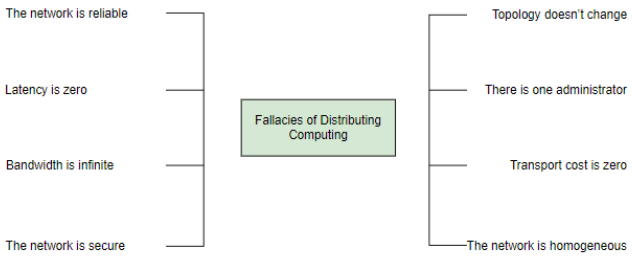
# Distributed Systems

http://csis.pace.edu/~marchese/CS865/Lectures/Chap1/Chapter1a.htm

A distributed system is a collection of nodes, that are programmable, asynchronous, autonomous, and failure-prone while communicating using an unreliable medium.

- The computers operate concurrently
- The computers fail independently
- The computers do not share a global clock(This 3rd condition means that our local system can not be considered as a distributed system even though it has separate concurrently running entities like GPU, CPU, memory, etc. because they share the common system clock)

There are eight such fallacies of distributed computing. The following illustration lists them.




False assumptions made by developers while developing software for distributed systems

## Difference between delivery and processing#

When we think about **exactly-once semantics**, it's useful to distinguish between the notions of delivery and processing.

In the context of the above discussion, let's consider **delivery** to be the arrival of the message at the destination node, at the hardware level.

Then, we consider **processing** to be the handling of this message from the software application layer of the node.

In most cases, we care more about how many times a node processes a message, than about how many times it delivers it. For instance, in our previous email example, we were mainly interested in whether the application would display the same email twice, and not whether it would receive it twice.

As the previous examples demonstrated, it's impossible to have *exactly-once delivery* in a distributed system. However, it's still sometimes possible to have *exactly-once processing*.

In the end, it's important for us to understand the difference between these two notions, and clarify what we refer to when we talk about exactly-once semantics.

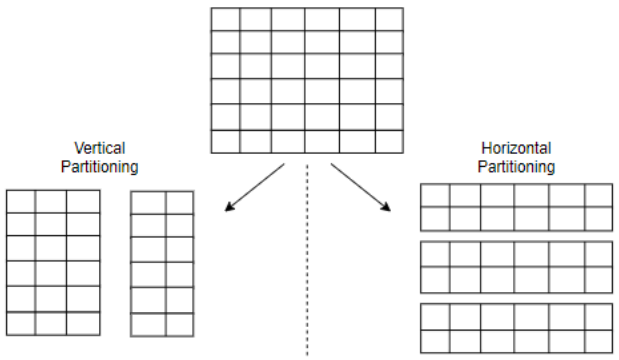
## Other delivery semantics#

As a last note, it's easy to see that we can easily implement **at-most-once** delivery semantics and **at-least-once** delivery semantics.

We can achieve the *at-most-once* delivery when we send every message only one time, no matter what happens. Meanwhile, we can achieve the *at-least-once* delivery when we send a message continuously until we get an acknowledgment from the recipient.

# Partitioning

- Range-based: A-B on node K…
- Hash-based: hash(an attribute) % no of nodes
- Consistent hashing




Vertical partitioning vs horizontal partitioning

## Replication

- Single master replication - A single master is responsible for all writes and propagates the writes to followers in sync or async way.
- Multi-master replication - All nodes are equal, there's no single master. Any node can handle the write and would sync all peers eagerly or lazily. If there's a conflict e.g. a node receives a stale value of the same key from another node, it can always be reconciled and fixed either eagerly or lazily - the client can tell which version is latest(e.g. Shopping cart), last write wins, any other conflict resolution algorithm.

## Consistency models

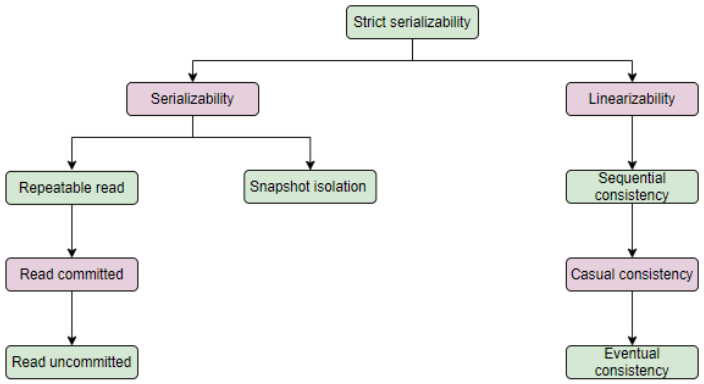
- Linearizability - When we use the asynchronous replication technique, we make the system linearizable. However, that means that the first write operation takes longer until the new value has propagated to the rest of the nodes. Remember the latency-consistency trade-off from the PACELC theorem. The client always reads the last written value.
- Sequential - For example, in a social networking application, we usually do not care what's the ordering of posts between some of our friends. However, we still expect posts from a single friend to be displayed in the right order (i.e., the one they published them at).
- Causal - We may want to display comments out of chronological order if it means that every comment is displayed after the comment it replies to. This is expected since there is a cause-and-effect relationship between a comment and the comments that constitute replies to it.
- Eventual

## Anomalies

- Dirty Write - A dirty write occurs when a transaction overwrites a value that was previously written by another transaction that is still in-flight and has not been committed yet.
- Dirty Read - A dirty read occurs when a transaction reads a value that has been written by another transaction that has not yet been committed.
- Fuzzy or non-repeatable reads - A fuzzy or non-repeatable read occurs when a value is retrieved twice during a transaction (without it being updated in the same transaction), and the value is different.
- Phantom read - A phantom read occurs when a transaction does a predicate-based read, and another transaction writes or removes a data item matched by that predicate while the first transaction is still in flight. If that happens, then the first transaction might be acting again on stale data or inconsistent data.
- Lost update - A lost update occurs when two transactions read the same value and then try to update it to two different values. The end result is that one of the two updates survives, but the process executing the other update is not informed that its update did not take effect Thus it is called a lost update.
- Read skew - A read skew occurs when there are integrity constraints between two data items that seem to be violated because a transaction can only see partial results of another transaction.
- Write Skew - A write skew occurs when two transactions read the same data, but then modify disjoint sets of data e.g. two people removing themselves from the ROTA in two independent and concurrent transactions.

### Isolation levels and prevented anomalies

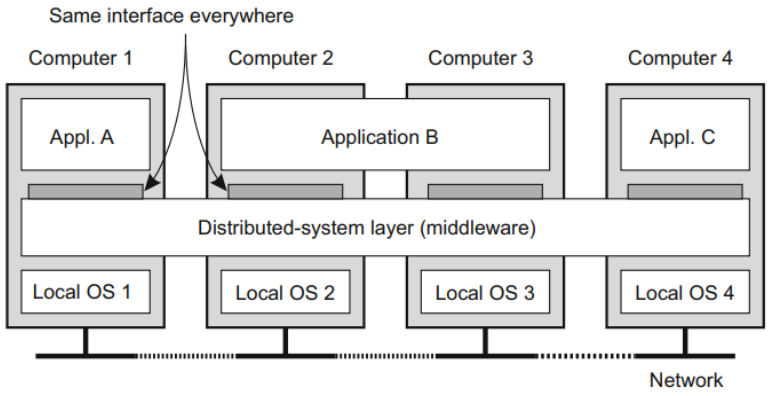
| | Dirty Writes | Dirty Reads | Fuzzy Reads | Phantom Reads | Lost Updates | Read Skew | Write Skew |
|---|---|---|---|---|---|---|---|
| Read Uncommitted | not possible | possible | possible | possible | possible | possible | possible |
| Read Committed | not possible | not possible | possible | possible | possible | possible | possible |
| Snapshot Isolation | not possible | not possible | not possible | possible | not possible | not possible | possible |
| Repeatable Read | not possible | not possible | not possible | possible | not possible | not possible | not possible |
| Serializability | not possible | not possible | not possible | not possible | not possible | not possible | not possible |




Hierarchy of consistency levels and isolation models, where the strictness level decreases from top to bottom

## Middleware

To assist the development of distributed applications, distributed systems are often organized to have a separate layer of software that is logically placed on top of the respective operating systems of the computers that are part of the system. This organization is leading to what is known as middleware. In a sense, middleware is the same to a distributed system as what an operating system is to a computer: a manager of resources offering its applications to efficiently share and deploy those resources across a network.




## Transparency

Transparency is some aspect of the distributed system that is hidden from the user (programmer, system developer, user, or application program). Transparency is provided by including some set of mechanisms in the distributed system at a layer below the interface where transparency is required. A number of basic transparencies have been defined for a distributed system. It is important to realize that not all of these are appropriate for every system, or are available at the same level of interface. In fact, all transparencies have an associated cost, and it is extremely important for the distributed system implementor to be aware of this.

**Table 1** Different forms of transparency in a distributed system (see ISO [31])

| Transparency | Description |
|---|---|
| Access | Hide differences in data representation and how an object is accessed |
| Location | Hide where an object is located |
| Relocation | Hide that an object may be moved to another location while in use |
| Migration | Hide that an object may move to another location |
| Replication | Hide that an object is replicated |
| Concurrency | Hide that an object may be shared by several independent users |
| Failure | Hide the failure and recovery of an object |

A data-intensive application is typically built from standard building blocks that provide the commonly needed functionality. For example, many applications need to:

- Store data so that they, or another application, can find it again later (databases)
- Remember the result of an expensive operation, to speed up reads (caches)
- Allow users to search data by keyword or filter it in various ways (search indexes)
- Send a message to another process, to be handled asynchronously (stream processing)
- Periodically crunch a large amount of accumulated data (batch processing)

# Reliability

- Typical expectations include:
- The application performs the function that the user expected.
- It can tolerate the user making mistakes or using the software in unexpected ways.
- Its performance is good enough for the required use case, under the expected load and data volume.
- The system prevents any unauthorized access and abuse.

The things that can go wrong are called faults, and systems that anticipate faults and can cope with them are called fault-tolerant or resilient. The former term is slightly misleading: it suggests that we could make a system tolerant of every possible kind of fault, which in reality is not feasible. So it only makes sense to talk about tolerating certain types of faults.
Note that a fault is not the same as a failure.
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 usually best to design fault-tolerance mechanisms that prevent faults from causing failures.

# Scalability

Scalability is the term we use to describe a system's ability to cope with increased load.

## Describing Performance

Once you have described the load on your system, you can investigate what happens when the load increases. You can look at it in two ways:

1. 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?
2. 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

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.
We, therefore, need to think of response time not as a single number, but as a distribution of values that you can measure.
Distributing load across multiple machines is also known as a shared-nothing architecture.

# Relational Versus Document Databases

If the data in your application has a document-like structure (i.e., a tree of one-to-many relationships, where typically the entire tree is loaded at once), then it's probably a good idea to use a document model. The relational technique of shredding—splitting a document-like structure into multiple tables (like positions, education, and contact_info)—can lead to cumbersome schemas and unnecessarily complicated application code.
The poor support for joins in document databases may or may not be a problem, depending on the application.
However, if your application does use many-to-many relationships, the document model becomes less appealing.
For highly interconnected data, the document model is awkward, the relational model is acceptable, and graph models are the most natural.
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.
The schema-on-read approach is advantageous if the items in the collection don't all have the same structure for some reason (i.e., the data is heterogeneous)

## Data locality for queries

A document is usually stored as a single continuous string, encoded as JSON, XML, or a binary variant thereof (such as MongoDB's BSON). If your application often needs to access the entire document(for example, to render it on a web page), there is a performance advantage to this storage locality.
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.

## Graph-Like Data Models

The relational model can handle simple cases of many-to-many relationships, but as the connections within your data become more complex, it becomes more natural to start modeling your data as a graph.

### 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)

# Storage and retrieval

## Hash Indexes

An important trade-off in storage systems: well-chosen indexes speed up read queries, but every index slows down writes.

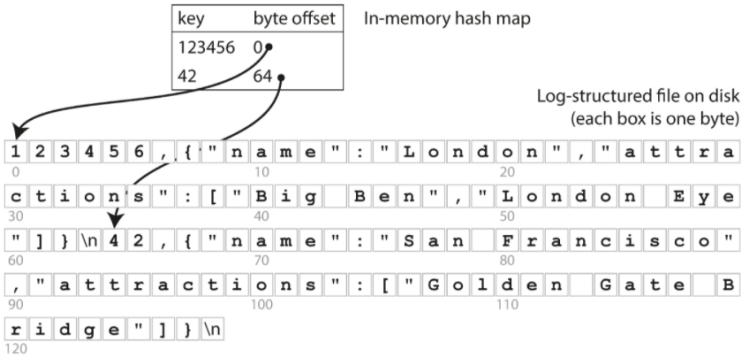



*Figure 3-1. Storing a log of key-value pairs in a CSV-like format, indexed with an in-memory hash map.*

This may sound simplistic, but it is a viable approach. In fact, this is essentially what Bitcask(the default storage engine in Riak) does. Bitcask offers high-performance reads and writes, subject to the requirement that all the keys fit in the available RAM since the hash map is kept completely in memory.
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, as illustrated in Figure 3-2. Compaction Means throwing away duplicate keys in the log and keeping only the most recent update for each key.
After the merging process is complete, we switch read requests to use the newly merged segment instead of the old segments—and then the old segment files can simply be deleted.
Each segment now has its own in-memory hash table, mapping keys to file offsets. In order to find the value for a key, we first check the most recent segment's hash map; if the key is not present we check the second-most-recent segment, and so on. The merging process keeps the number of segments small, so lookups don't need to check many hash maps.

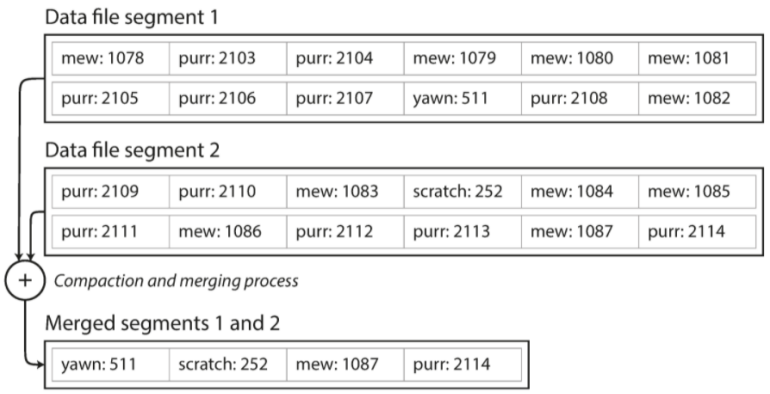



*Figure 3-3. Performing compaction and segment merging simultaneously.*

## Deleting records

If you want to delete a key and its associated value, you have to append a special deletion record to the data file (sometimes called a tombstone). When log segments are merged, the tombstone tells the merging process to discard any previous values for the deleted key.

## Crash recovery

If the database is restarted, the in-memory hash maps are lost. In principle, you can restore each segment's hashmap by reading the entire segment file from beginning to end and noting the offset of the most recent value for every key as you go along. However, that might take a long time if the segment files are large, which would make server restarts painful. Bitcask speeds up recovery by storing a snapshot of each segment's hashmap on disk, which can be loaded into memory more quickly.

## Partially written records

The database may crash at any time, including halfway through appending a record to the log. Bitcask files include checksums, allowing such corrupted parts of the log to be detected and ignored.

## Concurrency control

As writes are appended to the log in strictly sequential order, a common implementation choice is to have only one writer thread. Data file segments are append-only and otherwise immutable, so they can be read concurrently by multiple threads.

But an append-only design turns out to be good for several reasons:

- Appending and segment merging are sequential write operations, which are generally much faster than random writes, especially on magnetic spinning-disk hard drives. To some extent, sequential writes are also preferable on flash-based solid-state drives (SSDs)
- Concurrency and crash recovery are much simpler if segment files are append-only or immutable. For Example, you don't have to worry about the case where a crash happened while a value was being overwritten, leaving you with a file containing part of the old and part of the new value spliced together.
- Merging old segments avoids the problem of data files getting fragmented over time.

# SSTables and LSM-Trees

Now we can make a simple change to the format of our segment files: we require that the sequence of key-value pairs is sorted by key. We call this format Sorted String Table or SSTable for short.
SSTables have several big advantages over log segments with hash indexes:

- Merging segments is simple and efficient, even if the files are bigger than the available memory. The approach is like the one used in the merge sort algorithm, you start reading the input files side by side, look at the first key in each file, copy the lowest key (according to the sort order) to the output file, and repeat. This produces a new merged segment file, also sorted by key.
- In order to find a particular key in the file, you no longer need to keep an index of all the keys in memory. Say you're looking for the key handiwork, but you don't know the exact offset of that key in the segment file. However, you know the offsets for the keys handbag and handsome, and because of the sorting, you know that handiwork must appear between those two. This means you can jump to the offset for the handbag and scan from there until you find handiwork (or not, if the key is not present in the file). You still need an in-memory index to tell you the offsets for some of the keys, but it can be sparse: one key for every few kilobytes of segment file is sufficient because a few kilobytes can be scanned very quickly.

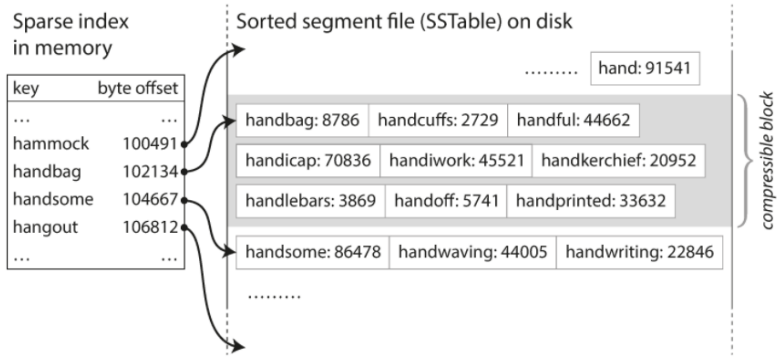



*Figure 3-5. An SSTable with an in-memory index.*

- 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. Each entry of the sparse in-memory index then points at the start of a compressed block. Besides saving disk space, compression also reduces the I/O bandwidth use.

## Constructing and maintaining SSTables

How do you get your data to be sorted by key in the first place? Our incoming writes can occur in any order. Maintaining a sorted structure on a disk is possible, but maintaining it in memory is much easier.
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 (for example, a red-black tree). This in-memory tree is sometimes called a memtable.
- When the memtable gets bigger than some threshold—typically a few megabytes—write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance.
- 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.

This scheme works very well. It only suffers from one problem: if the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost. In order to avoid that problem, we can keep a separate log on disk to which every write is immediately appended, just like in the previous section. 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.

## LSM storage engines

Originally this indexing structure was described by Patrick O'Neil et al. under the name Log-Structured Merge-Tree (or LSM-Tree), building on earlier work on log-structured file systems. Storage engines that are based on this principle of merging and compacting sorted files are often called LSM storage engines.

## Performance optimizations

- LSM-tree algorithm can be slow when looking up keys that do not exist in the database: you have to check the memtable, then the segments all the way back to the oldest (possibly having to read from disk for each one) before you can be sure that the key does not exist. In order to optimize this kind of access, storage engines often use additional Bloom filters.
- There are also different strategies to determine the order and timing of how SSTables are compacted and merged. The most common options are size-tiered and leveled compaction. LevelDB and RocksDBuse leveled compaction (hence the name of LevelDB), HBase uses size-tiered, and Cassandra supports both. In size-tiered compaction, newer and smaller SSTables are successively merged into older and larger SSTables. In leveled compaction, the key range is split up into smaller SSTables and older data is moved into separate "levels," which allows the compaction to proceed more incrementally and use less disk space.
- Even when the dataset is much bigger than the available memory it continues to work well. Since data is stored in sorted order, you can efficiently perform range queries (scanning all keys above some minimum and up to some maximum), and because the disk writes are sequential the LSM-tree can support remarkably high write throughput.

# B-Trees

They remain the standard index implementation in almost all relational databases, and many non-relational databases use them too. Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key-value lookup and range queries.
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. This design corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks. Each page can be identified using an address or location, which allows one page to refer to another—similar to a pointer, but on disk instead of in memory.
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 (any references to that page remain valid). If you want to add a new key, you need to find the page whose range encompasses the new key and add it to that page. If there isn't enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdivision of key ranges

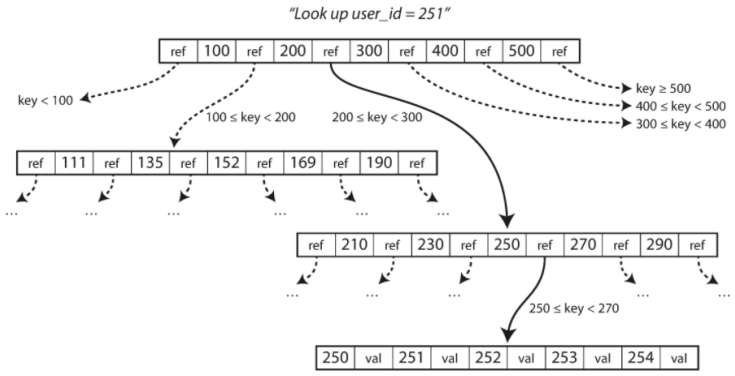



*Figure 3-6. Looking up a key using a B-tree index.*

This algorithm ensures that the tree remains balanced: a B-tree with n keys always has a depth of O(log n). 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.)

The basic underlying write operation of a B-tree is to overwrite a page on the disk with new data. This is in stark contrast to log-structured indexes such as LSM-trees, which only append to files (and eventually delete obsolete files) but never modify files in place. Moreover, some operations require several different pages to be overwritten. For example, if you split a page because an insertion caused it to be overfull, you need to write the two pages that were split, and also overwrite their parent page to update the references to the two-child pages. This is a dangerous operation because if the database crashes after only some of the pages have been written, you end up with a corrupted index (e.g., there may be an orphan page that is not a child of any parent).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. When the database comes back up after a crash, this log is used to restore the B-tree back to a consistent state. 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).

## 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.
- A B-tree index must write every piece of data at least twice: once to the write-ahead log, and once to the tree page itself (and perhaps again as pages are split). There is also overhead from having to write an entire page at a time, even if only a few bytes in that page changed.
- Moreover, LSM-trees are typically able to sustain higher write throughput than B-trees, partly because they sometimes have lower write amplification (although this depends on the storage engine configuration and workload), and partly because they sequentially write compact SSTable files rather than having to overwrite several pages in the tree.
- B-tree storage engines leave some disk space unused due to fragmentation: when a page is split or when a row cannot fit into an existing page, some space in a page remains unused. Since LSM-trees are not page-oriented and periodically rewrite SSTables to remove fragmentation, they have lower storage overheads, especially when using leveled compaction.
- A downside of log-structured storage is that the compaction process can sometimes interfere with the performance of ongoing reads and writes. Even though storage engines try to perform compaction incrementally and without affecting concurrent access, disks have limited resources, so it can easily happen that a request needs to wait while the disk finishes an expensive compaction operation.
- Another issue with compaction arises at high write throughput: the disk's finite write bandwidth needs to be shared between the initial write (logging and flushing a memtable to disk) and the compaction threads running in the background.
- 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 until you run out of disk space and reads also slow down because they need to check more segment files.

## 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 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.
The heap file approach is common because it avoids duplicating data when multiple secondary indexes are present: each index just references a location in the heap file, and the actual data is kept in one place.
When updating a value without changing the key, the heap file approach can be quite efficient: the record can be overwritten in place.
The situation is more complicated if the new value is larger, as it probably needs to be moved to a new location in the heap where there is enough space. In that case, either all indexes need to be updated to point at the new heap location of the record, or a forwarding pointer is left behind in the old heap location.

- Clustered index (storing all row data within the index)
- Non-clustered index (storing only references to the data within the index)

As with any kind of duplication of data, clustered indexes can speed up reads, but they require additional storage and can add overhead on writes.

Databases also need to go to additional effort to enforce transactional guarantees, because applications should not see inconsistencies due to the duplication.

## Multi-column indexes

Multi-dimensional indexes are a more general way of querying several columns at once. A standard B-tree or LSM-tree index is not able to answer that kind of query efficiently: it can give you either all the restaurants in a range of latitudes (but at any longitude), or all the restaurants in a range of longitudes (but anywhere between the North and South poles), but not both simultaneously. More commonly, specialized spatial indexes such as **R-trees** are used.

## Keeping everything in memory

As RAM becomes cheaper, the cost-per-gigabyte argument is eroded. Many datasets are simply not that big, so it's quite feasible to keep them entirely in memory, potentially distributed across several machines. This has led to the development of in-memory databases.
But other in-memory databases aim for durability, which can be achieved with special hardware (such as battery-powered RAM), by writing a log of changes to disk, by writing periodic snapshots to disk, or by replicating the in-memory state to other machines. When an in-memory database is restarted, it needs to reload its state, either from disk or over the network from a replica (unless special hardware is used). Despite writing to disk, it's still an in-memory database, because the disk is merely used as an append-only log for durability, and reads are served entirely from memory.
The performance advantage of in-memory databases is not due to the fact that they don't need to read from disk. Even a disk-based storage engine may never need to read from disk if you have enough memory because the operating system caches recently used disk blocks in memory anyway. Rather, they can be faster because they can avoid the overheads of encoding in-memory data structures in a form that can be written to disk.

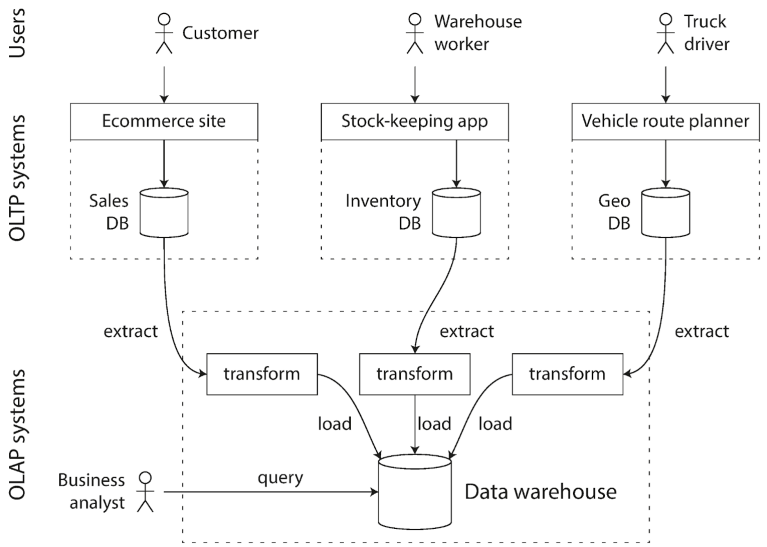
## Transaction Processing or Analytics?

*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 |

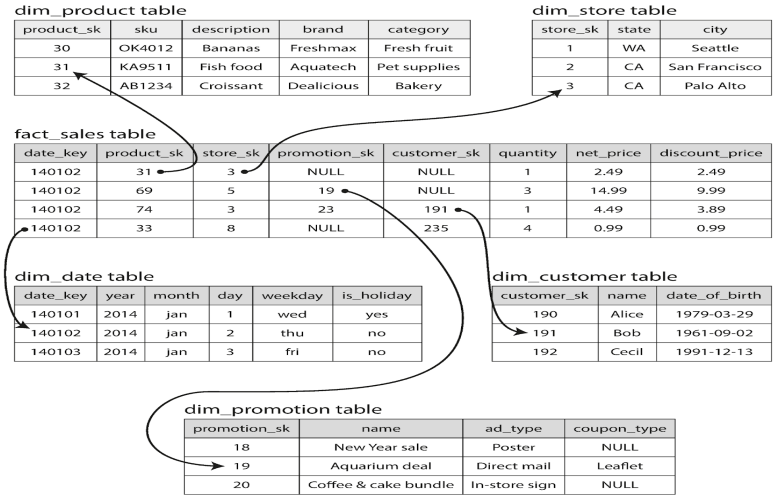
These OLTP systems are usually expected to be highly available and to process transactions with low latency, since they are often critical to the operation of the business. Database administrators therefore closely guard their OLTP databases. They are usually reluctant to let business analysts run ad hoc analytic queries on an OLTP database, since those queries are often expensive, scanning large parts of the dataset, which can harm the performance of concurrently executing transactions. The data warehouse contains a read-only copy of the data in all the various OLTP systems in the company.

On the surface, a data warehouse and a relational OLTP database look similar, because they both have a SQL query interface. However, the internals of the systems can look quite different, because they are optimized for very different query patterns. Many database vendors now focus on supporting either transaction processing or analytics workloads, but not both.




## Stars and Snowflakes: Schemas for Analytics

At the center of the schema is a so-called fact table. Each row of the fact table represents an event that occurred at a particular time(here, each row represents a customer's purchase of a product).
Some of the columns in the fact table are attributes, such as the price at which the product was sold and the cost of buying it from the supplier (allowing the profit margin to be calculated). Other columns in the fact table are foreign key references to other tables, called dimension tables. As each row in the fact table represents an event, the dimensions represent the who, what, where, when, how, and why of the event.
The name "star schema" comes from the fact that when the table relationships are visualized, the fact table is in the middle, surrounded by its dimension tables; the connections to these tables are like the rays of a star. A variation of this template is known as the snowflake schema, where dimensions are further broken down into subdimensions.



dim_product table

| product_sk | sku | description | brand | category |
|---|---|---|---|---|
| 30 | OK4012 | Bananas | Freshmax | Fresh fruit |
| 31 | KA9511 | Fish food | Aquatech | Pet supplies |
| 32 | AB1234 | Croissant | Dealicious | Bakery |

dim_store table

| store_sk | state | city |
|---|---|---|
| 1 | WA | Seattle |
| 2 | CA | San Francisco |
| 3 | CA | Palo Alto |

fact_sales table

| date_key | product_sk | store_sk | promotion_sk | customer_sk | quantity | net_price | discount_price |
|---|---|---|---|---|---|---|---|
| 140102 | 31 | 3 | NULL | NULL | 1 | 2.49 | 2.49 |
| 140102 | 69 | 5 | 19 | NULL | 3 | 14.99 | 9.99 |
| 140102 | 74 | 3 | 23 | 191 | 1 | 4.49 | 3.89 |
| 140102 | 33 | 8 | NULL | 235 | 4 | 0.99 | 0.99 |

dim_date table

| date_key | year | month | day | weekday | is_holiday |
|---|---|---|---|---|---|
| 140101 | 2014 | jan | 1 | wed | yes |
| 140102 | 2014 | jan | 2 | thu | no |
| 140103 | 2014 | jan | 3 | fri | no |

dim_customer table

| customer_sk | name | date_of_birth |
|---|---|---|
| 190 | Alice | 1979-03-29 |
| 191 | Bob | 1961-09-02 |
| 192 | Cecil | 1991-12-13 |

dim_promotion table

| promotion_sk | name | ad_type | coupon_type |
|---|---|---|---|
| 18 | New Year sale | Poster | NULL |
| 19 | Aquarium deal | Direct mail | Leaflet |
| 20 | Coffee & cake bundle | In-store sign | NULL |

# Column-Oriented Storage

A row-oriented storage engine still needs to load all of those rows from the disk into memory, parse them, and filter out those that don't meet the required conditions. That can take a long time.
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. If each column is stored in a separate file, a query only needs to read and parse those columns that are used in that query, which can save a lot of work.
The column-oriented storage layout relies on each column file containing the rows in the same order. Thus, if you need to reassemble an entire row, you can take the 23rd entry from each of the individual column files and put them together to form the 23rd row of the table.

## Column Compression

Column values often look quite repetitive, which is a good sign for compression. Often, the number of distinct values in a column is small compared to the number of rows. We can now take a column with n distinct values and compress them efficiently using different algorithms.
"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."
A big bottleneck is the bandwidth for getting data from disk into memory and finally to the CPU cache. 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. Column compression allows more rows from a column to fit in the same amount of L1 cache.

## 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 so the data needs to be sorted an entire row at a time, even though it is stored by column.
Another advantage of sorted order is that it can help with the compression of columns. If the primary sort column does not have many distinct values, then after sorting, it will have long sequences where the same value is repeated many times in a row. A simple run-length encoding could compress that column down to a few kilobytes—even if the table has billions of rows.

## 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.
Fortunately, we have already seen a good solution earlier in this chapter: LSM-trees. 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

In a relational data model, it is often defined like a standard (virtual) view: a table-like object whose contents are the results of some query. The difference is that 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 the underlying data changes, a materialized view needs to be updated, because it is a denormalized copy of the data. The database can do that automatically, but such updates make writes more expensive, which is why materialized views are not often used in OLTP databases. In read-heavy data warehouses, they can make more sense.

# Replication

There are several reasons why you might want to replicate data:

- To keep data geographically close to your users (and thus reduce access latency)
- To allow the system to continue working even if some of its parts have failed (and thus increase availability)
- To scale out the number of machines that can serve read queries (and thus increase read throughput)

Three popular algorithms for replicating changes between nodes: single-leader, multi-leader, and leaderless replication.

## Leaders and Followers

Each node that stores a copy of the database is called a replica. Every write to the database needs to be processed by every replica; otherwise, the replicas would no longer contain the same data.
The most common solution for this is called leader-based replication (also known as active/passive or master-slave replication).

- Leader (also known as master or primary)
- Followers (read replicas, slaves, secondaries, or hot standbys)

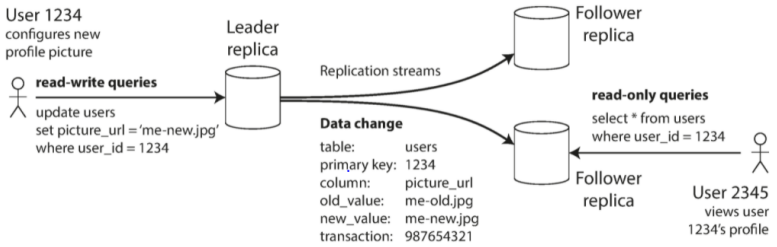



*Figure 5-1. Leader-based (master–slave) replication.*

# Synchronous Versus Asynchronous Replication

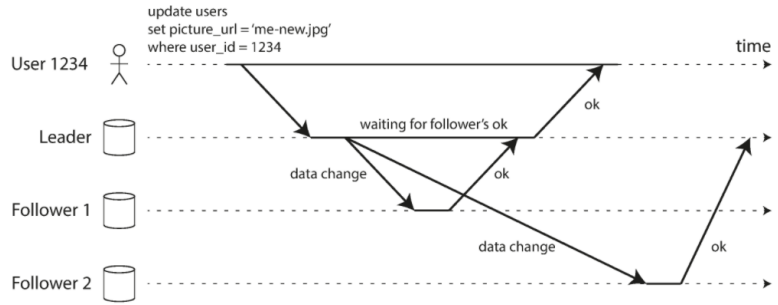



*Figure 5-2. Leader-based replication with one synchronous and one asynchronous follower.*

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.
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 impracticable for all followers to be synchronous. In practice, if you enable synchronous replication on a database, it usually means that one of the followers is synchronous, and the others are asynchronous. This configuration is sometimes also called semi-synchronous.

# Setting Up New Followers

Conceptually, the process looks like this:

- Take a consistent snapshot of the leader's database at some point in time—if possible, without taking a lock on the entire database. Most databases have this feature, as it is also required for backups.
- Copy the snapshot to the new follower node.
- 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. That position has various names: for example, PostgreSQL calls it the log sequence number, and MySQL calls it the binlog coordinates.
- When the follower has processed the backlog of data changes since the snapshot, we say it has caught up. It can now continue to process data changes from the leader as they happen.

# Handling Node Outages

## Follower failure: Catch-up recovery

On its local disk, each follower keeps a log of the data changes it has received from the leader. The follower can recover quite easily: from its log, it knows the last transaction that was processed before the fault occurred. Thus, 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

Handling a failure of the leader is trickier: one of the followers needs to be promoted to be the new leader, clients need to be reconfigured to send their writes to the new leader, and the other followers need to start consuming data changes from the new leader. This process is called failover.
An automatic failover process usually consists of the following steps:

- Determining that the leader has failed - Most systems simply use a timeout: nodes frequently bounce messages back and forth between each other, and if a node doesn't respond for some period of time—say, 30seconds—it is assumed to be dead.
- Choosing a new leader - The best candidate for leadership is usually the replica with the most up-to-date data changes from the old leader (to minimize any data loss). Getting all the nodes to agree on a new leader is a consensus problem.
- Reconfiguring the system to use the new leader - If the old leader comes back, it might still believe that it is the leader, not realizing that the other replicas have forced it to step down. 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 a split-brain. If both leaders accept writes, and there is no process for resolving conflicts, data is likely to be lost or corrupted. As a safety catch, some systems have a mechanism to shut down one node if two leaders are detected. However, if this mechanism is not carefully designed, you can end up with both nodes being shut down.
- What is the right timeout before the leader is declared dead? A longer timeout means a longer time to recover in the case where the leader fails. However, if the timeout is too short, there could be unnecessary failovers.

# Implementation of Replication Logs

How does leader-based replication work under the hood?

## 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. For a relational database, this means that every INSERT, UPDATE or DELETE statement is forwarded to followers, and each follower parses and executes the SQL statement as if it had been received from a client.
There are various ways in which this approach to replication can break down:

- Any statement that calls a nondeterministic function, such as NOW() to get the current date and time or RAND() to get a random number, is likely to generate a different value on each replica.
- If statements use an auto-incrementing column, or if they depend on the existing data into the database (e.g., UPDATE … WHERE <some condition>), they must be executed in exactly the same order on each replica, or else they may have a different effect. This can be limiting when there are multiple concurrently executing transactions.
- Statements that have side effects (e.g., triggers, stored procedures, user-defined functions) may result in different side effects occurring on each replica unless the side effects are absolutely deterministic.

## Write-ahead log (WAL) shipping

The log is an append-only sequence of bytes containing all writes to the database. We can use the exact same log to build a replica on another node: besides writing the log to disk, the leader also sends it across the network to its followers. When the follower processes this log, it builds a copy of the exact same data structures as found on the leader.
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

An alternative is to use different log formats for replication and for the storage engine, which allows the replication log to be decoupled from the storage engine internals. This kind of replication log is called a logical log, to distinguish it from the storage engine's (physical)data representation.
A logical log for a relational database is usually a sequence of records describing writes to database tables at the granularity of a row:

- For an inserted row, the log contains the new values of all columns.
- For a deleted row, the log contains enough information to uniquely identify the row that was deleted.
- For an updated row, the log contains enough information to uniquely identify the updated row and the new values of all columns (or at least the new values of all columns that changed).

A transaction that modifies several rows generates several such log records, followed by a record indicating that the transaction was committed.

Since a logical log is decoupled from the storage engine internals, it allows the leader and the follower to run different versions of the database software, or even different storage engines.
A logical log format is also easier for external applications to parse. This aspect is useful if you want to send the contents of a database to an external system, such as a data warehouse for offline analysis, or for building custom indexes and caches. This technique is called **change data capture**.

# Problems with Replication Lag

Leader-based replication requires all writes to go through a single node, but read-only queries can go to any replica. In this read-scaling architecture, you can increase the capacity for serving read-only requests simply by adding more followers.
If an application reads from an asynchronous follower, it may see outdated information if the follower has fallen behind.
This inconsistency is just a temporary state - the followers will eventually catch up and become consistent with the leader. For that reason, this effect is known as **eventual consistency**.

## Reading Your Own Writes

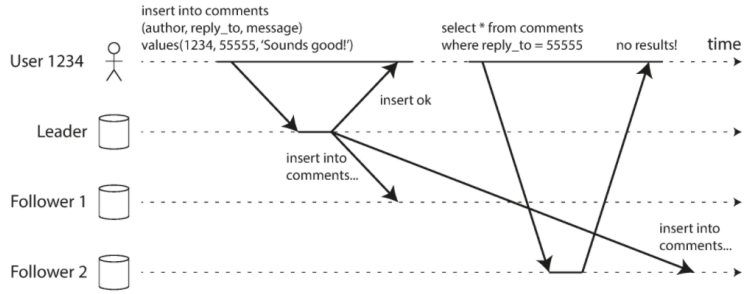



*Figure 5-3. A user makes a write, followed by a read from a stale replica. To prevent this anomaly, we need read-after-write consistency.*

In this situation, we need read-after-write consistency, also known as read-your-writes consistency. It makes no promises about other users: other users' updates may not be visible until some later time. However, it reassures the user that their own input has been saved correctly.
How can we implement read-after-write consistency in a system with leader-based replication?

- When reading something that the user may have modified, read it from the leader; otherwise, read it from a follower. For example, user profile information on a social network is normally only editable by the owner of the profile, not by anybody else. Thus, a simple rule is: always read the user's own profile from the leader and any other users' profiles from a follower.

- If most things in the application are potentially editable by the user, that approach won't be effective, as most things would have to be read from the leader (negating the benefit of read scaling). In that case, other criteria may be used to decide whether to read from the leader. For Example, 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.
- The client can remember the timestamp of its most recent write—then the system can ensure that the replica serving any reads for that user reflects updates at least until that timestamp. If the replica is not sufficiently up to date, either the read can be handled by another replica or the query can wait until the replica has caught up.
- If your replicas are distributed across multiple data centers (for geographical proximity to users or for availability), there is additional complexity. Any request that needs to be served by the leader must be routed to the data center that contains 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.
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, because the code running on one device doesn't know what updates have happened on the other device. This metadata will need to be centralized.
- If your replicas are distributed across different data centers, there is no guarantee that connections from different devices will be routed to the same data center. (For example, if the user's desktop computer uses the home broadband connection and their mobile device uses the cellular data network, the devices' network routes may be completely different.) If your approach requires reading from the leader, you may first need to route requests from all of a user's devices to the same data center.

## Monotonic Reads

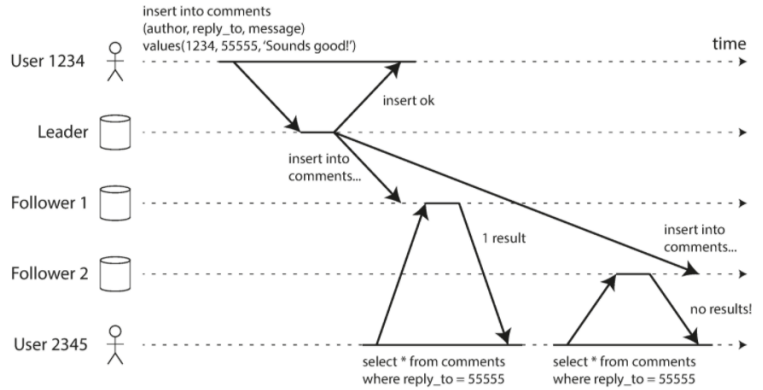



*Figure 5-4. A user first reads from a fresh replica, then from a stale replica. Time appears to go backward. To prevent this anomaly, we need monotonic reads.*

Monotonic reads is a lesser guarantee than strong consistency, but a stronger guarantee than eventual consistency. Monotonic reads mean that if one user makes several reads in sequence, they will not read older data after having previously read newer data.
One way of achieving monotonic reads is to make sure that each user always makes their reads from the same replica (For example, the replica can be chosen based on a hash of the user ID, rather than randomly).

## Consistent Prefix Reads

This 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.

# Multi-Leader Replication

Leader-based replication has one major downside: there is only one leader. If you can't connect to the leader for any reason, you can't write to the database.
A natural extension of the leader-based replication model is to allow more than one node to accept writes.
We call this a multi-leader configuration (also known as master-master or active/active replication). In this setup, each leader simultaneously acts as a follower to the other leaders.

## Multi-datacenter operation

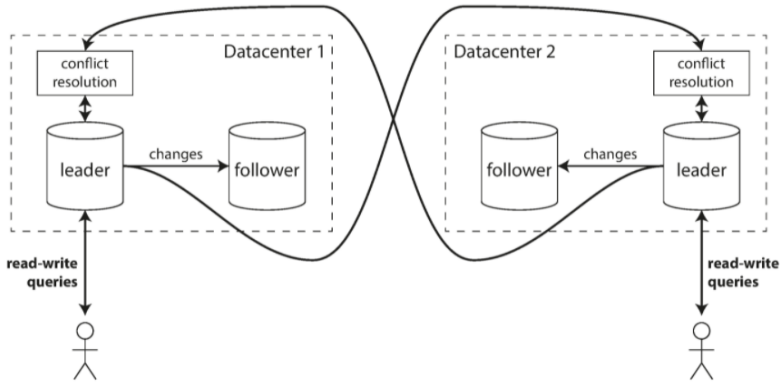



*Figure 5-6. Multi-leader replication across multiple datacenters.*

### Advantages

- Performance
  In a single-leader configuration, every write must go over the internet to the data center with the leader. In a multi-leader configuration, every write can be processed in the local datacenter and is replicated asynchronously to the other data centers. Thus, the inter-datacenter network delay is hidden from users, which means the perceived performance may be better.
- Tolerance of data center outages
  In a single-leader configuration, if the datacenter with the leader fails, failover can promote a follower in another datacenter to be the leader. In a multi-leader configuration, each datacenter can continue operating independently of the others, and replication catches up when the failed datacenter comes back online.
- Tolerance of network problems
  Traffic between data centers usually goes over the public internet, which may be less reliable than the local network within a data center. A single-leader configuration is very sensitive to problems in this inter-datacenter link because writes are made synchronously over this link. A multi-leader configuration with asynchronous replication can usually tolerate network problems better: a temporary network interruption does not prevent writes from being processed.

### Disadvantages

- The same data may be concurrently modified in two different data centers, and those write conflicts must be resolved.
- Auto Incrementing keys, triggers, and integrity constraints can be problematic. For this reason,multi-leader replication is often considered dangerous territory that should be avoided if possible.

## Handling Write Conflicts

The biggest problem with multi-leader replication is that write conflicts can occur, which means that conflict resolution is required.

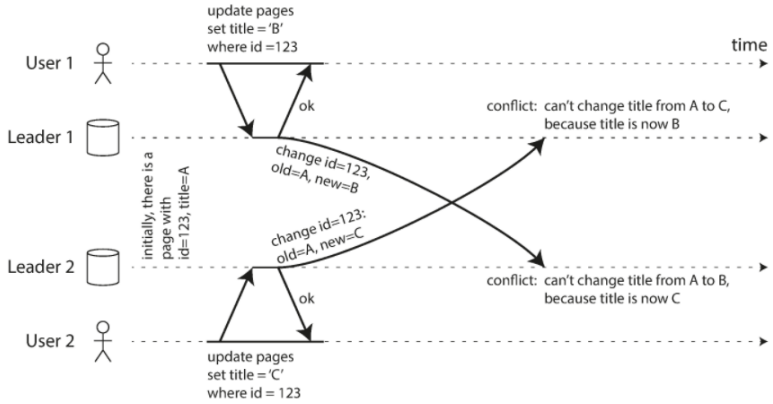



*Figure 5-7. A write conflict caused by two leaders concurrently updating the same record.*

### Synchronous versus asynchronous conflict detection

In a single-leader database, the second writer will either block and wait for the first write to complete, or abort the second write transaction, forcing the user to retry the write. On the other hand, in a multi-leader setup, both writes are successful, and the conflict is only detected asynchronously at some later point in time. At that time, it may be too late to ask the user to resolve the conflict.
In principle, you could make the conflict detection synchronous—i.e., wait for the write to be replicated to all replicas before telling the user that the write was successful. However, by doing so, you would lose the main advantage of multi-leader replication: allowing each replica to accept writes independently.

### 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. Since many implementations of multi-leader replication handle conflict quite poorly, avoiding conflicts is a frequently recommended approach.

## Multi-Leader Replication Topologies

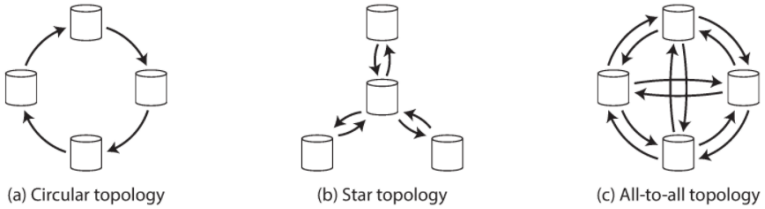



*Figure 5-8. Three example topologies in which multi-leader replication can be set up.*

In circular and star topologies, a write may need to pass through several nodes before it reaches all replicas. Therefore, nodes need to forward data changes they receive from other nodes. To Prevent infinite replication loops, each node is given a unique identifier, and in the replication log, each write is tagged with the identifiers of all the nodes it has passed through. When a node receives a data change that is tagged with its own identifier, that data change was ignored, because the node knows that it has already been processed.
A problem with circular and star topologies is that if just one node fails, it can interrupt the flow of replication messages between other nodes, causing them to be unable to communicate until the node is fixed.
The fault tolerance of a more densely connected topology (such as all-to-all) is better because it allows messages to travel along different paths, avoiding a single point of failure. On the other hand, 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 “overtake” others.

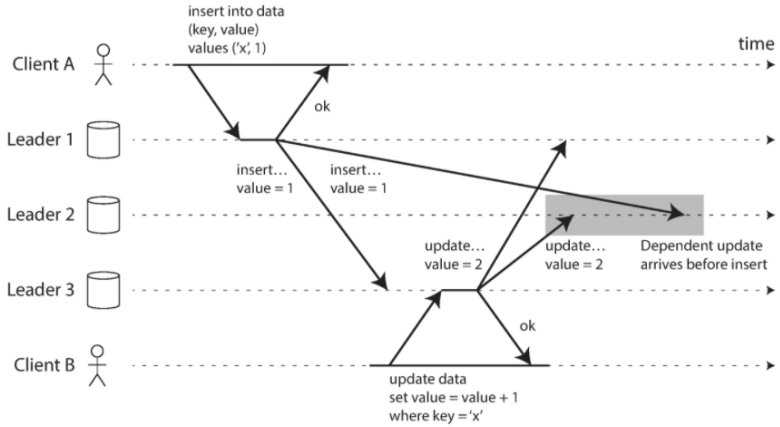



*Figure 5-9. With multi-leader replication, writes may arrive in the wrong order at some replicas.*

To order these events correctly, a technique called version vectors can be used.

# Leaderless Replication

Amazon used it for its in-house Dynamo system. Riak, Cassandra, and Voldemort are open source datastores with leaderless replication models inspired by Dynamo, so this kind of database is also known as Dynamo-style.

## Writing to the Database When a Node Is Down

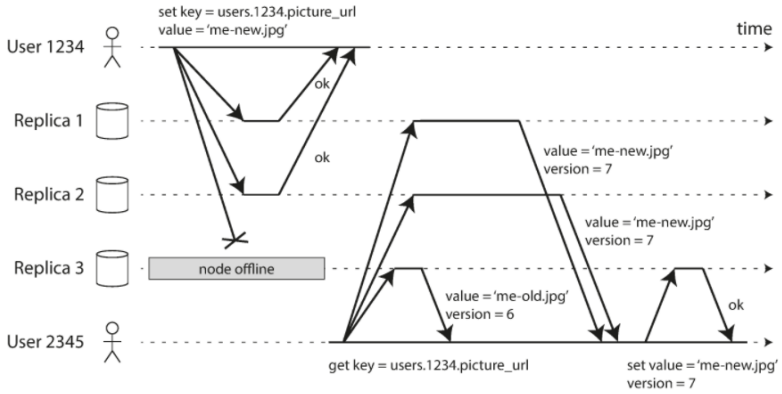



*Figure 5-10. A quorum write, quorum read, and read repair after a node outage.*

## Read repair and anti-entropy

The replication system should ensure that eventually all the data is copied to every replica. After an unavailable node comes back online, how does it catch up on the writes that it missed?
Two mechanisms are often used in Dynamo-style datastores:

### Read repair

When a client makes a read from several nodes in parallel, it can detect any stale responses. For example, user 2345 gets a version 6 value from replica 3 and a version 7 value from replicas 1 and 2. The client sees that replica 3 has a stable 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. Unlike the replication log in leader-based replication, this anti-entropy process does

not copy writes in any particular order, and there may be a significant delay before data is copied.

## Quorums for reading and writing

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, we expect to get an up-to-date value when reading, because at least one of the r nodes we're reading from must be up to date. Reads and writes that obey these r and w values are called quorum reads and writes.
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

Often, r and w are chosen to be a majority (more than n/2) of nodes, because that ensures w + r > n while still tolerating up to n/2 (rounded down) node failures. But quorums are not necessarily majorities—it only matters that the sets of nodes used by the read and write operations overlap in at least one node.

- With a smaller w and r, you are more likely to read stale values. On the upside, this configuration allows lower latency and higher availability.
- Even with w + r > n, there are likely to be edge cases where stale values are returned:
- If two writes occur concurrently, it is not clear which one happened first.
- If a write happens concurrently with a read, the write may be reflected on only some of the replicas. In this case, it's undetermined whether the read returns the old or the new value.
- If a write succeeded on some replicas but failed on others (for example because the disks on some nodes are full), and overall succeeded on fewer than w replicas, it is not rolled back on the replicas where it succeeded. This means that if a write was reported as failed, subsequent reads may or may not return the value from that write.
- If a node carrying a new value fails, and its data is restored from a replica carrying an old value.

You usually do not get the guarantees(reading your writes, monotonic reads, or consistent prefix reads). Stronger guarantees generally require transactions or consensus.

### Monitoring staleness

In systems with leaderless replication, there is no fixed order in which writes are applied, which makes monitoring more difficult. Moreover, if the database only uses read repair (no anti-entropy), there is no limit to how old a value might be.

### Sloppy Quorums and Hinted Handoff

A network interruption can easily cut off a client from a large number of database nodes. Although those nodes are alive, and other clients may be able to connect to them, to a client that is cut off from the database nodes, they might as well be dead. In this situation, it's likely that fewer than w or r reachable nodes remain, so the client can no longer reach a quorum.

In a large cluster (with significantly more than n nodes) it's likely that the client can connect to some database nodes during the network interruption, just not to the nodes that it needs to assemble a quorum for a particular value. In that case, database designers face a trade-off:

1. Is it better to return errors to all requests for which we cannot reach a quorum of w or r nodes?
2. Or should we accept writes anyway, and write them to some nodes that are reachable but aren't among the n nodes on which the value usually lives?

The latter is known as a **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.
Once the network interruption is fixed, any writes that one node temporarily accepted on behalf of another node are sent to the appropriate "home" nodes. This is called **hinted handoff**.
Sloppy quorums are particularly useful for increasing write availability: as long as any w nodes are available, the database can accept writes. However, this means that even when w + r > n, you cannot be sure to read the latest value for a key, because the latest value may have been temporarily written to some nodes outside of n.
Thus, a sloppy quorum actually isn't a quorum at all in the traditional sense. It's only an assurance of durability, namely that the data is stored on w nodes somewhere. There is no guarantee that a read of r nodes will see it until the hinted handoff has been completed.

## Detecting Concurrent Writes

A and B, simultaneously writing to a key X in a three-node datastore:

- Node 1 receives the write from A, but never receives the write from B due to a transient outage.
- Node 2 first receives the write from A, then the write from B.
- Node 3 first receives the write from B, then the write from A.

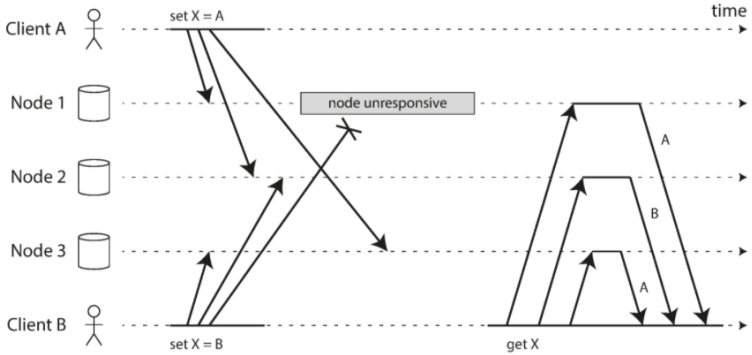



*Figure 5-12. Concurrent writes in a Dynamo-style datastore: there is no well-defined ordering.*

If each node simply overwrote the value for a key whenever it received a write request from the client, the nodes would become permanently inconsistent, as shown by the final get

request: node 2 thinks that the final value of X is B, whereas the other nodes think that the value is A.

### Last write wins (discarding concurrent writes)

Even though the writes don't have a natural ordering, we can force an arbitrary order on them. For Example, we can attach a timestamp to each write, pick the biggest timestamp as the most "recent," and discard any writes with an earlier timestamp. This conflict resolution algorithm, called last write wins (LWW), is the only supported conflict resolution method in Cassandra, and an optional feature in Riak.LWW achieves the goal of eventual convergence, but at the cost of durability: if there are several concurrent writes to the same key, even if they were all reported as successful to the client(because they were written to w replicas), only one of the writes will survive and the others will be silently discarded.

### The "happens-before" relationship and concurrency

An operation A happens before another operation B if B knows about A, or depends on A, or builds upon A in some way. Whether one operation happens before another operation is the key to defining what concurrency means. In fact, we can simply say that two operations are concurrent if neither happens before the other (i.e., neither knows about the other).
Thus, whenever you have two operations A and B, there are three possibilities: either A happens before B, B happened before A, or A and B are concurrent. What we need is an algorithm to tell us whether two operations are concurrent or not. If one operation happened before another, the later operation should overwrite the earlier operation, but if the operations are concurrent, we have a conflict that needs to be resolved.

# Partitioning

For very large datasets, or very high query throughput, that is not sufficient: we need to break the data up into partitions, also known as sharding.
What we call a partition here is called a shard in MongoDB, Elasticsearch, and SolrCloud; it's known as a region in HBase, a tablet in Bigtable, a vnode in Cassandra and Riak, and a vBucket in Couchbase.
Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes. This means that, even though each record belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance.

## Partitioning of Key-Value Data

If the partitioning is unfair, so that some partitions have more data or queries than others, we call it skewed.
A partition with a disproportionately high load is called a hot spot.

### Partitioning by Key Range

One way of partitioning is to assign a continuous range of keys (from some minimum to some maximum)to each partition. If You know the boundaries between the ranges, you can easily determine which partition contains a given key. If you also know which partition is assigned to which node, then you can make your request directly to the appropriate node.

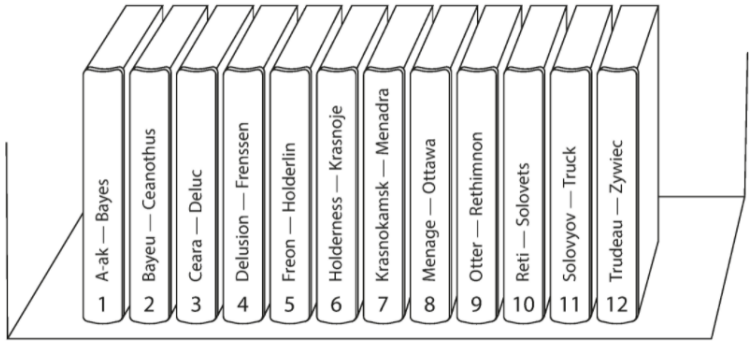



*Figure 6-2. A print encyclopedia is partitioned by key range.*

Within each partition, we can keep keys in sorted order ("SSTables and LSM-Trees"). This has the advantage that range scans are easy, and you can treat the key as a concatenated index in order to fetch several related records in one query ("Multi-column indexes").
However, the downside of key range partitioning is that certain access patterns can lead to hotspots. For example, if the key is a timestamp, then the partitions correspond to ranges of time—e.g., one partition per day, all the writes end up going to the same partition (the one for today), so that partition can be overloaded with writes while others sit idle.

## Partitioning by Hash of Key

Because of this risk of skew and hot spots, many distributed datastores use a hash function to determine the partition for a given key.
For partitioning purposes, the hash function need not be cryptographically strong: for example, MongoDB uses MD5, Cassandra uses Murmur3, and Voldemort uses the Fowler–Noll–Vo function.

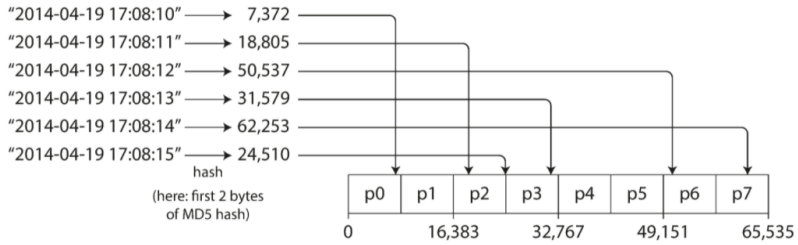



*Figure 6-3. Partitioning by hash of key.*

Unfortunately, however, 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. Keys that were once adjacent are now scattered across all the partitions, so their sort order is lost. In MongoDB, if you have enabled hash-based sharding mode, any range query has to be sent to all partitions.
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. A query therefore cannot search for a range of values within the first column of a compound key, but if it specifies a fixed value for the first column, it can perform an efficient range scan over the other columns of the key.

# 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

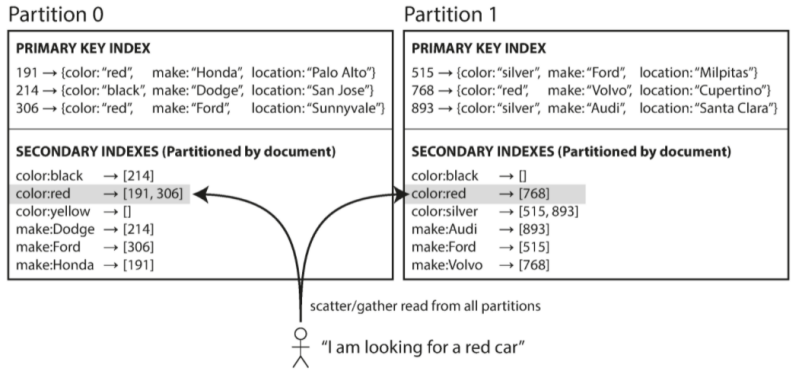



*Figure 6-4. Partitioning secondary indexes by document.*

In this indexing approach, each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition. Whenever you write to the database—to add, remove, or update a document—you only need to deal with the partition that contains the document ID that you are writing. 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. Even if you query the partitions in parallel, scatter/gather is prone to tail latency amplification.Nevertheless, it is widely used: MongoDB,Riak,Cassandra,Elasticsearch,and VoltDB all use document-partitioned secondary indexes.

## Partitioning Secondary Indexes by Term

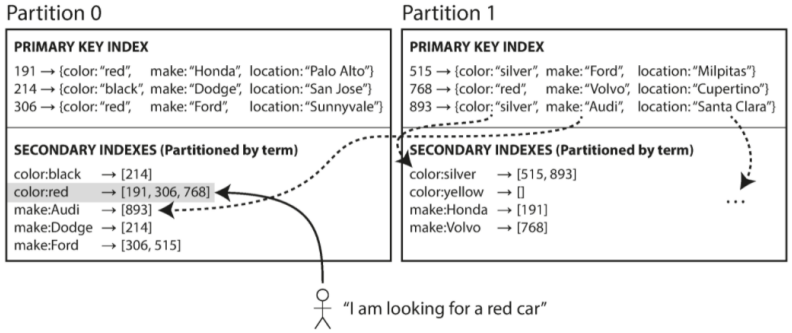



*Figure 6-5. Partitioning secondary indexes by term.*

Rather than each partition having its own secondary index (a local index), we can construct a global index that covers data in all partitions. However, we can't just store that index on one node, since it would likely become a bottleneck and defeat the purpose of partitioning. A global index must also be partitioned, but it can be partitioned differently from the primary key index.
For E.g. red cars from all partitions appear under color:red in the index, but the index is partitioned so that colors starting with the letters a to r appear in partition 0 and colors starting with s to z appear in partition 1.
The advantage of a global (term-partitioned) index over a document-partitioned index is that it can make reads more efficient: rather than doing scatter/gather over all partitions, a client only needs to make a request to the partition containing the term that it wants. However, the downside of a global index is that writes are slower and more complicated, because a write to a single document may now affect multiple partitions of the index (every term in the document might be on a different partition, on a different node).
In an ideal world, the index would always be up to date, and every document written to the database would immediately be reflected in the index. However, in a term-partitioned index, that would require a distributed transaction across all partitions affected by a write, which is not supported in all databases.
In practice, updates to global secondary indexes are often asynchronous (that is, if you read the index shortly after a write, the change you just made may not yet be reflected in the index).

# Rebalancing Partitions

The process of moving load from one node in the cluster to another is called rebalancing.
No matter which partitioning scheme is used, rebalancing is usually expected to meet some minimum requirements:

- After rebalancing, the load (data storage, read and write requests) should be shared fairly between the nodes in the cluster.
- While rebalancing is happening, the database should continue accepting reads and writes.
- No more data than necessary should be moved between nodes, to make rebalancing fast and to minimize the network and disk I/O load.

There are a few different ways of assigning partitions to nodes:

## 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.
For example, say hash(key) = 123456.  If you initially have 10 nodes, that key starts out on node 6(because 123456 mod 10 = 6). When you grow to 11 nodes, the key needs to move to node 3 (123456 mod 11 = 3).
Such frequent moves make rebalancing excessively expensive.

## Fixed number of partitions

There is a fairly simple solution: create many more partitions than there are nodes, and assign several partitions to each node. For example, a database running on a cluster of 10 nodes may be split into 1,000 partitions from the outset so that approximately 100 partitions are assigned to each node.
Now, 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. If a node is removed from the cluster, the same happens in reverse.

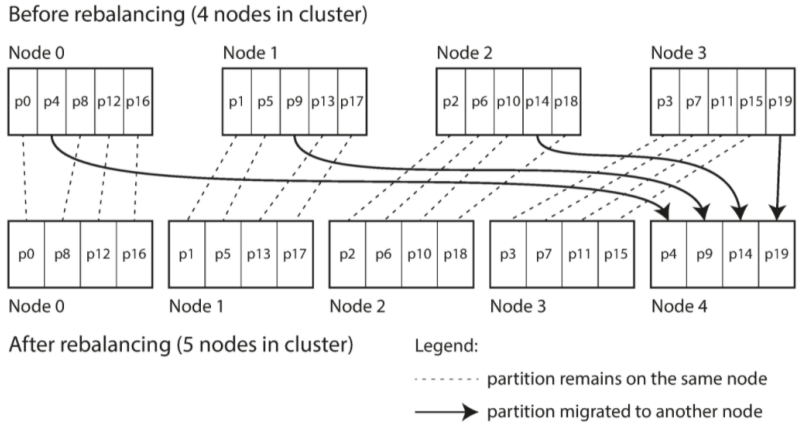



*Figure 6-6. Adding a new node to a database cluster with multiple partitions per node.*

The number of partitions does not change, nor does the assignment of keys to partitions. The only thing that changes is the assignment of partitions to nodes. This change of assignment is not immediate—it takes some time to transfer a large amount of data over the network—so the old assignment of partitions is used for any reads and writes that happen while the transfer is in progress.

## Dynamic partitioning

Key range–partitioned databases such as HBase and RethinkDB create partitions dynamically. When a partition grows to exceed a configured size (on HBase, the default is10 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.
However, a caveat is that an empty database starts off with a single partition since there is no a priori information about where to draw the partition boundaries. While the dataset is small—until it hits the point at which the first partition is split—all writes have to be processed by a single node while the other nodes sit idle. To mitigate this issue, HBase and MongoDB allow an initial set of partitions to be configured on an empty database (this is called pre-splitting).

## Partitioning proportionally to nodes

A third option, used by Cassandra and Ketama, is to make the number of partitions proportional to the number of nodes—in other words, to have a fixed number of partitions per node. 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. Since a larger data volume generally requires a larger number of nodes to store, this approach also keeps the size of each partition fairly stable.

## Request Routing

We have now partitioned our dataset across multiple nodes running on multiple machines. But there remains an open question: when a client wants to make a request, how does it know which node to connect to? As partitions are rebalanced, the assignment of partitions to nodes changes.

This is an instance of a more general problem called service discovery, which isn't limited to just databases.

On a high level, there are a few different approaches to this problem:

- Allow clients to contact any node (e.g., via a round-robin load balancer). If that node coincidentally owns the partition to which the request applies, it can handle the request directly; otherwise, it forwards the request to the appropriate node, receives the reply, and passes the reply along to the client.
- Send all requests from clients to a routing tier first, which determines the node that should handle each request and forwards it accordingly. This routing tier does not itself handle any requests; it only acts as a partition-aware load balancer.
- Require that clients be aware of the partitioning and the assignment of partitions to nodes. In this case, a client can connect directly to the appropriate node, without an intermediary.

Many distributed data systems rely on a separate coordination service such as ZooKeeper to keep track of this cluster metadata.

For example, LinkedIn's Espresso uses Helix for cluster management (which in turn relies on ZooKeeper), implementing a routing tier. HBase, SolrCloud, and Kafka also use ZooKeeper to track partition assignments. Cassandra and Riak take a different approach: they use a gossip protocol among the nodes to disseminate any changes in the cluster state.

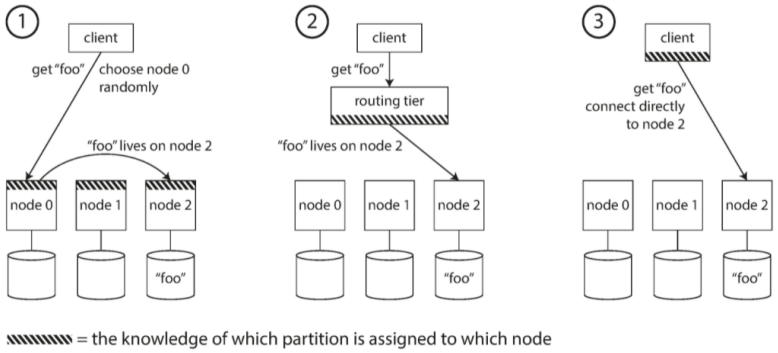



*Figure 6-7. Three different ways of routing a request to the right node.*

# Transaction

## The Meaning of ACID

### Atomicity

In the context of ACID, atomicity is not about concurrency. It does not describe what happens if several processes try to access the same data at the same time. Rather, ACID atomicity describes what happens if a client wants to make several writes, but a fault occurs after some of the writes have been processed. The ability to abort a transaction on an error and have all writes from that transaction discarded is the defining feature of ACID atomicity.

### Consistency

In the context of ACID, consistency refers to an application-specific notion of the database being in a “good state.” The idea of ACID consistency is that you have certain statements about your data (invariants) that must always be true—for example, in an accounting system, credits and debits across all accounts must always be balanced.
However, this idea of consistency depends on the application’s notion of invariants, and it’s the application's responsibility to define its transactions correctly so that they preserve consistency. This is not something that the database can guarantee: if you write bad data that violates your invariants, the database can’t stop you.
Atomicity, isolation, and durability are properties of the database, whereas consistency (in the ACID sense) is a property of the application. The application may rely on the database’s atomicity and isolation properties in order to achieve consistency, but it’s not up to the database alone.

### Isolation

Most databases are accessed by several clients at the same time. That is no problem if they are reading and writing different parts of the database, but if they are accessing the same database records, you can run into concurrency problems (race conditions).

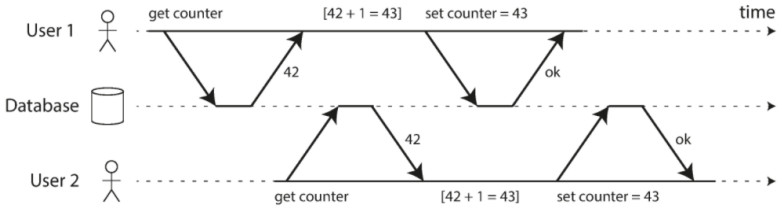



*Figure 7-1. A race condition between two clients concurrently incrementing a counter.*

Isolation in the sense of ACID means that concurrently executing transactions are isolated from each other. The classic database textbooks formalize isolation as serializability, which means that each transaction can pretend that it is the only transaction running on the entire database. The database ensures that when the transactions have been committed, the result is the same as if they had run serially (one after another), even though in reality they may have run concurrently.

### Durability

Durability is the promise that once a transaction has been committed successfully, any data it has written will not be forgotten, even if there is a hardware fault or the database crashes. In a single-node database, durability typically means that the data has been written to nonvolatile storage such as a hard drive or SSD. It usually also involves a write-ahead log or similar (see“Making B-trees reliable”), which allows recovery in the event that the data structures on the disk are corrupted. In a replicated database, durability may mean that the data has been successfully copied to some number of nodes.

#### Replication and Durability

- If you write to disk and the machine dies, even though your data isn’t lost, it is inaccessible until you either fix the machine or transfer the disk to another machine. Replicated systems can remain available.
- A correlated fault—a power outage or a bug that crashes every node on a particular input—can knock out all replicas at once, losing any data that is only in memory. Writing to disk is therefore still relevant for in-memory databases.
- In an asynchronously replicated system, recent writes may be lost when the leader becomes unavailable.
- When the power is suddenly cut, SSDs in particular have been shown to sometimes violate the guarantees they are supposed to provide.
- Subtle interactions between the storage engine and the filesystem implementation can lead to bugs that are hard to track down, and may cause files on disk to be corrupted after a crash.
- Data on a disk can gradually become corrupted without this being detected. If data has been corrupted for some time, replicas and recent backups may also be corrupted. In This case, you will need to try to restore the data from a historical backup.
- One study of SSDs found that between 30% and 80% of drives develop at least one bad block during the first four years of operation. Magnetic hard drives have a lower rate of bad sectors, but a higher rate of complete failure than SSDs.
- When a worn-out SSD (that has gone through many writes/erase cycles) is disconnected from power, it can start losing data within a timescale of weeks to months, depending on the temperature.

In practice, there is no one technique that can provide absolute guarantees.

# Single-Object and Multi-Object Operations

## The need for multi-object transactions

There are some use cases in which single-object inserts, updates, and deletes are sufficient. However, in many other cases writes to several different objects need to be coordinated:

- In a relational data model, a row in one table often has a foreign key reference to a row in another table. (Similarly, in a graph-like data model, a vertex has edges to other vertices.) Multi-object transactions allow you to ensure that these references remain valid.
- In a document data model, the fields that need to be updated together are often within the same document, which is treated as a single object—no multi-object transactions are needed when updating a single document. However, document databases lacking join functionality also encouraged denormalization. When denormalized information needs to be updated, you need to update several documents in one go. Transactions are very useful in this situation to prevent denormalized data from going out of sync.
- In databases with secondary indexes (almost everything except pure key-value stores), the indexes also need to be updated every time you change a value. These indexes are different database objects from a transaction point of view: for example, without transaction isolation, it's possible for a record to appear in one index but not another, because the update to the second index hasn't happened yet.

# Handling errors and aborts

In particular, datastores with leaderless replication work much more on a "best-effort" basis, which could be summarized as "the database will do as much as it can, and if it runs into an error, it won't undo something it has already done"—so it's the application's responsibility to recover from errors.
Although retrying an aborted transaction is a simple and effective error handling mechanism, it isn't perfect:

- If the transaction actually succeeded, but the network failed while the server tried to acknowledge the successful commit to the client (so the client thinks it failed), then retrying the transaction causes it to be performed twice—unless you have an additional application-level deduplication mechanism in place.
- If the error is due to overload, retrying the transaction will make the problem worse, not better. To avoid such feedback cycles, you can limit the number of retries, use exponential backoff, and handle overload-related errors differently from other errors (if possible).
- It is only worth retrying after transient errors (for example due to deadlock, isolation violation, temporary network interruptions, and failover); after a permanent error (e.g., constraint violation) a retry would be pointless.

# Weak Isolation Levels

In theory, isolation should make your life easier by letting you pretend that no concurrency is happening: serializable isolation means that the database guarantees that transactions have the same effect as if they ran serially (i.e., one at a time, without any concurrency). 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 read committed. It makes two guarantees:

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

### No dirty reads

Imagine a transaction has written some data to the database, but the transaction has not yet been committed or aborted. Can another transaction see that uncommitted data? If yes, that is called a dirty read.
This means that any writes by a transaction only become visible to others when that transaction commits (and then all of its writes become visible at once).

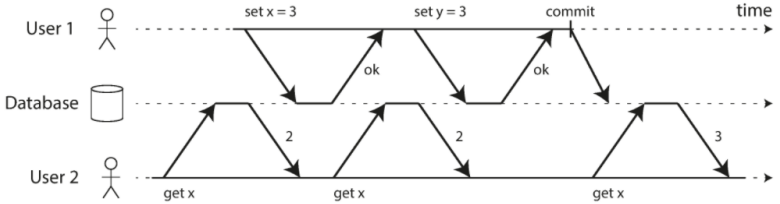



*Figure 7-4. No dirty reads: user 2 sees the new value for x only after user 1's transaction has committed.*

There are a few reasons why it's useful to prevent dirty reads:

- Seeing the database in a partially updated state is confusing to users and may cause other transactions to take incorrect decisions.
- If a transaction aborts, any writes it has made need to be rolled back. If the database allows dirty reads, that means a transaction may see data that is later rolled back—i.e., which is never actually committed to the database.

### No dirty writes

What happens if two transactions concurrently try to update the same object in a database? What happens if the earlier write is part of a transaction that has not yet been committed, so the later write overwrites an uncommitted value? This is called a dirty write.
If transactions update multiple objects, dirty writes can lead to a bad outcome.

### Implementing read committed

Most commonly, databases prevent dirty writes by using row-level locks: when a transaction wants to modify a particular object (row or document), it must first acquire a lock on that object. It must then hold that lock until the transaction is committed or aborted.
How do we prevent dirty reads? One option would be to use the same lock and to require any transaction that wants to read an object to briefly acquire the lock and then release it again immediately after reading.
However, the approach of requiring read locks does not work well in practice, because one long-running write transaction can force many other transactions to wait until the long-running transaction has been completed, even if the other transactions only read and do not write anything to the database. This harms the response time of read-only transactions and is bad for operability.
For that reason, most databases use this approach- For every object that is written, the database remembers both the old committed value and the new value set by the transaction that currently holds the write lock. While the transaction is ongoing, any other transactions that read the object are simply given the old value. Only when the new value is committed do transactions switch over to reading the new value.

## Snapshot Isolation and Repeatable Read

Say Alice has $1,000 of savings at a bank, split across two accounts with $500 each. Now a transaction transfers $100 from one of her accounts to the other. If she is unlucky enough to look at her list of account balances at the same moment as that transaction is being processed, she may see one account balance at a time before the incoming payment has arrived (with a balance of $500), and the other account after the outgoing transfer has been made (the new balance being $400). To Alice, it now appears as though she only has a total of $900 in her accounts—it seems that $100 has vanished into thin air. This anomaly is called read skew, and it is an example of a non-repeatable read. Read skew is considered acceptable under read committed isolation.
In Alice's case, this is not a lasting problem, because she will most likely see consistent account balances if she reloads the online banking website a few seconds later. However, some situations cannot tolerate such temporary inconsistency:

- BackupsTaking
  A backup requires making a copy of the entire database, which may take hours on a large database. During the time that the backup process is running, writes will continue to be made to the database. Thus, you could end up with some parts of the backup containing an older version of the data, and other parts containing a newer version. If you need to restore from such a backup, the inconsistencies (such as disappearing money) become permanent.
- Analytic queries and integrity checks
  Sometimes, you may want to run a query that scans over large parts of the database. Such queries are common in analytics or maybe part of a periodic integrity check that everything is in order (monitoring for data corruption). These queries are likely to return nonsensical results if they observe parts of the database at different points in time.

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. It is very hard to reason about the meaning of a query if the data on which it operates is changing at the same time as the query is executing. When a transaction can see a consistent snapshot of the database, frozen at a particular point in time, it is much easier to understand.

### Implementing snapshot isolation

Like read committed isolation, implementations of snapshot isolation typically use write locks to prevent dirty writes, which means that a transaction that makes a write can block the progress of another transaction that writes to the same object. However, reads do not require any locks. From a performance point of view, a key principle of snapshot isolation is readers never block writers, and writers never block readers. This allows a database to handle long-running read queries on a consistent snapshot at the same time as processing write normally, without any lock contention between the two.
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).
If a database only needed to provide read committed isolation, but not snapshot isolation, it would be sufficient to keep two versions of an object: the committed version and the overwritten-but-not-yet-committed version.

### Visibility rules for observing a consistent snapshot

By carefully defining visibility rules, the database can present a consistent snapshot of the database to the application. This works as follows:

- At the start of each transaction, the database makes a list of all the other transactions that are in progress (not yet committed or aborted) at that time. Any writes that those transactions have made are ignored, even if the transactions subsequently commit.
- Any writes made by aborted transactions are ignored.
- Any writes made by transactions with a later transaction ID (i.e., which started after the current transaction started) are ignored, regardless of whether those transactions have been committed.
- All other writes are visible to the application's queries.

## Preventing Lost Updates

The lost update problem can occur if an application reads some value from the database, modifies it, and writes back the modified value (a read-modify-write cycle). If two transactions do this concurrently, one of the modifications can be lost, because the second write does not include the first modification. (We sometimes say that the later write clobbers the earlier write.) This Pattern occurs in various different scenarios:

- Incrementing a counter or updating an account balance (requires reading the current value, calculating the new value, and writing back the updated value)

- Making a local change to a complex value, e.g., adding an element to a list within a JSON document(requires parsing the document, making the change, and writing back the modified document)
- Two users editing a wiki page at the same time, where each user saves their changes by sending the entire page contents to the server, overwriting whatever is currently in the database.

### Atomic write operations

Many databases provide atomic update operations, which remove the need to implement read-modify-write cycles in application code. Not all writes can easily be expressed in terms of atomic operations—for example, updates to a wiki page involve arbitrary text editing.
Atomic operations are usually implemented by taking an exclusive lock on the object when it is read so that no other transaction can read it until the update has been applied. This technique is sometimes known as cursor stability. Another option is to simply force all atomic operations to be executed on a single thread.

### Explicit locking

Another option for preventing lost updates, if the database's built-in atomic operations don't provide the necessary functionality, is for the application to explicitly lock objects that are going to be updated. Then the application can perform a read-modify-write cycle.

### Automatically detecting lost updates

Atomic operations and locks are ways of preventing lost updates by forcing the read-modify-write cycles to happen sequentially. 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.

### Compare-and-set

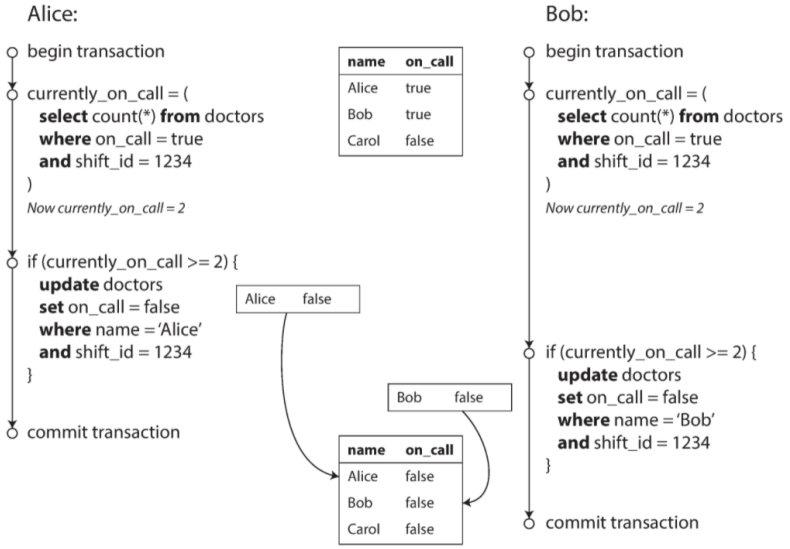
In databases that don't provide transactions, you sometimes find an atomic compare-and-set operation. The purpose of this operation is to avoid lost updates by allowing an update to happen only if the value has not changed since you last read it. If the current value does not match what you previously read, the update has no effect, and the read-modify-write cycle must be retried.

### Conflict resolution and replication

Locks and compare-and-set operations assume that there is a single up-to-date copy of the data. However, databases with multi-leader or leaderless replication usually allow several writes to happen concurrently and replicate them asynchronously, so they cannot guarantee that there is a single up-to-date copy of the data. Thus, techniques based on locks or compare-and-set do not apply in this context.
A common approach in such replicated databases is to allow concurrent writes to create several conflicting versions of a value (also known as siblings) and to use application code or special data structures to resolve and merge these versions after the fact.

On the other hand, the last write wins (LWW) conflict resolution method is prone to lost updates.

## Write Skew and Phantoms




In each transaction, your application first checks that two or more doctors are currently on call; if yes, it assumes it's safe for one doctor to go off the call. Since the database is using snapshot isolation, both checks return 2, so both transactions proceed to the next stage. Alice updates her own record to take herself off the call, and Bob updates his own record likewise. Both transactions commit, and now no doctor is on call. Your requirement of having at least one doctor on call has been violated.
This anomaly is called write skew. It is neither a dirty write nor a lost update because the two transactions are updating two different objects (Alice's and Bob's on-call records, respectively).

### Phantoms causing write skew

- A SELECT query checks whether some requirement is satisfied by searching for rows that match some search condition.
- Depending on the result of the first query, the application code decides how to continue.
- If the application decides to go ahead, it makes a write (INSERT, UPDATE, or DELETE) to the database and commits the transaction.

If you were to repeat the SELECT query from step 1 after committing the write, you would get a different result, because the write changed the set of rows matching the search condition.

This effect, where a write in one transaction changes the result of a search query in another transaction, is called a phantom.

## 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. Thus, the database guarantees that if the transactions behave correctly when run individually, they continue to be correct when run concurrently—in other words, the database prevents all possible race conditions.
Most databases that provide serializability today use one of three techniques:

- Literally executing transactions in serial order (“Actual Serial Execution”)
- Two-phase locking (“Two-Phase Locking (2PL)”)
- Optimistic concurrency control techniques such as serializable snapshot isolation (“Serializable Snapshot Isolation (SSI)”)

### Actual Serial Execution

The simplest way of avoiding concurrency problems is to remove the concurrency entirely: to execute only one transaction at a time, in serial order, on a single thread. By doing so, we completely sidestep the problem of detecting and preventing conflicts between transactions: the resulting solution is by definition serializable.
Executing all transactions serially makes concurrency control much simpler, but limits the transaction throughput of the database to the speed of a single CPU core on a single machine. Read-only transactions may execute elsewhere, using snapshot isolation, but for applications with high write throughput, the single-threaded transaction processor can become a serious bottleneck.
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.
- 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.
- Cross-partition transactions are possible, but there is a hard limit to the extent to which they can be used.

### Two-Phase Locking (2PL)

While two-phase locking (2PL) sounds very similar to two-phase commit (2PC), they are completely different things.
Two-phase locking is similar but makes the lock requirements much stronger. Several transactions are allowed to concurrently read the same object as long as nobody is writing to it. But as soon as anyone wants to write (modify or delete) an object, exclusive access is required:

- 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. (Reading an old version of the object)

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

#### 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 lock is used as follows:

- If a transaction wants to read an object, it must first acquire the lock in shared mode. Several Transactions are allowed to hold the lock in shared mode simultaneously, but if another transaction already has an exclusive lock on the object, these transactions must wait.
- If a transaction wants to write to an object, it must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time (either in shared or in exclusive mode), so if there is an existing lock on the object, the transaction must wait.
- If a transaction first reads and then writes an object, it may upgrade its shared lock to an exclusive lock. The upgrade works the same as getting an exclusive lock directly.
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort). This is where the name “two-phase” comes from: the first phase(while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released.

#### Performance of two-phase locking

Transaction throughput and response times of queries are significantly worse under two-phase locking than under weak isolation. This is partly due to the overhead of acquiring and releasing all those locks, but more importantly due to reduced concurrency.
Even if you make sure that you keep all your transactions short, a queue may form if several transactions want to access the same object.
If there is contention in the workload. It May take just one slow transaction or one transaction that accesses a lot of data and acquires many locks, to cause the rest of the system to grind to a halt.
Deadlocks occur much more frequently under 2PL serializable isolation (depending on the access patterns of your transaction). This can be an additional performance problem: when a transaction is aborted due to deadlock and is retried, it needs to do its work all over again.

### Serializable Snapshot Isolation (SSI)

On the one hand, we have implementations of serializability that don’t perform well (two-phase locking) or don’t scale well (serial execution). On the other hand, we have weak isolation levels that have good performance but are prone to various race conditions (lost

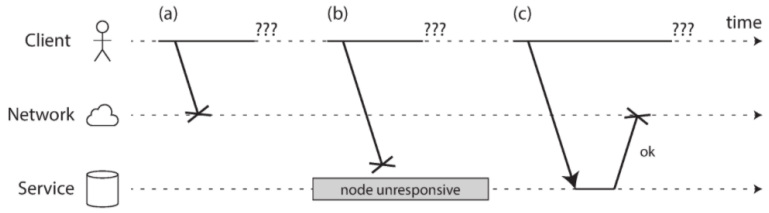
updates, write skew, phantoms, etc.). Are Serializable isolation and good performance fundamentally at odds with each other?
An algorithm called serializable snapshot isolation (SSI) is very promising. As SSI is so young compared to other concurrency control mechanisms, it is still proving its performance in practice.
Under snapshot isolation, the result from the original query may no longer be up-to-date by the time the transaction commits, because the data may have been modified in the meantime.
When the application makes a query (e.g., “How many doctors are currently on call?”), the database doesn't know how the application logic uses the result of that query. To be safe, the database needs to assume that any change in the query result (the premise) means that writes in that transaction may be invalid. In other words, there may be a causal dependency between the queries and the writes in the transaction. In order to provide serializable isolation, the database must detect situations in which a transaction may have acted on an outdated premise and abort the transaction in that case.
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)

# The Trouble with Distributed Systems

In a distributed system, there may well be some parts of the system that are broken in some unpredictable way, even though other parts of the system are working fine. This is known as a partial failure. The difficulty is that partial failures are nondeterministic: if you try to do anything involving multiple nodes and the network, it may sometimes work and sometimes unpredictable fail. You may not even know whether something succeeded or not, as the time it takes for a message to travel across a network is also non-deterministic!
**In distributed systems, suspicion, pessimism, and paranoia pay off.**

## Unreliable Networks




*re 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. However, when a timeout occurs, you still don’t know whether the remote node got your request or not (and if the request is still queued somewhere, it may still be delivered to the recipient, even if the sender has given up on it).
You do need to know how your software reacts to network problems and ensure that the system can recover from them. It may make sense to deliberately trigger network problems and test the system’s response (this is the idea behind Chaos Monkey).

## Timeouts and Unbounded Delays

If a timeout is the only sure way of detecting a fault, then how long should the timeout be? There is unfortunately no simple answer. A long timeout means a long wait until a node is declared dead (and during this time, users may have to wait or see error messages). A short timeout detects faults faster but carries a higher risk of incorrectly declaring a node dead when in fact it has only suffered a temporary slowdown (e.g., due to a load spike on the node or the network).
Prematurely declaring a node dead is problematic: if the node is actually alive and in the middle of performing some action (for example, sending an email), and another node takes over, the action may end up being performed twice.

When a node is declared dead, its responsibilities need to be transferred to other nodes, which places additional load on other nodes and the network. If the system is already struggling with high load, declaring nodes dead prematurely can make the problem worse.

# Unreliable Clocks

In a distributed system, time is a tricky business, because communication is not instantaneous: it takes time for a message to travel across the network from one machine to another. The time when a message is received is always later than the time when it is sent, but due to variable delays in the network, we don't know how much later. This fact sometimes makes it difficult to determine the order in which things happened when multiple machines are involved.
Moreover, each machine on the network has its own clock, which is an actual hardware device: usually a quartz crystal oscillator. These devices are not perfectly accurate, so each machine has its own notion of time, which may be slightly faster or slower than on other machines.

## 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.
**Time-of-day clocks**
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). For example,clock_gettime(CLOCK_REALTIME) on Linux and System.currentTimeMillis() in Java return the number of seconds (or milliseconds) since the epoch: midnight UTC on January 1, 1970, according to the Gregorian calendar, not counting leap seconds. Some systems use other dates as their reference point. Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine(ideally) means the same as a timestamp on another machine.
However, time-of-day clocks also have various oddities, as described in the next section. In particular, if the local clock is too far ahead of the NTP server, it may be forcibly reset and appear to jump back to a previous point in time.
**Monotonic clocks**
A monotonic clock is suitable for measuring a duration (time interval), such as a timeout or a service's response time: clock_gettime(CLOCK_MONOTONIC) on Linux and System.nanoTime() in Java are monotonic clocks, for example. The name comes from the fact that they are guaranteed to always move forward (whereas a time-of-day clock may jump back in time).
However, the absolute value of the clock is meaningless: it might be the number of nanoseconds since the computer was started, or something similarly arbitrary. In Particular, it makes no sense to compare monotonic clock values from two different computers, because they don't mean the same thing.
In a distributed system, using a monotonic clock for measuring elapsed time (e.g., timeouts) is usually fine, because it doesn't assume any synchronization between different nodes' clocks and is not sensitive to slight inaccuracies of measurement.

## Clock Synchronization and Accuracy

Monotonic clocks don't need synchronization, but time-of-day clocks need to be set according to an NTP server or other external time sources in order to be useful.

- The quartz clock in a computer is not very accurate: it drifts (runs faster or slower than it should). Clock drift varies depending on the temperature of the machine. Google assumes a clock drift of 200 ppm (parts per million) for its servers, which is equivalent to a 6 ms drift for a clock that is resynchronized with a server every 30seconds, or 17 seconds drift for a clock that is resynchronized once a day. This drift limits the best possible accuracy you can achieve, even if everything is working correctly.
- If a computer's clock differs too much from an NTP server, it may refuse to synchronize, or the local clock will be forcibly reset. Any Applications observing the time before and after this reset may see time go backward or suddenly jump forward.
- If a node is accidentally firewalled off from NTP servers, the misconfiguration may go unnoticed for some time. Anecdotal evidence suggests that this does happen in practice.
- NTP synchronization can only be as good as the network delay, so there is a limit to its accuracy when you're on a congested network with variable packet delays. One experiment showed that a minimum error of 35 ms is achievable when synchronizing over the internet, though occasional spikes in network delay lead to errors of around a second.
- Leap seconds result in a minute that is 59 seconds or 61 seconds long, which messes up timing assumptions in systems that are not designed with leap seconds in mind.
- In virtual machines, the hardware clock is virtualized, which raises additional challenges for applications that need accurate timekeeping. When a CPU core is shared between virtual machines, each VM is paused for tens of milliseconds while another VM is running. From an application's point of view, this pause manifests itself as the clock suddenly jumping forward.

## Timestamps for ordering events

If two clients write to a distributed database, who got there first? Which write is the more recent one?

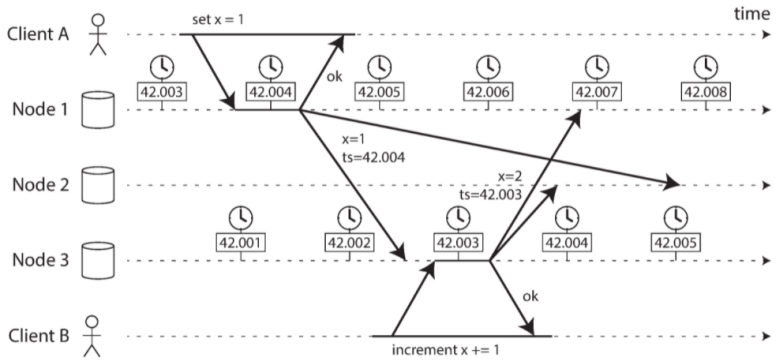



*Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.*

So-called logical clocks, which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events.

# 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. A node can only find out what state another node is in (what data it has stored, whether it is correctly functioning, etc.) by exchanging messages with it. If a remote node doesn't respond, there is no way of knowing what state it is in, because problems in the network cannot reliably be distinguished from problems at a node.

## The Truth Is Defined by the Majority

Imagine a network with an asymmetric fault: a node is able to receive all messages sent to it, but any outgoing messages from that node are dropped or delayed. Even though that node is working perfectly well, and is receiving requests from other nodes, the other nodes cannot hear its responses. After some timeout, the other nodes declare it dead, because they haven't heard from the node.

In a slightly less nightmarish scenario, the semi-disconnected node may notice that the messages it is sending are not being acknowledged by other nodes, and so realize that there must be a fault in the network. Nevertheless, the node is wrongly declared dead by the other nodes, and the semi-disconnected node cannot do anything about it.

As a third scenario, imagine a node that experiences a long stop-the-world garbage collection pause. All of the node's threads are preempted by the GC and paused for one minute, and consequently, no requests are processed and no responses are sent. The other nodes wait, retry, grow impatient, and eventually declare the node dead and load it onto the hearse. Finally, the GC finishes, and the node's threads continue as if nothing had happened.

The moral of these stories is that a node cannot necessarily trust its own judgment of a situation. A distributed system cannot exclusively rely on a single node, because a node may fail at any time, potentially leaving the system stuck and unable to recover. Instead, many distributed algorithms rely on a quorum, that is, voting among the nodes.

## System Model and Reality

Three system models are in common use:

Synchronous model

The synchronous model assumes bounded network delay, bounded process pauses, and bounded clock error. This does not imply exactly synchronized clocks or zero network delay; it just means you know that network delay, pauses, and clock drift will never exceed some fixed upper bound. The synchronous model is not a realistic model of most practical systems, because unbounded delays and pauses do occur.

Partially synchronous model

Partial synchrony means that a system behaves like a synchronous system most of the time, but it sometimes exceeds the bounds for network delay, process pauses, and clock drift. This is a realistic model of many systems: most of the time, networks and processes are quite well behaved—otherwise we would never be able to get anything done—but we have to reckon with the fact that any timing assumptions may be shattered occasionally. When this happens, network delay, pauses, and clock error may become arbitrarily large.

Asynchronous model

In this model, an algorithm is not allowed to make any timing assumptions—in fact, it doesn't even have a clock (so it cannot use timeouts). Some algorithms can be designed for the asynchronous model, but it is very restrictive.

Moreover, besides timing issues, we have to consider node failures. The three most common system models for nodes are:

Crash-stop faults

In the crash-stop model, an algorithm may assume that a node can fail in only one way, namely by crashing. This means that the node may suddenly stop responding at any moment, and thereafter that node is gone forever—it never comes back.

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 (i.e., nonvolatile disk storage) that is preserved across crashes, while the in-memory state is assumed to be lost.

Byzantine (arbitrary) faultsNodes may do absolutely anything, including trying to trick and deceive other nodes, as described in the last section.

# Consistency and Consensus

## Linearizability

In an eventually consistent database, if you ask two different replicas the same question at the same time, you may get two different answers. Wouldn't it be a lot simpler if the database could give the illusion that there is only one replica (i.e., only one copy of the data)? Then every client would have the same view of the data, and you wouldn't have to worry about replication lag.
This is the idea behind linearizability(also known as atomic consistency, strong consistency, immediate consistency, or external consistency).

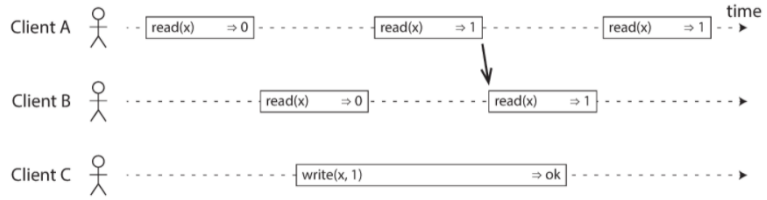



*Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.*

In a linearizable system, we imagine that there must be some point in time (between the start and end of the write operation) at which the value of x atomically flips from 0 to 1. Thus, if one client's read returns the new value 1, all subsequent reads must also return the new value, even if the write operation has not yet been completed.
This timing dependency is illustrated with an arrow in Figure 9-3. Client A is the first to read the new value, 1. Just after A's read returns, B begins a new read. Since B's read occurs strictly after A's read, it must also return 1, even though the write by C is still ongoing.

## Linearizability Versus Serializability

- Serializability is an isolation property of transactions, where every transaction may read and write multiple objects (rows, documents, records). It guarantees that transactions behave the same as if they had executed in some serial order (each transaction running to completion before the next transaction starts). It is okay for that serial order to be different from the order in which transactions were actually run.
- Linearizability is a recency guarantee on the reads and writes of a register (an individual object). It doesn't group operations together into transactions.

A database may provide both serializability and linearizability, and this combination is known as strict serializability or strong one-copy serializability (strong-1SR).
Implementations of serializability based on two-phase locking or actual serial execution are typically linearizable.
However, serializable snapshot isolation is not linearizable: by design, it makes reads from a consistent snapshot, to avoid lock contention between readers and writers. The whole point of a consistent snapshot is that it does not include writes that are more recent than the snapshot, and thus reads from the snapshot are not linearizable.

## Uses of Linearizability

There are a few areas in which linearizability is an important requirement for making a system work correctly:
**Locking and leader election** 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** This situation is actually similar to a lock: when a user registers for your service, you can think of them acquiring a "lock" on their chosen username. The operation is also very similar to an atomic compare-and-set, setting the username to the ID of the user who claimed it, provided that the username is not already taken.
Similar issues arise if you want to ensure that a bank account balance never goes negative, or that you don't sell more items than you have in stock in the warehouse, or that two people don't concurrently book the same seat on a flight or in a theater. These constraints all require there to be a single up-to-date value (the account balance, the stock level, the seat occupancy) that all nodes agree on.
**Cross-channel timing dependencies** For example, say you have a website where users can upload a photo, and a background process resizes the photos to lower resolution for faster download (thumbnails).

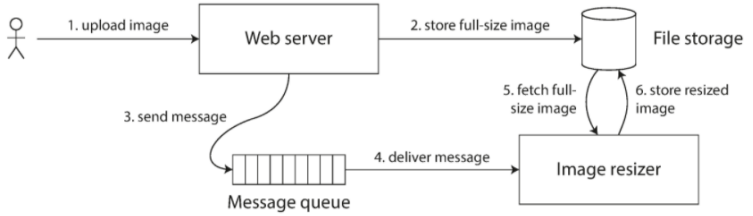



*Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.*

The image resizer needs to be explicitly instructed to perform a resizing job, and this instruction is sent from the webserver to the resizer via a message queue. The web server does not place the entire photo on the queue, since most message brokers are designed for small messages, and a photo may be several megabytes in size. Instead, the photo is first written to a file storage service, and once the write is complete, the instruction to the resizer is placed on the queue.
If the file storage service is linearizable, then this system should work fine. If it is not linearizable, there is the risk of a race condition: the message queue (steps 3 and 4) might be faster than the internal replication inside the storage service. In this case, when the resizer fetches the image (step 5), it might see an old version of the image or nothing at all. If it processes an old version of the image, the full-size and resize images in the file storage become permanently inconsistent.

This problem arises because there are two different communication channels between the web server and the resizer: the file storage and the message queue. Without the recency guarantee of linearizability, race conditions between these two channels are possible.

## Implementing Linearizable Systems

The most common approach to making a system fault-tolerant is to use replication.

- **Single-leader replication (potentially linearizable)** If you make reads from the leader, or from synchronously updated followers, they have the potential to be linearizable.
- Consensus algorithms (linearizable)
- **Multi-leader replication (not linearizable)** Because they concurrently process writes on multiple nodes and asynchronously replicate them to other nodes.
- Leaderless replication (probably not linearizable)

## Linearizability and quorums

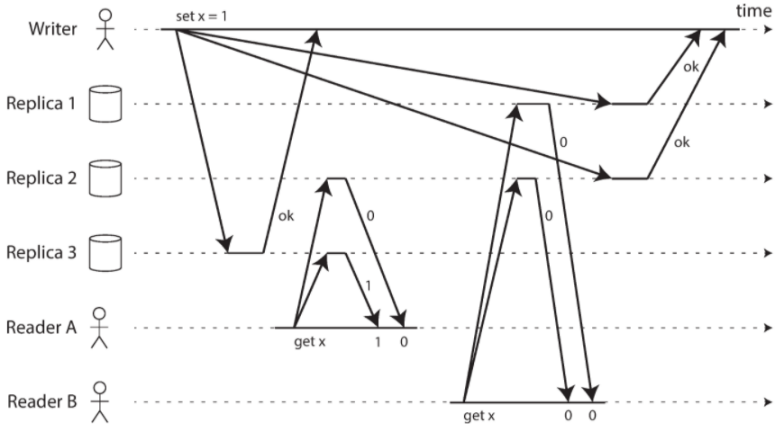



*Figure 9-6. A nonlinearizable execution, despite using a strict quorum.*

The quorum condition is met (w + r > n), but this execution is nevertheless not linearizable: B's request begins after A's request completes, but B returns the old value while A returns the new value.
Interestingly, it is possible to make Dynamo-style quorums linearizable at the cost of reduced performance: a reader must perform read repair synchronously, before returning results to the application, and a writer must read the latest state of a quorum of nodes before sending its writes.
Cassandra does wait for read repair to complete on quorum reads, but it loses linearizability if there are multiple concurrent writes to the same key, due to its use of last-write-wins conflict resolution.

## The Cost of Linearizability

With a multi-leader database, each datacenter can continue operating normally: since writes from one datacenter are synchronously replicated to the other, the writes are simply queued up and exchanged when network connectivity is restored.
On the other hand, if single-leader replication is used, then the leader must be in one of the data centers. Any writes and any linearizable reads must be sent to the leader—thus, for any clients connected to a follower data center, those read and write requests must be sent synchronously over the network to the leader datacenter. If the network between datacenters is interrupted in a single-leader setup, clients connected to follower data centers cannot contact the leader, so they cannot make any writes to the database, nor any linearizable reads.
They can still make reads from the follower, but they might be stale(non-linearizable). If the application requires linearizable reads and writes, the network interruption causes the application to become unavailable in the data centers that cannot contact the leader.

## The CAP theorem

The trade-off is as follows:

- 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.

This Insight is popularly known as the CAP theorem.
CAP is sometimes presented as Consistency, Availability, Partition tolerance: pick 2 out of 3. Unfortunately, putting it this way is misleading because network partitions are a kind of fault, so they aren't something about which you have a choice: they will happen whether you like it or not.
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**.

## Linearizability and network delays

Linearizability is slow—and this is true all the time, not only during a network fault. In a network with highly variable delays, like most computer networks, the response time of linearizable reads and writes is inevitably going to be high. A faster algorithm for linearizability does not exist, but weaker consistency models can be much faster, so this trade-off is important for latency-sensitive systems.

# Ordering Guarantees

## Ordering and Causality

Ordering helps preserve causality. Causality imposes an ordering on events: cause comes before effect; a message is sent before that message is received; the question comes before the answer. 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(assuming it has not been deleted in the meantime).

## The causal order is not a 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. For example, natural numbers are totally ordered: if I give you any two numbers, say 5 and 13, you can tell me that 13 is greater than 5.
The difference between a total order and a partial order is reflected in different database consistency models:

- **Linearizability** In a linearizable system, we have a total order of operations: if the system behaves as if there is only a single copy of the data, and every operation is atomic, this means that for any two operations we can always say which one happened first.
- **Causality** Two events are ordered if they are causally related (one happened before the other), but they are incomparable if they are concurrent. This means that causality defines a partial order, not a total order: some operations are ordered with respect to each other, but some are incomparable.

There are no concurrent operations in a linearizable datastore: there must be a single timeline along which all operations are totally ordered. There might be several requests waiting to be handled, but the datastore ensures that every request is handled atomically at a single point in time, acting on a single copy of the data, along a single timeline, without any concurrency.
Concurrency would mean that the timeline branches and merges again—and in this case, operations on different branches are incomparable (i.e., concurrent).

## Linearizability is stronger than causal consistency

Linearizability implies causality: any system that is linearizable will preserve causality correctly. However, making a system linearizable can harm its performance and availability, especially if the system has significant network delays.
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. In many cases, systems that appear to require linearizability in fact only really require causal consistency, which can be implemented more efficiently.

## Capturing causal dependencies

In order to maintain causality, you need to know which operation happened before which other operation. This is a partial order: concurrent operations may be processed in any order, but if one operation happened before another, then they must be processed in that order on every replica. Thus, when a replica processes an operation, it must ensure that all causally preceding operations (all operations that happened before) have already been processed; if some preceding operation is missing, the later operation must wait until the preceding operation has been processed.

### Sequence Number Ordering

We can use sequence numbers or timestamps to order events. Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a total order: that is, every operation has a unique sequence number, and you can always compare two sequence numbers to determine which is greater (i.e., which operation happened later).In particular, we can create sequence numbers in a total order that is consistent with causality:we promise that if operation A causally happened before B, then A occurs before B in the total order (A has a lower sequence number than B). Concurrent operations may be ordered arbitrarily.

### Non Causal sequence number generators

If there is not a single leader (perhaps because you are using a multi-leader or leaderless database, or because the database is partitioned), it is less clear how to generate sequence numbers for operations.

- Each node can generate its own independent set of sequence numbers. For example, if you have two nodes, one node can generate only odd numbers and the other only even numbers.
- You can preallocate blocks of sequence numbers. For example, node A might claim the block of sequence numbers from 1 to 1,000, and node B might claim the block from 1,001 to 2,000.

They generate a unique, approximately increasing sequence number for each operation. However, they all have a problem: the sequence numbers they generate are not consistent with causality. 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). Two nodes may sometimes have the same counter value, but by including the node ID in the timestamp, each timestamp is made unique.
The key idea about Lamport timestamps, which makes them consistent with causality, is the following: 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.
Lamport timestamps always enforce a total ordering. From the total ordering of Lamport timestamps, you cannot tell whether two operations are concurrent or whether they are causally dependent.

## Timestamp ordering is not sufficient

Consider a system that needs to ensure that a username uniquely identifies a user account. Total ordering(by Lamport Timestamps) is not sufficient when a node has just received a request from a user to create a username and needs to decide right now whether the request should succeed or fail. At that moment, the node does not know whether another node is concurrently in the process of creating an account with the same username, and what timestamp that other node may assign to the operation.
In order to be sure that no other node is in the process of concurrently creating an account with the same username and a lower timestamp, you would have to check with every other node to find out which timestamps it has generated.If one of the other nodes has failed or cannot be reached due to a network problem, this system would grind to a halt.
The problem here is that the total order of operations only emerges after you have collected all of the operations. If another node has generated some operations, but you don't yet know what they are,you cannot construct the final ordering of operations.
To conclude: in order to implement something like a uniqueness constraint for usernames, it's not sufficient to have a total ordering of operations—you also need to know when that order is finalized.

## Total Order Broadcast

Total order broadcast(or atomic 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.A correct algorithm for total order broadcast must ensure that the reliability and ordering properties are always satisfied, even if a node or the network is faulty.

Of course, messages will not be delivered while the network is interrupted, but an algorithm can keep retrying so that the messages get through when the network is eventually repaired (and then they must still be delivered in the correct order).

### Using total order broadcast

Total order broadcast is exactly what you need for database replication: if every message represents write to the database, and every replica processes the same writes in the same order, then the replicas will remain consistent with each other (aside from any temporary replication lag). This principle is known as state machine replication.

If every message represents a deterministic transaction to be executed as a stored procedure, and if every node processes those messages in the same order, then the partitions and replicas of the database are kept consistent with each other.
An important aspect of total order broadcast is that the order is fixed at the time the messages are delivered: a node is not allowed to retroactively insert a message into an earlier position in the order if subsequent messages have already been delivered. This fact makes total order broadcast stronger than timestamp ordering.

### Implementing linearizable storage using total order broadcast

Total order broadcast is asynchronous: messages are guaranteed to be delivered reliably in a fixed order, but there is no guarantee about when a message will be delivered (so one recipient may lag behind the others). By contrast, linearizability is a recency guarantee: a read is guaranteed to see the latest value written.
You can implement such a linearizable compare-and-set operation as follows by using total order broadcast as an append-only log:

- Append a message to the log, tentatively indicating the username you want to claim.
- Read the log, and wait for the message you appended to be delivered back to you.
- Check for any messages claiming the username that you want. If the first message for your desired username is your own message, then you are successful: you can commit the username claim (perhaps by appending another message to the log) and acknowledge it to the client. If the first message for your desired username is from another user, you abort the operation.

Because log entries are delivered to all nodes in the same order, if there are several concurrent writes, all nodes will agree on which one came first. Choosing the first of the conflicting writes as the winner and aborting later ones ensures that all nodes agree on whether a write was committed or aborted.

### Implementing total order broadcast using linearizable storage

The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation. Alternatively, an atomic compare-and-set operation would also do the job.
The algorithm is simple: for every message you want to send through total order broadcast, you increment and get the linearizable integer and then attach the value you got from the register as a sequence number to the message. You can then send the message to all nodes (resending any lost messages), and the recipients will deliver the messages consecutively by sequence number.
Note that, unlike Lamport timestamps, the numbers you get from incrementing the linearizable register form a sequence with no gaps. Thus, if a node has delivered message 4 and receives an incoming message with a sequence number of 6, it knows that it must wait for message 5 before it can deliver message 6. The same is not the case with Lamport timestamps—in fact, this is the key difference between total order broadcast and timestamp ordering.

# Distributed Transactions and Consensus

There are a number of situations in which it is important for nodes to agree. For example:

- Leader election
- Atomic commit In a database that supports transactions spanning several nodes or partitions, we have the problem that a transaction may fail on some nodes but succeed on others. If we want to maintain transaction atomicity (in the sense of ACID), we have to get all nodes to agree on the outcome of the transaction: either they all abort/rollback (if anything goes wrong) or they all commit (if nothing goes wrong). This instance of consensus is known as the atomic commit problem.

## Atomic Commit and Two-Phase Commit (2PC)

### From single-node to distributed atomic commit

For transactions that execute at a single database node, atomicity is commonly implemented by the storage engine. When the client asks the database node to commit the transaction, the database makes the transaction's writes durable (typically in a write-ahead log) and then appends a commit record to the log on disk. If the database crashes in the middle of this process, the transaction is recovered from the log when the node restarts: if the commit record was successfully written to disk before the crash, the transaction is considered committed; if not, any writes from that transaction are rolled back.
However, what if multiple nodes are involved in a transaction? For example, perhaps you have a multi-object transaction in a partitioned database, or a term-partitioned secondary index (in which the index entry may be on a different node from the primary data).
In these cases, it is not sufficient to simply send a commit request to all of the nodes and independently commit the transaction on each one. In doing so, it could easily happen that the commit succeeds on some nodes and fails on other nodes, which would violate the atomicity guarantee.
For This reason, a node must only commit once it is certain that all other nodes in the transaction are also going to commit.

## Introduction to two-phase commit

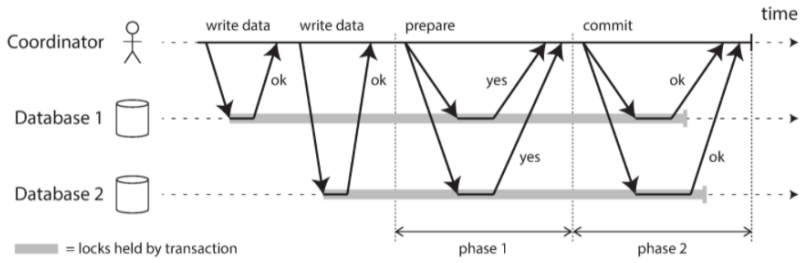



*Figure 9-9. A successful execution of two-phase commit (2PC).*

A two-phase commit is an algorithm for achieving atomic transaction commit across multiple nodes—i.e., to ensure that either all nodes commit or all nodes abort. 2PC is used internally in some databases and also made available to applications in the form of XA transactions (which are supported by the Java Transaction API, for example)
2PC uses a new component that does not normally appear in single-node transactions: a coordinator (also known as transaction manager).
A 2PC transaction begins with the application reading and writing data on multiple database nodes, as normal. We call these database nodes participants in the transaction. When the application is ready to commit, the coordinator begins phase 1: it sends a prepare request to each of the nodes, asking them whether they are able to commit. The coordinator then tracks the responses from the participants:

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

### A system of promises

To understand why it works, we have to break down the process in a bit more detail:

- When the application wants to begin a distributed transaction, it requests a transaction ID from the coordinator. This transaction ID is globally unique.
- The application begins a single-node transaction on each of the participants and attaches the globally unique transaction ID to the single-node transaction. All reads and writes are done in one of these single-node transactions. If anything goes wrong at this stage (for example, a node crashes or a request times out), the coordinator or any of the participants can abort.
- When the application is ready to commit, the coordinator sends a prepare request to all participants, tagged with the global transaction ID. If any of these requests fails or times out, the coordinator sends an abort request for that transaction ID to all participants.
- When a participant receives the prepare request, it makes sure that it can definitely commit the transaction under all circumstances. This includes writing all transaction data to disk (a crash, a power failure, or running out of disk space is not an acceptable excuse for refusing to commit later), and checking for any conflicts or constraint violations. By replying "yes" to the coordinator, the node promises to commit the transaction without error if requested.
- When the coordinator has received responses to all prepare requests, it makes a definitive decision on whether to commit or abort the transaction (committing only if all participants voted "yes"). The coordinator must write that decision to its transaction log on disk so that it knows which way it decided in case it subsequently crashes. This is called the commit point.
- Once the coordinator's decision has been written to disk, the commit or abort request is sent to all participants. If this request fails or times out, the coordinator must retry forever until it succeeds. If the decision was to commit, that decision must be enforced, no matter how many retries it takes. If a participant has crashed in the meantime, the transaction will be committed when it recovers—since the participant voted "yes," it cannot refuse to commit when it recovers.

Thus, 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.)

### Coordinator failure

If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction. But once the participant has received a prepare request and voted “yes,” it can no longer abort unilaterally—it must wait to hear back from the coordinator whether the transaction was committed or aborted. If the coordinator crashes or the network fails at this point, the participant can do nothing but wait. A participant’s transaction in this state is called in doubt or uncertain.
Without hearing from the coordinator, the participant has no way of knowing whether to commit or abort. In principle, the participants could communicate among themselves to find out how each participant voted and come to some agreement, but that is not part of the 2PC protocol.
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. Any transactions that don’t have a commit record in the coordinator’s log are aborted. Thus, the commit point of 2PC comes down to a regular single-node atomic commit on the coordinator.

## Distributed Transactions in Practice

On the one hand, distributed transactions are seen as providing an important safety guarantee that would be hard to achieve otherwise; on the other hand, they are criticized for causing operational problems, killing performance.
Two quite different types of distributed transactions are often conflated:
**Database-internal distributed transactions** Some distributed databases (i.e., databases that use replication and partitioning in their standard configuration) support internal transactions among the nodes of that database.
**Heterogeneous distributed transactions** In a heterogeneous transaction, the participants are two or more different technologies: for example, two databases from different vendors, or even non-database systems such as message brokers. A distributed transaction across these systems must ensure atomic commit, even though the systems may be entirely different under the hood.

### XA transactions

X/Open XA (short for eXtended Architecture) is a standard for implementing two-phase commit across heterogeneous technologies. XA is not a network protocol—it is merely a C API for interfacing with a transaction coordinator. Bindings for this API exist in other languages; for example, in the world of Java EE applications, XA transactions are implemented using the Java Transaction API (JTA).

### Holding locks while in doubt

Database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty reads. In addition, if you want serializable isolation, a database using two-phase locking would also have to take a shared lock on any rows read by the transaction.
The database cannot release those locks until the transaction commits or aborts them. Therefore, when using a two-phase commit, a transaction must hold onto the locks throughout the time it is in doubt. If the coordinator has crashed and takes 20 minutes to start up again, those locks will be held for 20 minutes. If the coordinator's log is entirely lost for some reason, those locks will be held forever—or at least until the situation is manually resolved by an administrator. While those locks are held, no other transaction can modify those rows. Depending on the database, other transactions may even be blocked from reading those rows.

### Recovering from coordinator failure

In practice, orphaned in-doubt transactions do occur—that is, transactions for which the coordinator cannot decide the outcome for whatever reason (e.g., because the transaction log has been lost or corrupted due to a software bug). These transactions cannot be resolved automatically, so they sit forever in the database, holding locks and blocking other transactions. Even rebooting your database servers will not fix this problem, since a correct implementation of 2PC must preserve the locks of an in-doubt transaction even across restarts (otherwise it would risk violating the atomicity guarantee). The only way out is for an administrator to manually decide whether to commit or rollback the transactions.
Many XA implementations have an emergency escape hatch called heuristic decisions: allowing a participant to unilaterally decide to abort or commit an in-doubt transaction without a definitive decision from the coordinator.

## 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 uniform agreement and integrity properties define the core idea of consensus: everyone decides on the same outcome, and once you have decided, you cannot change your mind.
The termination property formalizes the idea of fault tolerance. Even if some nodes fail, the other nodes must still reach a decision.
The best-known fault-tolerant consensus algorithms are Viewstamped Replication (VSR), Paxos, Raft, and Zab.
Most of these algorithms decide on a sequence of values, which makes them total order broadcast algorithms.
So, total order broadcast is equivalent to repeated rounds of consensus (each consensus decision corresponding to one message delivery):

- Due to the agreement property of consensus, all nodes decide to deliver the same messages in the same order.
- Due to the integrity property, messages are not duplicated.
- Due to the validity property, messages are not corrupted and not fabricated out of thin air.
- Due to the termination property, messages are not lost.

Viewstamped Replication, Raft, and Zab implement total order broadcast directly.

### 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 (called the ballot number in Paxos, view number inViewstamped Replication, and term number in Raft) 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.

Before a leader is allowed to decide anything, it must first check that there isn't some other leader with a higher epoch number that might take a conflicting decision. For every decision that a leader wants to make, it must send the proposed value to the other nodes and wait for a quorum of nodes to respond in favor of the proposal.

Thus, we have two rounds of voting: once to choose a leader and a second time to vote on a leader's proposal.

Thus, if the vote on a proposal does not reveal any higher-numbered epoch, the current leader can conclude that no leader election with a higher epoch number has happened, and therefore be sure that it still holds the leadership. It can then safely decide the proposed value.

### Limitations of consensus

- The process by which nodes vote on proposals before they are decided is a kind of synchronous replication.
- Consensus systems always require a strict majority to operate. If a network failure cuts off some nodes from the rest, only the majority portion of the network can make progress, and the rest is blocked.
- Most consensus algorithms assume a fixed set of nodes that participate in voting, which means that you can't just add or remove nodes in the cluster.
- Consensus systems generally rely on timeouts to detect failed nodes. In environments with highly variable network delays, especially geographically distributed systems, it often happens that a node falsely believes the leader to have failed due to a transient network issue.

# Batch Processing

**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. Jobs often take a while (from a few minutes to several days), so there normally isn't a user waiting for the job to finish. Instead, batch jobs are often scheduled to run periodically (for example, once a day). The primary performance measure of a batch job is usually throughput (the time it takes to crunch through an input dataset of a certain size).

# Stream Processing