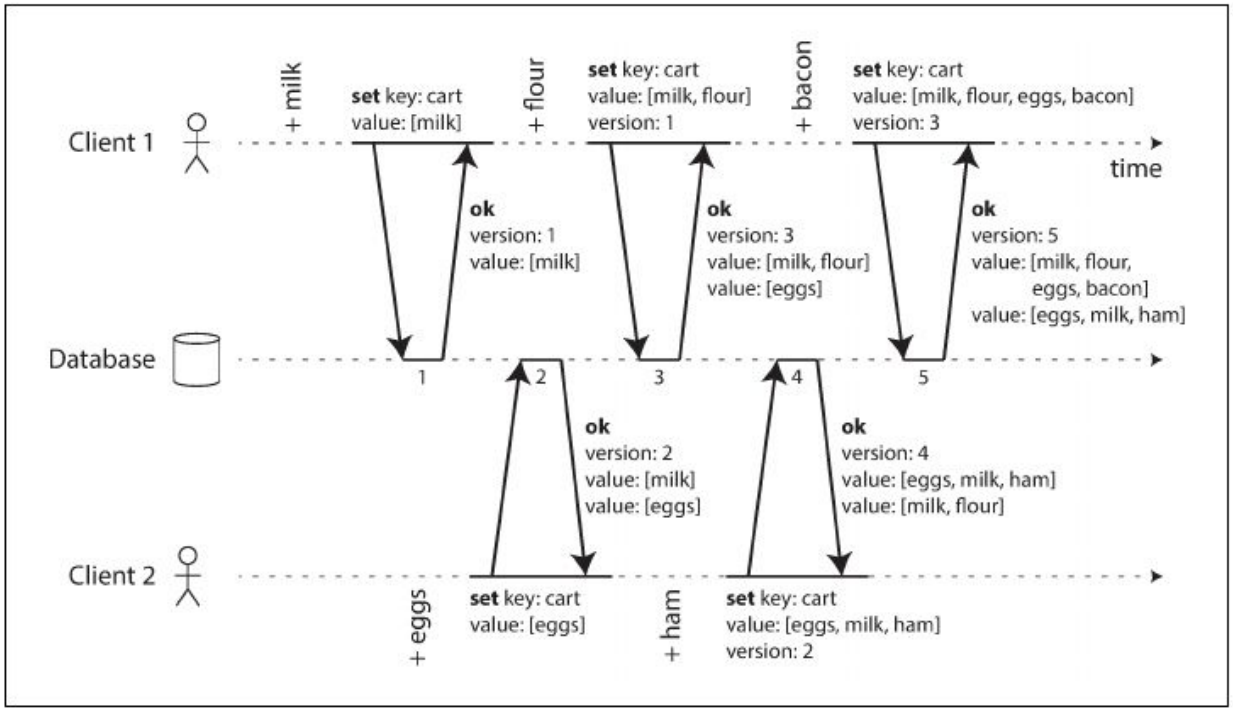


Figure 5-13. Capturing causal dependencies between two clients concurrently editing a shopping cart.

The algorithm works as follows:

- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written.
- When a client reads a key, the server returns all values that have not been overwritten, as well as the latest version number. A client must read a key before writing.
- When a client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read.
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value), but it must keep all values with a higher version number (because those values are concurrent with the incoming write).

Version vectors

How does the algorithm change when there are multiple replicas, but no leader?

- We need to use a version number per replica as well as per key. Each replica increments its own version number when processing a write, and also keeps track of the version numbers it has seen from each of the other replicas.
- The collection of version numbers from all the replicas is called a version vector.
- The version vector allows the database to distinguish between overwrites and concurrent writes

Chapter 6 Partitioning

Partitioning and Replication

- Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes.
- A node may store more than one partition.
- The choice of partitioning scheme is mostly independent of the choice of replication scheme.

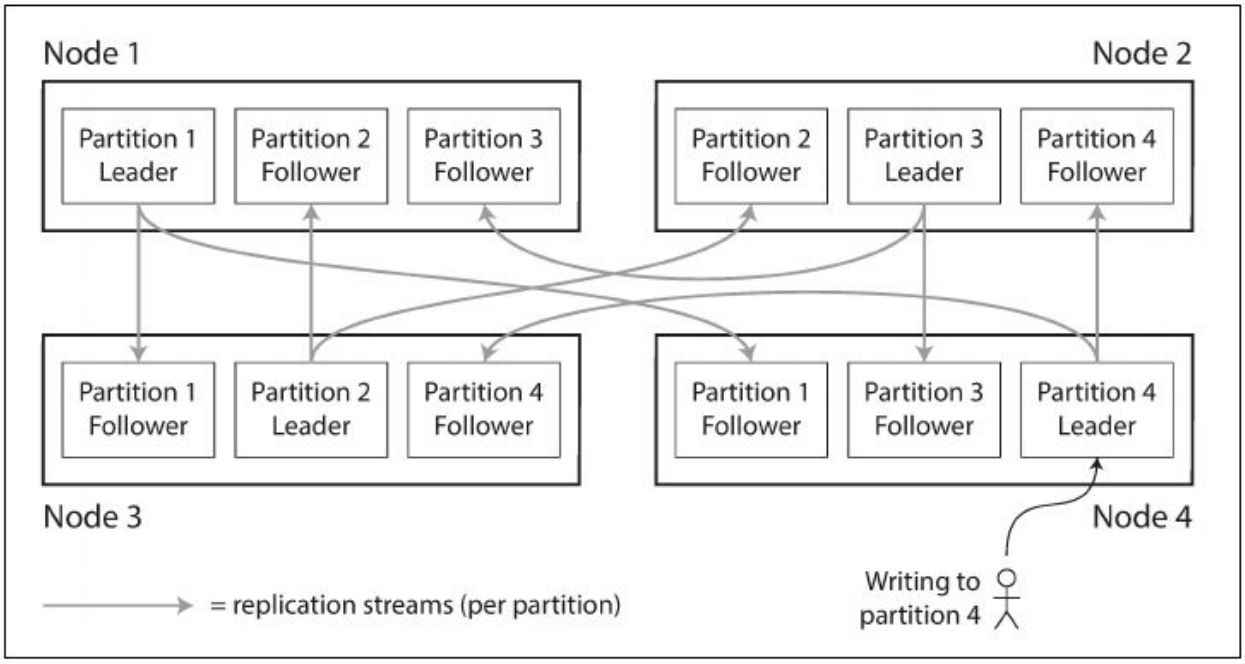


Figure 6-1. Combining replication and partitioning: each node acts as leader for some partitions and follower for other partitions.

Partitioning of Key-Value Data

- Our goal with partitioning is to spread the data and the query load evenly across nodes.
- If the partitioning is unfair, so that some partitions have more data or queries than others, we call it skewed.

Partitioning by Key Range

- One way of partitioning is to assign a continuous range of keys to each partition.

- If you know the boundaries between the ranges, you can easily determine which partition contains a given key.
- In order to distribute the data evenly, the partition boundaries need to adapt to the data.
- Within each partition, we can keep keys in sorted order (see “SSTables and LSM-Trees ”). This has the advantage that range scans are easy.
- The downside of key range partitioning is that certain access patterns can lead to hot spots.

Partitioning by Hash of Key

- A good hash function takes skewed data and makes it uniformly distributed.
- This technique is good at distributing keys fairly among the partitions. The partition boundaries can be evenly spaced, or they can be chosen pseudorandomly (in which case the technique is sometimes known as consistent hashing).
- By using the hash of the key for partitioning we lose a nice property of key-range partitioning: the ability to do efficient range queries.
- Cassandra achieves a compromise between the two partitioning strategies. A table in Cassandra can be declared with a compound primary key consisting of several columns. Only the first part of that key is hashed to determine the partition, but the other columns are used as a concatenated index for sorting the data in Cassandra’s SSTables.

Skewed Workloads and Relieving Hot Spots

- In the extreme case where all reads and writes are for the same key, you still end up with all requests being routed to the same partition(Eg:celebrity with millions of followers).

Partitioning and Secondary Indexes

- The problem with secondary indexes is that they don’t map neatly to partitions. There are two main approaches to partitioning a database with secondary indexes: document-based partitioning and term-based partitioning.

Partitioning Secondary Indexes by Document

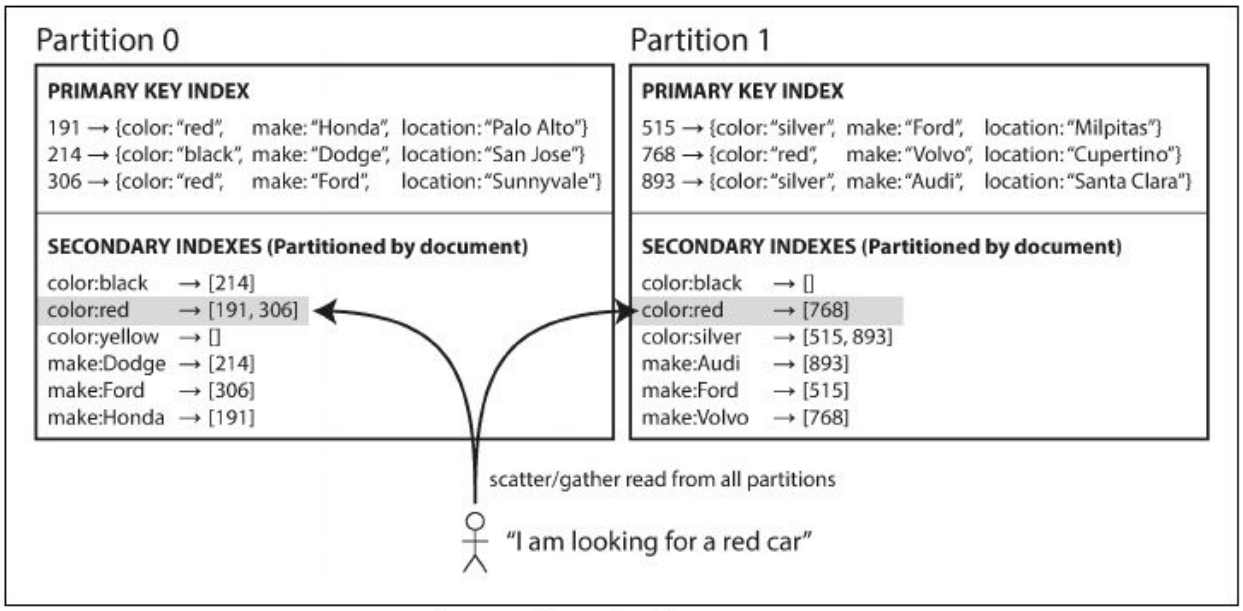


Figure 6-4. Partitioning secondary indexes by document.

- In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition.
- For that reason, a document-partitioned index is also known as a local index.
- This approach to querying a partitioned database is sometimes known as scatter/gather, and it can make read queries on secondary indexes quite expensive.

Partitioning Secondary Indexes by Term

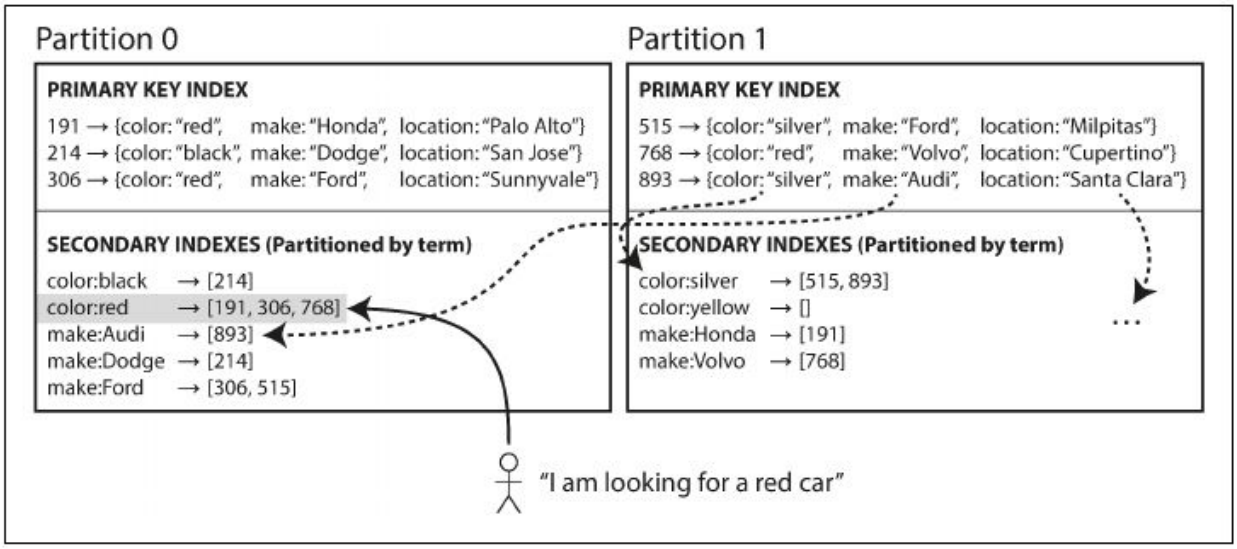


Figure 6-5. Partitioning secondary indexes by term.

- A global index must also be partitioned, but it can be partitioned differently from the primary key index.
- We call this kind of index term-partitioned, because the term we're looking for determines the partition of the index.
- Partitioning by the term itself can be useful for range scans,whereas partitioning on a hash of the term gives a more even distribution of load.
- The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient.Downside writes are slower and complicated.
- In practice, updates to global secondary indexes are often asynchronous.

Rebalancing Partitions

The process of moving load from one node in the cluster to another is called rebalancing.
Minimum requirements rebalancing should meet :

- After rebalancing,the load should be sharded fairly.
- While rebalancing is happening,the database should continue accepting read/writes.
- It should be fast and minimize network and I/O load.

Strategies for Rebalancing

How not to do it: hash mod N

- The problem with the mod N approach is that if the number of nodes N changes, most of the keys will need to be moved from one node to another.

Fixed number of partitions

- Create many more partitions than there are nodes, and assign several partitions to each node.
- If a node is added to the cluster, the new node can steal a few partitions from every existing node until partitions are fairly distributed once again.

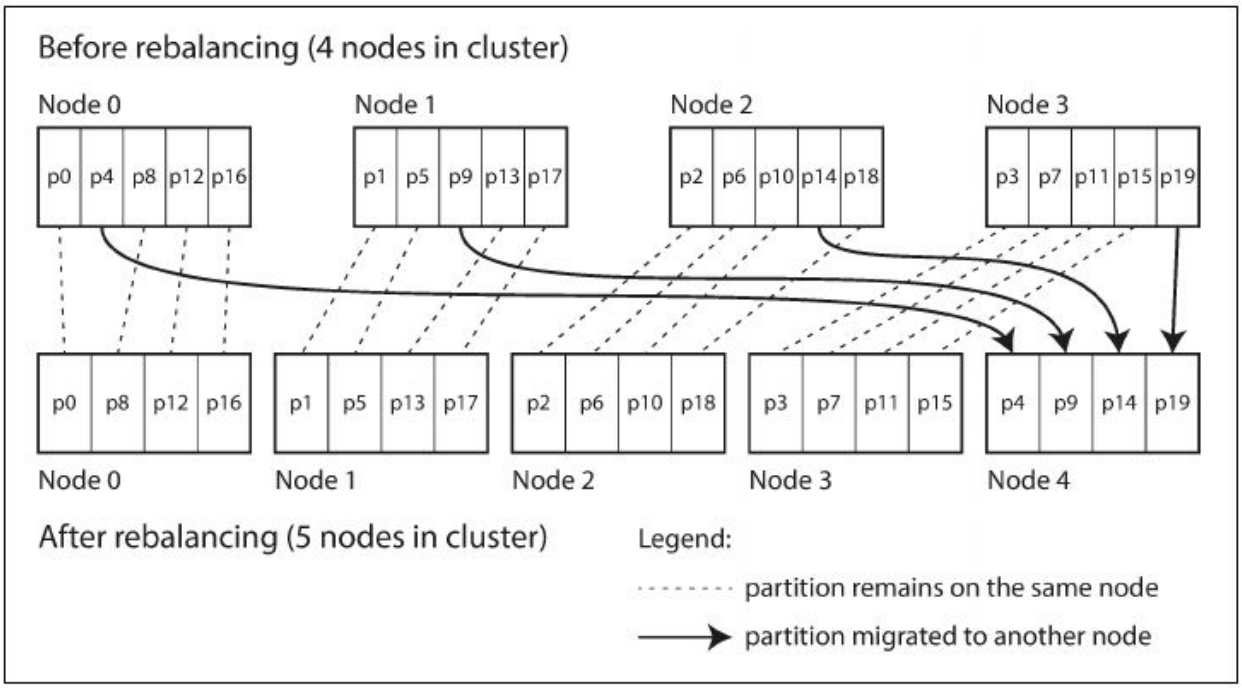


Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.

- You can even account for mismatched hardware in your cluster: by assigning more partitions to nodes that are more powerful, you can force those nodes to take a greater share of the load.
- If partitions are very large, rebalancing and recovery from node failures become expensive. But if partitions are too small, they incur too much overhead. The best performance is achieved when the size of partitions is “just right”.

Dynamic partitioning

- When a partition grows to exceed a configured size (on HBase, the default is 10 GB), it is split into two partitions so that approximately half of the data ends up on each side of the split. Conversely, if lots of data is deleted and a partition shrinks below some threshold, it can be merged with an adjacent partition.
- An advantage of dynamic partitioning is that the number of partitions adapts to the total data volume.
- Initial set of partitions to be configured on an empty database (this is called pre-splitting). Pre-splitting requires that you already know what the key distribution is going to look like.
- Dynamic partitioning is not only suitable for key range–partitioned data, but can equally well be used with hash-partitioned data.

Partitioning proportionally to nodes

- In this case, the size of each partition grows proportionally to the dataset size while the number of nodes remains unchanged, but when you increase the number of nodes, the partitions become smaller again.
- When a new node joins the cluster, it randomly chooses a fixed number of existing partitions to split, and then takes ownership of one half of each of those split partitions while leaving the other half of each partition in place(For this hash partitioning is required).
- The randomization can produce unfair splits, but when averaged over a larger number of partitions (in Cassandra, 256 partitions per node by default), the new node ends up taking a fair share of the load from the existing nodes.

Operations: Automatic or Manual Rebalancing

- It can be a good thing to have a human in the loop for rebalancing.For example, Couchbase, Riak, and Voldemort generate a suggested partition assignment automatically, but require an administrator to commit it before it takes effect.

Request Routing

- There remains an open question: when a client wants to make a request, how does it know which node to connect to?This is an instance of a more general problem called service discovery.

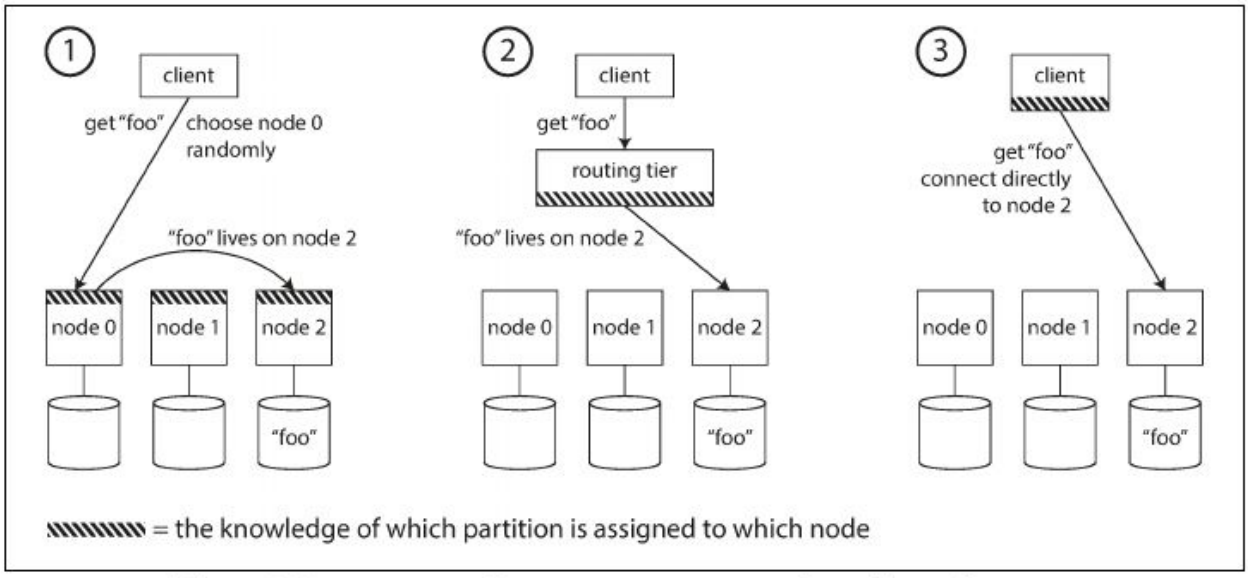
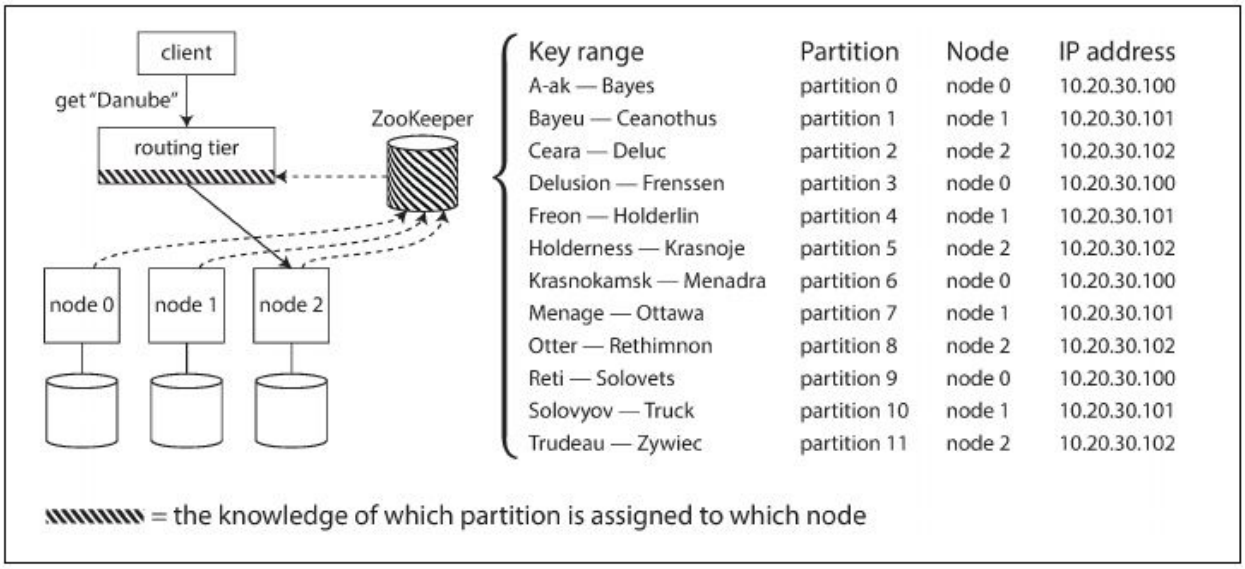


Figure 6-7. Three different ways of routing a request to the right node.

- How does the component making the routing decision (which may be one of the nodes, or the routing tier, or the client) learn about changes in the assignment of partitions to nodes?
- Many distributed data systems rely on a separate coordination service such as Zoo-Keeper to keep track of this cluster metadata.

- Each node registers itself in ZooKeeper, and ZooKeeper maintains the authoritative mapping of partitions to nodes.Whenever a partition changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date.



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.

- Cassandra and Riak take a different approach: they use a gossip protocol among the nodes to disseminate any changes in cluster state(Approach 1). This model puts more complexity in the database nodes but avoids the dependency on an external coordination service such as ZooKeeper.

Chapter 7 Transactions

The Meaning of ACID

Atomicity

- If the writes are grouped together into an atomic transaction, and the transaction cannot be completed (committed) due to a fault, then the transaction is aborted and the database must discard or undo any writes it has made so far in that transaction.

Consistency

- In the context of ACID, consistency refers to an application-specific notion of the database being in a “good state.”
- Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application.

Isolation

- Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other: they cannot step on each other's toes.

Durability

- Durability is the promise that once a transaction has committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes.

Weak Isolation Levels

- Databases have long tried to hide concurrency issues from application developers by providing transaction isolation.
- Serializable isolation has a performance cost, and many databases don't want to pay that price .
- It's therefore common for systems to use weaker levels of isolation, which protect against some concurrency issues, but not all.

Read Committed

The most basic level of transaction isolation is read committed.It makes two guarantees:

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

Implementing Read Committed

- Most commonly, databases prevent dirty writes by using row-level locks.
- To prevent dirty reads there are two options :
    - One option is to use the same lock for read as well.But it does not work well in practice as one long write transaction would force many read transactions to wait.
    - For that reason most databases remembers the old committed value and the new value set by the transaction that currently holds the write lock.
- However read committed does not prevent the race condition between two counter increments(Lost Update Problem)
- Read skew is considered acceptable in Read Committed.

Snapshot Isolation and Repeatable Read

- Problem that can occur in Read Committed :

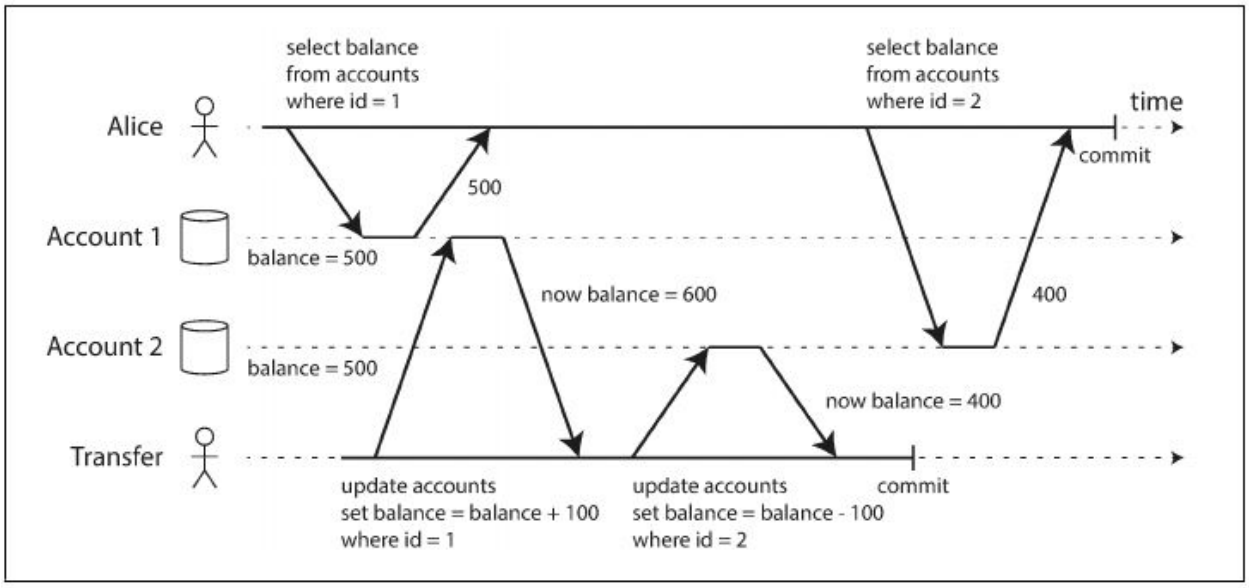


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

- Some situations cannot tolerate such temporary inconsistency:
  - Backups
  - Analytic queries and integrity checks
- Snapshot isolation is the most common solution to this problem. The idea is that each transaction reads from a consistent snapshot of the database—that is, the transaction sees all the data that was committed in the database at the start of the transaction. Even if the data is subsequently changed by another transaction, each transaction sees only the old data from that particular point in time.
- Snapshot isolation is a boon for long-running, read-only queries such as backups and analytics.

Implementing Snapshot Isolation

- A key principle of snapshot isolation is readers never block writers, and writers never block readers.
- The database must potentially keep several different committed versions of an object, because various in-progress transactions may need to see the state of the database at different points in time. Because it maintains several versions of an object side by side, this technique is known as multi-version concurrency control (MVCC).
- A typical approach is that read committed uses a separate snapshot for each query, while snapshot isolation uses the same snapshot for an entire transaction.

Implementing Snapshot Isolation

- When a transaction is started, it is given a unique, always-increasing transaction ID (txid).

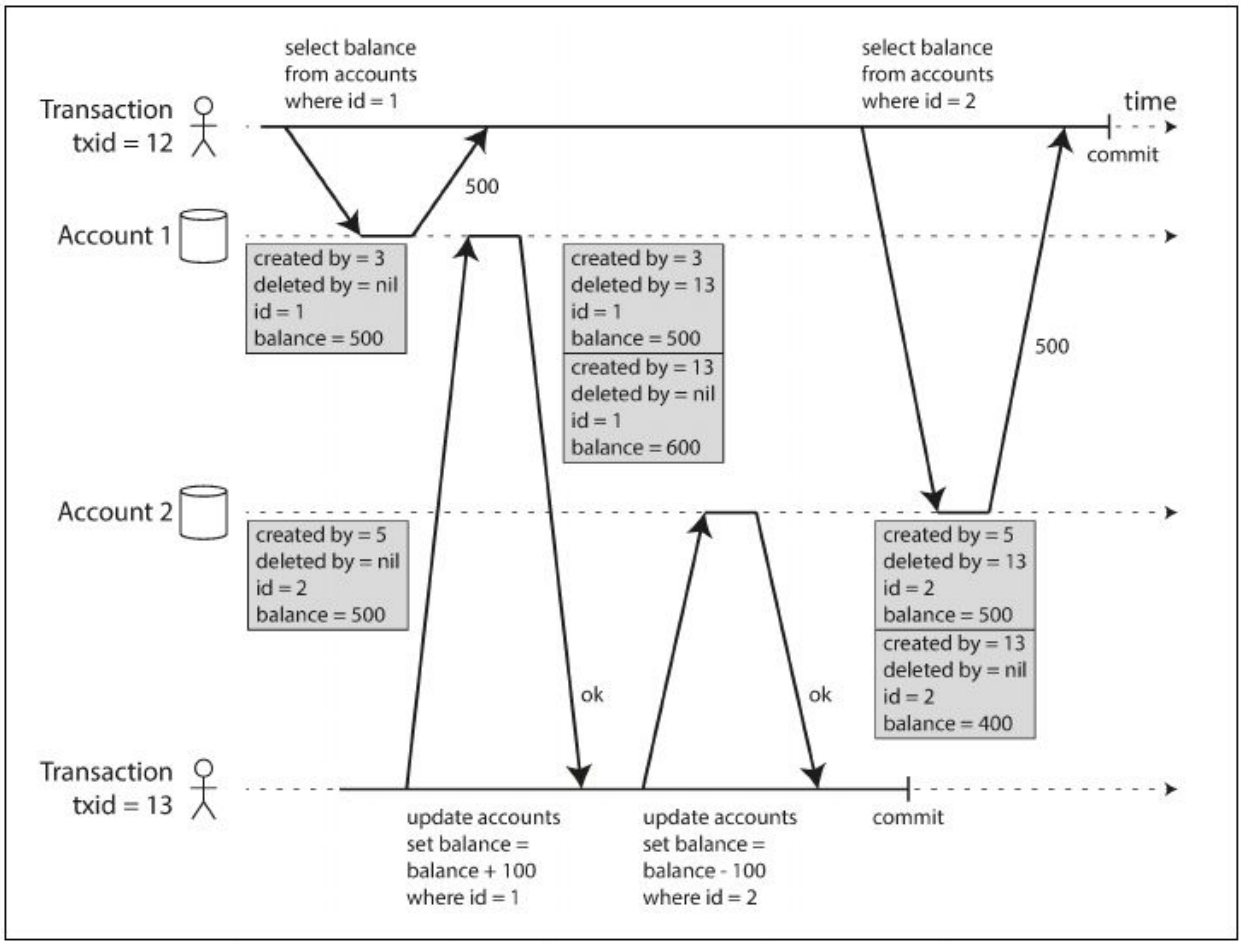


Figure 7-7. Implementing snapshot isolation using multi-version objects.

- An update is internally translated into a delete and a create.It is marked for deletion by setting the deleted_by field to the ID of the transaction that requested the deletion.Later the garbage collection process removes it.
- An object is visible if both of the following conditions are true:
    - At the time when the reader's transaction started, the transaction that created the object had already committed.
    - The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.
- By never updating values in place but instead creating a new version every time a value is changed, the database can provide a consistent snapshot while incurring only a small overhead.

Indexes and snapshot isolation

- How do indexes work in a multi-version database? One option is to have the index simply point to all versions of an object and require an index query to filter out any object versions that are not visible to the current transaction.

- Another approach is used in CouchDB, Datomic, and LMDB. Although they also use B-trees,they use an append-only/copy-on-write variant that does not overwrite pages of the tree when they are updated, but instead creates a new copy of each modified page.
- Any pages that are not affected by a write do not need to be copied, and remain immutable.
- With append-only B-trees, every write transaction (or batch of transactions) creates a new B-tree root, and a particular root is a consistent snapshot of the database at the point in time when it was created.
- There is no need to filter out objects based on transaction IDs because subsequent writes cannot modify an existing B-tree.
- However this approach requires garbage collection and compaction.

Preventing Lost Updates

- The isolation levels discussed so far have ignored the issues of two transactions writing concurrently(Like lost update).

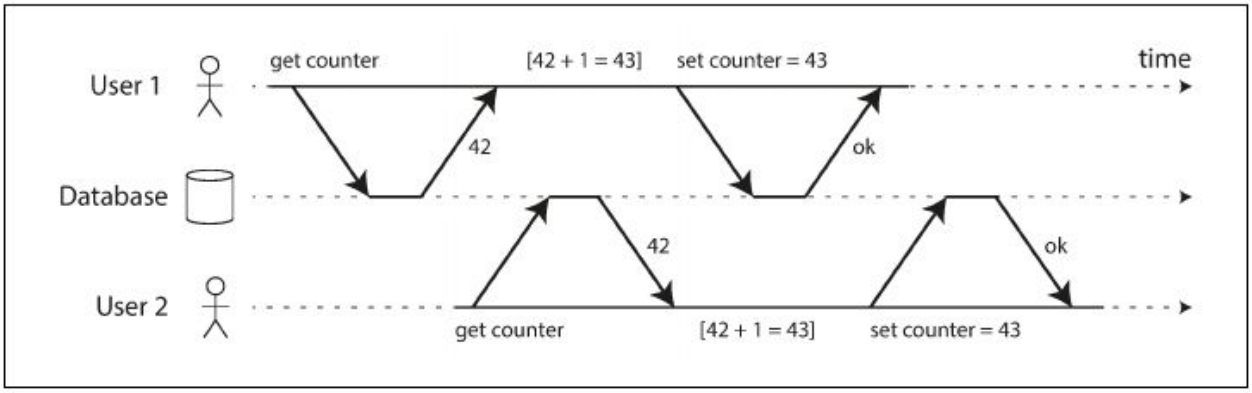


Figure 7-1. A race condition between two clients concurrently incrementing a counter.

Variety of Solutions have been developed:

Atomic write operations

- Many databases provide atomic update operations, which remove the need to implement read-modify-write cycles in application code.

Explicit locking

- The application can perform a read-modify-write cycle, and if any other transaction tries to concurrently read the same object, it is forced to wait until the first read-modify-write cycle has completed.

Automatically detecting lost updates

- An alternative is to allow them to execute in parallel and, if the transaction manager detects a lost update, abort the transaction and force it to retry its read-modify-write cycle.

- An advantage of this approach is that databases can perform this check efficiently in conjunction with snapshot isolation.

Compare-and-set

- The purpose of this operation is to avoid lost updates by allowing an update to happen only if the value has not changed since you last read it.

Conflict resolution and replication

- Locks and compare-and-set operations assume that there is a single up-to-date copy of the data.
- We can allow concurrent writes and then resolve or merge those versions after the fact.
- Atomic operations can work well in a replicated context, especially if they are commutative.

Write Skew and Phantoms

- It is neither a dirty write nor a lost update, because the two transactions are updating two different objects. It is neither a dirty write nor a lost update, because the two transactions are updating two different objects.
- You can think of write skew as a generalization of the lost update problem. Write skew can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects). In the special case where different transactions update the same object, you get a dirty write or lost update anomaly (depending on the timing).
- Automatically preventing write skew requires true serializable isolation.
- Example of write skew : Claiming a username.
- The effect, where a write in one transaction changes the result of a search query in another transaction, is called a phantom. Snapshot isolation avoids phantoms in read-only queries, but in read-write transactions like the examples we discussed phantoms can lead to particularly tricky cases of write skew.

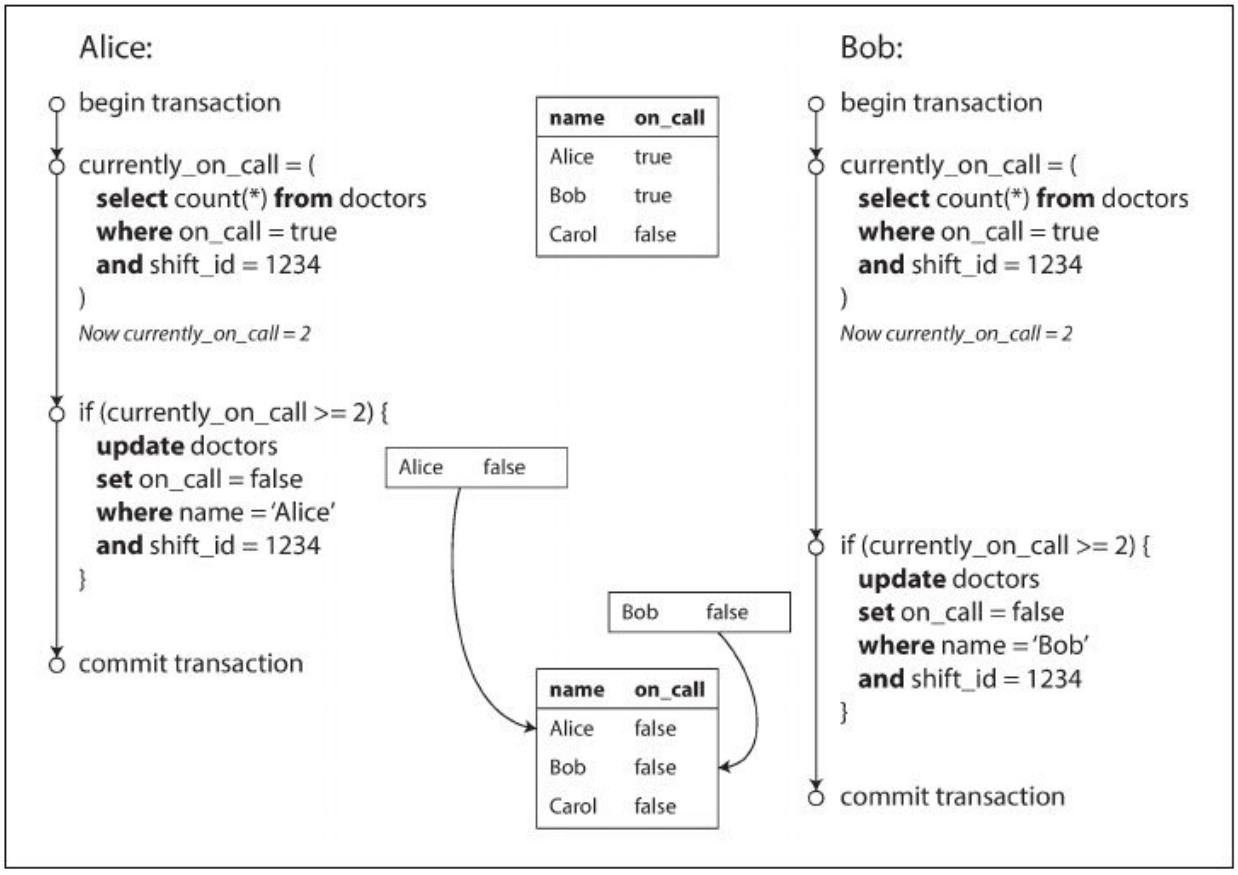


Figure 7-8. Example of write skew causing an application bug.

Materializing conflicts

- If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?
- This approach is called materializing conflicts, because it takes a phantom and turns it into a lock conflict on a concrete set of rows that exist in the database
- Unfortunately it's hard and error-prone.A serializable isolation level is much preferable in most cases.

Serializability

- Serializable isolation is usually regarded as the strongest isolation level.It guarantees that even though transactions may execute in parallel, the end result is the same as if they had executed one at a time, serially, without any concurrency.
- Most databases that provide serializability today use one of three techniques,
    - Actual Serial Execution
    - Two-phase locking
    - Serializable Snapshot Isolation (SSI)

Actual Serial Execution

- Systems with single-threaded serial transaction processing don't allow interactive multi-statement transactions. Instead, the application must submit the entire transaction code to the database ahead of time, as a stored procedure.

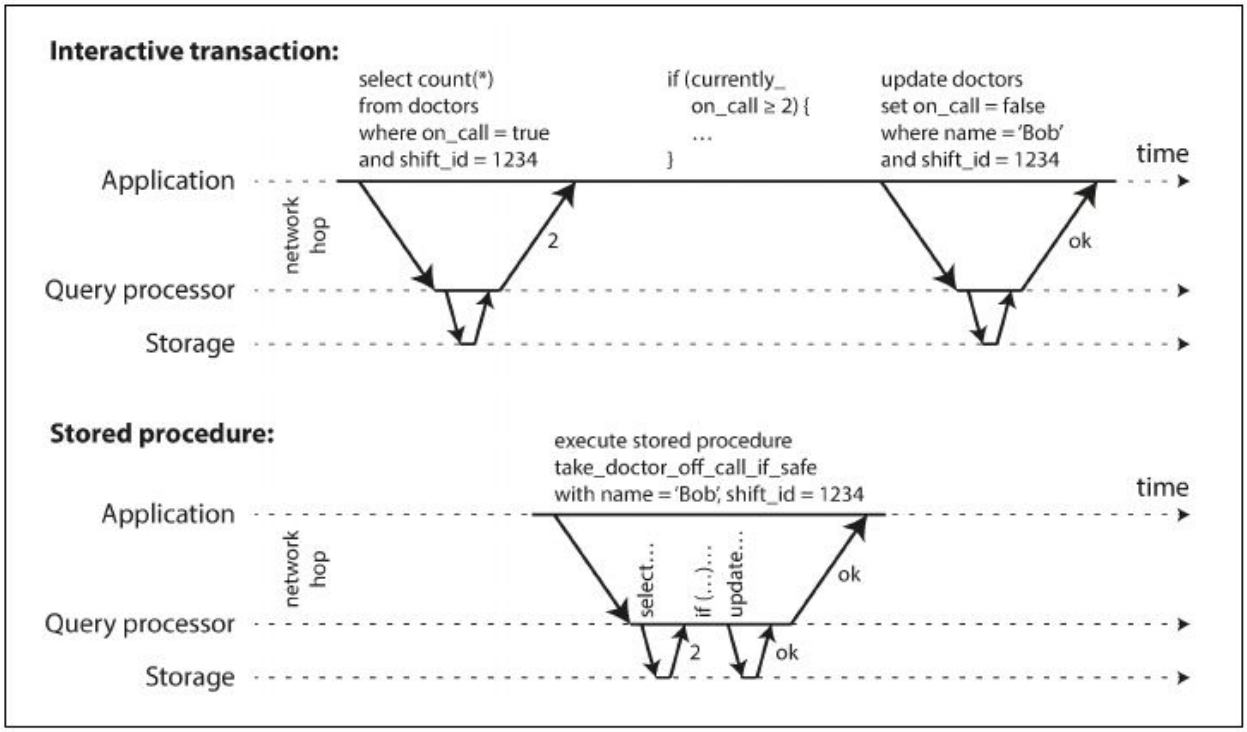


Figure 7-9. The difference between an interactive transaction and a stored procedure

Serial execution of transactions has become a viable way of achieving serializable isolation within certain constraints:

- Every transaction must be small and fast, because it takes only one slow transaction to stall all transaction processing.
- It is limited to use cases where the active dataset can fit in memory. Rarely accessed data could potentially be moved to disk, but if it needed to be accessed in a single-threaded transaction, the system would get very slow.If a transaction needs to access data that's not in memory, the best solution may be to abort the transaction, asynchronously fetch the data into memory while continuing to process other transactions, and then restart the transaction when the data has been loaded. This approach is known as anti-caching.
- Write throughput must be low enough to be handled on a single CPU core, or else transactions need to be partitioned without requiring cross-partition coordination.

Two-Phase Locking (2PL)

- In 2PL, writers don't just block other writers; they also block readers and vice versa.

- If transaction A has read an object and transaction B wants to write to that object, B must wait until A commits or aborts before it can continue. (This ensures that B can't change the object unexpectedly behind A's back.)
- If transaction A has written an object and transaction B wants to read that object, B must wait until A commits or aborts before it can continue.
- 2PL provides serializability, it protects against all the race conditions discussed earlier, including lost updates and write skew.

Implementation of two-phase locking

- The blocking of readers and writers is implemented by having a lock on each object in the database. The lock can either be in shared mode or in exclusive mode.
- The database automatically detects deadlocks between transactions and aborts one of them so that the others can make progress. The aborted transaction needs to be retried by the application

Performance of two-phase locking

- The big downside of two-phase locking, and the reason why it hasn't been used by everybody since the 1970s, is performance: transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation.

Predicate Locks

- Predicate Locks works similarly to the shared/exclusive lock described earlier, but rather than belonging to particular object (e.g., one row in a table), it belongs to all objects that match some search condition.
- The key idea here is that a predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms). If two-phase locking includes predicate locks, the database prevents all forms of write skew and other race conditions, and so its isolation becomes serializable.

Index-range locks

- Unfortunately, predicate locks do not perform well: if there are many locks by active transactions, checking for matching locks becomes time-consuming. For that reason, most databases with 2PL actually implement index-range locking (also known as next- key locking)
- The database can simply attach a shared lock to the index entry(Eg: indicating that a transaction has searched for bookings of room 123).
- An approximation of the search condition is attached to one of the indexes. Now, if another transaction wants to insert, update, or delete a booking for the same room and/or an overlapping time period, it will have to update the same part of the index. In the process of doing so, it will encounter the shared lock, and it will be forced to wait until the lock is released.

Serializable Snapshot Isolation (SSI)

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

Pessimistic versus optimistic concurrency control

- Two-phase locking is a so-called pessimistic concurrency control mechanism: it is based on the principle that if anything might possibly go wrong (as indicated bylock held by another transaction), it's better to wait until the situation is safe again before doing anything. It is like mutual exclusion, which is used to protect data structures in multi-threaded programming.
- Serial execution is, in a sense, pessimistic to the extreme.
- Serializable snapshot isolation is an optimistic concurrency control technique. Optimistic in this context means that instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out all right.
- If contention between transactions not too high, optimistic concurrency control techniques tend to perform better than pessimistic ones.

Decisions based on an outdated premise

- The transaction is taking an action based on a premise.
- How does the database know if a query result might have changed? There are two cases to consider:
    - Detecting reads of a stale MVCC object version (uncommitted write occurred before the read)
    - Detecting writes that affect prior reads (the write occurs after the read)

Detecting stale MVCC reads

- In order to prevent this anomaly, the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules. When the transaction wants to commit, the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted.

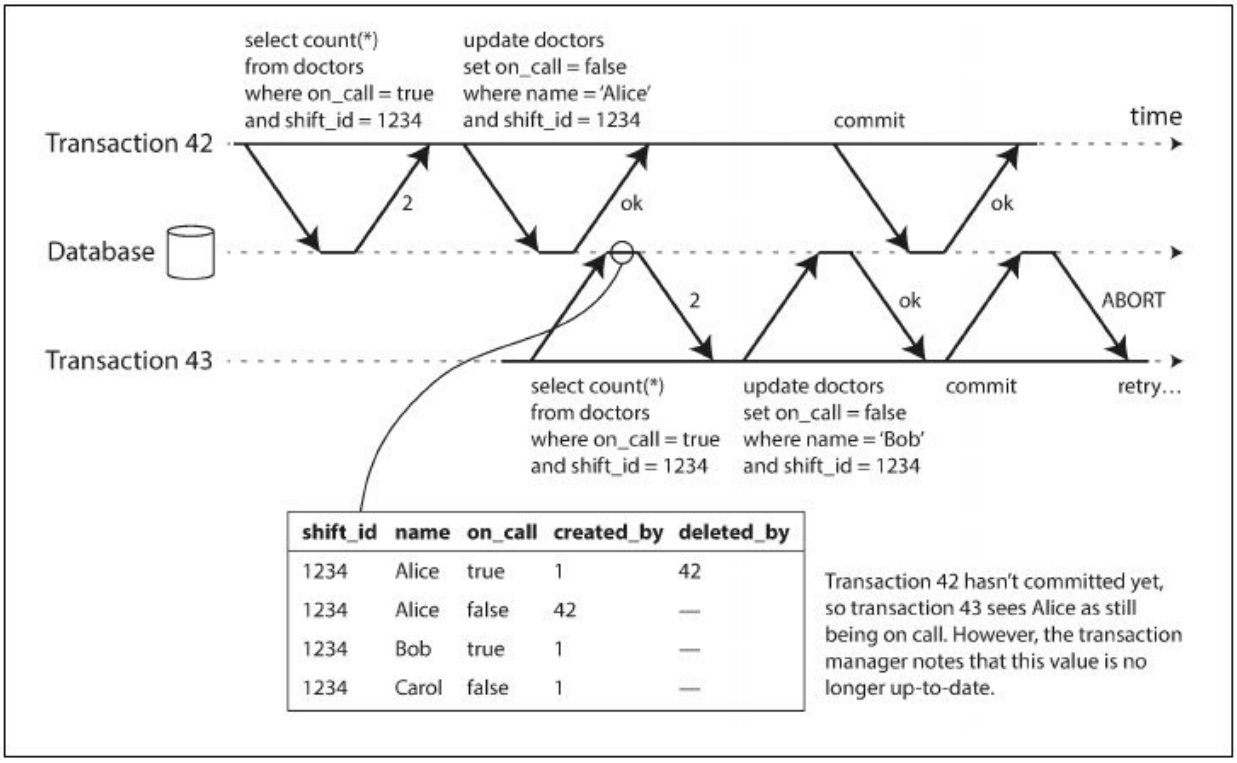


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

- By avoiding unnecessary aborts, SSI preserves snapshot isolation's support for long-running reads from a consistent snapshot.

Detecting writes that affect prior reads

- When another transaction modifies data after it has been read.
- When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data.
- In Figure 7-11, transaction 43 notifies transaction 42 that its prior read is outdated, and vice versa. Transaction 42 is first to commit, and it is successful: although transaction 43's write affected 42, 43 hasn't yet committed, so the write has not yet taken effect. However, when transaction 43 wants to commit, the conflicting write from 42 has already been committed, so 43 must abort.

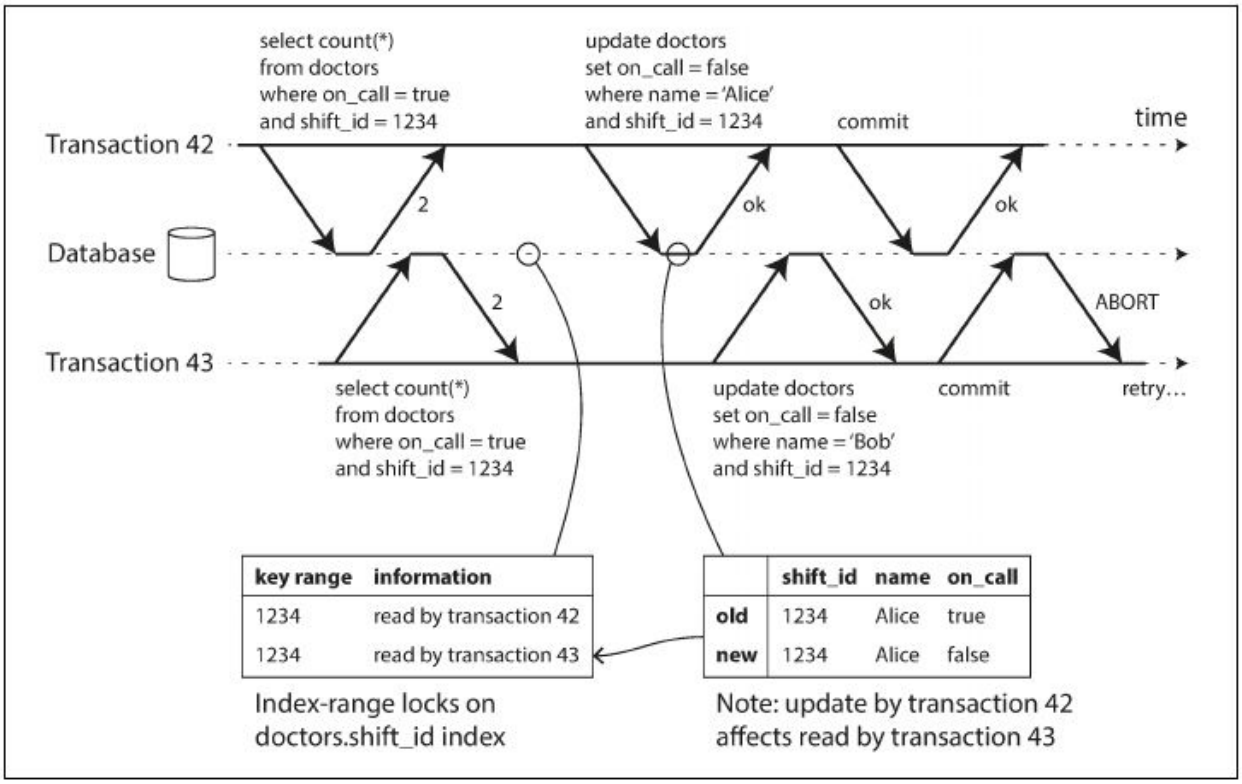


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

Performance of serializable snapshot isolation

- Like under snapshot isolation, writers don't block readers, and vice versa.
- SSI is not limited to the throughput of a single CPU core.
- SSI requires that read-write transactions be fairly short (long-running read-only transactions may be okay).
- However, SSI is probably less sensitive to slow transactions than two-phase locking or serial execution.

Chapter 8 The Trouble with Distributed Systems

Unreliable Networks

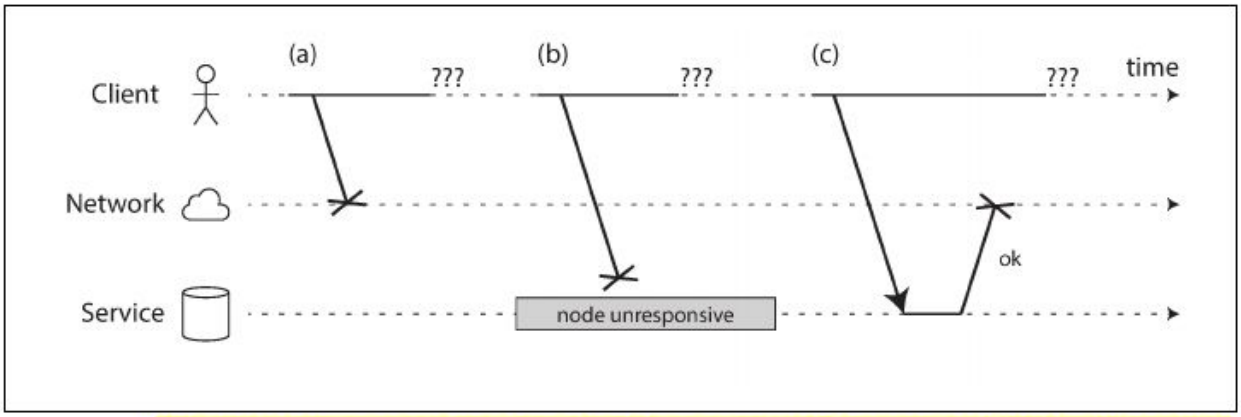


Figure 8-1. If you send a request and don't get a response, it's not possible to distinguish whether (a) the request was lost, (b) the remote node is down, or (c) the response was lost.

- The usual way of handling this issue is a timeout: after some time you give up waiting and assume that the response is not going to arrive.

Network Faults in Practice

- When one part of the network is cut off from the rest due to a network fault, that is sometimes called a network partition or netsplit or network fault.
- If the error handling of network faults is not defined and tested, arbitrarily bad things could happen

Detecting Faults

- Unfortunately, the uncertainty about the network makes it difficult to tell whether a node is working or not.

Timeouts and Unbounded Delays

- If a timeout is the only sure way of detecting a fault, then how long should the time-out be?If the timeout is long, the waiting period is more and if it's less the node may be declared dead incorrectly.
- Most systems we work with have neither of those guarantees: asynchronous networks have unbounded delays (that is, they try to deliver packets as quickly as possible, but there is no upper limit on the time it may take for a packet to arrive), and most server implementations cannot guarantee that they can handle requests within some maximum time.

Network congestion and queueing

- The variability of packet delays on computer networks is most often due to queueing

TCP Versus UDP

- Some latency-sensitive applications, such as videoconferencing and Voice over IP (VoIP), use UDP rather than TCP. It's a trade-off between reliability and variability of delays: as UDP does not perform flow control and does not retransmit lost packets, it avoids some of the reasons for variable network delays.
- UDP is a good choice in situations where delayed data is worthless.
- Rather than using configured constant timeouts, systems can continually measure response times and their variability (jitter), and automatically adjust timeouts according to the observed response time distribution.

Synchronous vs Asynchronous Networks

- This(fixed telephone network) kind of network is synchronous: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been reserved in the next hop of the network. And because there is no queueing, the maximum end-to-end latency of the network is fixed. We call this a bounded delay.
- Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays in the network.
- If you wanted to transfer a file over a circuit, you would have to guess a bandwidth allocation.
- With careful use of quality of service (QoS, prioritization and scheduling of packets) and admission control (rate-limiting senders), it is possible to emulate circuit switching on packet networks, or provide statistically bounded delay
- Latency guarantees are achievable in certain environments, if resources are statically partitioned.
- Variable delays in networks are not a law of nature, but simply the result of a cost/benefit trade-off.

Unreliable Clocks

Monotonic Versus Time-of-Day Clocks

- Modern computers have at least two different kinds of clocks: a time-of-day clock and a monotonic clock.
- A time-of-day clock does what you intuitively expect of a clock: it returns the current date and time according to some calendar (also known as wall-clock time).
- Time-of-day clocks are usually synchronized with NTP, which means that a time‑stamp from one machine (ideally) means the same as a timestamp on another machine.
- A monotonic clock is suitable for measuring a duration.

- They are guaranteed to always move forward (whereas a time-of-day clock may jump back in time).
- It makes no sense to compare monotonic clock values from two different computers, because they don't mean the same thing.
- By default, NTP allows the clock rate to be speeded up or slowed down by up to 0.05%, but NTP cannot cause the monotonic clock to jump forward or backward.

Clock Synchronization and Accuracy

- The quartz clock in a computer is not very accurate: it drifts.Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an NTP server.
- If your NTP daemon is misconfigured, or a firewall is blocking NTP traffic, the clock error due to drift can quickly become large.

Relying on Synchronized Clocks

Timestamps for ordering events

- It's important to be aware that the definition of "recent timestamp" depends on a local time-of-day clock, which may well be incorrect.
- NTP's synchronization accuracy is itself limited by the network round-trip time, in addition to other sources of error such as quartz drift. For correct ordering, you would need the clock source to be significantly more accurate than the thing you are measuring
- So-called logical clocks, which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events

Clock readings have a confidence interval

- It doesn't make sense to think of a clock reading as a point in time—it is more like a range of times, within a confidence interval: for example, a system may be 95% confident that the time now is between 10.3 and 10.5 seconds past the minute.
- Unfortunately, most systems don't expose this uncertainty(confidence interval).
- An interesting exception is Google's TrueTime API in Spanner, which explicitly reports the confidence interval on the local clock. When you ask it for the current time, you get back two values: [earliest, latest], which are the earliest possible and the latest possible timestamp.

Synchronized clocks for global snapshots

- Spanner implements snapshot isolation across datacenters in this way:
    - It uses the clock's confidence interval as reported by the TrueTime API, and is based on the following observation: if you have two confidence intervals, each consisting of an earliest and latest possible timestamp (A = [Aearliest, Alatest] and B = [Bearliest, Blatest]), and those two intervals do not overlap (i.e., Aearliest < Alatest < Bearliest < Blatest), then B definitely happened after A—there can be no doubt. Only if the intervals overlap are we unsure in which order A and B happened.

- In order to ensure that transaction timestamps reflect causality, Spanner deliberately waits for the length of the confidence interval before committing a read-write transaction.

Process Pauses

- The running thread can be preempted at any point and resume it at some later time, without the thread even noticing. The problem is similar to making multi-threaded code on a single machine thread-safe: you can't assume anything about timing, because arbitrary context switches and parallelism.
- In embedded systems, real-time means that a system is carefully designed and tested to meet specified timing guarantees in all circumstances.For most server-side data processing systems, real-time guarantees are simply not economical or appropriate.

Knowledge, Truth, and Lies

- A node in the network cannot know anything for sure—it can only make guesses based on the messages it receives (or doesn't receive) via the network.

The Truth Is Defined by the Majority.

Fencing Tokens

- When using a lock or lease to protect access to some resource, such as the file storage, we need to ensure that a node that is under a false belief of being "the chosen one" cannot disrupt the rest of the system. A fairly simple technique that achieves this goal is called fencing.

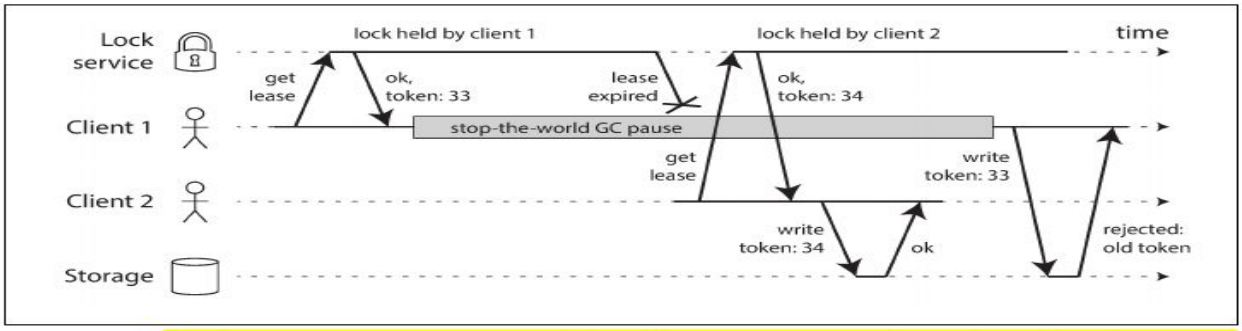


Figure 8-5. Making access to storage safe by allowing writes only in the order of increasing fencing tokens.

- Note that this mechanism requires the resource itself to take an active role in checking tokens by rejecting any writes with an older token than one that has already been processed—it is not sufficient to rely on clients checking their lock status themselves.

Byzantine Faults

- Distributed systems problems become much harder if there is a risk that nodes may "lie" (send arbitrary faulty or corrupted responses).Such behavior is known as a Byzantine fault, and the

problem of reaching consensus in this untrusting environment is known as the Byzantine Generals Problem.

- A system is Byzantine fault-tolerant if it continues to operate correctly even if some of the nodes are malfunctioning and not obeying the protocol, or if malicious attackers are interfering with the network.
- Web applications do need to expect arbitrary and malicious behavior of clients that are under end-user control, such as web browsers.

System Model and Reality

With regard to timing assumptions, three system models are in common use:

1. Synchronous model - The synchronous model assumes bounded network delay, bounded process pauses, and bounded clock error.Not a realistic model.
2. Partially synchronous model - Partial synchrony means that a system behaves like a synchronous system most of the time(Realistic model).
3. Asynchronous model - In this model, an algorithm is not allowed to make any timing assumptions—in fact, it does not even have a clock (so it cannot use timeouts)(restrictive).

The three most common system models for nodes are:

1. Crash-stop faults - In the crash-stop model, an algorithm may assume that a node can fail in only one way, namely by crashing.
2. Crash-recovery faults - We assume that nodes may crash at any moment, and perhaps start responding again after some unknown time. In the crash-recovery model, nodes are assumed to have stable storage
3. Byzantine (arbitrary) faults

For modeling real systems, the partially synchronous model with crash-recovery faults is generally the most useful model.
Safety is often informally defined as nothing bad happens, and liveness as something good eventually happens.
For distributed algorithms, it is common to require that safety properties always hold;However, with liveness properties we are allowed to make caveats.

Chapter 9 Consistency and Consensus

- Most replicated databases provide at least “eventual consistency(convergence)”, which means that if you stop writing to the database and wait for some unspecified length of time, then eventually all read requests will return the same value.
- Transaction isolation is primarily about avoiding race conditions due to concurrently executing transactions, whereas distributed consistency is mostly about coordinating the state of replicas in the face of delays and faults.

Linearizability

- Basic idea of Linearizability is to make a system appear as if there were only one copy of the data, and all operations on it are atomic.
- In a linearizable system, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written.
- Linearizability is a recency guarantee on reads and writes of a register (an individual object). It doesn't group operations together into transactions, so it does not prevent problems such as write skew.
- A database may provide both serializability and linearizability, and this combination is known as strict serializability or strong one-copy serializability
- Serializable snapshot isolation is not linearizable.

Relying on Linearizability

There are a few areas in which linearizability is an important requirement for making a system work correctly :

Locking and leader election

- No matter how the lock is implemented, it must be linearizable: all nodes must agree which node owns the lock; otherwise it is useless.
- Coordination services like Apache ZooKeeper and etcd are often used to implement distributed locks and leader election. They use consensus algorithms to implement linearizable operations in a fault-tolerant way

Constraints and uniqueness guarantees

- A hard uniqueness constraint, such as the one you typically find in relational databases, requires linearizability. Other kinds of constraints, such as foreign key or attribute constraints, can be implemented without requiring linearizability.

Implementing Linearizable Systems

The different replication methods and if they can be made linearizable :

- Single-leader replication (potentially linearizable)
- Consensus algorithms (linearizable)
- Multi-leader replication (not linearizable)
- Leaderless replication (probably not linearizable)

The Cost of Linearizability

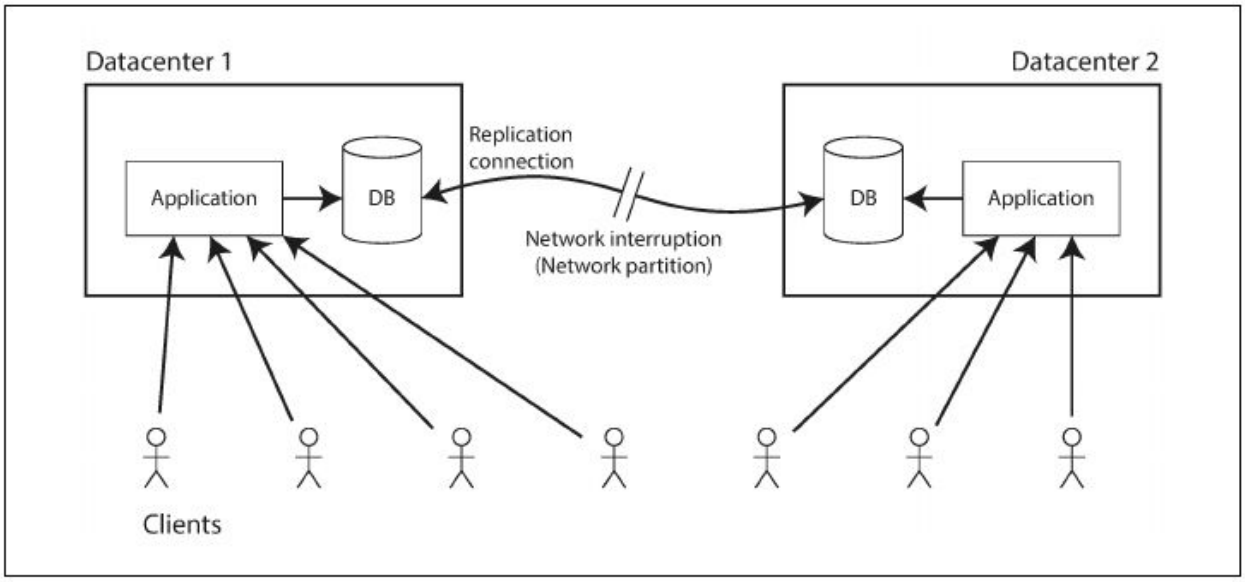


Figure 9-7. A network interruption forcing a choice between linearizability and availability.

CAP Theorem

- If your application requires linearizability, and some replicas are disconnected from the other replicas due to a network problem, then some replicas cannot process requests while they are disconnected: they must either wait until the network problem is fixed, or return an error (either way, they become unavailable).
- If your application does not require linearizability, then it can be written in a way that each replica can process requests independently, even if it is disconnected from other replicas (e.g., multi-leader). In this case, the application can remain available in the face of a network problem, but its behavior is not linearizable.
- Thus, applications that don't require linearizability can be more tolerant of network problems. This insight is popularly known as the CAP theorem
- At times when the network is working correctly, a system can provide both consistency (linearizability) and total availability. When a network fault occurs, you have to choose between either linearizability or total availability. Thus, a better way of phrasing CAP would be either Consistent or Available when Partitioned.
- Although CAP has been historically influential, it has little practical value for designing systems because it only considers one consistency model(linearizability) and one kind of fault(network partition) and not about network delays,dead nodes or other trade-offs.
- If you want linearizability, the response time of read and write requests is at least proportional to the uncertainty of delays in the network.

Ordering Guarantees

Ordering and Causality

- Causality imposes an ordering on events: cause comes before effect; a message is sent before that message is received.
- If a system obeys the ordering imposed by causality, we say that it is causally consistent. For example, snapshot isolation provides causal consistency: when you read from the database, and you see some piece of data, then you must also be able to see any data that causally precedes it.
- The causal order is not total order : A total order allows any two elements to be compared, so if you have two elements, you can always say which one is greater and which one is smaller.Linearizability -> total order, Causality -> partial order.
- Thus Linearizability is stronger than causal consistency but it harms the performance and availability.
- 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

Sequence Number Ordering

- Actually keeping track of all causal dependencies can become impractical.We can create sequence numbers in a total order that is consistent with causality:vii we promise that if operation A causally happened before B, then A occurs before B in the total order
- In single-leader replication the leader can simply increment a counter for each operation, and thus assign a monotonically increasing sequence number to each operation in the replication log.
- If there is not a single leader there are various operations :
    - Each node can generate its own independent set of sequence numbers.
    - Attach a timestamp from a time-of-day clock
    - Preallocate blocks of sequence numbers
- The causality problems occur because these sequence number generators do not correctly capture the ordering of operations across different nodes

Lamport timestamps

- Each node has a unique identifier, and each node keeps a counter of the number of operations it has processed. The Lamport timestamp is then simply a pair of (counter, node ID).
- Lamport timestamps provide a total ordering consistent with causality.
- A Lamport timestamp bears no relationship to a physical time-of-day clock, but it provides total ordering: if you have two timestamps, the one with a greater counter value is the greater timestamp; if the counter values are the same, the one with the greater node ID is the greater timestamp.
- Every node and every client keeps track of the maximum counter value it has seen so far, and includes that maximum on every request. When a node receives a request or response with a maximum counter value greater than its own counter value, it immediately increases its own counter to that maximum.

- As long as the maximum counter value is carried along with every operation, this scheme ensures that the ordering from the Lamport timestamps is consistent with causality, because every causal dependency results in an increased timestamp.
- Although Lamport timestamps define a total order of operations that is consistent with causality, they are not quite sufficient to solve many common problems in distributed systems.Eg: In order to implement something like a uniqueness constraint for user-names, it's not sufficient to have a total ordering of operations—you also need to know when that order is finalized.
- This idea of knowing when your total order is finalized is captured in the topic of total order broadcast.

Total Order Broadcast

- Total order broadcast is usually described as a protocol for exchanging messages between nodes. Informally, it requires that two safety properties always be satisfied:
    - Reliable delivery - No messages are lost: if a message is delivered to one node, it is delivered to all nodes.
    - Totally ordered delivery - Messages are delivered to every node in the same order.
- Consensus services such as ZooKeeper and etcd actually implement total order broadcast.

Distributed Transactions and Consensus

Two-Phase Commit (2PC)

- 2PC uses a new component that does not normally appear in single-node transactions: a coordinator (also known as transaction manager).
- The protocol contains two crucial "points of no return": when a participant votes "yes," it promises that it will definitely be able to commit later (although the coordinator may still choose to abort); and once the coordinator decides, that decision is irrevocable. Those promises ensure the atomicity of 2PC. (Single-node atomic commit lumps these two events into one: writing the commit record to the transaction log.)

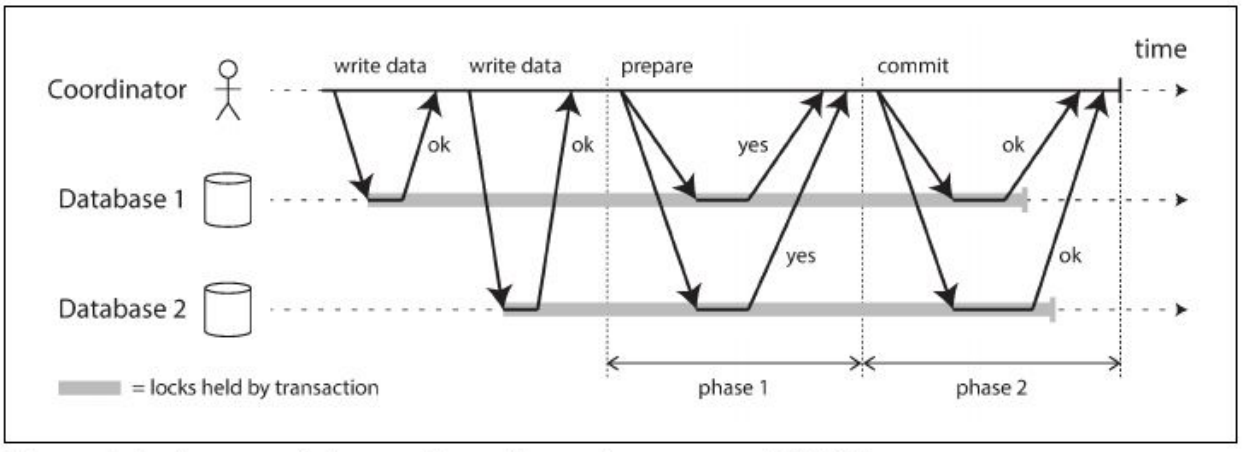


Figure 9-9. A successful execution of two-phase commit (2PC).

- The only way 2PC can complete is by waiting for the coordinator to recover. This is why the coordinator must write its commit or abort decision to a transaction log on disk before sending commit or abort requests to participants: when the coordinator recovers, it determines the status of all in-doubt transactions by reading its transaction log.

Three-phase commit

- Two-phase commit is called a blocking atomic commit protocol due to the fact that 2PC can become stuck waiting for the coordinator to recover.
- 3PC assumes a network with bounded delay and nodes with bounded response times; in most practical systems with unbounded network delay and process pauses(non-blocking commit requires a perfect failure detector).

Distributed Transactions in Practice

- Two types of distributed transactions are present :
    - Database-internal distributed transactions - Eg : distributed databases
    - Heterogeneous distributed transactions - different technologies
- Database-internal transactions do not have to be compatible with any other system,so they can use any protocol and apply optimizations specific to that particular technology.
- X/Open XA (short for eXtended Architecture) is a standard for implementing two-phase commit across heterogeneous technologies.
- Distributed transactions thus have a tendency of amplifying failures, which runs counter to our goal of building fault-tolerant systems.

Fault-Tolerant Consensus

- The consensus problem is normally formalized as follows: one or more nodes may propose values, and the consensus algorithm decides on one of those values.
- A consensus algorithm must satisfy the following properties :
    - Uniform agreement - No two nodes decide differently.
    - Integrity - No node decides twice.
    - Validity - If a node decides value v, then v was proposed by some node.
    - Termination - Every node that does not crash eventually decides some value.
- The termination property is subject to the assumption that fewer than half of the nodes are crashed or unreachable.
- Total order broadcast is equivalent to repeated rounds of consensus (each consensus decision corresponding to one message delivery)

Epoch numbering and quorums

- All of the consensus protocols discussed so far internally use a leader in some form or another, but they don’t guarantee that the leader is unique. Instead, they can make a weaker guarantee: the protocols define an epoch number and guarantee that within each epoch, the leader is unique.

- Every time the current leader is thought to be dead, a vote is started among the nodes to elect a new leader. This election is given an incremented epoch number, and thus epoch numbers are totally ordered and monotonically increasing. If there is a conflict between two different leaders in two different epochs (perhaps because the previous leader actually wasn't dead after all), then the leader with the higher epoch number prevails.
- The biggest differences are that in 2PC the coordinator is not elected, and that fault-tolerant consensus algorithms only require votes from a majority of nodes, whereas 2PC requires a "yes" vote from every participant.

Membership and Coordination Services

- Features that ZooKeeper provide that are useful in distributed system :
    - Linearizable atomic operations - lease
    - Total ordering of operations - fencing token
    - Failure detection - ephemeral nodes
    - Change notifications
- Allocating work to node :
    - Trying to perform majority votes over so many nodes would be terribly inefficient. Instead, 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. Thus, ZooKeeper provides a way of "outsourcing" some of the work of coordinating nodes (consensus, operation ordering, and failure detection) to an external service.
- Service Discovery :
    - ZooKeeper, etcd, and Consul are also often used for service discovery—that is, to find out which IP address you need to connect to in order to reach a particular service.
- Membership services :
    - A membership service determines which nodes are currently active and live members of a cluster.

Chapter 10 Batch Processing

Three different types of system :

1. Services (online systems) - A service waits for a request or instruction from a client to arrive. When one is received, the service tries to handle it as quickly as possible and sends a response back.
2. Batch processing systems (offline systems) - A batch processing system takes a large amount of input data, runs a job to process it, and produces some output data.Run periodically.
3. Stream Processing (near-real-time systems) - A stream job operates on events shortly after they happen(Low Latency).

Batch Processing with Unix Tools

The simple chain of Unix commands we saw earlier easily scales to large datasets, without running out of memory. The bottleneck is likely to be the rate at which the input file can be read from disk.

The Unix Philosophy

- A uniform interface :
    - If you want to be able to connect any program's output to any program's input, that means that all programs must use the same input/output interface.In Unix that interface is file)
- Separation of logic and wiring
- Transparency and experimentation
- Even though Unix tools are quite blunt, simple tools compared to a query optimizer of a relational database, they remain amazingly useful, especially for experimentation.
- However, the biggest limitation of Unix tools is that they run only on a single machine, and that's where tools like Hadoop come in.

MapReduce and Distributed Filesystems

- HDFS is based on the shared-nothing principle.
- HDFS consists of a daemon process running on each machine, exposing a network service that allows other nodes to access files stored on that machine
- A central server called the NameNode keeps track of which file blocks are stored on which machine. Thus, HDFS conceptually creates one big filesystem that can use the space on the disks of all machines running the daemon.
- The main difference from pipelines of Unix commands is that MapReduce can parallelize a computation across many machines, without you having to write code to explicitly handle the parallelism.
- Putting the computation near the data : it saves copying the input file over the network, reducing network load and increasing locality.
- It is very common for MapReduce jobs to be chained together into workflows, such that the output of one job becomes the input to the next job.

Reduce-Side Joins and Grouping

- When we talk about joins in the context of batch processing, we mean resolving all occurrences of some association within a dataset. For example, we assume that a job is processing the data for all users simultaneously, not merely looking up the data for one particular user (which would be done far more efficiently with an index).

- Instead of making random requests for data over the network, it is better to load a copy of the data locally.

Sort-merge joins

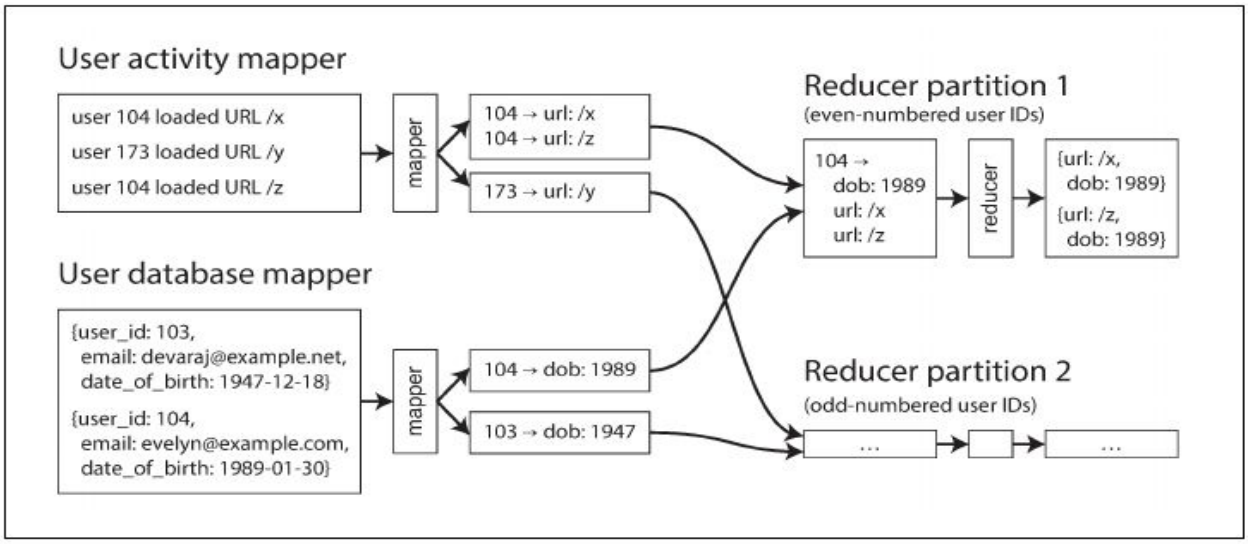


Figure 10-3. A reduce-side sort-merge join on user ID. If the input datasets are partitioned into multiple files, each could be processed with multiple mappers in parallel.

- Since the reducer processes all of the records for a particular user ID in one go, it only needs to keep one user record in memory at any one time, and it never needs to make any requests over the network. This algorithm is known as a sort-merge join, since mapper output is sorted by key, and the reducers then merge together the sorted lists of records from both sides of the join.

Bringing related data together in the same place

- In a sort-merge join, the mappers and the sorting process make sure that all the necessary data to perform the join operation for a particular user ID is brought together in the same place: a single call to the reducer.

GROUP BY

- The simplest way of implementing such a grouping operation with MapReduce is to set up the mappers so that the key-value pairs they produce use the desired grouping key. The partitioning and sorting process then brings together all the records with the same key in the same reducer. Thus, grouping and joining look quite similar when implemented on top of MapReduce.

Handling skew

- The pattern of “bringing all records with the same key to the same place” breaks down if there is a very large amount of data related to a single key.The skewed join method in Pig first runs a sampling job to determine which keys are hot. When performing the actual join, the mappers send any records relating to a hot key to one of several reducers, chosen at random (in contrast to conventional MapReduce, which chooses a reducer deterministically based on a hash of the key).

Map-Side Joins

- If you can make certain assumptions about your input data, it is possible to make joins faster by using a so-called map-side join.

Broadcast hash joins

- The simplest way of performing a map-side join applies in the case where a large dataset is joined with a small dataset(to fit into memory).
- This simple but effective algorithm is called a broadcast hash join: the word broadcast reflects the fact that each mapper for a partition of the large input reads the entirety of the small input and the word hash reflects its use of a hash table.

Partitioned hash joins

- If the inputs to the map-side join are partitioned in the same way, then the hash join approach can be applied to each partition independently.
- For example, mapper 3 first loads all users with an ID ending in 3 into a hash table, and then scans over all the activity events for each user whose ID ends in 3.

Map-side merge joins

- Another variant of a map-side join applies if the input datasets are not only partitioned in the same way, but also sorted based on the same key. In this case, it does not matter whether the inputs are small enough to fit in memory, because a mapper can perform the same merging operation that would normally be done by a reducer: reading both input files incrementally.

The Output of Batch Workflows

- Building Search Indexes
- Key-value stores as batch process output

Comparing Hadoop to Distributed Databases

- Diversity of storage
    - Hadoop opened up the possibility of indiscriminately dumping data into HDFS, and only later figuring out how to process it further. By contrast, MPP databases typically require careful up-front modeling of the data and query patterns before importing the data into the database's proprietary storage format.Thus, Hadoop has often been used for implementing ETL processes.
- Diversity of processing models
    - The Hadoop ecosystem includes both random-access OLTP databases such as HBase and MPP-style analytic databases such as Impala.Neither of them use Map-Reduce but use HDFS,but they can coexist and be integrated in the same system.
- Designing for frequent faults

Beyond MapReduce

Materialization of Intermediate State

MapReduce's approach of fully materializing intermediate state has downsides compared to Unix pipes:

- A MapReduce job can only start when all tasks in the preceding jobs (that generate its inputs) have completed.
- Mappers are often redundant
- Storing intermediate state in a distributed filesystem means those files are replicated across several nodes, which is often overkill for such temporary data.

Dataflow engines :

- They handle an entire workflow as one job, rather than breaking it up into independent subjobs.Since they explicitly model the flow of data through several processing stages, these systems are known as dataflow engines.
- You can use dataflow engines to implement the same computations as MapReduce workflows, and they usually execute significantly faster due to the optimizations.

Graphs and Iterative Processing

The Pregel processing model

- In Pregel: one vertex can "send a message" to another vertex, and typically those messages are sent along the edges in a graph.
- The difference from MapReduce is that in the Pregel model, a vertex remembers its state in memory from one iteration to the next, so the function only needs to process new incoming messages.
- Fault tolerance
- Parallel execution

High-Level APIs and Languages

- Besides the obvious advantage of requiring less code, these high-level interfaces also allow interactive use, in which you write analysis code incrementally in a shell and run it frequently to observe what it is doing.

Chapter 11 Stream Processing

Transmitting Event Streams

- In a stream processing context, a record is more commonly known as an event, but it is essentially the same thing: a small, self-contained, immutable object containing the details of something that happened at some point in time.
- In batch processing, a file is written once and then potentially read by multiple jobs. Analogously, in streaming terminology, an event is generated once by a producer (also known as a publisher or sender), and then potentially processed by multiple consumers (subscribers or recipients).

Tools for the purpose of delivering event notifications :

Messaging Systems

- A common approach for notifying consumers about new events is to use a messaging system: a producer sends a message containing the event, which is then pushed to consumers.
- To differentiate the systems, ask these two questions :
    - What happens if the producers send messages faster than the consumers can process them? Broadly speaking, there are three options: the system can drop messages, buffer messages in a queue, or apply backpressure (also known as flow control; i.e., blocking the producer from sending more messages).
    - What happens if nodes crash or temporarily go offline—are any messages lost? As with databases, durability may require some combination of writing to disk and/or replication.
- Although the direct messaging systems(ZeroMQ,HTTP or RPC) work well in the situations for which they are designed, they generally require the application code to be aware of the possibility of message loss. The faults they can tolerate are quite limited.
- A widely used alternative is to send messages via a message broker (also known as a message queue), which is essentially a kind of database that is optimized for handling message streams.A consequence of queueing is also that consumers are generally asynchronous: when a producer sends a message, it normally only waits for the broker to confirm that it has buffered the message and does not wait for the message to be processed by consumers.

Multiple consumers

- When multiple consumers read messages in the same topic, two main patterns of messaging are used:

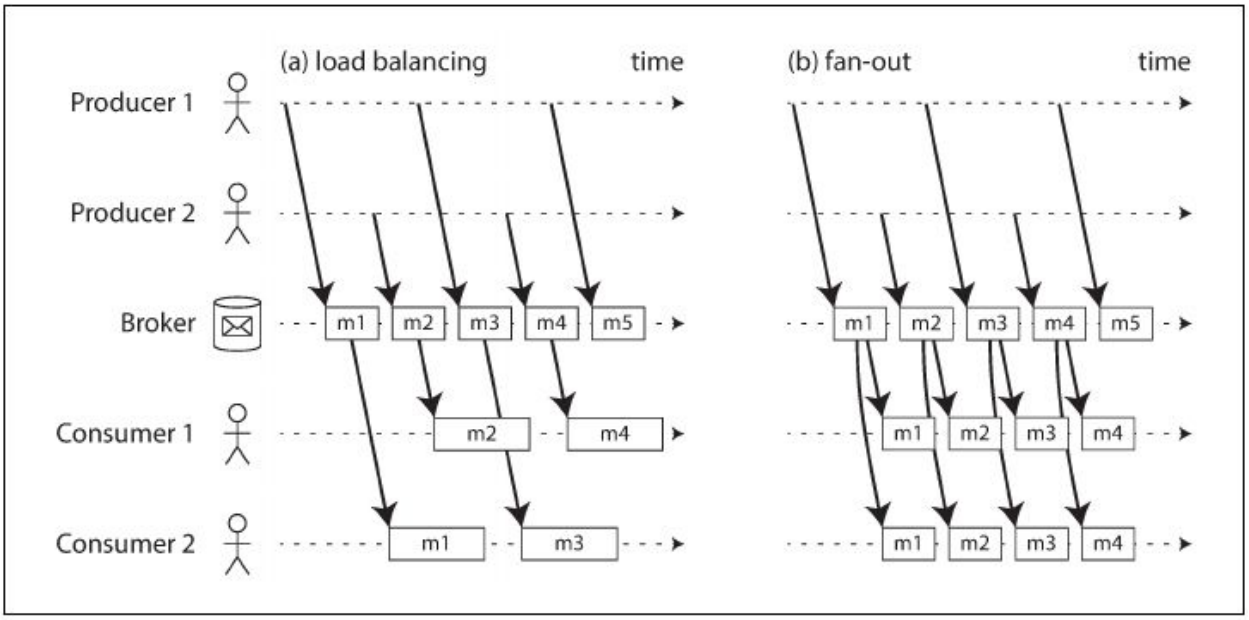


Figure 11-1. (a) Load balancing: sharing the work of consuming a topic among consumers; (b) fan-out: delivering each message to multiple consumers.

- The two patterns can be combined: for example, two separate groups of consumers may each subscribe to a topic, such that each group collectively receives all messages, but within each group only one of the nodes receives each message.

Acknowledgments and redelivery

- In order to ensure that the message is not lost, message brokers use acknowledgments: a client must explicitly tell the broker when it has finished processing a message so that the broker can remove it from the queue.

Partitioned Logs

- The idea behind log-based message brokers is combining the durable storage approach of databases with the low-latency facilities of messaging.

Using logs for message storage

- In order to scale to higher throughput than a single disk can offer, the log can be partitioned. Different partitions can then be hosted on different machines, making each partition a separate log that can be read and written independently from other partitions. A topic can then be defined as a group of partitions that all carry messages of the same type(Eg : Apache Kafka,Twitter's Distributed Log).

- In situations where messages may be expensive to process and you want to parallelize processing on a message-by-message basis, and where message ordering is not so important, the JMS/AMQP style of message broker is preferable. On the other hand, in situations with high message throughput, where each message is fast to process and where message ordering is important, the log-based approach works very well.

Consumer Offsets

- Consuming a partition sequentially makes it easy to tell which messages have been processed: all messages with an offset less than a consumer's current offset have already been processed, and all messages with a greater offset have not yet been seen.
- Thus, the broker does not need to track acknowledgments for every single message— it only needs to periodically record the consumer offsets. The reduced bookkeeping overhead and the opportunities for batching and pipelining in this approach help increase the throughput of log-based systems

Disk Space usage

- If you only ever append to the log, you will eventually run out of disk space. To reclaim disk space, the log is actually divided into segments, and from time to time old segments are deleted or moved to archive storage.
- You can monitor how far a consumer is behind the head of the log, and raise an alert if it falls behind significantly.

Databases and Streams

We can also go in reverse: take ideas from messaging and streams, and apply them to databases.

Keeping Systems in Sync

- As the same or related data appears in several different places(database,search index,..), they need to be kept in sync with one another.
- If periodic full database dumps are too slow, an alternative that is sometimes used is dual writes.However,dual writes has some serious problems : race condition, concurrency, atomicity,etc.

Change Data Capture

- More recently, there has been growing interest in change data capture (CDC), which is the process of observing all data changes written to a database and extracting them in a form in which

they can be replicated to other systems.

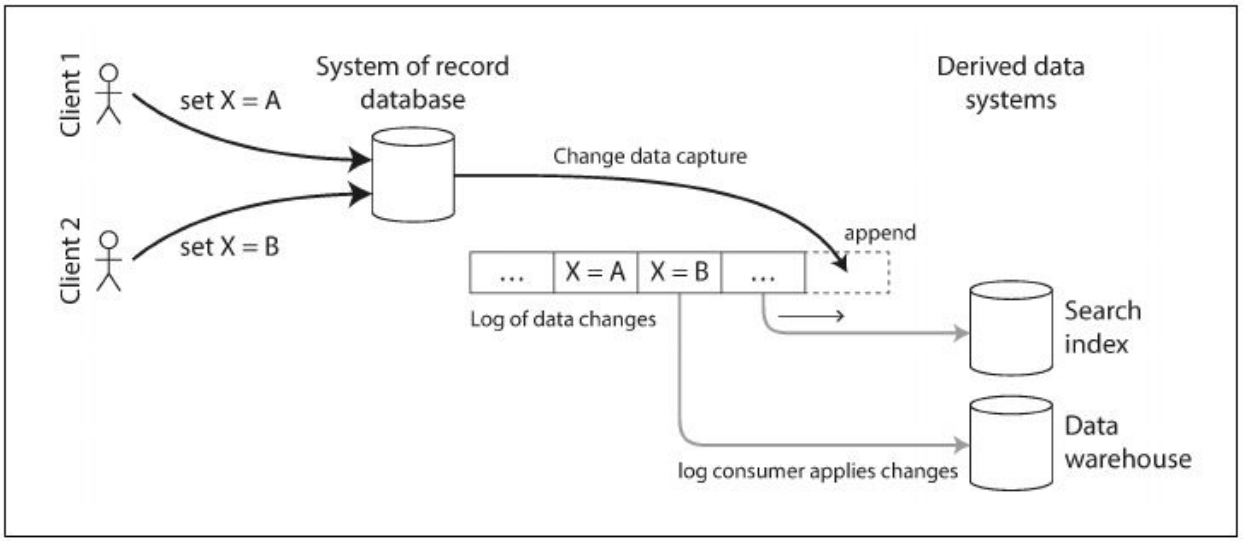


Figure 11-5. Taking data in the order it was written to one database, and applying the changes to other systems in the same order.

- Like message brokers, change data capture is usually asynchronous: the system of record database does not wait for the change to be applied to consumers before committing it.
- Increasingly, databases are beginning to support change streams as a first-class interface, rather than the typical retrofitted and reverse-engineered CDC efforts.
- Kafka Connect is an effort to integrate change data capture tools for a wide range of database systems with Kafka.

Event Sourcing

- Similarly to change data capture, event sourcing involves storing all changes to the application state as a log of change events.The biggest difference is that : In CDC the log of changes is extracted from the database at low level.In event sourcing events are designed to reflect things that happened at the application level, rather than low-level state changes.
- Event sourcing is a powerful technique for data modeling: from an application point of view it is more meaningful to record the user's actions as immutable events, rather than recording the effect of those actions on a mutable database.

Deriving current state from the event log

- An event typically expresses the intent of a user action, not the mechanics of the state update that occurred as a result of the action.Log compaction is not possible in the same way : you need the full history of events to reconstruct the final state.So some mechanism of snapshot is used.

State, Streams, and Immutability

- Mutable state and an append-only log of immutable events do not contradict each other: they are two sides of the same coin. The log of all changes, the changelog, represents the evolution of state over time.
- Immutable events also capture more information than just the current state. For example, on a shopping website, a customer may add an item to their cart and then remove it again. Although the second event cancels out the first event from the point of view of order fulfillment, it may be useful to know for analytics purposes that the customer was considering a particular item but then decided against it.
- Moreover, by separating mutable state from the immutable event log, you can derive several different read-oriented representations from the same log of events. This works just like having multiple consumers of a stream.
- You gain a lot of flexibility by separating the form in which data is written from the form it is read, and by allowing several different read views. This idea is sometimes known as command query responsibility segregation (CQRS).
- The biggest downside of event sourcing and change data capture is that the consumers of the event log are usually asynchronous, so there is a possibility that a user may make a write to the log, then read from a log-derived view and find that their write has not yet been reflected in the read view. We discussed this problem and potential solutions previously in “Reading Your Own Writes”.

Processing Streams

- Processing the stream usually means three options :
    1. Writing the events to another system.
    2. You can push the events to users in some way.
    3. To produce one or more output streams.

Uses of Stream Processing

- Stream processing has long been used for monitoring purposes, where an organization wants to be alerted if certain things happen(Eg : Fraud Detection).

Complex event processing

- CEP allows you to specify rules to search for certain patterns of events in a stream.
- Usually, a database stores data persistently and treats queries as transient: when a query comes in, the database searches for data matching the query, and then forgets about the query when it has finished. CEP engines reverse these roles: queries are stored long-term, and events from the input streams continuously flow past them in search of a query that matches an event pattern.

Stream analytics

- The boundary between CEP and stream analytics is blurry, but as a general rule, analytics tends to be less interested in finding specific event sequences and is more oriented toward aggregations and statistical metrics over a large number of events.
- The time interval over which you aggregate is known as a window.

Maintaining materialized views

Search on streams

Reasoning About Time

Event time versus processing time

- Message delays can also lead to unpredictable ordering of messages. For example, say a user first makes one web request (which is handled by web server A), and then a second request (which is handled by server B). A and B emit events describing the requests they handled, but B's event reaches the message broker before A's event does. Now stream processors will first see the B event and then the A event, even though they actually occurred in the opposite order.

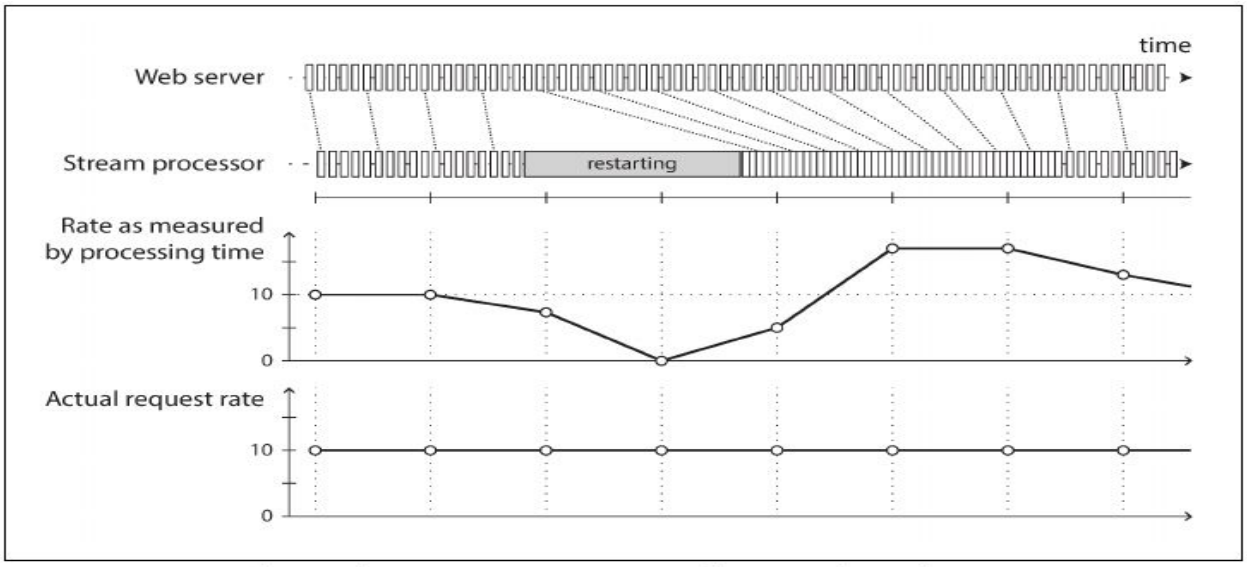


Figure 11-7. Windowing by processing time introduces artifacts due to variations in processing rate.

Knowing when you're ready

To deal with straggler events there are two options : Ignore the straggler events or Publish a correction.

Whose clock are you using, anyway?

- Assigning timestamps to events is even more difficult when events can be buffered at several points in the system.
- In this context, the timestamp on the events should really be the time at which the user interaction occurred, according to the mobile device's local clock. However, the clock on a user-controlled device often cannot be trusted, as it may be accidentally or deliberately set to the wrong time.The time at which the event was received by the server (according to the server's clock) is more likely to be accurate, since the server is under your control, but less meaningful in terms of describing the user interaction.
- To adjust for incorrect device clocks, one approach is to log three timestamps :
    - The time at which the event occurred, according to the device clock.
    - The time at which the event was sent to the server, according to the device clock.
    - The time at which the event was received by the server, according to the server clock.
- By subtracting the second timestamp from the third, you can estimate the offset between the device clock and the server clock.You can then apply that offset to the event timestamp, and thus estimate the true time at which the event actually occurred.

Types of windows

Once you know how the timestamp of an event should be determined, the next step is to decide how windows over time periods should be defined.

- Tumbling window - A tumbling window has a fixed length, and every event belongs to exactly one window.
- Hopping window - A hopping window also has a fixed length, but allows windows to overlap in order to provide some smoothing.
- Sliding window - A sliding window contains all the events that occur within some interval of each other.
- Session window - Unlike the other window types, a session window has no fixed duration.

Stream Joins

The fact that new events can appear anytime on a stream makes joins on streams more challenging than in batch jobs.

Stream-stream join (window join)

- In order to measure search quality, you need accurate click-through rates, for which you need both the search events and the click events.
- To implement this type of join, a stream processor needs to maintain state: for example, all the events that occurred in the last hour, indexed by session ID. Whenever a search event or click event occurs, it is added to the appropriate index, and the stream processor also checks the other index to see if another event for the same session ID has already arrived. If there is a matching event, you emit an event saying which search result was clicked. If the search event expires

without you seeing a matching click event, you emit an event saying which search results were not clicked.

Stream-table join (stream enrichment)

- To perform this join(Eg: user activity as stream and user info as table), the stream process needs to look at one activity event at a time, look up the event's user ID in the database, and add the profile information to the activity event.
- The database lookup could be remote or loaded locally in stream processor.
- In the latter case, the local copy needs to be up to date.This issue can be solved by CDC.
- A stream-table join is actually very similar to a stream-stream join; the biggest difference is that for the table changelog stream, the join uses a window that reaches back to the "beginning of time"

Table-table join (materialized view maintenance)

- Eg : A timeline cache: a kind of per-user "inbox" to which tweets are written as they are sent, so that reading the timeline is a single lookup.
- The join of the streams corresponds directly to the join of the tables in that query. The timelines are effectively a cache of the result of this query, updated every time the underlying tables change.

Time-dependence of joins

- The three types of joins described here (stream-stream, stream-table, and table-table) have a lot in common: they all require the stream processor to maintain some state (search and click events, user profiles, or follower list) based on one join input, and query that state on messages from the other join input. The order of the events that maintain the state is important.
- If state changes over time, and you join with some state, what point in time do you use for the join.Eg : If a user updates their profile, which activity events are joined with the old profile (processed before the profile update), and which are joined with the new profile(processed after the profile update).
- Such time dependence can occur in many places.

Fault Tolerance

In stream processing,waiting until a task is finished before making its output visible is not an option, because a stream is infinite and so you can never finish processing it.

Microbatching and checkpointing

- One solution is to break the stream into small blocks, and treat each block like a miniature batch process. This approach is called microbatching.
- The batch size is typically around one second, which is the result of a performance compromise: smaller batches incur greater scheduling and coordination overhead, while larger batches mean a longer delay before results of the stream processor become visible.

- A variant approach, used in Apache Flink, is to periodically generate rolling checkpoints of state and write them to durable storage.

Atomic commit revisited

- In order to give the appearance of exactly-once processing in the presence of faults, we need to ensure that all outputs and side effects of processing an event take effect if and only if the processing is successful.

Idempotence

- An idempotent operation is one that you can perform multiple times, and it has the same effect as if you performed it only once.

Rebuilding state after a failure

- One option is to keep the state in a remote datastore and replicate it, although having to query a remote database for each individual message can be slow.
- An alternative is to keep state local to the stream processor, and replicate it periodically.
- In some cases, it may not even be necessary to replicate the state, because it can be rebuilt from the input streams.

Chapter 12 The Future of Data Systems

Data Integration

- Every piece of software, even a so-called “general-purpose” database, is designed for a particular usage pattern.
- Faced with this profusion of alternatives, the first challenge is then to figure out the mapping between the software products and the circumstances in which they are a good fit.
- Distributed transactions use atomic commit to ensure that changes take effect exactly once, while log-based systems are often based on deterministic retry and idempotence
- The outputs of batch and stream processes are derived datasets such as search indexes, materialized views, recommendations to show to users, aggregate metrics, and so on.
- Asynchrony is what makes systems based on event logs robust: it allows a fault in one part of the system to be contained locally, whereas distributed transactions abort if any one participant fails, so they tend to amplify failures by spreading them to the rest of the system.

The lambda architecture

- The core idea of the lambda architecture is that incoming data should be recorded by appending immutable events to an always-growing dataset, similarly to event sourcing.
- From these events, read-optimized views are derived. The lambda architecture proposes running two different systems in parallel: a batch processing system such as Hadoop MapReduce, and a separate stream processing system such as Storm.
- In the lambda approach, the stream processor consumes the events and quickly produces an approximate update to the view; the batch processor later consumes the same set of events and produces a corrected version of the derived view.
- Problems with lambda architecture :
    - Having to maintain the same logic to run both in a batch and in a stream processing framework is significant additional effort.
    - Although it is great to have the ability to reprocess the entire historical dataset, doing so frequently is expensive on large datasets.

Unbundling Databases

Composing Data Storage Technologies

- I speculate that there are two avenues by which different storage and processing tools can nevertheless be composed into a cohesive system:
    1. Federated databases: unifying reads
        - It is possible to provide a unified query interface to a wide variety of underlying storage engines and processing methods—an approach known as a federated database or polystore.
    2. Unbundled databases: unifying writes
        - Making it easier to reliably plug together storage systems (e.g., through change data capture and event logs) is like unbundling a database's index-maintenance features in a way that can synchronize writes across disparate technologies.
        - The unbundled approach follows the Unix tradition of small tools that do one thing well, that communicate through a uniform low-level API (pipes), and that can be composed using a higher-level language (the shell).

Making unbundling work

- Federation and unbundling are two sides of the same coin: composing a reliable, scalable, and maintainable system out of diverse components.
- Transactions within a single storage or stream processing system are feasible, but when data crosses the boundary between different technologies, I believe that an asynchronous event log with idempotent writes is a much more robust and practical approach.
- The big advantage of log-based integration is loose coupling between the various components, which manifests itself in two ways:

    - At a system level, asynchronous event streams make the system as a whole more robust to outages or performance degradation of individual components.
    - At a human level, unbundling data systems allows different software components and services to be developed, improved, and maintained independently from each other by different teams.
- The goal of unbundling is not to compete with individual databases on performance for particular workloads; the goal is to allow you to combine several different databases in order to achieve good performance for a much wider range of workloads than is possible with a single piece of software. It's about breadth, not depth.

What's missing?

- We don't yet have the unbundled-database equivalent of the Unix shell.For example, I would love it if we could simply declare mysql | elasticsearch, by analogy to Unix pipes, which would be the unbundled equivalent of CREATE INDEX: it would take all the documents in a MySQL database and index them in an Elasticsearch cluster.
- Similarly, it would be great to be able to precompute and update caches more easily.

Designing Applications Around Dataflow

- The approach of unbundling databases by composing specialized storage and processing systems with application code is also becoming known as the "database inside-out".
- When a record in a database changes, we want any index for that record to be automatically updated, and any cached views or aggregations that depend on the record to be automatically refreshed.

Separation of application code and state

- Most web applications today are deployed as stateless services, in which any user request can be routed to any application server, and the server forgets everything about the request once it has sent the response.
- The trend has been to keep stateless application logic separate from state management (databases).
- Application code responds to state changes in one place by triggering state changes in another place.
- Unbundling the database means taking this idea and applying it to the creation of derived datasets outside of the primary database: caches, full-text search indexes, machine learning, or analytics systems. We can use stream processing and messaging systems for this purpose.The important thing to keep in mind is that maintaining derived data is not the same as asynchronous job execution, for which messaging systems are traditionally designed.
- This application code can do the arbitrary processing that built-in derivation functions in databases generally don't provide. Like Unix tools chained by pipes, stream operators can be composed to build large systems around dataflow. Each operator takes streams of state changes as input, and produces other streams of state changes as output.

Stream processors and services

- Composing stream operators into dataflow systems has a lot of similar characteristics to the microservices approach. However, the underlying communication mecha nism is very different: one-directional, asynchronous message streams rather than synchronous request/response interactions.
- For example, say a customer is purchasing an item that is priced in one currency but paid for in another currency. In order to perform the currency conversion, you need to know the current exchange rate.This operation could be implemented in two ways:
    - In the microservices approach, the code that processes the purchase would probably query an exchange-rate service or database in order to obtain the current rate for a particular currency.
    - In the dataflow approach, the code that processes purchases would subscribe to a stream of exchange rate updates ahead of time, and record the current rate in a local database whenever it changes. When it comes to processing the purchase, it only needs to query the local database.
- Not only is the dataflow approach faster, but it is also more robust to the failure of another service.

Observing Derived State

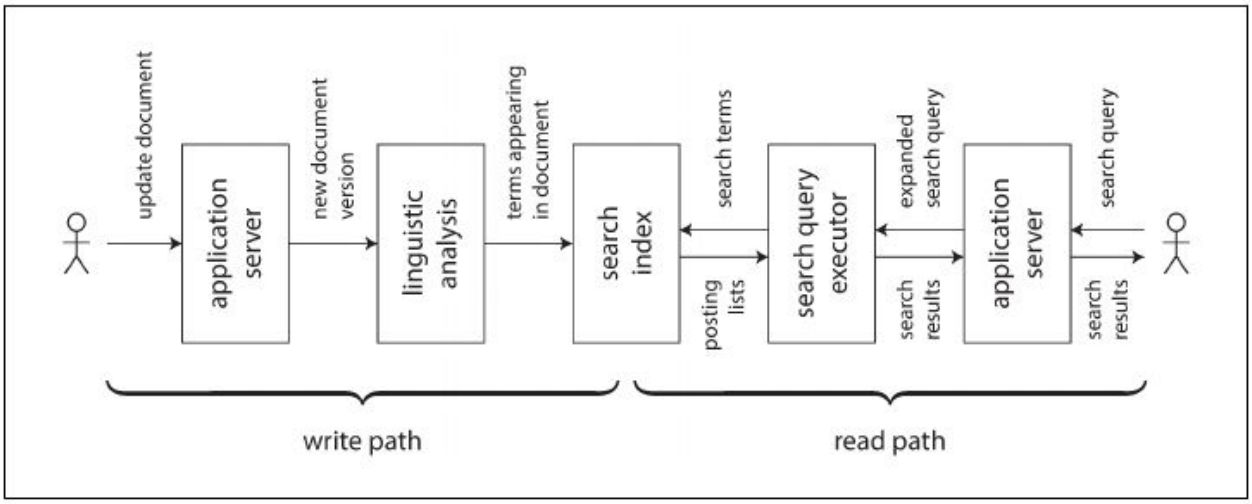


Figure 12-1. In a search index, writes (document updates) meet reads (queries).

- Taken together, the write path and the read path encompass the whole journey of the data, from the point where it is collected to the point where it is consumed.
- The write path is similar to eager evaluation, and the read path is similar to lazy evaluation.
- The derived dataset is the place where the write path and the read path meet.It represents the tradeoff between the amount of work that needs to be done at write time and the amount that needs to be done at read time.

Stateful, offline-capable clients

- In offline-first applications that do as much as possible using a local database on the same device, without requiring an internet connection, and sync with remote servers in the background when a network connection is available.

Pushing state changes to clients

- In terms of our model of write path and read path, actively pushing state changes all the way to client devices means extending the write path all the way to the end user.
- When a client is first initialized, it would still need to use a read path to get its initial state, but thereafter it could rely on a stream of state changes sent by the server.

End-to-end event streams

- Facebook's toolchain of React, Flux, and Redux, already manage internal client-side state by subscribing to a stream of events representing user input or responses from a server, structured similarly to event sourcing.
- It would be very natural to extend this programming model to also allow a server to push state-change events into this client-side event pipeline.
- Some applications, such as instant messaging and online games, already have such a "real-time" architecture.
- In order to extend the write path all the way to the end user, we would need to fundamentally rethink the way we build many of these systems: moving away from request/ response interaction and toward publish/subscribe dataflow.

Reads are events too

- It is also possible to represent read requests as streams of events, and send both the read events and the write events through a stream processor; the processor responds to read events by emitting the result of the read to an output stream.
- When both the writes and the reads are represented as events, and routed to the same stream operator in order to be handled, we are in fact performing a stream-table join between the stream of read queries and the database.

Aiming for Correctness

The End-to-End Argument for Databases

- Exactly-once execution of an operation
- Duplicate suppression
- Operation identifiers
    - To make the operation idempotent through several hops of network communication, it is not sufficient to rely just on a transaction mechanism provided by a database— you need

to consider the end-to-end flow of the request. For example, you could generate a unique identifier for an operation (such as a UUID) and include it as a hidden form field in the client application, or calculate a hash of all the relevant form fields to derive the operation ID. If the web browser submits the POST request twice, the two requests will have the same operation ID.

- The end-to-end argument(Sometimes an incomplete version of the function provided by the communication system may be useful as a performance enhancement.)
    - We just need to remember that the low-level reliability features are not by themselves sufficient to ensure end-to-end correctness.

Timeliness and Integrity

- More generally, I think the term consistency conflates two different requirements that are worth considering separately:
    - Timeliness
        - Timeliness means ensuring that users observe the system in an up-to-date state.
        - The CAP theorem (see “The Cost of Linearizability” on page 335) uses consistency in the sense of linearizability, which is a strong way of achieving timeliness. Weaker timeliness properties like read-after-write consistency.
    - Integrity
        - Integrity means absence of corruption; i.e., no data loss, and no contradictory or false data.
- Violations of timeliness are “eventual consistency,” whereas violations of integrity are “perpetual inconsistency.”
- I am going to assert that in most applications, integrity is much more important than timeliness. Violations of timeliness can be annoying and confusing, but violations of integrity can be catastrophic.

Loosely interpreted constraints

- However, another thing to realize is that many real applications can actually get away with much weaker notions of uniqueness : If two people concurrently register the same username or book the same seat, you can send one of them a message to apologize, and ask them to choose a different one. This kind of change to correct a mistake is called a compensating transaction.
- In many business contexts, it is actually acceptable to temporarily violate a constraint and fix it up later by apologizing.

Coordination-avoiding data systems

- Dataflow systems can maintain integrity guarantees on derived data without atomic commit, linearizability, or synchronous cross-partition coordination.
- Although strict uniqueness constraints require timeliness and coordination, many applications are actually fine with loose constraints that may be temporarily violated and fixed up later, as long as integrity is preserved throughout.

- Such coordination-avoiding data systems have a lot of appeal: they can achieve better performance and fault tolerance than systems that need to perform synchronous coordination.
- Another way of looking at coordination and constraints: they reduce the number of apologies you have to make due to inconsistencies, but potentially also reduce the performance and availability of your system, and thus potentially increase the number of apologies you have to make due to outages. You cannot reduce the number of apologies to zero, but you can aim to find the best trade-off for your needs.

Trust, but Verify

- Checking the integrity of data is known as auditing.For example, large-scale storage systems such as HDFS and Amazon S3 do not fully trust disks: they run background processes that continu ally read back files, compare them to other replicas, and move files from one disk to another, in order to mitigate the risk of silent corruption.
- Don't just blindly trust that it is all working.

Designing for auditability

- Event-based systems can provide better auditability. In the event sourcing approach, user input to the system is represented as a single immutable event, and any resulting state updates are derived from that event.
- Having continuous end-to-end integrity checks gives you increased confidence about the correctness of your systems, which in turn allows you to move faster.
- Cryptographic auditing and integrity checking often relies on Merkle trees, which are trees of hashes that can be used to efficiently prove that a record appears in some dataset (and a few other things). Outside of the hype of cryptocurrencies, certificate transparency is a security technology that relies on Merkle trees to check the validity of TLS/SSL certificates.

Doing the Right Thing

- A technology is not good or bad in itself—what matters is how it is used and how it affects people.

Predictive Analytics

- As data-driven decision making becomes more widespread, we will need to figure out how to make algorithms accountable and transparent, how to avoid reinforcing existing biases, and how to fix them when they inevitably make mistakes.

Feedback loops

- When services become good at predicting what content users want to see, they may end up showing people only opinions they already agree with, leading to echo chambers in which stereotypes, misinformation, and polarization can breed.

- When predictive analytics affect people's lives, particularly pernicious problems arise due to self-reinforcing feedback loops.

Privacy and Tracking

- The user is given a free service and is coaxed into engaging with it as much as possible. The tracking of the user serves not primarily that individual, but rather the needs of the advertisers who are funding the service.
- When surveillance is used to determine things that hold sway over important aspects of life, such as insurance coverage or employment, it starts to appear less benign.
- When surveillance is used to determine things that hold sway over important aspects of life, such as insurance coverage or employment, it starts to appear less benign. Moreover, data analysis can reveal surprisingly intrusive things: for example, the movement sensor in a smartwatch or fitness tracker can be used to work out what you are typing.
- Having privacy does not mean keeping everything secret; it means having the freedom to choose which things to reveal to whom, what to make public, and what to keep secret.
- When data is extracted from people through surveillance infrastructure, privacy rights are not necessarily eroded, but rather transferred to the data collector. Companies that acquire data essentially say "trust us to do the right thing with your data," which means that the right to decide what to reveal and what to keep secret is transferred from the individual to the company.

As software and data are having such a large impact on the world, we engineers must remember that we carry a responsibility to work toward the kind of world that we want to live in: a world that treats people with humanity and respect. I hope that we can work together toward that goal.