Chapter 1. Introduction

Reliability: Fault Tolerance. - "continuing to work correctly, even when things go wrong."
TODO: (MTTF, MTTR)

Netflix Chaos Monkey:

Scalability:
Load:
Twitter(2012):
Post tweet - (4.6k requests/sec on average, over 12k requests/sec at peak)
View Timeline: (300k requests/sec).

Since load at write time is less, do a Fan out at write time to Timeline-Cache, so that we can process 300k reads/sec.

In a batch processing system such as Hadoop - we usually care about *throughput*

Imp: Average response time in terms of Mean, but the mean is not a very good metric if you want to know your "typical" response time, because it doesn't tell you how many users actually experienced that delay. PERCENTILES are more useful.
Median = p50
99the p = p99
99.9 = p999

Approaches for Coping with Load:

vertical scaling, moving to a more powerful machine) and *scaling out* (*horizontal scaling - hybrid approach is good.*

elastic, meaning that they can automatically add computing resources when they detect a load increase

- While distributing stateless services across multiple machines is fairly straightforward, taking stateful data systems from a single node to a distributed setup

For example, a system that is designed to handle 100,000 requests per second, each 1 kB in size, looks very different from a system that is designed for 3 requests per minute, each 2 GB in size—even though the two systems have the same data throughput

Maintainability

Operability
Simplicity
Evolvability

Simplicity: Managing Complexity

abstraction

Chapter 2. Data Models and Query Languages

Data Model:

Most applications are built by layering one data model on top of another. Each layer hides the complexity of the layers below by providing a clean data model. These abstractions allow different groups of people to work effectively.

Not Only SQL has a few driving forces:

- Greater scalability
- preference for free and open source software
- Specialised query optimisations
- Desire for a more dynamic and expressive data model

With a SQL model, if data is stored in a relational tables, an awkward translation layer is translated, this is called *impedance mismatch*. OO -> tables (ORM frameworks help)

Resume:

1:M relationship in SQL:

a) separate relationship tables (jobs, education)

b) JSON column

c) Text/Blob, interpreted by apple.

JSON for resume is more appropriate.

Document-oriented databases like MongoDB [9], RethinkDB [10], CouchDB [11], and Espresso [12] support this data model

```
{
  "user_id":     251,

  "first_name":  "Bill",

  "last_name":   "Gates",

  "summary":     "Co-chair of the Bill & Melinda Gates... Active
blogger.",

  "region_id":   "us:91",

  "industry_id": 131,

  "photo_url":   "/p/7/000/253/05b/308dd6e.jpg",

  "positions": [

    {"job_title": "Co-chair", "organization": "Bill & Melinda Gates
Foundation"},

    {"job_title": "Co-founder, Chairman", "organization": "Microsoft"}

  ],

  "education": [

    {"school_name": "Harvard University",       "start": 1973, "end":
1975},

    {"school_name": "Lakeside School, Seattle", "start": null, "end":
null}

```

```
  ],

  "contact_info": {

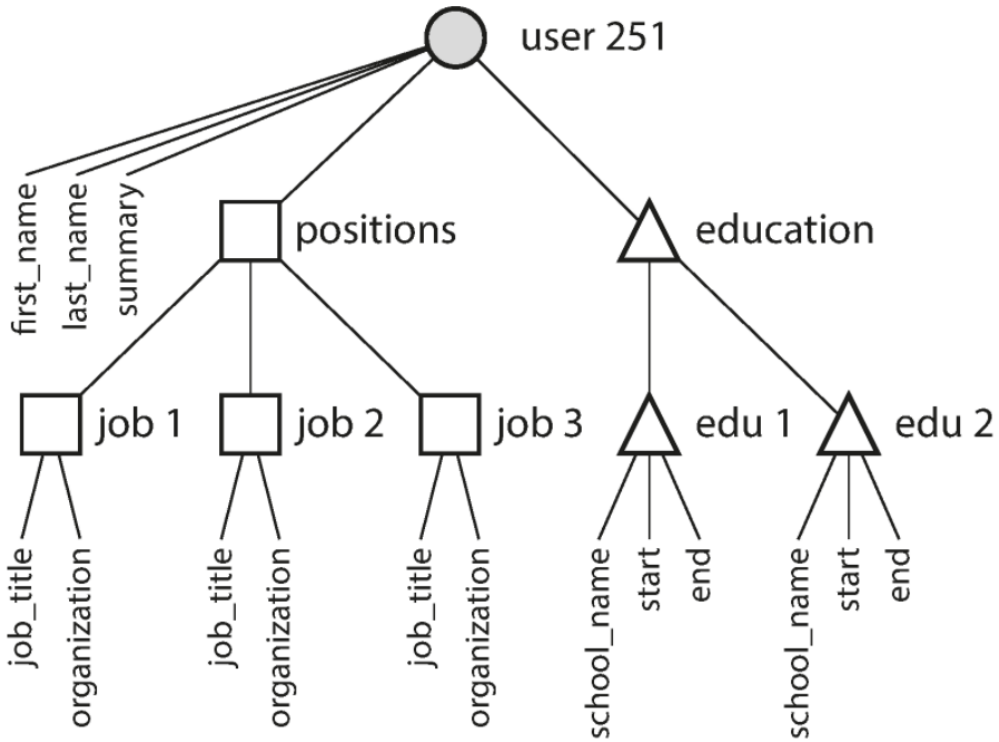
    "blog":    "http://thegatesnotes.com",

    "twitter": "http://twitter.com/BillGates"

  }

}
```

1:M -> Tree structure and JSON fits naturally.



Many-to-One and Many-to-Many Relationships:

<The most useful of these ??>

Adv of normalized storing:

- Consistent style and spelling across profiles
- Avoiding ambiguity
- Ease of updating
- Localization support e.g. when the site is transformed into other langauges
- Better search

Relational Versus Document Databases Today

<Which do you think is the biggest factor among these ?? >
Locality: <which one ?>
JSON representation has better *locality* than the multi-table SQL schema. All the relevant information is in one place, and one query is sufficient.

Joins: <which one ?>
Apple side joins in Json are slow and not natural. SQL has inbuilt support.

Schema flexibility:
schema-on-read -> dynamic runtime type checking *schema-on-write*

The main arguments in favour of the document data model are schema flexibility, better performance due to locality, and sometimes closer data structures to the ones used by the applications. The relation model counters by providing better support for joins, and many-to-one and many-to-many relationships.

Query languages for data:

SQL is a ***declarative*** query language. In an ***imperative*** *language*, you tell the computer to perform certain operations in order.

In a declarative query language you just specify the pattern of the data you want, but not *how* to achieve that goal.

Declarative >> ??

Query Optimizer (a new index added doesn’t change the query)

Parallelization

MapReduce querying

Graph-like data models:

If many-to-many relationships are very common in your application, it becomes more natural to start modelling your data as a graph.

A graph consists of *vertices* (*nodes* or *entities*) and *edges* (*relationships* or *arcs*).

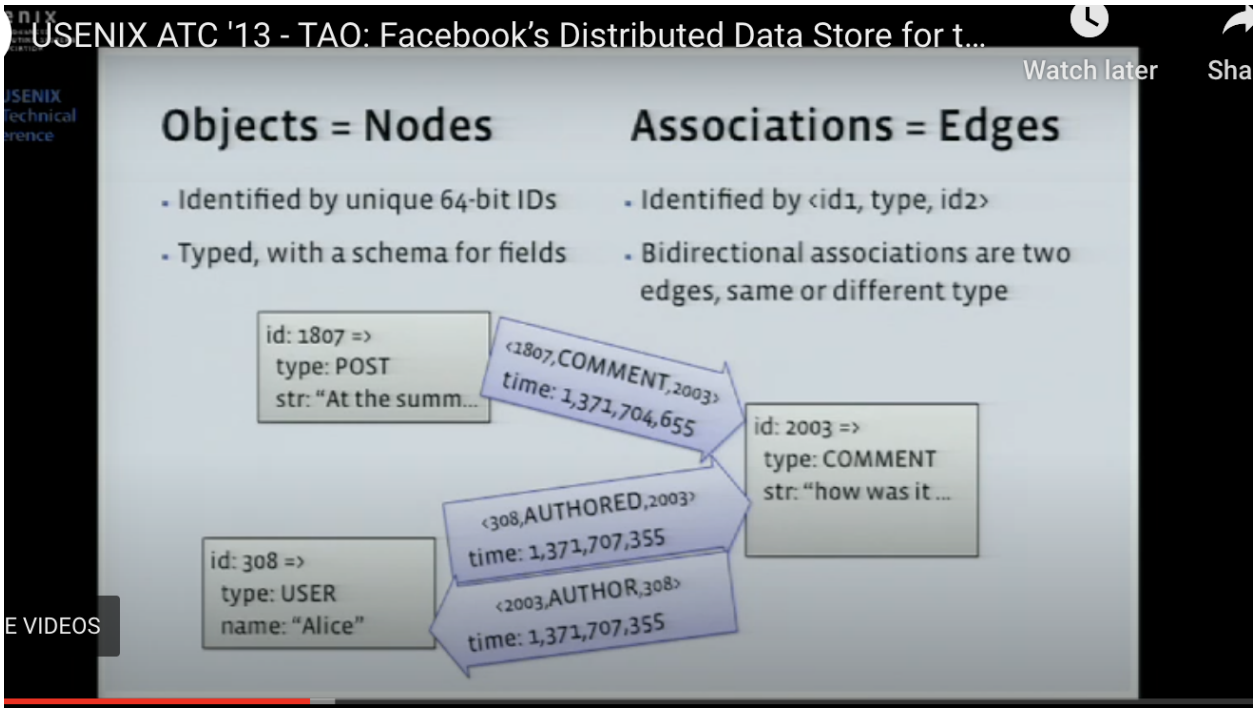
Best example ???

Social Network. (TAO by FB)

So in 2009, engineers at the social networking site started working on a database architecture that could perform better than its then-current relational database backed by Memcached for in-memory caching and MySQL for persistent storage. They built a graph database. The fruit of their efforts, The Associations and Objects (TAO) distributed data store, is a system purpose-built for the storage, expansion and, most importantly, delivery of the complex web of relationships among people, places and things that Facebook represents

We continue to use MySQL to manage persistent storage for TAO objects and associations.

We chose eventual consistency as the default consistency model for TAO.



Property graphs

Each vertex consists of:

- Unique identifier
- Outgoing edges
- Incoming edges
- Collection of properties (key-value pairs)

Each edge consists of:

- Unique identifier
- Vertex at which the edge starts (*tail vertex*)
- Vertex at which the edge ends (*head vertex*)
- Label to describe the kind of relationship between the two vertices
- A collection of properties (key-value pairs)

Graphs provide a great deal of flexibility for data modelling. Graphs are good for evolvability.

Chapter 3. Storage and Retrieval

log-structured storage engines, and *page-oriented* storage engines

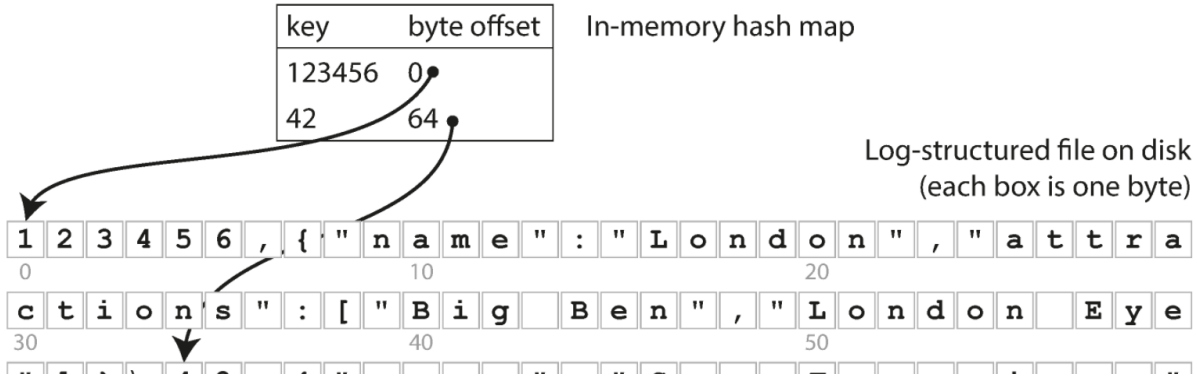
KV STore

Use just append only logs for writes.

Pros/cons ? Good perf for writes. Bad for reads.

Soln: Use Indexes.

Hash Index:



Problems ?

1. Can all keys fit in memory ? (Has to for this solution)
2. Evergrowing log file size ? (Segments of 1 GB and compact)
3. ()Every segment has its own index.

4. For reads, we have to start from the latest segment index and move backwards in search.

Details:

1. Append only logs: Sequenential Writes (HDD vs SSD)
1. File format: Binary - (length + raw string)
2. Tombstone for deletes
3. Snapshots for crash recovery
4. Checksums for inlaid writes due to crash.
5. Concurrency: one write, multiple readers (thinking deeper , they definitely operate at different offset, so never a need for locks)

CONS:

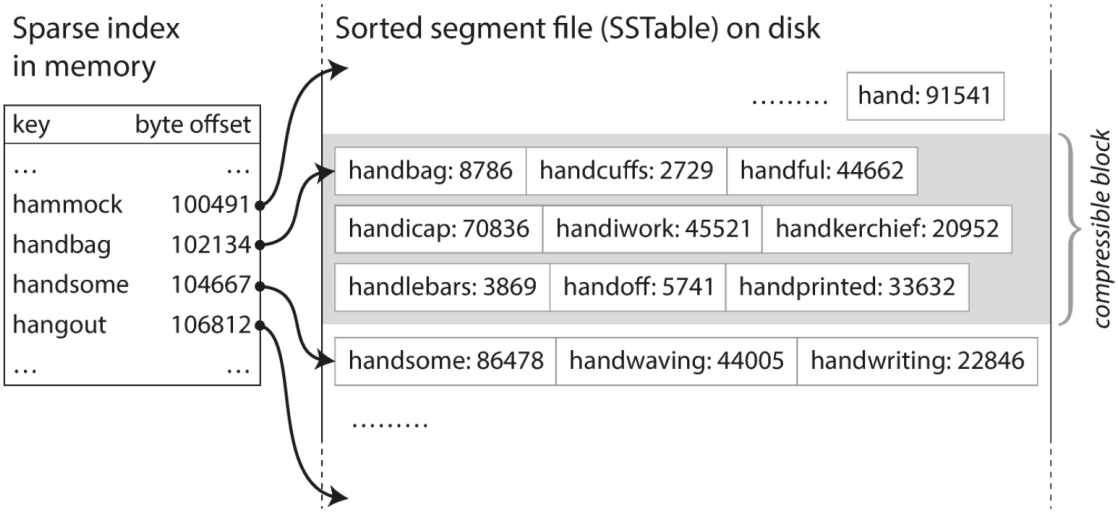
Hash index: No range query

Index needs to fit in memory.

SSTables and LSM-Trees :

Sorted String Tables:
Each Segment File: sorted by keys.

Sparse index:
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,



Since read requests need to scan over several key-value pairs in the requested range anyway, it is possible to group those records into a block and compress it before writing it to disk (indicated by the shaded area in Figure 3-5). 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.

Steps:

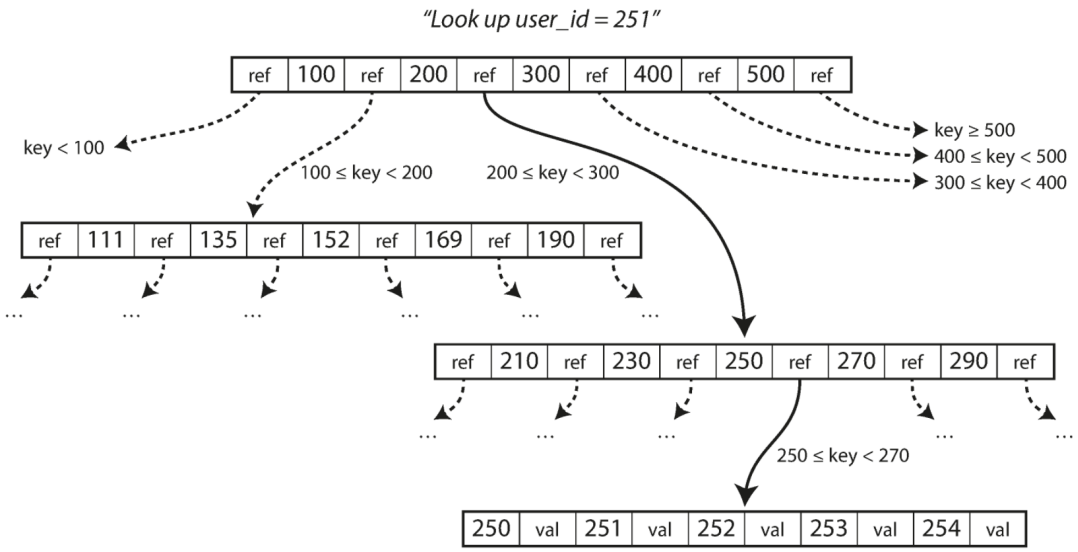
1. This in-memory tree is sometimes called a *memtable*
2. When the memtable gets bigger than some threshold—typically a few megabytes—write it out to disk as an SSTable file.
3. In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment,
4. From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values
5. WAL for recovery.
6. Bloom filter for each SSTable.

LevelDB [6] and RocksDB Cassandra and HBase Lucene (key-val index term-docList is maintained in memtable and stable)

Issue with compaction: GC/Heap pressure (Garbage Collection in Cassandra)

B-Trees:

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.



-> Eventually we get down to a page containing individual keys (a *leaf page*), which either contains the value for each key inline or contains references to the pages where the values can be found.

branching factor - several hundreds.

WAL - redo log for crash recovery.
protecting the tree's data structures with *latches* (lightweight locks).

B TREE optimizations:

- Instead of overwriting pages and maintaining a WAL for crash recovery, some databases (like LMDB) use a copy-on-write scheme [21]. A modified page is written to a different location, and a new version of the parent pages in the tree is created, pointing at the new location. This approach is also useful for concurrency control
- Many B-tree implementations therefore try to lay out the tree so that leaf pages appear in sequential order on disk.
- Sibling pointers

Rule of Thumb:
As a rule of thumb, LSM-trees are typically faster for writes, whereas B-trees are thought to be faster for reads [23]. Reads are typically slower on LSM-trees because they have to check several different data structures and SSTables at different stages of compaction.

write amplification:-

- 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)
- Log-structured indexes also rewrite data multiple times due to repeated compaction and merging of SSTables
- SSDs have issue with writing multiple times, due to erasure overhead. (And limits on number of erasures performed)

LSM vs Btrees:
Pros of LSM

1. LSM: Seq writes and less write amp.
2. LSM-trees can be compressed better, while Btree pages can cause fragmentation due to splits.

Cons of LSM:

1. Compaction can interfere with direct reads/writes, esp at the tail reflecting in p99+ latencies. Btrees are more predictable.
2. Disk bandwidth is shared between
    1. Memtable to sstable writes
    2. Compaction writes

Secondary indexes on non-unique fields.

Indexes:

1. Clustered,
2. non-clustered (refers to data in a heap file or in a clustered primary index)
3. Hybrid , covering index, only few columns are actually stored along with index.

4, Pros: Reads are fast, writes have to be slower (clustered, covering, due to duplication). What other issues with duplication ?

Multi column Index:
(Not supported by trees or lsm) RTrees are used.

Fuzzy Index:
LevelDB -> Sparse collection of key for a SSTable
Lucene -> FSA over a Trie.

In-memory DB:

- Use disk for WAL for crash recovery and periodic snapshots

the performance advantage of in-memory databases is not due to the fact that they don't need to read from disk. Then ??
Encoding in memory DS -> disk form

Anti caching:

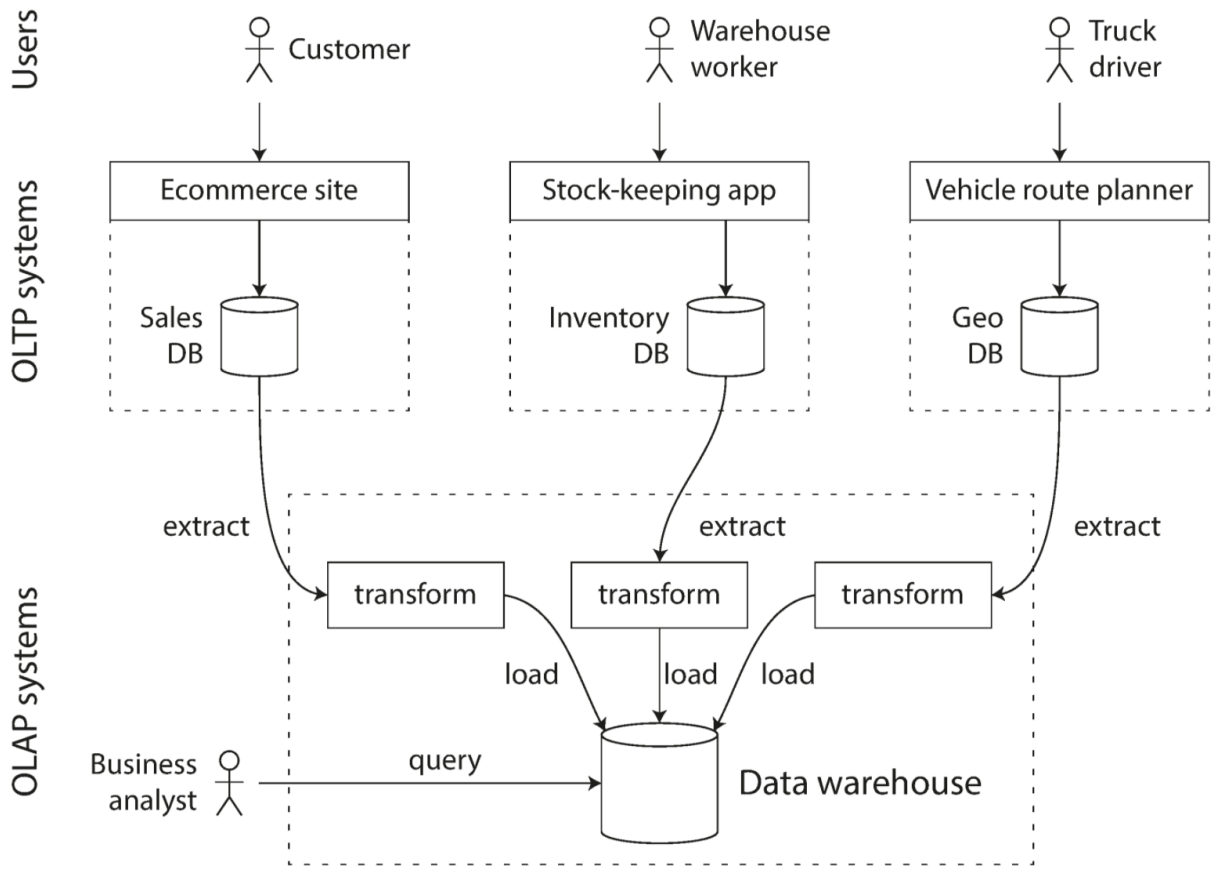
Unlike a traditional DBMS architecture, tuples do not reside in both places; each tuple is either in memory or in a disk block, but never in both places at the same time. In this new architecture, main memory, rather than disk, becomes the primary storage location. Rather than starting with data on disk and reading hot data into the cache, data starts in memory and cold data is evicted to the anti-cache on disk. This approach is similar to virtual memory swapping in operating systems (OS). With virtual memory, when the amount of data exceeds the amount of available memory, cold data is written out to disk in pages, typically in least recently used (LRU) order. When the evicted page is accessed, it is read back in, possibly causing other pages to be evicted. This allows the amount of virtual memory to exceed the amount of physical memory allocated to a process. Similarly, anti-caching allows the amount of data to exceed the available memory by evicting cold data to disk in blocks. If data access is skewed, the working set will remain in main memory.

- Fine-Grained Eviction: At tuple level, rather than whole pages as done by virtual memory
- Non blocking reads : no page faults (which block a process). Abort Tx immediately and restart at a later point.

OLAP vs OLTP:

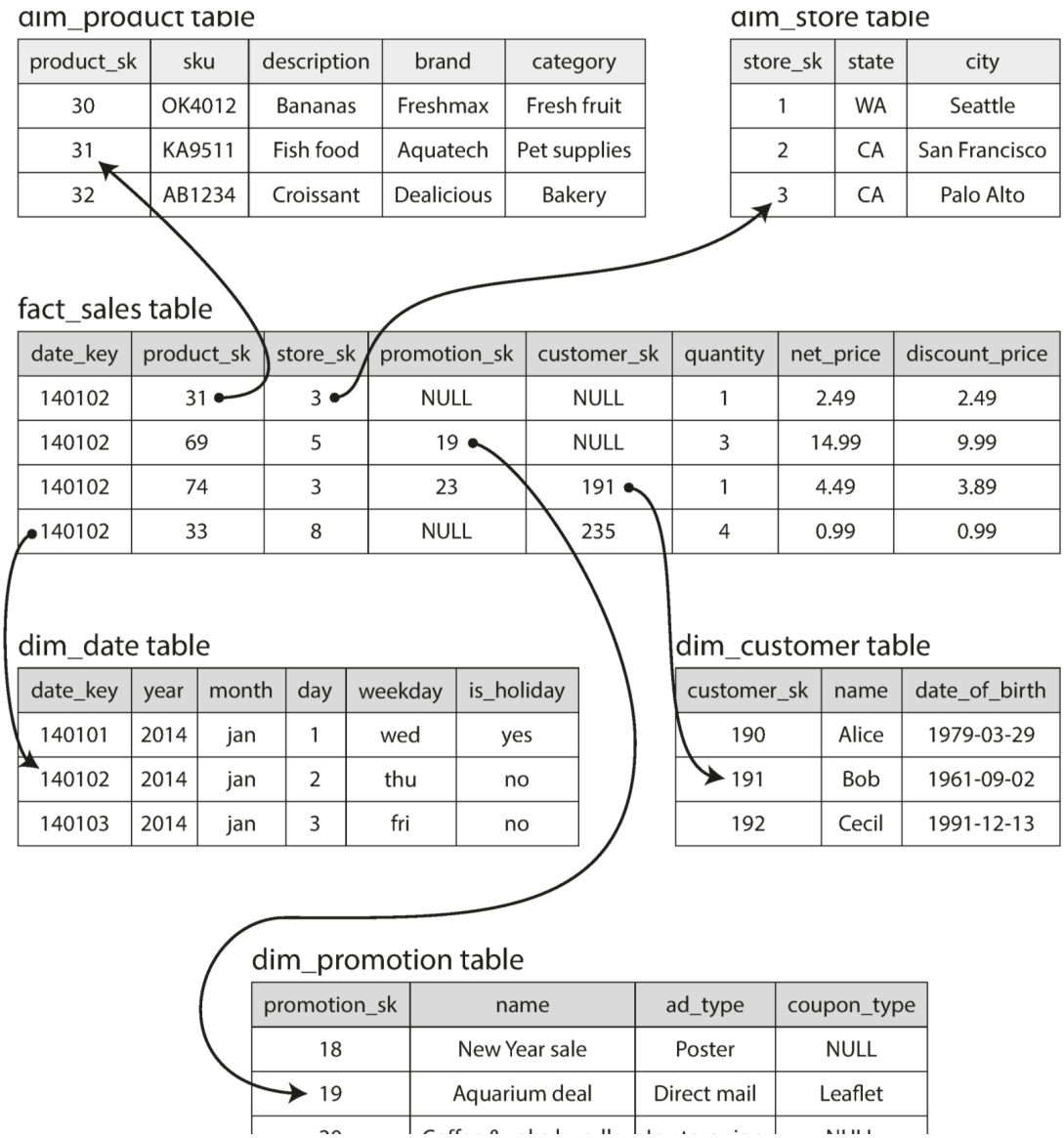
Property	Transaction processing systems (OLTP)	Analytic systems (OLAP)
Main read pattern	Small number of records per query, fetched by key	Aggregate over large number of records
Main write pattern	Random-access, low-latency writes from user input	Bulk import (ETL) or event stream
Primarily used by	End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

Data Warehousing:



ETL (extract, transform, load)

star schema (also known as *dimensional modeling:*



- facts are captured as individual events
- each row in the fact table represents an event, the dimensions represent the *who*, *what*, *where*, *when*, *how*, and *why* of the event
- 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*
- *snowflake schema:* where dimensions are further broken down into subdimensions. For example, there could be separate tables for brands and

product categories, and each row in the `dim_product` table could reference the brand and category as foreign keys, rather than storing them as strings in the `dim_product` table. Snowflake schemas are more normalized than star schemas

Column-Oriented Storage

- **Consider trillions of rows for fact tables with 100+ columns.**
- 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
- Column compression: e.g. *bitmap encoding:*

Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

-
- We can now take a column with *n* distinct values and turn it into *n* separate bitmaps: one bitmap for each distinct value, with one bit for each row. The bit is 1 if the row has that value, and 0 if not.
  - `WHERE product_sk IN (30, 68, 69):`
  - Load the three bitmaps for `product_sk = 30`, `product_sk = 68`, and `product_sk = 69`, and calculate the bitwise *OR* of the three bitmaps, which can be done very efficiently.
- *`WHERE product_sk = 31 AND store_sk = 3:`*
  - Load the bitmaps for `product_sk = 31` and `store_sk = 3`, and calculate the bitwise *AND*. This works because the columns contain the rows in the same order, so the *k*th bit in one column's bitmap corresponds to the same row as the *k*th bit in another column's bitmap.

- 2 bandwidths for large data

    - Disk to memory b/w
    - Memory to cpu b/w
- TODO: SIMD, Efficient use of L1 cache, Vectorized Processing.

Define a sorted order for columns, primary sort column, secondary …..
Then sort all the rows using that order,
Then store the columns,

SEVERAL DIFFERENT SORT ORDERS (very clever, C-Store, Vertica uses this)

Any how we have replicas, why not each replica store in different sort orders, so that a query can be served by an appropriate sort odered replica.

Writing to Column-Oriented Storage:

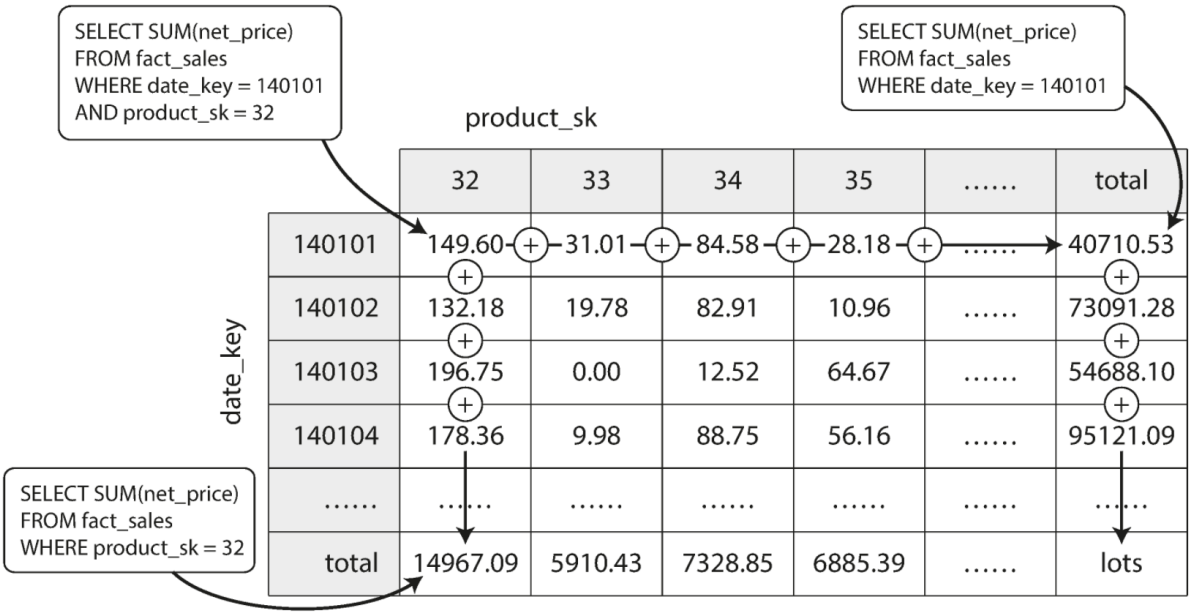
Use the same LSM concept - Vertica

materialized view:
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

- When the underlying data changes, a materialized view needs to be updated, because it is a denormalized copy of the data.

OLAP Cube:

A common special case of a materialized view is known as a *data cube* or *OLAP cube* [64]. It is a grid of aggregates grouped by different dimensions. Figure 3-12 shows an example.



You can now draw a two-dimensional table, with dates along one axis and products along the other. Each cell contains the aggregate (e.g., `SUM`) of an attribute (e.g., `net_price`) of all facts with that date-product combination.

Chapter 4 - Encoding and Evolution

(Cut from - https://github.com/ResidentMario/designing-data-intensive-applications-notes/blob/master/Chapter%204%20---%20Encoding%20and%20Evolution.ipynb)

- In memory we consider data in terms of the data structures they live in.
- On disc we work with memory which has been **encoded** into a sequence of bytes somehow.
- The process of transliteration between these two formats is known as serialization, encoding, or marshalling. A specific format is a **data serialization format**

Human-readable data interchange formats

- JSON and XML (and CSV, and the other usual suspects) are common **data interchange formats**, meant to be moved between application boundaries.
- These formats are considered to be lowest common denominators, however.
- They have parsing problems. For example, it's often impossible to difficult to determine the type of an object.
- Being human-readable, they are also inefficient in resource terms when performing network transfers.

Binary data interchange formats

Google Protobufs and Apache Thrift
machine-generate APIs for working with the data in your language of choice

- In the context of data interchange formats this is known as **schema evolution**.

Thrift and Protobuf

- Fields in Protobuf (and in Thrift) are identified by field IDs.
- These field IDs can be omitted if the field is not required, in which case they will simply be removed from the encoded byte sequence.
- But once assigned, they cannot be changed, as doing so would change the meaning of past data encodings.
- This provides backwards compatibility. There is one catch however. You cannot mark new fields required, for the obvious reason that doing so would cause all old encodings to fail.
- How about forward compatibility? Every tag is provided a type annotation. When an old reader reads new data, and finds a type it doesn't know about, it skips it, using the type annotation to determine how many bytes to skip.

```
message Person {

    required string user_name       = 1;

    optional int64  favorite_number = 2;

```

```
    repeated string interests       = 3;

}
```

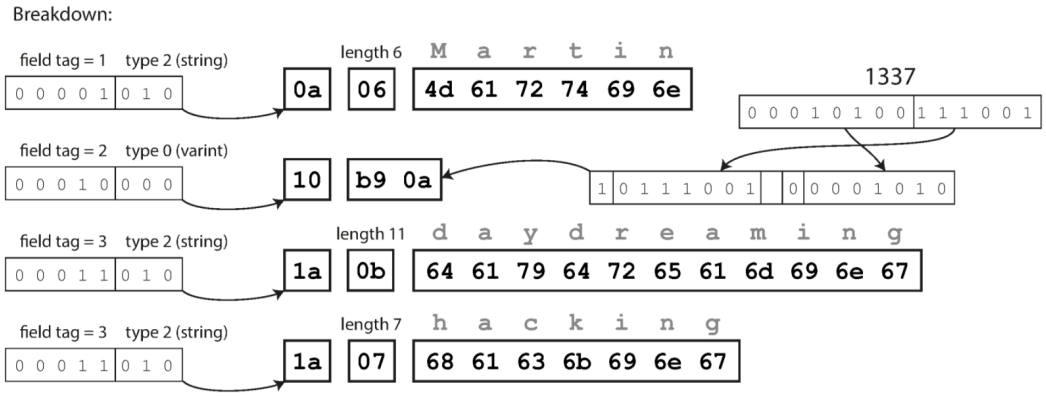


Figure 4-4. Example record encoded using Protocol Buffers.

AVRO:

- Apache Avro, is that these formats must be resiliant to differences between the **reader schema** and the **writer schema**. The challenge is to have a reader that understands every possible version of the writer.
- Avro requires you provide version information on read, which the other two formats do not require. This is additional overhead, as you must either encode that information in the file or provide it through some other means (the former is better if the file is big, and the latter if the file(s) are small).
- This allows Avro to omit data tags. This in turn makes Avro much easier to use with a dynamic schema, e.g. one that is changing all the time.

Data flow

1. Databases:

Databases are the first data flow concept.

Both backwards and forward compatibility matters in a database. Backwards compatibility is important because a database is fundamentally messages to your future self. Forward compatibility matters because when you perform rolling updates, your database will be accessed concurrently by both newer and older versions of software.

2. REST / RPC:

RESTful Services

RPC have issue with latency and n/w unreliability. => Retries => Idempotent on duplicate attempts

3. Message Queues:

    Loose coupling

TODO: UTF-8
TODO: REST Vs RPC
TODO: HTTP/2

Chapter 5. Replication

Reasons:

1. Geo access nearby
2. Availability
3. Read Throughput

Issue:
How to propagate changes to all replicas.

Types:

- *single-leader*,
- *multi-leader*, and
- *leaderless* replication

Characteristics:

- Sync
- Async

Leaders and Followers: (SINGLE LEADER)

Leaders/Followers
Active/Passive
Master/Slave

1. Write always to the master
2. Replication log is sent to followers/slaves
3. Read is accepted by master and slaves.

1. Sync Followers: Wait till all followers update and acknowledge before terming the write as successful.

(Not practical, as if a node is down, then the write blocks)

2. Semi-syncrhonous (most used approach): One node is in Sync, rest are replicated Async.

Setting Up New Followers:

1. consistent snapshot of the leader's database at some point in time—if possible, without taking a lock on the entire databaseon the entire database. Most databases have this feature, as it is also required for backups.
2. Copy the snapshot to the new follower node.
3. The follower connects to the leader and requests all the data changes that have happened since the snapshot was taken. This requires that the snapshot is associated with an exact position in the leader's replication log. That position has various names: for example, PostgreSQL calls it the *log sequence number*, and MySQL calls it the *binlog coordinates*.
4. When the follower has processed the backlog of data changes since the snapshot, we say it has *caught up*.

Handling Node Outages:

FOLLOWER FAILURE: CATCH-UP RECOVERY:

1. sync with server based on last synced log.

LEADER FAILURE: FAILOVER

1. *Determining that the leader has failed : timeout*
2. *Leader election*
3. *Reconfiguring the system to use the new leader*

Failover Issues:

- If asynchronous replication is used the replica that is elected the leader may be "missing" some transaction history which occurred in the leader. Totally ordered consistency goes wrong.
- You can discard writes in this case, but this introduces additional complexity onto any coordinating services that are also aware of those writes (such as cache invalidation).
- It is possible for multiple nodes to believe they are the leader (Byzantine generals). This is dangerous as it allows concurrent writes to different parts of the system. E.g. **split brain**.
- Care must be taken in setting the timeout. Timeouts that are too long will result in application delay for users. Timeouts that are too short will cause unnecessary failovers during high load periods.

Implementation of Replication Logs:

1. **STATEMENT-BASED REPLICATION:**
    1. Issue with now() and rand(), non-pure Fns()
2. **(WAL) SHIPPING**:
    1. (both LSM and Btrees already have WAL): Record dumps are a data storage implementation detail byte by byte with offset info. That's an issue in upgrades and needs downtime, as you can't do this [Upgrade followers first and then do FAILOVER]
3. **LOGICAL (ROW-BASED) LOG REPLICATION**
    1. Basically a sequence of records describing writes to database tables at the granularity of a row
    2. Decoupled logical log with storage engine format.

4. Application level replication:
    1. Capture the change, and do partial/selective/full replication at application layer. E.g. Triggers

Problems with Replication Lag:

Fewer writes and Lots of reads :

- Scale by adding many read replicas.
- Only works with async replication => replication lag

Issues:

1. Reading Your Own Writes:

read-after-write consistency, also known as *read-your-writes consistency* : users can see their own modified data on refresh

Solution:

1. easiest way is to simply have the leader handle read requests for user-owned data.
2. track the time of the last update and, for one minute after the last update, make all reads from the leader
3. 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
4. *cross-device* read-after-write consistency: (Still, handling multiple clients - hard)

2.Monotonic Reads:

(No going back in time):

Another issue occurs if a user is reading from replica, and then switches to another replica that is further out of sync than the previous one. This creates the phenomenon of going backwards in time, which is bad.

- An easy way to get monotonic reads is to have the user always read from the same replica (e.g. instead of giving them a random one).

3. Consistent Prefix Reads (causal inconsistency):

Psychic answering 🙂

1. Mainly in partitioned DBs.
2. A: Partition 1
   B: Partition 2
   Conv:
   A: Time ?
   B: 1.00 pm
   Client:

   B: 1.00 pm (Data coming from Partition 2 follower with little lag)

   A: Time ? (Data coming from Partition 1 follower with huge lag)

Solution:

- One solution is to make sure that any writes that are causally related to each other are written to the same partitio
- There are also algorithms that explicitly keep track of causal dependencies, a topic that we will return to in "The "happens-before" relationship and concurrency"

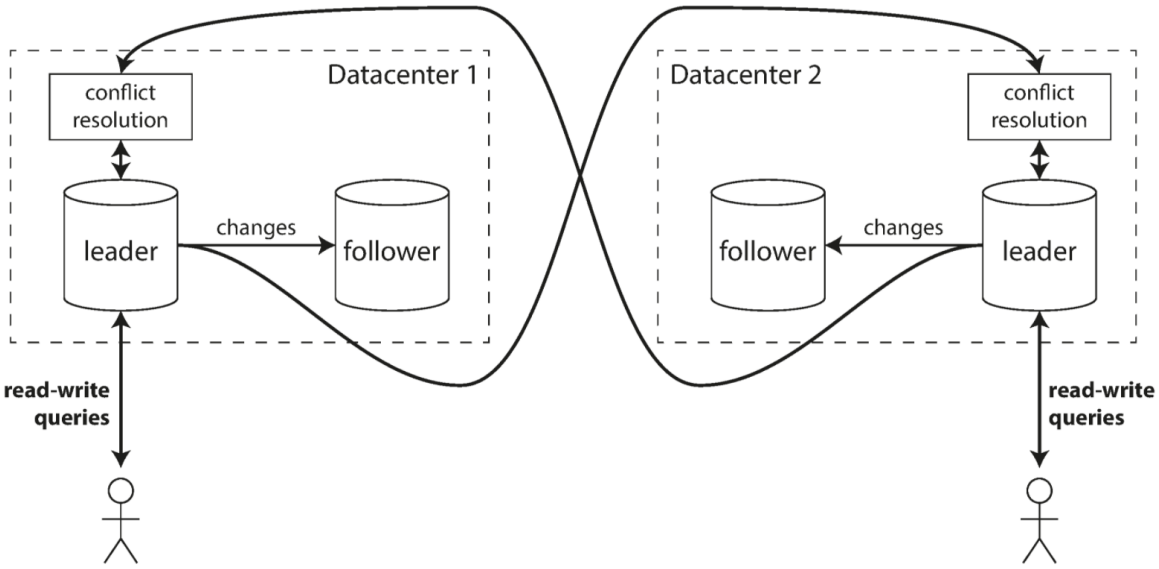
Multi-Leader Replication:

Why ?

- **Single leader - SPOF.**

(*master–master* or *active/active replication*)

Mainly for Multi DC.



- In a single leader, writes from across GEO, take long time, as writes are synchronous, this is an issue.
- Entire DC outage doesn't affect multi master.

Issues:

- The fundamental new problem that multi-leader architectures introduce is the fact that concurrent authoritative writes may occur on multiple leaders.

- Thus your application has to implement some kind of **conflict resolution**. This can occur on the backend side, or it may be implemented on the application side. It may even require user intervention or prompting.

Use Cases:

1. **CLIENTS WITH OFFLINE OPERATION:**

- calendar apps on your mobile phone (Its like a DC that's offline a lot)

2. COLLABORATIVE EDITING: However, for faster collaboration, you may want to make the unit of change very small (e.g., a single keystroke) and avoid locking. This approach allows multiple users to edit simultaneously, but it also brings all the challenges of multi-leader replication, including requiring conflict resolution

Handling Write Conflicts:

A -> B by user1
A -> C by user2
At the same time, both succeeds at different leaders.

Conflict Avoidance:
Record affinity to a particular leader. But Geo-locality is lost and failure scenarios are an exception.

Conflict Resolution:

- **LWW - > Last Write Wins, if each write Is associated with a sortable ID based on timestamp.**
- **Append rather than overwrite when conflict is detected.**
- **External repair process.**

CUSTOM CONFLICT RESOLUTION LOGIC: (Most widely used @ APP Level):

On write (by script)

As soon as the database system detects a conflict in the log of replicated changes, it calls the conflict handler.

On read (by application detecting multiple returns)

When a conflict is detected, all the conflicting writes are stored. The next time the data is read, these multiple versions of the data are returned to the application. The application may prompt the user or automatically resolve the conflict, and write the result back to the database. CouchDB works this way, for example.

Other ways of conflict resolution:

- *Conflict-free replicated datatypes* (CRDTs) [32, 38] are a family of data structures for sets, maps (dictionaries), ordered lists, counters
- *Mergeable persistent data structures* [41] track history explicitly, similarly to the Git version control system, and use a three-way merge function (whereas CRDTs use two-way merges).
- *Operational transformation* [42] is the conflict resolution algorithm behind collaborative editing applications such as Etherpad [30] and Google Docs [31]. It was designed particularly for concurrent editing of an ordered list of items, such as the list of characters that constitute a text document.

In All -All topology: where a leader sends its writes to all other leaders.

Messages may arrive out of order.

e.g update before the actual insert.

Solution: Use Vector Clocks.

Leaderless Replication: (Dynamo, Cassandra)

In some leaderless implementations, the client directly sends its writes to several replicas, while in others, a coordinator node does this on behalf of the client

READ REPAIR AND ANTI-ENTROPY:

Read repair

When a client makes a read from several nodes in parallel, it can detect any stale responses. For example, in Figure 5-10, 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 stale value and writes the newer value back to that replica. This approach works well for values that are frequently read.

Anti-entropy process

In addition, some datastores have a background process that constantly looks for differences in the data between replicas and copies any missing data from one replica to another. 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:

Quorum consistency:
2 things:
a) how many node failures can be tolerated (n-r) and (n-w)

b) how to get consistent reads

r + w > n (no. of replicas) [guaranteed 1 overlap]

- 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
- a workload with few writes and many reads may benefit from setting *w* = *n* and *r* = 1. This makes reads faster, but has the disadvantage ??
- Normally, reads and writes are always sent to all *n* replicas in parallel. The parameters *w* and *r* determine how many nodes we wait for—i.e., how many of the *n* nodes need to report success before we consider the read or write to be successful.

Issues in Quorum consistency: (situations where stale reads can still happen)

- Sloppy Quorum, can write to a different node other than replica
- Clock Skew
- Concurrent Read write (unsure wether new or old value is returned by read)
- On failure to write in at least w replicas, no rollback is done.
- If a node carrying a new value fails, and its data is restored from a replica carrying an old value, the number of replicas storing the new value may fall below *w*, breaking the quorum condition.

Sloppy Quorums and Hinted Handoff:

- Is it better to return errors to all requests for which we cannot reach a quorum of *w* or *r* nodes?
- 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?
- Sloppy quorums are particularly useful for increasing write availability, but they can lead to stale reads.
- Handoff occurs when the downed nodes come back

Quorum and **MULTI-DATACENTER OPERATION:**

CAssandra:

n is spread across data centers, but r and w we try to get it from the local datacenter

RIAK:

n is all within datacenter and additional replicas outside data center.

Detecting Concurrent Writes:

Even with strict quorum , conflicts bound to occur. Why ?

- Different nodes may see writes in different orders.
- in concurrent writes, only one will eventually prevail.
    - Which one ? (Conflict resolution - LLW, even if its a failed write)

THE "HAPPENS-BEFORE" RELATIONSHIP AND CONCURRENCY: (Causal)

- An operation A *happens before* another operation B if B knows about A, or depends on A, or builds upon A in some way.
- Concurrent operations - (Not necessarily at the same time), as long as there is no happens before relationship, the 2 operations are logically concurrent, (at same time — ? Can't tell exactly due to clock skew)

- The server maintains a version number for every key, increments the version number every time that key is written, and stores the new version number along with the value written.

- When a client reads a key, the server returns all values that have not been overwritten, as well as the latest version number. A client must read a key before writing.
- When a client writes a key, it must include the version number from the prior read, and it must merge together all values that it received in the prior read. (The response from a write request can be like a read, returning all current values, which allows us to chain several writes like in the shopping cart example.)
- When the server receives a write with a particular version number, it can overwrite all values with that version number or below (since it knows that they have been merged into the new value), but it must keep all values with a higher version number (because those values are concurrent with the incoming write).

[

- When a write includes the version number from a prior read, that tells us which previous state the write is based on.
- ESSENCE - When you write, you tell which state the write is based on, and also you merge the previous state with the current value you want to write.
- Both on read and write, you get back all the valid reads.

]

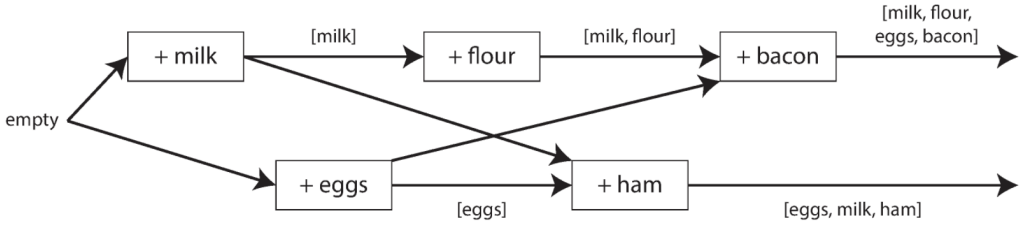


Figure 5-14. Graph of causal dependencies in Figure 5-13.

Now on the last read, the client gets

[milk, flour, eggs, bacon] AND [eggs, milk, ham]

The client/application choses to merge.

Version vectors:

The collection of version numbers from all the replicas is called a *version vector*

When a replica updates a key it will increment only it's own entry in the vector. An replica's entry is a summary of all the updates that actor has performed. As each entry is a summary of an replica's updates, the whole Version Vector summarises all the updates to the key from all the replicas in the system.

```
A = [{a, 3}, {b, 2}, {c, 1}]
B = [{a, 4}, {b, 2}, {c, 1}]
```

In Version Vector **`A`** above three replicas have updated the key. Actor **`a`** has issued three updates, **`b`** two, and **`c`** has issued one.

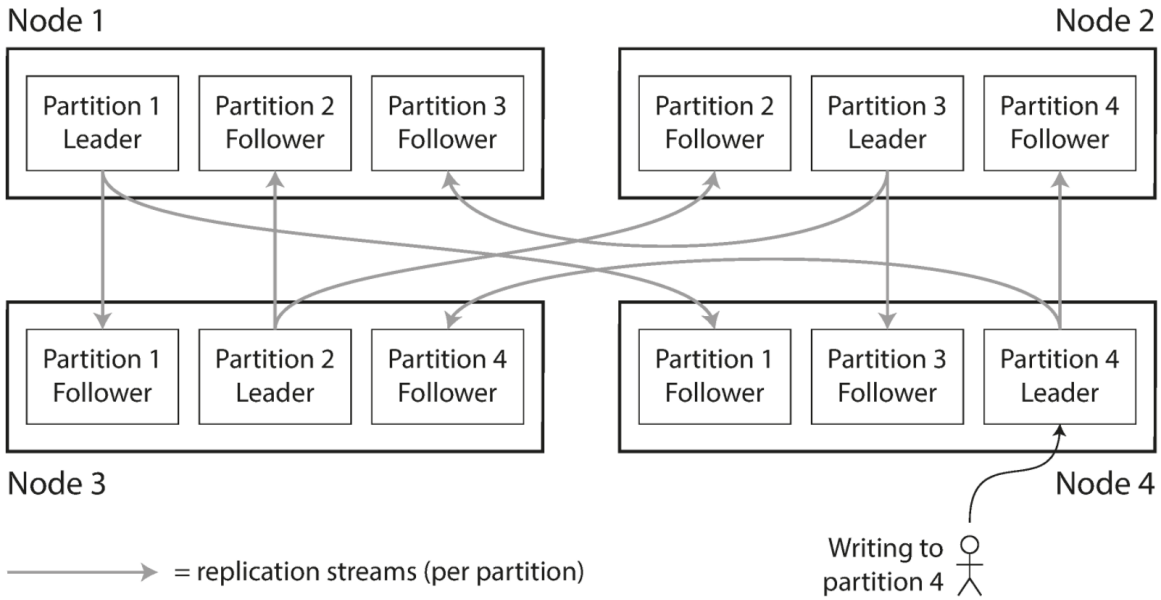
A = B or A > B or A < B or A conflict B.

https://riak.com/posts/technical/vector-clocks-revisited/index.html?p=9545.html

Chapter 6. Partitioning

Partitioning and Replication

Partitioning is usually combined with replication so that copies of each partition are stored on multiple nodes.



A partition with disproportionately high load is called a *hot spot*.

Partitioning by Key Range:

HBAse

- **Hotspots.**
- **If key is timestamp_day alone -> bad, include sensor_name in key.**

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.

- Range of hashes to Node => Consistent Hashing.
- Lose range query capability in hash partitioning.
- Cassandra achieves a compromise between the two partitioning strategies [PartitionKey + Clustering Keys]

HotSpots:
hashing a key to determine its partition can help reduce hot spots. However, it can't avoid them entirely: in the extreme case where all reads and writes are for the same key, you still end up with all requests being routed to the same partition.

Soln:

- responsibility of the application to reduce the skew. For example, if one key is known to be very hot, a simple technique is to add a random number to the beginning or end of the key. Just a

two-digit decimal random number would split the writes to the key evenly across 100 different keys, allowing those keys to be distributed to different partitions.

- it only makes sense to append the random number for the small number of hot keys;

Partitioning Secondary Indexes by Document:

- each partition is completely separate: each partition maintains its own secondary indexes, covering only the documents in that partition
- Local index
- querying a partitioned database is sometimes known as *scatter/gather*, and it can make read queries on secondary indexes quite expensive

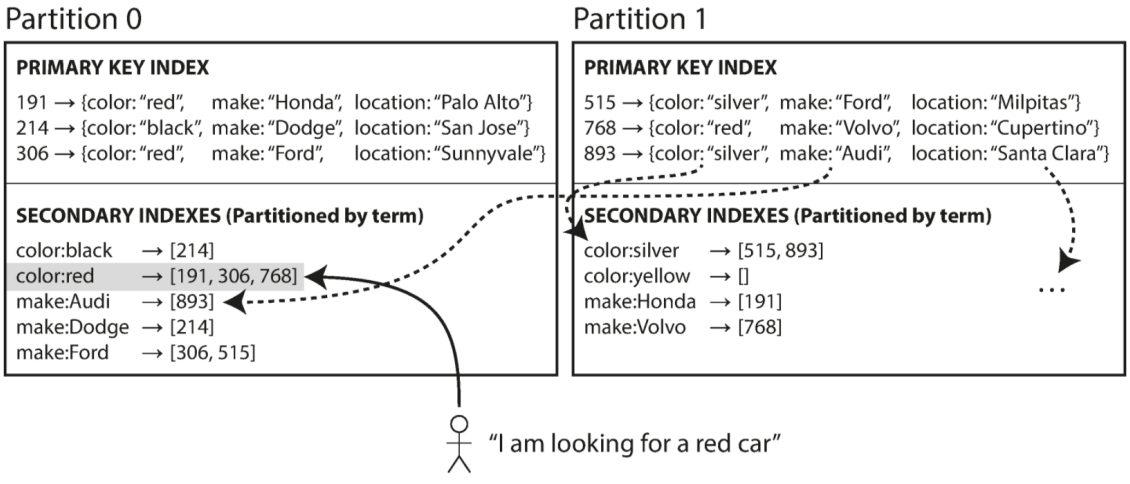


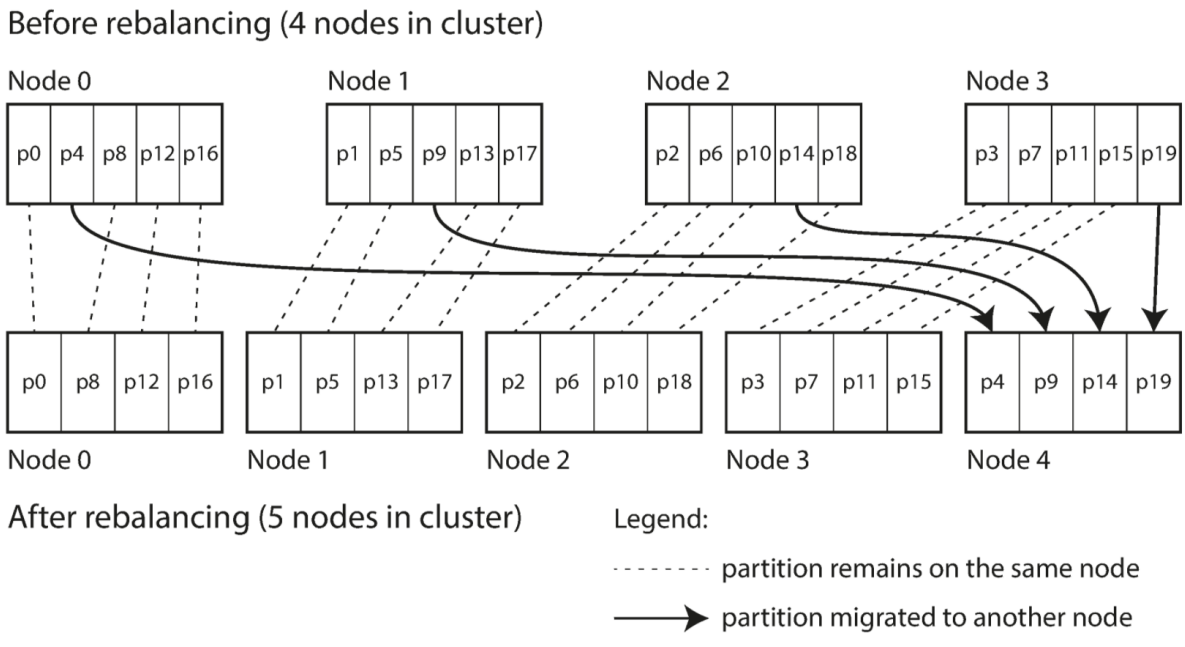
Figure 6-5. Partitioning secondary indexes by term.

Partitioning Secondary Indexes by Term

- Global Index
- Separate partitioning scheme for the secondary index.
- it can make reads more efficient:
- writes are slower 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).

Rebalancing Partitions:

FIXED NUMBER OF PARTITIONS:



DYNAMIC PARTITIONING (RANGE BASED PARTITIONING)

HBASE.

- the number of partitions is proportional to the size of the dataset, and if a partition exceeds a size, its split (just like BTree)
- In the case of key-range partitioning, pre-splitting requires that you already know what the key distribution is going to look like

Cassandra - Consistent Hashing.

Request Routing

- *service discovery*

a) any node can act as a routing node
b) separate routing tier
c) smart clients

- Each node registers itself in **ZooKeeper**, and ZooKeeper maintains the authoritative mapping of partitions to nodes. Other actors, such as the routing tier or the partitioning-aware client, can subscribe to this information in ZooKeeper. Whenever a partition changes ownership, or a node is added or removed, ZooKeeper notifies the routing tier so that it can keep its routing information up to date
- Gossip Protocol by Cassandra

Chapter 7. Transactions

ACID:

- Atomicity: All or Nothing.
- Consistency: App level characteristic, user makes sure data is consistent in whatever sense it means to them.
- Isolation: (Serializability)Operations are **isolated** if they cannot see the results of partially applied work by other operations. In other words, if two operations happen concurrently, any one will see the state either before or after the changes precipitated by its peer, but never in the middle.
    - serializable isolation is rarely used
    - Oracle “serializable,” - *snapshot isolation*, which is a weaker guarantee than serializability
- A database is **durable** if it doesn't lose data. In practice nothing is really durable, but there are different levels of guarantees.

- An operation is **single-object** if it only touches one record in the database (one row, one document, one value; whatever the smallest unit of thing is). It is **multi-object** if it touches multiple records.
- Basically all stores implement single-object operation isolation and atomicity. Not doing so would allow for partial updates to records in cases of failures (e.g. half-inserting an updated JSON blob right before a crash), which is Very Bad.
- Isolated and atomic multi-object operations ("strong transactions") are more difficult to implement because they require maintaining groups of operations, they get in the way of high availability, and they are hard to make work across partitions.

Issues with Retrying:

- Non idempotent
- 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.
- If the transaction also has side effects outside of the database, those side effects may happen even if the transaction is aborted.

Weak Isolation Levels:

- *serializable* isolation means that the database guarantees that transactions have the same effect as if they ran *serially*

Read Committed

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

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

NO DIRTY READS:

any writes by a transaction only become visible to others when that transaction commits
2 bad scenarios

- Unread email count, unread emails
- Rolled back writes

How ? 2 versions, old value, uncommitted value.

NO DIRTY WRITES:

if two transactions concurrently try to update the same object in a database

the later write overwrites an uncommitted value

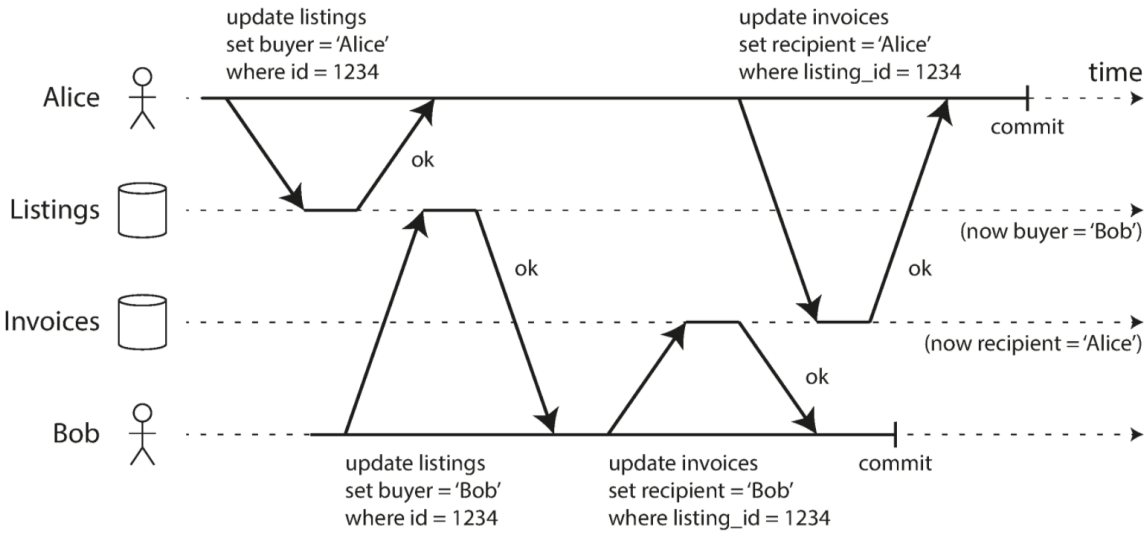


Figure 7-5. With dirty writes, conflicting writes from different transactions can be mixed up.

How ? Row level lock. (2nd trans. Either waits for lock or aborts)

Issue with Read Committed: (Read Skew)

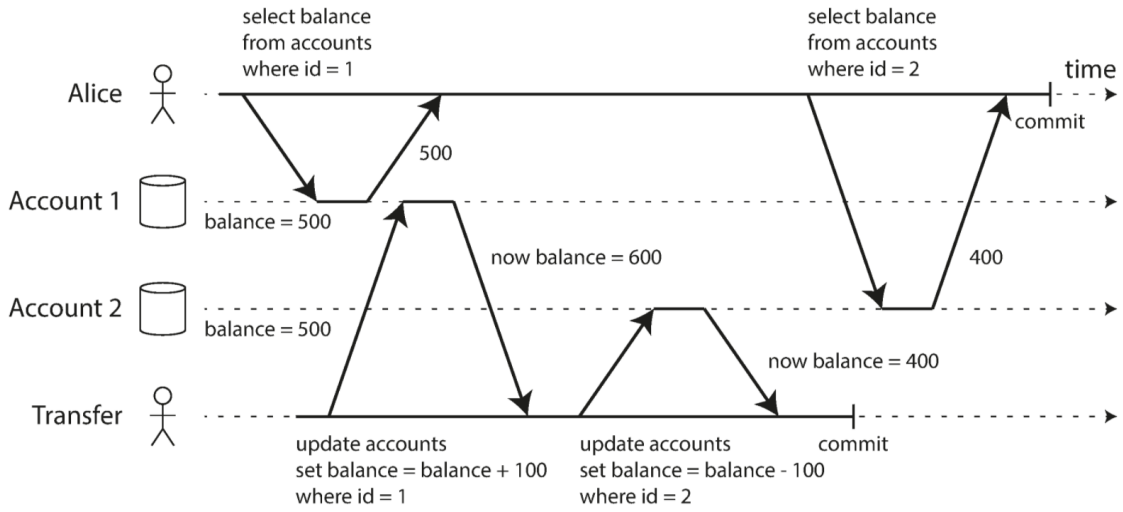


Figure 7-6. Read skew: Alice observes the database in an inconsistent state.

nonrepeatable read (Something changed as this trans. Was executing => You read again and see a different value.)

Snapshot Isolation and Repeatable Read:

Frozen:

Snapshot isolation [28] 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.

Esp helpful in long running queries like
a) backups
b) analytical queries

- a key principle of snapshot isolation is *readers never block writers, and writers never block readers*

Solution: MVCC

- 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.
- unique, always-increasing[vii] transaction ID (`txid`)

(HOW ?)

VISIBILITY RULES FOR OBSERVING A CONSISTENT SNAPSHOT

When a transaction reads from the database, transaction IDs are used to decide which objects it can see and which are invisible. By carefully defining visibility rules, the database can present a consistent snapshot of the database to the application. This works as follows:

1. 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.
2. Any writes made by aborted transactions are ignored.
3. 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 committed.
4. All other writes are visible to the application's queries.

(LOGIC)

Put another way, an object is visible if both of the following conditions are true:

- At the time when the reader's transaction started, the transaction that created the object had already committed.
- The object is not marked for deletion, or if it is, the transaction that requested deletion had not yet committed at the time when the reader's transaction started.

INDEXES AND SNAPSHOT ISOLATION:

- the index simply point to all versions of an object and require an index query to filter out any object versions that are not visible to the current transaction.
- *append-only/copy-on-write* variant of Btrees.=> every write transaction (or batch of transactions) creates a new B-tree root,

Preventing Lost Updates:

- The read committed (read skews issue) and snapshot isolation levels (solves read skew) we've discussed so far have been primarily about the guarantees of what a read-only transaction can see in the presence of concurrent writes.
- We have mostly ignored the issue of two transactions writing concurrently—we have only discussed dirty writes (see "No dirty writes"), one particular type of write-write conflict that can occur.
- lost update problem: *read-modify-write cycle* -
    - ++ *counters, bank accts*
    - 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

Lost Update Solutions:

- ATOMIC WRITE OPERATIONS: (db itself provides):

- 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.

- EXPLICIT LOCKING:

- the application to explicitly lock objects that are going to be updated
-

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

```
BEGIN TRANSACTION;

SELECT * FROM figures
  WHERE name = 'robot' AND game_id = 222
  FOR UPDATE; ❶

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

COMMIT;
```

- AUTOMATICALLY DETECTING LOST UPDATES: (MOST COMMONLY USED)

- 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

- ```
  -- This may or may not be safe, depending on th
  UPDATE wiki_pages SET content = 'new content'
    WHERE id = 1234 AND content = 'old content';
  ```
  ```

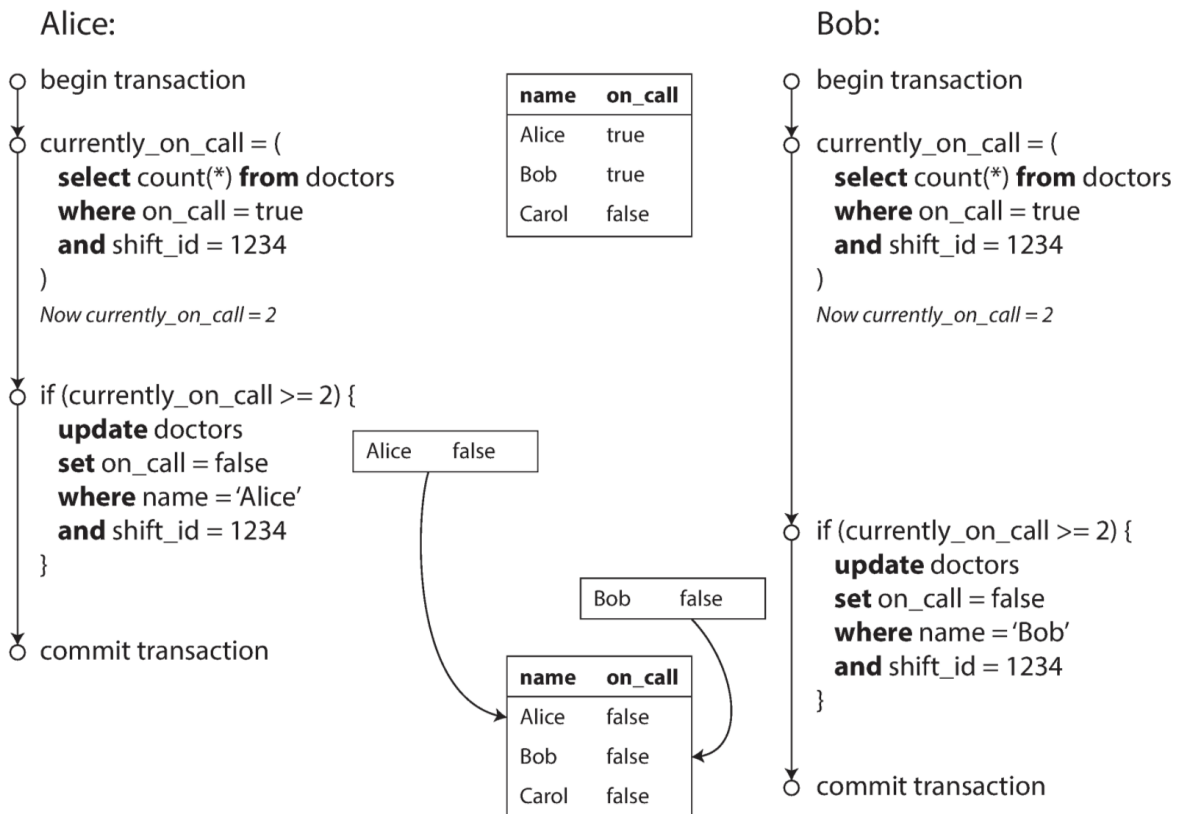
CONFLICT RESOLUTION AND REPLICATION:

- as discussed in "Detecting Concurrent Writes", 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.
- LWW - prone to lost updates

Write Skew and Phantoms:

Seen so far:

- Dirty Writes
- Lost Updates



- Write skew can occur if two transactions read the same objects, and then update some of those objects (different transactions may update different objects).
- In the special case where different transactions update the same object, you get a dirty write or lost update anomaly (depending on the timing).

- SOLN: explicitly lock the rows that the transaction depends on.

```
BEGIN TRANSACTION;

SELECT * FROM doctors
  WHERE on_call = true
  AND shift_id = 1234 FOR UPDATE; ❶

UPDATE doctors
  SET on_call = false
  WHERE name = 'Alice'
  AND shift_id = 1234;

COMMIT;
```

MORE EXAMPLES OF WRITE SKEW

- *Meeting room booking system:*

```
BEGIN TRANSACTION;

-- Check for any existing bookings that overlap with the period of noon-1pm
SELECT COUNT(*) FROM bookings
  WHERE room_id = 123 AND
    end_time > '2015-01-01 12:00' AND start_time < '2015-01-01 13:00';

-- If the previous query returned zero:
INSERT INTO bookings
  (room_id, start_time, end_time, user_id)
  VALUES (123, '2015-01-01 12:00', '2015-01-01 13:00', 666);

COMMIT;
```

- Unfortunately, snapshot isolation does not prevent another user from concurrently inserting a conflicting meeting

PHANTOMS CAUSING WRITE SKEW

All of these examples follow a similar pattern:

1. A `SELECT` query checks whether some requirement is satisfied by searching for rows that match some search condition (there are at least two doctors on call, there are no existing bookings for that room at that time, the position on the board doesn't already have another figure on it, the username isn't already taken, there is still money in the account).
2. Depending on the result of the first query, the application code decides how to continue (perhaps to go ahead with the operation, or perhaps to report an error to the user and abort).
3. If the application decides to go ahead, it makes a write (`INSERT`, `UPDATE`, or `DELETE`) to the database and commits the transaction.

PHANTOM:

- The effect of this write changes the precondition of the decision of step 2. In other words, 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 (there is now one fewer doctor on call, the meeting room is now booked for that time, the position on the board is now taken by the figure that was moved, the username is now taken, there is now less money in the account).

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

SOLN: Lock all objects that result in the select query.

ISSUE:

MATERIALIZING CONFLICTS

If the problem of phantoms is that there is no object to which we can attach the locks, perhaps we can artificially introduce a lock object into the database?

For example, in the meeting room booking case you could imagine creating a table of time slots and rooms. Each row in this table corresponds to a particular room for a particular time period (say, 15 minutes). You create rows for all possible combinations of rooms and time periods ahead of time, e.g. for the next six months

- the additional table isn't used to store information about the booking—it's purely a collection of locks

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.

3 Techniques:

- Literally executing transactions in a serial order (see "Actual Serial Execution")
- Two-phase locking (see "Two-Phase Locking (2PL)"), which for several decades was the only viable option
- Optimistic concurrency control techniques such as serializable snapshot isolation (see "Serializable Snapshot Isolation (SSI)")

Actual Serial Execution:

- around 2007—decided that a single-threaded loop for executing transactions was feasible
- RAM became cheap enough that for many use cases it is now feasible to keep the entire active dataset in memory (see "Keeping everything in memory"). When all data that a transaction needs to access is in memory, transactions can execute much faster than if they have to wait for data to be loaded from disk.
- Database designers realized that OLTP transactions are usually short and only make a small number of reads and writes (see "Transaction Processing or Analytics?").

- single-partition transactions. => can execute parallel.

Two-Phase Locking (2PL):

- 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, like in Figure 7-4, is not acceptable under 2PL.)

- In 2PL, writers don't just block other writers; they also block readers and vice versa. Snapshot isolation has the mantra *readers never block writers, and writers never block readers* (see "Implementing snapshot isolation"), which captures this key difference between snapshot isolation and two-phase locking.

- because 2PL provides serializability, it protects against all the race conditions discussed earlier, including lost updates and write skew.

Implementation:

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 any 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:

- **The performance for transaction throughput and response time of queries are significantly worse under two-phase locking than under weak isolation.**
- A transaction may have to wait for several others to complete before it can do anything.
- Databases running 2PL can have unstable latencies, and they can be very slow at high percentiles. One slow transaction, or one

transaction that accesses a lot of data and acquires many locks can cause the rest of the system to halt.

PREDICATE LOCKS:

- In "Phantoms causing write skew" we discussed the problem of *phantoms*—that is, one transaction changing the results of another transaction's search query. A database with serializable isolation must prevent phantoms.
- but rather than belonging to a particular object (e.g., one row in a table), it belongs to all objects that match some search condition

A predicate lock restricts access as follows:

- If transaction A wants to read objects matching some condition, like in that `SELECT` query, it must acquire a shared-mode predicate lock on the conditions of the query. If another transaction B currently has an exclusive lock on any object matching those conditions, A must wait until B releases its lock before it is allowed to make its query.
- If transaction A wants to insert, update, or delete any object, it must first check whether either the old or the new value matches any existing predicate lock. If there is a matching predicate lock held by transaction B, then A must wait until B has committed or aborted before it can continue.
- The key idea here is that a predicate lock applies even to objects that do not yet exist in the database, but which might be added in the future (phantoms). If two-phase locking includes predicate locks, the database prevents all forms of write skew and other race conditions, and so its isolation becomes serializable.

INDEX-RANGE LOCKS:

- predicate locks do not perform well: (lock on every object) if there are many locks by active transactions, checking for matching locks becomes time-consuming. For that reason, most databases with 2PL actually implement *index-range locking*
- It's safe to simplify a predicate by making it match a greater set of objects. For example, if you have a predicate lock for bookings of room 123 between noon and 1 p.m., you can approximate it by locking bookings for room 123 at any time, or you can approximate it by locking all rooms (not just room 123) between noon and 1 p.m. This is safe, because any write that matches the original predicate will definitely also match the approximations.
- In the room bookings database you would probably have an index on the `room_id` column, and/or indexes on `start_time` and `end_time` (otherwise the preceding query would be very slow on a large database):
- Say your index is on `room_id`, and the database uses this index to find existing bookings for room 123. Now the database can simply attach a shared lock to this index entry, indicating that a transaction has searched for bookings of room 123.
- Alternatively, if the database uses a time-based index to find existing bookings, it can attach a shared lock to a range of values in that index, indicating that a transaction has searched for bookings that overlap with the time period of noon to 1 p.m. on January 1, 2018.
- Either way, an approximation of the search condition is attached to one of the indexes. Now, if another transaction wants to insert, update, or delete a booking for the same room and/or an overlapping time period, it will have to update the same part of the index. In the process of doing so, it will encounter the shared lock, and it will be forced to wait until the lock is released.
- This provides effective protection against phantoms and write skew. Index-range locks are not as precise as predicate locks would be (they may lock a bigger range of objects than is strictly

necessary to maintain serializability), but since they have much lower overheads, they are a good compromise.

Serializable Snapshot Isolation (SSI)

PESSIMISTIC VERSUS OPTIMISTIC CONCURRENCY CONTROL:

- *mutual exclusion vs*
- instead of blocking if something potentially dangerous happens, transactions continue anyway, in the hope that everything will turn out all right.
- When a transaction wants to commit, the database checks whether anything bad happened (i.e., whether isolation was violated);
- if contention between transactions is not too high, optimistic concurrency control techniques tend to perform better than pessimistic ones
- On top of snapshot isolation, SSI adds an algorithm for detecting serialization conflicts among writes and determining which transactions to abort.

DECISIONS BASED ON AN OUTDATED PREMISE

When we previously discussed write skew in snapshot isolation (see "Write Skew and Phantoms"), we observed a recurring pattern: a

transaction reads some data from the database, examines the result of the query, and decides to take some action (write to the database) based on the result that it saw. However, 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.

- Detecting reads of a stale MVCC object version (uncommitted write occurred before the read)
- Detecting writes that affect prior reads (the write occurs after the read)

DETECTING STALE MVCC READS:

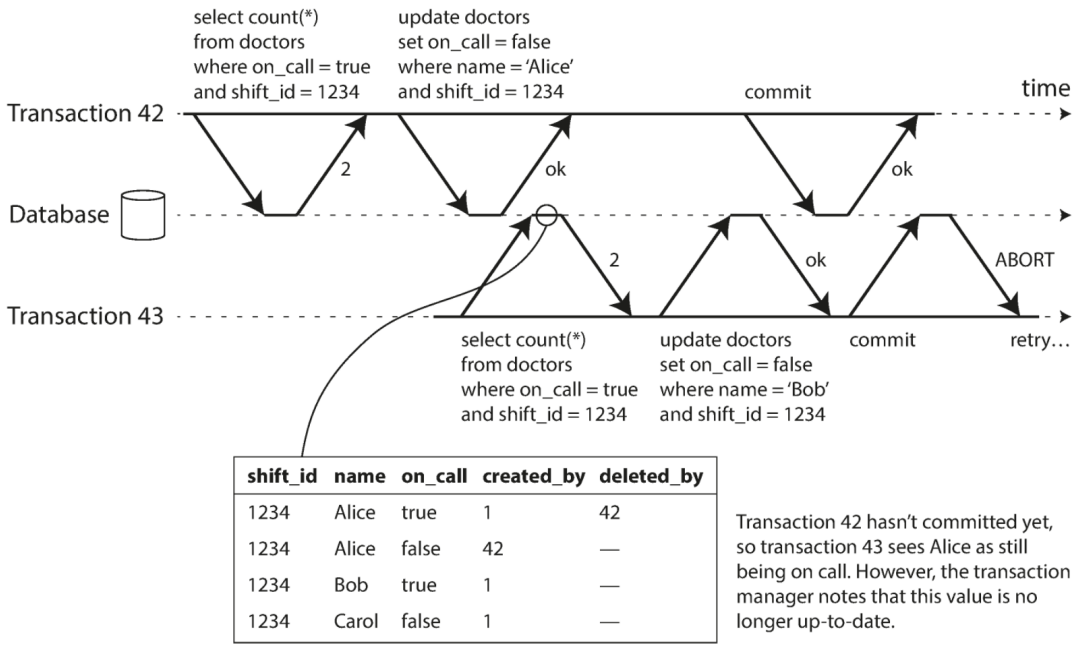


Figure 7-10. Detecting when a transaction reads outdated values from an MVCC snapshot.

- by the time transaction 43 wants to commit, transaction 42 has already committed. This means that the write that was ignored when reading from the consistent snapshot has now taken effect, and transaction 43's premise is no longer true.
- the database needs to track when a transaction ignores another transaction's writes due to MVCC visibility rules. When the transaction wants to commit, the database checks whether any of the ignored writes have now been committed. If so, the transaction must be aborted.

DETECTING WRITES THAT AFFECT PRIOR READS:

when another transaction modifies data after it has been read.

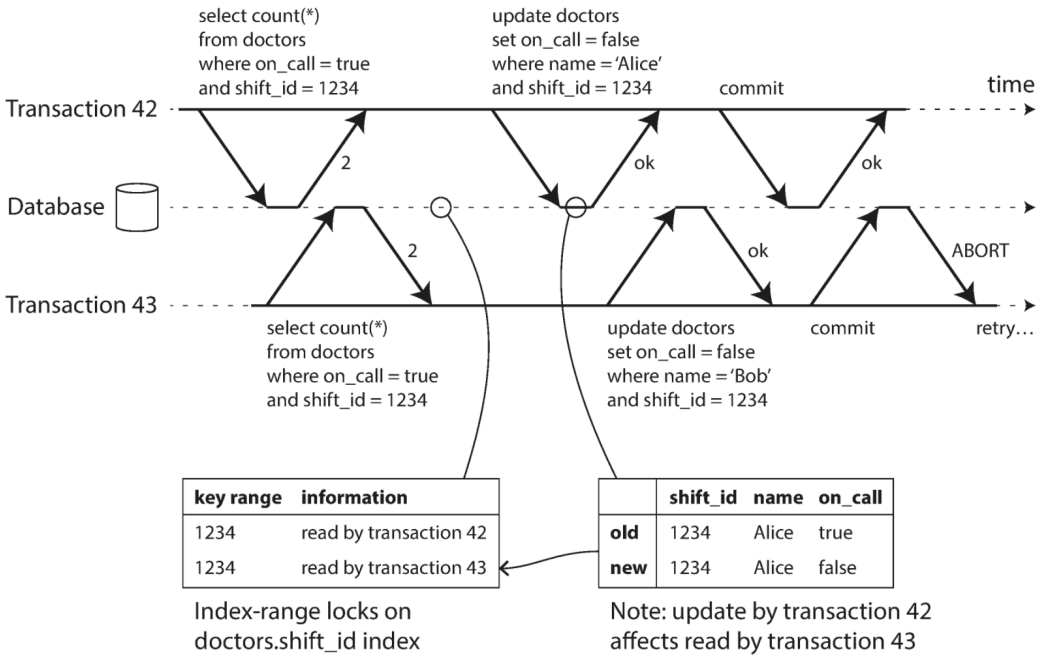


Figure 7-11. In serializable snapshot isolation, detecting when one transaction modifies another transaction's reads.

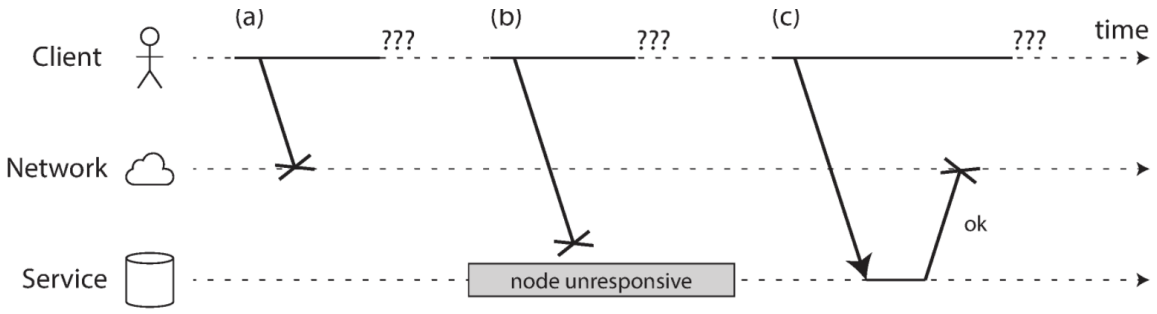
- the database can use the index entry 1234 to record the fact that transactions 42 and 43 read this data.
- When a transaction writes to the database, it must look in the indexes for any other transactions that have recently read the affected data. This process is similar to acquiring a write lock on the affected key range, but rather than blocking until the readers have committed, the lock acts as a tripwire: it simply notifies the transactions that the data they read may no longer be up to date.

Chapter 8. The Trouble with Distributed Systems

- Even in smaller systems consisting of only a few nodes, it's important to think about partial failure. In a small system, it's quite likely that most of the components are working correctly most of the time. However, sooner or later, some part of the system *will* become faulty
- error-correcting codes can deal with a small number of single-bit errors, but if your signal is swamped by interference, there is a fundamental limit to how much data you can get through your communication channel [13].
- TCP can hide packet loss, duplication, and reordering from you, but it cannot magically remove delays in the network.

Unreliable Networks:

- *shared-nothing systems*: i.e., a bunch of machines connected by a network



-

Network Faults in Practice:

- medium-sized datacenter found about 12 network faults per month, of which half disconnected a single machine, and half disconnected an entire rack
- **NETWORK PARTITIONS**
- When one part of the network is cut off from the rest due to a network fault, that is sometimes called a *network partition* or *netsplit*. In this book we'll generally stick with the more general term *network fault*

Detecting Faults:

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

Retries:
Application level/TCP

TCP: (SYN, SYN/ACK, ACK) ——> (FIN, FIN/ACK, ACK)

- TCP includes mechanisms to solve many of the problems that arise from packet-based messaging,
    - lost packets - Timeout/RetransmissionQueue/Retries
    - out of order packets - Seq No, Ack No/ ask for missing/ reassemble
    - duplicate packets, and
    - corrupted packets - Error control, CR
    - Flow control

TIMEOUTS:

Issue with detecting early/late ?

- Unnecessary failovers
- Long wait times to be stable again

NETWORK CONGESTION AND QUEUEING:

- the variability of packet delays on computer networks is most often due to queueing
    - If several different nodes simultaneously try to send packets to the same destination, the network switch must queue them up and feed them into the destination network link one by one
    - When a packet reaches the destination machine, if all CPU cores are currently busy, the incoming request from the network is queued by the operating system until the application is ready to handle it.
    - TCP performs *flow control* (also known as *congestion avoidance* or *backpressure*), in which a node limits its own rate of sending in order to avoid overloading a network link or the receiving node

TCP vs UDP:

- Some latency-sensitive applications, such as videoconferencing and Voice over IP (VoIP), use UDP rather than TCP. It's a trade-off between reliability and variability of delays:
    - as UDP does not perform flow control and does not retransmit lost packets, it avoids some of the reasons for variable network delays (although it is still susceptible to switch queues and scheduling delays).
- UDP is a good choice in situations where delayed data is worthless. Like VoIP phone call

- You can only choose timeouts experimentally.
- Systems can continually measure response times and their variability (jitter), and automatically adjust: Phi Accrual failure detector, which is used for example in Akka and Cassandra. TCP retransmission timeouts also work similarly.

Synchronous vs. Asynchronous Networks:

- Phone call network is **synchronous**: even as data passes through several routers, it does not suffer from queueing, because the 16 bits of space for the call have already been **reserved** in the next hop of the network.
- **bounded delay**: No queueing, the maximum end-to-end latency of the network is fixed.

- Can we not simply make network delays predictable?
    - Ethernet and IP are packet-switched protocols, which suffer from queueing and thus unbounded delays in the network. These protocols do not have the concept of a circuit.
    - Why do datacenter networks and the internet use packet switching?

        - The answer is that they are **optimized for bursty traffic**.
- With careful use of
    - quality of service (QoS, prioritization and scheduling of packets) and
    - admission control (rate-limiting senders),
    - it is possible to emulate circuit switching on packet networks, or provide statistically bounded delay.

- **Latency and Resource Utilization**:
        - Phone line: resource is divided up in a static way
        - Internet: shares network bandwidth dynamically
        - This approach has the downside of queueing, but the advantage is that it **maximizes utilization of the wire**.
        - A similar situation arises with CPUs.
    - **Latency guarantees** are achievable in certain environments, if resources are statically partitioned (e.g., dedicated hardware and exclusive bandwidth allocations).
        - Comes at the cost of reduced utilization
    - **Variable delays** in networks are not a law of nature, but simply the result of a **cost/benefit trade-off**.

Unreliable Clocks

- **Clocks and Time** are important. Applications depend on clocks in various ways.
- In a distributed system, time is a tricky business, because communication is not instantaneous.

- Each machine on the network has its own clock, which is an actual hardware device: usually a **quartz crystal oscillator**. But these devices are **not perfectly accurate**, so each machine has its own notion of time.
    - It is possible to synchronize clocks to some degree: the most commonly used mechanism is the **Network Time Protocol (NTP)**
        - which allows the computer clock to be adjusted according to the time reported by a group of servers . The servers in turn get their time from a more accurate time source, such as a GPS receiver.
    - This fact sometimes makes it difficult to determine the order in which things happened when multiple machines are involved.

- **Monotonic vs. 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**:
        - it returns the current date and time according to some calendar (also known as wall-clock time).
        - Time-of-day clocks are usually synchronized with NTP, which means that a timestamp from one machine (ideally) means the same as a timestamp on another machine.
    - **Monotonic clocks**:
        - is suitable for measuring a duration (time interval), such as a timeout or a service's response time.
        - they are **guaranteed to always move forward** (whereas a time-of- day clock may jump back in time).
        - NTP may adjust the frequency at which the monotonic clock moves forward (this is known as slewing the clock) if it detects that the computer's local quartz is moving faster or slower than the NTP server.

- **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 source in order to be useful.
    - Hardware clocks and NTP can be fickle beasts.
    - It is possible to achieve very good clock accuracy if you care about it sufficiently to invest significant resources.
        - E.g. using GPS receivers, the Precision Time Protocol (PTP), and careful deployment and monitoring
    - 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.
    - Any applications observing the time before and after this reset may see time go backward or suddenly jump forward.
    - 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
    - NTP clients are quite robust, because they query several servers and ignore outliers.

Relying on Synchronized Clocks:

- Robust software needs to be prepared to deal with incorrect clocks.
    - incorrect clocks easily go unnoticed.
- If some piece of software is relying on an accurately synchronized clock, the result is more likely to be silent and subtle data loss than a dramatic crash.
- Thus, if you use software that requires synchronized clocks, it is essential that you also carefully monitor the clock offsets between all the machines.

Timestamps for ordering events:

- Consider one particular situation in which it is tempting, but dangerous, to rely on clocks: ordering of events across multiple nodes. E.g. (LWW)

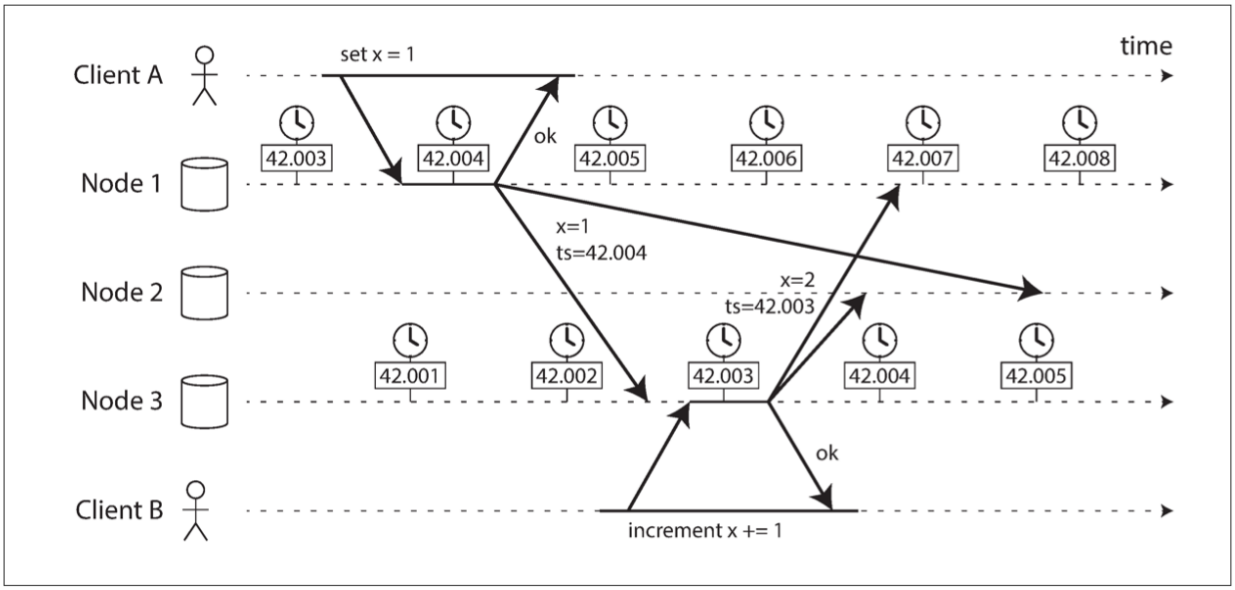


Figure 8-3. The write by client B is causally later than the write by client A, but B's write has an earlier timestamp.

- This conflict resolution strategy is called last write wins (LWW), and it is widely used in both multi-leader replication and leaderless databases such as Cassandra and Riak. (C: write to cache first?)
    - DB writes can mysteriously disappear;
- **Logical clocks**, which are based on incrementing counters rather than an oscillating quartz crystal, are a safer alternative for ordering events:
    - Logical clocks only measure the relative ordering of events (whether one event happened before or after another).

Clock readings have a confidence interval:

- Thus, it doesn't make sense to think of a clock reading as a point in time—it is more like a **range of times**, within a confidence interval.
- The uncertainty bound can be calculated based on your time source.
    - If you have a GPS receiver or atomic (caesium) clock directly attached to your computer, the expected error range is reported by the manufacturer.
    - If you're getting the time from a server, the uncertainty is based on the expected quartz drift since your last sync with the server, plus the NTP server's uncertainty, plus the network round-trip time to the server (to a first approximation, and assuming you trust the server).
- Google's TrueTime API in Spanner, which explicitly reports the confidence interval on the local clock.
    - When you ask it for the current time, you get back two values: [earliest, latest]

SYNCHRONIZED CLOCKS FOR GLOBAL SNAPSHOTS:

- With lots of small, rapid transactions, creating transaction IDs in a distributed system becomes an untenable bottleneck.
    - Can we use the timestamps from synchronized time-of-day clocks as transaction IDs?
        - If we could get the synchronization good enough, they would have the right properties: later transactions have a higher timestamp.
    - Spanner needs to keep the clock uncertainty as small as possible;
        - For this purpose, Google deploys a **GPS receiver or atomic clock** in each datacenter, allowing clocks to be synchronized to within about **7 ms**.

[Lets read SPANNER a bit, for tomorrow's session]

Process Pauses:

- Is it crazy to assume that a thread might be paused for so long? Unfortunately not. E.g.
  - Many programming language runtimes (such as the Java Virtual Machine) have a garbage collector (GC) that occasionally needs to stop all running threads.
  - In virtualized environments, a virtual machine can be suspended (pausing the execution of all processes and saving the contents of memory to disk) and resumed (restoring the contents of memory and continuing execution).
  - If the application performs synchronous disk access, a thread may be paused waiting for a slow disk I/O operation to complete
- All of these occurrences can preempt the running thread at any point and resume it at some later time, without the thread even noticing.

- **Response time guarantees**:
  - In some systems, there is a specified deadline by which the software must respond; if it doesn't meet the deadline, that may cause a failure of the entire system. These are so-called **Hard real-time systems**.
  - A **real-time operating system (RTOS)** that allows processes to be scheduled with a guaranteed allocation of CPU time in specified intervals is needed;
  - For most server-side data processing systems, real-time guarantees are simply not economical or appropriate.

- **Limiting the impact of garbage collection**:
  - An emerging idea is to **treat GC pauses like brief planned outages** of a node, and to let other nodes handle requests from clients while one node is collecting its garbage.

Chapter 9. Consistency and Consensus

We will assume that all the problems from Chapter 8 can occur:

- packets can be lost, reordered, duplicated,
- arbitrarily delayed in the network;
- clocks are approximate at best;
- and nodes can pause (e.g., due to garbage collection) or crash at any time.

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

This is the same approach as we used with transactions in Chapter 7: by using a transaction, the application can pretend:

- that there are no crashes (**atomicity**),
- that nobody else is concurrently accessing the database (**isolation**),
- that storage devices are perfectly reliable (**durability**).

One of the most important abstractions for distributed systems is **consensus**: that is, getting all of the nodes to agree on something.

- Once you have an implementation of consensus, applications can use it for various purposes.
- Correct implementations of consensus help avoid problems.
    - For example, say you have a database with single-leader replication. If the leader dies and you need to fail over to another node, the remaining database nodes can use consensus to elect a new leader.
        - it's important that there is only one leader, and that all nodes agree who the leader is. If two nodes both believe that they are the leader, that situation is called *split brain*, and it often leads to data loss. Correct implementations of consensus help avoid such problems.

Consistency Guarantees:

Problems with Replication Lag:

- Reading Your Own Writes

- Monotonic Reads
- Causal inconsistency

eventual consistency:

- which means that if you stop writing to the database and wait for some unspecified length of time, then eventually all read requests will return the same value

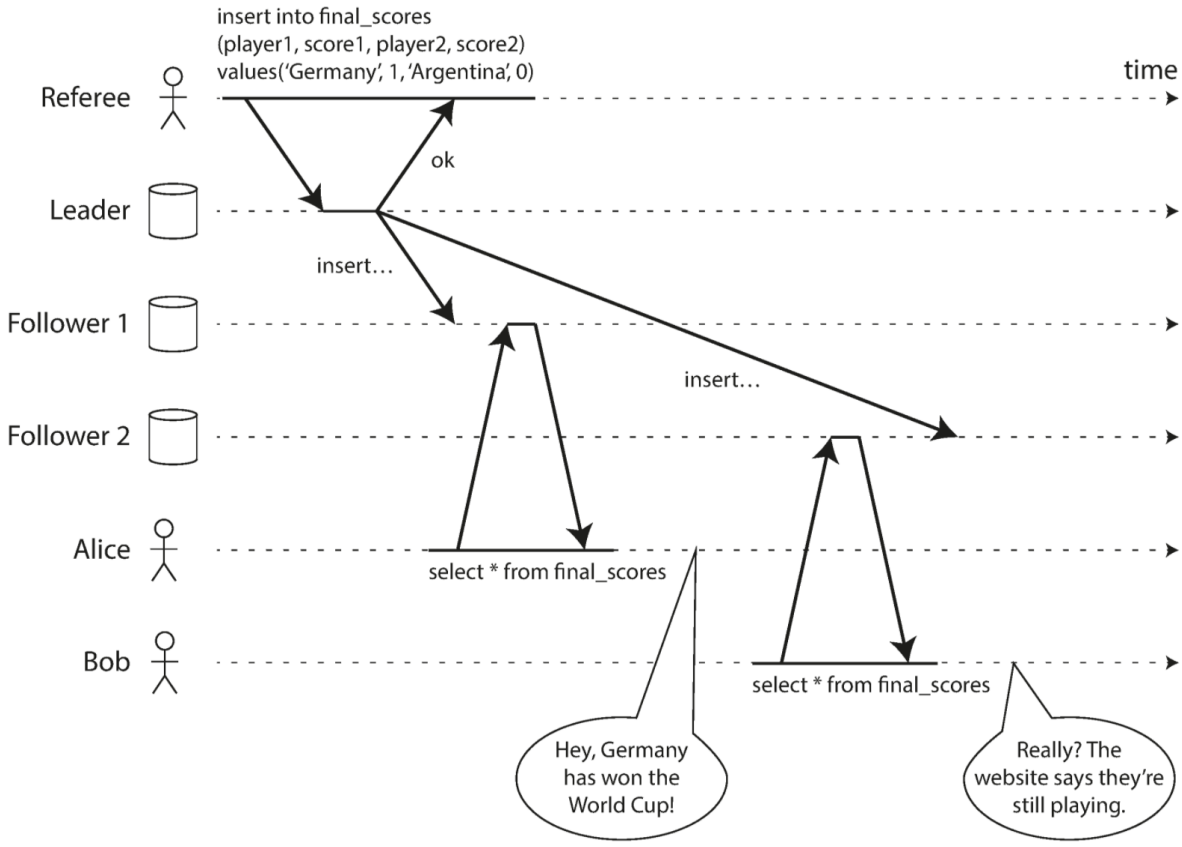
3 -> 4
W ----------
R — — — — —
3 3. 4. 4. 4 ……..

F. F F F TTTTTT.TT

Linearizability:

- Idea behind **Linearizability** (also known as **atomic consistency**, **strong consistency**, **immediate consistency**, or **external consistency**)
  - **The basic idea is to**
    - **make a system appear as if there were only one copy of the data, and**
    - **all operations on it are atomic.**
    - **recency guarantee**
  - **(even though there may be multiple replicas in reality, the application does not need to worry about them.)**

- **In a linearizable system**, as soon as one client successfully completes a write, all clients reading from the database must be able to see the value just written.
    - In other words, linearizability is a **recency guarantee**.



Issue above ?

Replicas with old and new values serving reads.

- **What Makes a System Linearizable**?
    - To make a system appear as if there is only a single copy of the data.

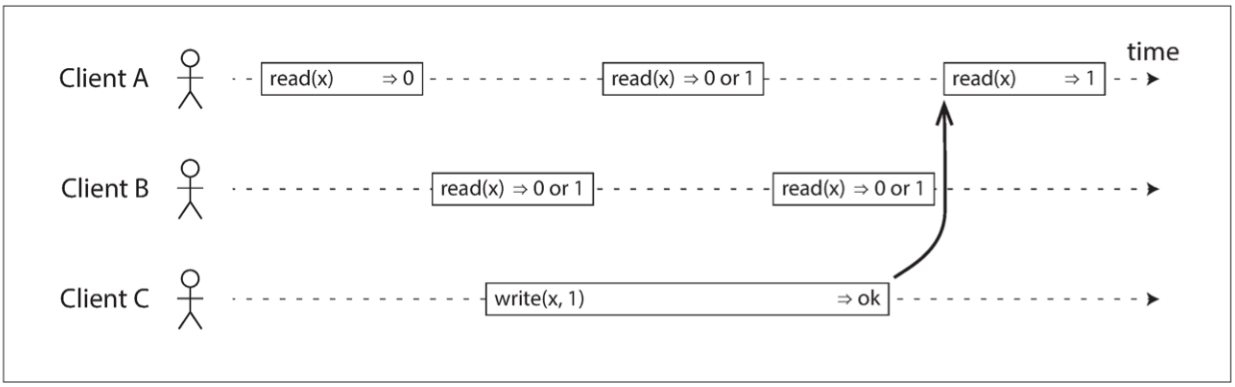


Figure 9-2. If a read request is concurrent with a write request, it may return either the old or the new value.

“If a Read request is concurrent with a write request, it may return either the old or new value”

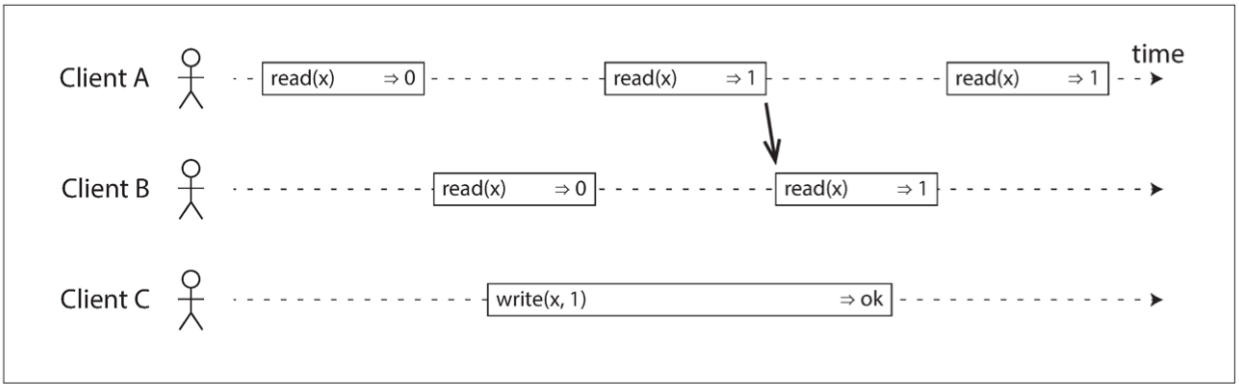


Figure 9-3. After any one read has returned the new value, all following reads (on the same or other clients) must also return the new value.

“After any one read has returned the new value, all following reads(by any client), must also return the new value"

- **atomic compare-and-set** (**cas**);

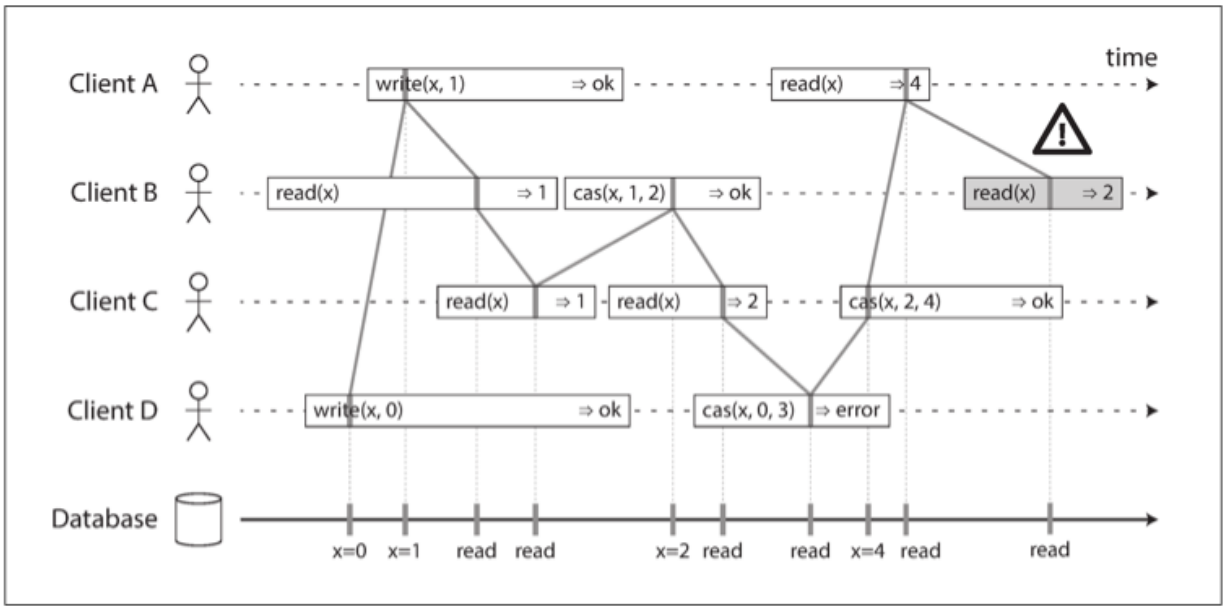


Figure 9-4. Visualizing the points in time at which the reads and writes appear to have taken effect. The final read by B is not linearizable.

- The requirement of linearizability is that the lines joining up the operation markers always move forward in time (from left to right), never backward.
- The final read by client B (in a shaded bar) is not linearizable. The operation is concurrent with C's *cas* write, which updates *x* from 2 to 4. In the absence of other requests, it would be okay for B's read to return 2. However, client A has already read the new value 4 before B's read started, so B is not allowed to read an older value than A.

Linearizability vs. Serializability:

- **Serializability** is an **isolation property of transactions**, where every transaction may read and write multiple objects (rows, documents, records).
    - 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 reads and writes of a register (an individual object).
    - It doesn't group operations together into transactions, so it does not prevent problems such as write skew unless you take additional measures such as materializing conflicts
  - 3 ways of serialization:
    - 2PL & actual serial execution are typically linearizable.
    - **serializable snapshot isolation** is **NOT** linearizable.
        - 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.
- A database may provide both **serializability** and **linearizability**, and this combination is known as strict serializability or strong one-copy serializability.

(The moment we read from cache or snapshot we lose serializability)

Relying on Linearizability:

LOCKING AND LEADER ELECTION:

- One way of electing a leader is to use a lock: every node that starts up tries to acquire the lock, and the one that succeeds becomes the leader
- No matter how this lock is implemented, it must be linearizable: all nodes must agree which node owns the lock; otherwise it is useless.

- ZooKeeper uses consensus algorithms to implement linearizable operations in a fault-tolerant way

CONSTRAINTS AND UNIQUENESS GUARANTEES

- If you want to enforce this constraint as the data is written (such that if two people try to concurrently create a user or a file with the same name, one of them will be returned an error), you need **linearizability**. (C: kind like “lock”)

CROSS-CHANNEL TIMING DEPENDENCIES

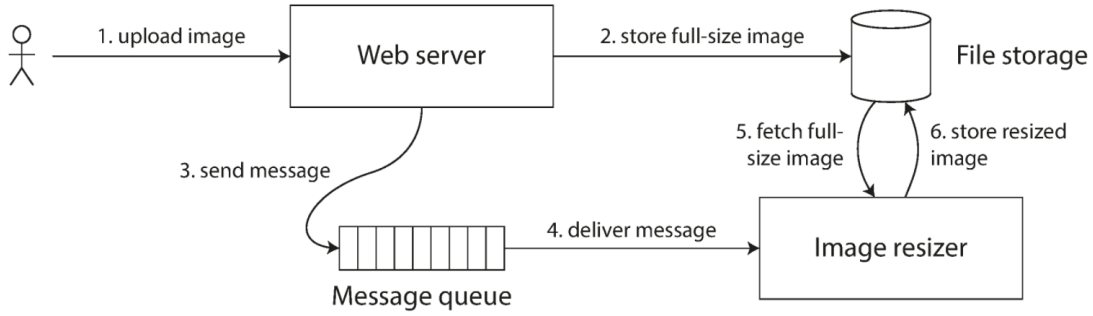


Figure 9-5. The web server and image resizer communicate both through file storage and a message queue, opening the potential for race conditions.

- If the file storage service is linearizable (1 replica vs n replicas), 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 in Figure 9-5) might be faster than the internal replication inside the storage service.

Implementing Linearizable Systems:

“linearizability essentially means “behave as though there is only a single copy of the data, and all operations on it are atomic,” “

So issues of linearizability occurs due to ===> Replication

Replication Methods: (Can they be linearizable ??)

- Single-leader replication (potentially linearizable)
  - In a system with single-leader replication, the leader has the primary copy of the data that is used for writes, and the followers maintain backup copies of the data on other nodes.
    - If you make reads from the leader,
    - or from synchronously updated followers,
  - they have the *potential* to be linearizable.[iv] However, not every single-leader database is actually linearizable, by design (e.g., because it uses snapshot isolation)

- *Consensus algorithms (linearizable)*
  - They bear a resemblance to single-leader replication.
  - One way of electing a leader is to use a lock: every node that starts up tries to acquire the lock, and the one that succeeds becomes the leader
    - No matter how this lock is implemented, it must be linearizable: all nodes must agree which node owns the lock; otherwise it is useless.
  - ZooKeeper uses consensus algorithms to implement linearizable operations in a fault-tolerant way
  - However, consensus protocols contain measures to prevent split brain and stale replicas. Thanks to these details, consensus

algorithms can implement linearizable storage safely. Etcd, ZooKeeper

- Multi-leader replication (not linearizable)
    - Systems with multi-leader replication are generally not linearizable, because they
        - concurrently process writes on multiple nodes
        - asynchronously replicate them to other nodes.
        - For this reason, they can produce conflicting writes that require resolution
- Leaderless replication (probably not linearizable)
    - For systems with leaderless replication people sometimes claim that you can obtain “strong consistency” by requiring quorum reads and writes ($w + r > n$). Depending on the exact configuration of the quorums, and depending on how you define strong consistency, this is not quite true.
    - “Last write wins” conflict resolution methods based on time-of-day clocks are almost certainly nonlinearizable,
        - because clock timestamps cannot be guaranteed to be consistent with actual event ordering due to clock skew
    - Sloppy quorums also ruin any chance of linearizability. Even with strict quorums, nonlinearizable behavior is possible, as demonstrated in the next section.

LINEARIZABILITY AND QUORUMS:

Intuitively, it seems as though strict quorum reads and writes should be linearizable in a Dynamo-style model. However, when we have variable network delays, it is possible to have race conditions

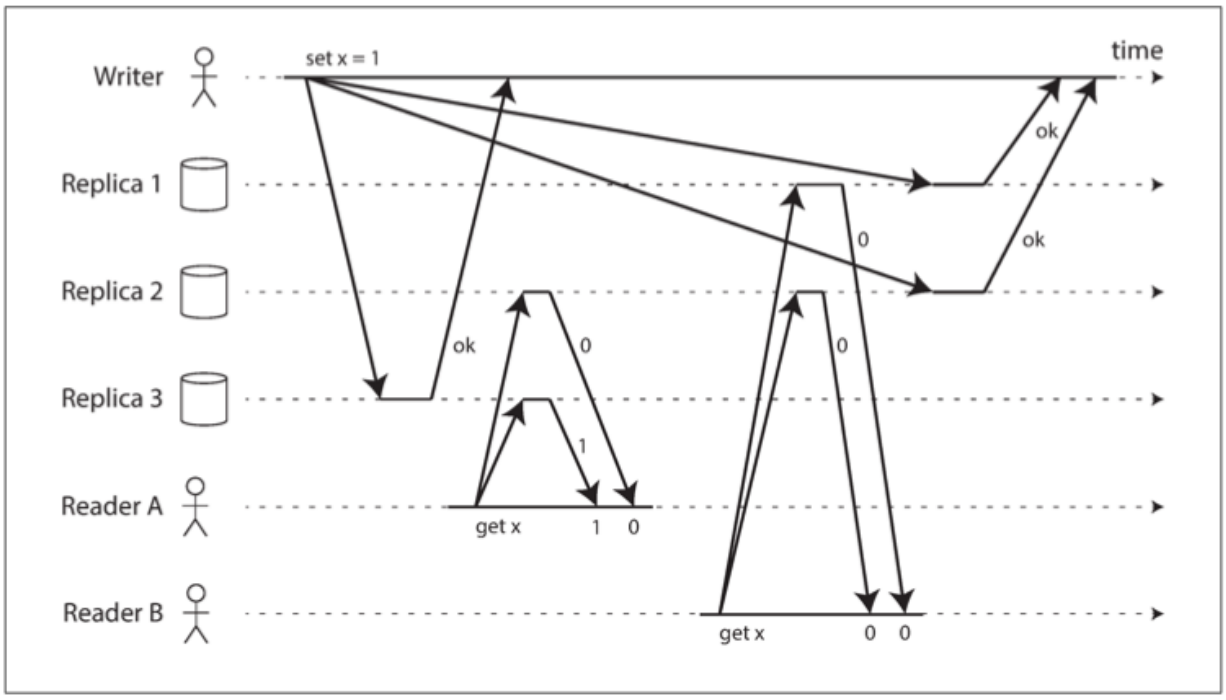


Figure 9-6. A nonlinearizable execution, despite using a strict quorum.

[Issue: In quorums, reads are happening as the writes are still progressing.]
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.

- it is safest to assume that a leaderless system with Dynamo-style replication does not provide linearizability.

Solution:

- 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 [If A had repaired, B would have read latest]
    - a writer must read the latest state of a quorum of nodes before sending its writes [Read before Write].

The Cost of Linearizability: (CAP Theorem)

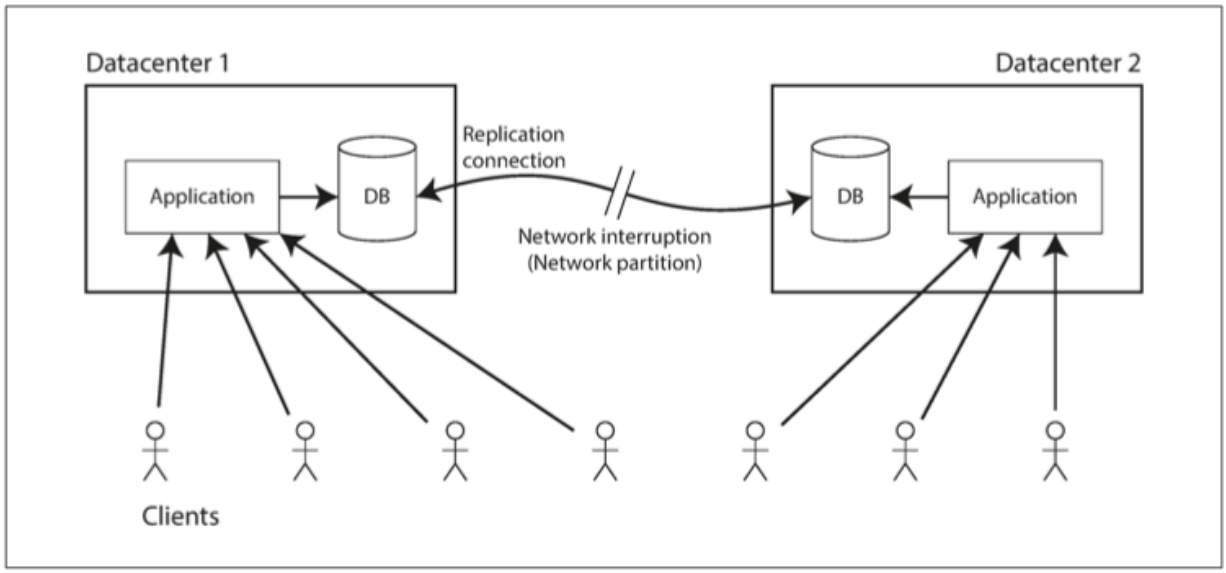


Figure 9-7. A network interruption forcing a choice between linearizability and availability.

[If you choose linearizability, you cannot be available all the time]

- 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.
- A better way of phrasing CAP would be either **Consistent** or **Available** when **Partitioned**. (aka, CP or AP)

Linearizability and network delays:

- Although linearizability is a useful guarantee, surprisingly few systems are actually linearizable in practice.
    - even RAM on a modern multi-core CPU is not linearizable: if a thread running on one CPU core writes to a memory address, and a thread on another CPU core reads the same address shortly afterward, it is not guaranteed to read the value written by the first thread (unless a *memory barrier* or *fence*
    - *Due to each cpu core maintaining its own memory cache/store buffer*
- Many distributed databases that choose not to provide linearizable guarantees: they do so primarily to increase performance, not so much for fault tolerance.
- Linearizability is **slow**—and this is true all the time, not only during a network fault.

Ordering Guarantees:

linearizable register behaves as if there is only a single copy of the data, and that every operation appears to take effect atomically at one point in time. This definition implies that operations are executed in some well-defined order.

Recap on ordering:

- main purpose of the leader in single-leader replication is to determine the *order of writes* in the replication log
- Serializability, is about ensuring that transactions behave as if they were executed in *some sequential order*.
- use of timestamps and clocks in distributed systems —> determine which one of two writes happened later.

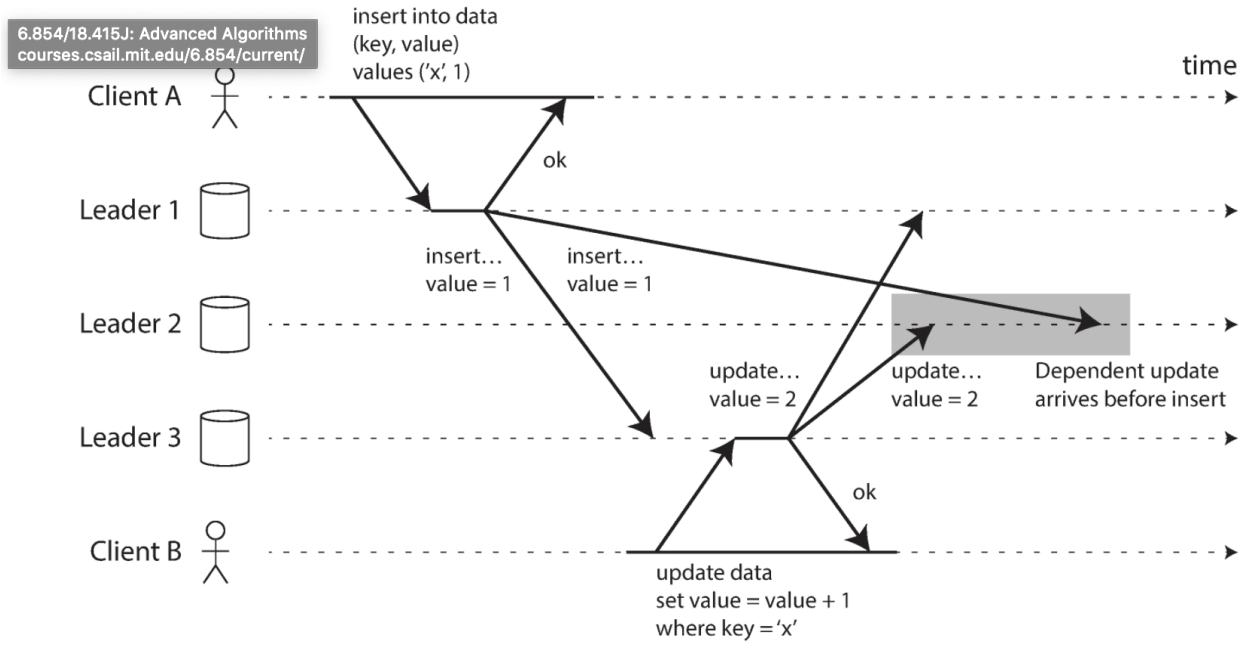
==> deep connections between ordering, linearizability, and consensus.

Ordering and Causality

There are several reasons why ordering keeps coming up, and one of the reasons is that it helps preserve *causality*.

Recap on causality:

- Consistent Prefix reads:
    - causal inconsistency,
- Serializability:
    - is about ensuring that transactions behave as if they were executed in *some sequential order*
- *Multi leader replication:*



- "Detecting Concurrent Writes":

    - we observed that if you have two operations A and B, there are three possibilities: either A happened before B, or B happened before A, or A and B are concurrent. This *happened before* relationship is another expression of causality: if A happened before B, that means B might have known about A, or built upon A, or depended on A. If A and B are concurrent, there is no causal link between them
- snapshot isolation for transactions:
    - a transaction reads from a consistent snapshot.
    - But what does "consistent" mean in this context? It means *consistent with causality*: if the snapshot contains an answer, it must also contain the question being answered
- write skew between transactions:
    - the action of going off call is causally dependent on the observation of who is currently on call.
    - SSI detects write skew by tracking the causal dependencies between transactions

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.
- mathematical sets are not totally ordered: is $\{a, b\}$ greater than $\{b, c\}$? Well, you can't really compare them, because neither is a subset of the other

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. This total ordering is illustrated as a timeline

Causality

- We said that two operations are concurrent if neither happened before the other
- 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.

Takeaways:

- there are no concurrent operations in a linearizable datastore:
- not a straight-line total order, but rather a jumble of different operations going on concurrently. The arrows in the diagram indicate causal dependencies—the partial ordering of operations.

LINEARIZABILITY IS STRONGER THAN CAUSAL CONSISTENCY

Recap on ordering:

- linearizability *implies* causality: any system that is linearizable will preserve causality correctly
- But linearizable can harm its performance and availability, especially if the system has significant network delays
- Linearizability is not the only way of preserving causality—there are other ways too.
    - A system can be causally consistent without incurring the performance hit of making it linearizable (in particular, the CAP theorem does not apply).

    - **In fact, 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 order to maintain causality, you need to know which operation happened before which other operation.
- In order to determine causal dependencies, we need some way of describing the "knowledge" of a node in the system.
- The techniques for determining which operation happened before which other operation are similar to what we discussed in "Detecting Concurrent Writes"
    - discussed causality in a leaderless datastore, where we need to detect concurrent writes to the same key in order to prevent lost updates.
    - Causal consistency goes further: it needs to track causal dependencies across the entire database, not just for a single key. Version vectors can be generalized to do this
- In order to determine the causal ordering,
    - the database needs to know which version of the data was read by the application. (e.g. Version Vectors, or conflict detection of SSI)
    - the version number from the prior operation is passed back to the database on a write

Sequence Number Ordering:

- Although causality is an important theoretical concept, actually keeping track of all causal dependencies can become impractical.
- Better way: we can use sequence numbers or timestamps to order events.
- A **timestamp** need not come from a time-of-day clock but "**Logical Clock**".
- Such sequence numbers or timestamps are compact (only a few bytes in size), and they provide a **total order.**

    - In a database with single-leader replication:
        - the replication log defines a total order of write operations that is consistent with causality.
        - The leader can simply increment a counter for each operation, and thus assign a monotonically increasing sequence number to each operation in the replication log.
        - If a follower applies the writes in the order they appear in the replication log, the state of the follower is always causally consistent (even if it is lagging behind the leader).
-
- **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. (e.g. one node odd, another node even; unique node identifier);
        - You can attach a timestamp from a time-of-day clock (physical clock) to each operation;
        - You can preallocate blocks of sequence numbers.
        - They all have a problem: **the sequence numbers they generate are not consistent with causality.**

LAMPORT TIMESTAMPS:

simple method for generating sequence numbers that *is* consistent with causality. It is called a *Lamport timestamp*

- 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.

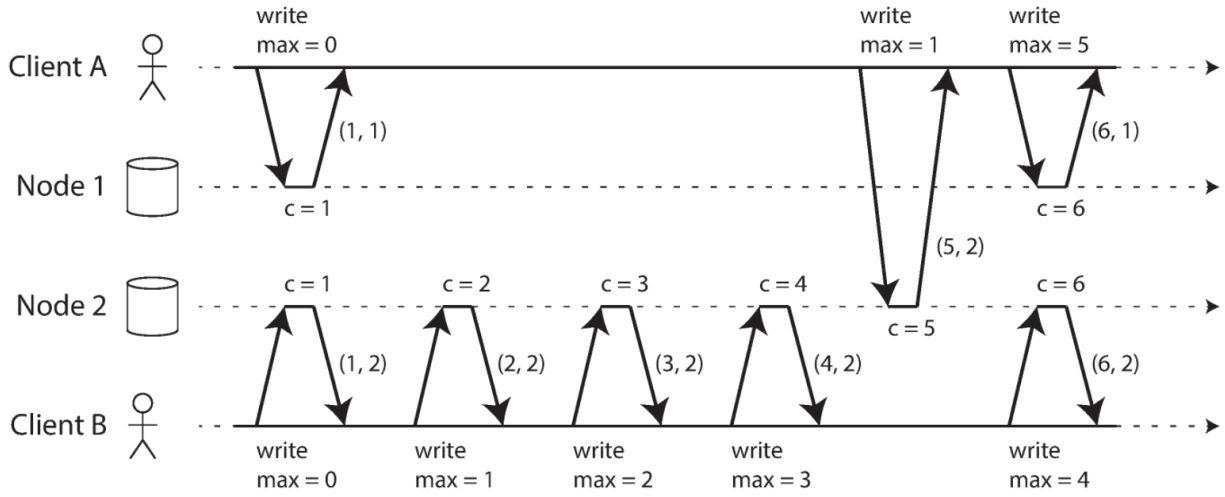


Figure 9-8. Lamport timestamps provide a total ordering consistent with causality.

if you have two timestamps,

- the one with a greater counter value is the greater timestamp;
- if the counter values are the same, the one with the greater node ID is the greater timestamp.

Key Idea:

- 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.
- **because every causal dependency results in an increased timestamp.**

Lamport vs version vectors:

- version vectors can distinguish whether two operations are concurrent or whether one is causally dependent on the other, whereas
- 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.
- (If 2 operations are causally dependent => the timestamp will tell this)
- BUT , based on timestamp alone => You can't figure this ,

Timestamp ordering is not sufficient:

- Comparing timestamps of usernames.
- However, this approach 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 generateIf one of the other nodes has failed or cannot be reached due to a network problem, this system would grind to a halt. This is not the kind of fault-tolerant system that we need.

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: the unknown operations from the other node may need to be inserted at various positions in the total order.

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. If you have an operation to create a username, and you are sure that no other node can insert a claim for the same username ahead

of your operation in the total order, then you can safely declare the operation successful.

- This idea of knowing when your total order is finalized is captured in the topic of **total order broadcast**.

Total Order Broadcast:

- If your program runs only on a single CPU core, it is easy to define a total ordering of operations: it is simply the order in which they were executed by the CPU.
- However, in a distributed system, getting all nodes to agree on the same total ordering of operations is tricky.
- ordering by timestamps or sequence numbers, but found that it is not as powerful as single-leader replication (if you use timestamp ordering to implement a uniqueness constraint, you cannot tolerate any faults, b**ecause you have to fetch the latest sequence number from every node**).
- single-leader replication determines a total order of operations by choosing one node as the leader and sequencing all operations on a single CPU core on the leader.
- The challenge then is how to scale the system if the throughput is greater than a single leader can handle, and also how to handle failover if the leader fails ====> *atomic broadcast*

Total order broadcast is usually described as a protocol for exchanging messages between nodes. Informally, it requires that two safety properties always be satisfied:

- *Reliable delivery*

No messages are lost: if a message is delivered to one node, it is delivered to all nodes.

- *Totally ordered delivery*

Messages are delivered to every node in the same order.

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.
- 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

Log-style (e.g. Consensus services such as ZooKeeper and etcd)

  - there is a strong connection between **total order broadcast and consensus**.
- Use cases:
  - Can be used for DB replication (aka. State machine replication) (replication log !!)
    - if every message represents a write to the database, and
    - every replica processes the same writes in the same order, then the replicas will remain consistent with each other
  - Can be used to implement serializable transactions.
    - if every message represents a deterministic transaction to be executed as a stored procedure,
    - if every node processes those messages in the same order, then the partitions and replicas of the database are kept consistent with each other

    - Total order broadcast is also useful for implementing a lock service that provides fencing tokens (monotonically increasing ids)
        - (see Every request to acquire the lock is appended as a message to the log,
        - and all messages are sequentially numbered in the order they appear in the log. The sequence number can then serve as a fencing token, because it is monotonically increasing. In ZooKeeper, this sequence number is called `zxid`

An important aspect of total order broadcast is that

- total order broadcast vs timestamp ordering:
    - 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.
- Log abstraction:
    - Another way of looking at total order broadcast is that it is a way of creating a *log* (as in a replication log, transaction log, or write-ahead log):
    - delivering a message is like appending to the log. Since all nodes must deliver the same messages in the same order, all nodes can read the log and see the same sequence of messages.

IMPLEMENTING LINEARIZABLE STORAGE USING TOTAL ORDER BROADCAST

- linearizable system there is a total order of operations.

- Does that mean linearizability is the same as total order broadcast? Not quite, but there are close links between the two.
    - 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.

However, if you have total order broadcast, you can build linearizable storage on top of it. For example, you can ensure that usernames uniquely identify user accounts.

What we want (Linearizable compare and set):

Imagine that for every possible

- username, you can have a linearizable register with an atomic compare-and-set operation.
- Every register initially has the value `null` (indicating that the username is not taken).
- When a user wants to create a username, you execute a compare-and-set operation on the register for that username, setting it to the user account ID, under the condition that the previous register value is `null`. If multiple users try to concurrently grab the same username, only one of the compare-and-set operations will succeed, because the others will see a value other than `null` (due to linearizability).

You can implement such a linearizable compare-and-set operation as follows by using total order broadcast as an append-only log

1. Append a message to the log, tentatively indicating the username you want to claim.

2. Read the log, and wait for the message you appended to be delivered back to you. xi
3. 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. A similar approach can be used to implement serializable multi-object transactions on top of a log

While this procedure ensures linearizable writes, it doesn't guarantee linearizable reads—if you read from a store that is asynchronously updated from the log, it may be stale.

To make reads linearizable, there are a few options:

- You can sequence reads through the log by appending a message, reading the log, and performing the actual read when the message is delivered back to you **(the delivery is ordered and linearized and you have waited out the replication lag and now you can read and return)**. The message's position in the log thus defines the point in time at which the read happens. (Quorum reads in etcd work somewhat like this [16].)
- If the log allows you to fetch the position of the latest log message in a linearizable way, you can query that position, wait for all entries up to that position to be delivered to you, and then perform the read. (This is the idea behind ZooKeeper's `sync()` operation [15].)
- You can make your read from a replica that is synchronously updated on writes, and is thus sure to be up to date. (This technique is used in chain replication [63]; see also "Research on Replication".)

IMPLEMENTING TOTAL ORDER BROADCAST USING LINEARIZABLE STORAGE

The last section showed how to build a linearizable compare-and-set operation from total order broadcast. We can also turn it around, assume that we have linearizable storage, and show how to build total order broadcast from it.

Assume (we have):

The easiest way is to assume you have a linearizable register that stores an integer and that has an atomic increment-and-get operation [28]. Alternatively, an atomic compare-and-set operation would also do the job.

The algorithm is simple: (Use linearizable integer and wait for the next seq. number)

- 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.

How hard could it be to make a linearizable integer with an atomic increment-and-get operation? As usual, if things never failed, it would be easy: you could just keep it in a variable on one node. The problem lies in handling the

situation when network connections to that node are interrupted, and restoring the value when that node fails [59]. In general, if you think hard enough about linearizable sequence number generators, you inevitably end up with a consensus algorithm.

This is no coincidence: it can be proved that a linearizable compare-and-set (or increment-and-get) register and total order broadcast are both *equivalent to consensus* [28, 67]. That is, if you can solve one of these problems, you can transform it into a solution for the others. This is quite a profound and surprising insight!

Distributed Transactions and Consensus

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

Leader election

- In a database with single-leader replication, all nodes need to agree on which node is the leader.
- The leadership position might become contested if some nodes can't communicate with others due to a network fault. In this case, consensus is important to avoid a bad failover, resulting in a split brain situation in which two nodes both believe themselves to be the leader
- If there were two leaders, they would both accept writes and their data would diverge, leading to inconsistency and data loss.

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/roll back (if anything goes wrong) or they all commit (if nothing goes wrong). This instance of consensus is known as the *atomic commit* problem
    - Updating primary and distributed secondary index

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

SINGLE-NODE :

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

MULTI NODE (2-PC):

- If some nodes commit the transaction but others abort it, the nodes become inconsistent with each other
- And once a transaction has been committed on one node, it cannot be retracted again if it later turns out that it was aborted on another node.
- The reason for this rule is that once data has been committed, it becomes visible to other transactions, and thus other clients may start relying on that data; this principle forms the basis of *read committed* isolation

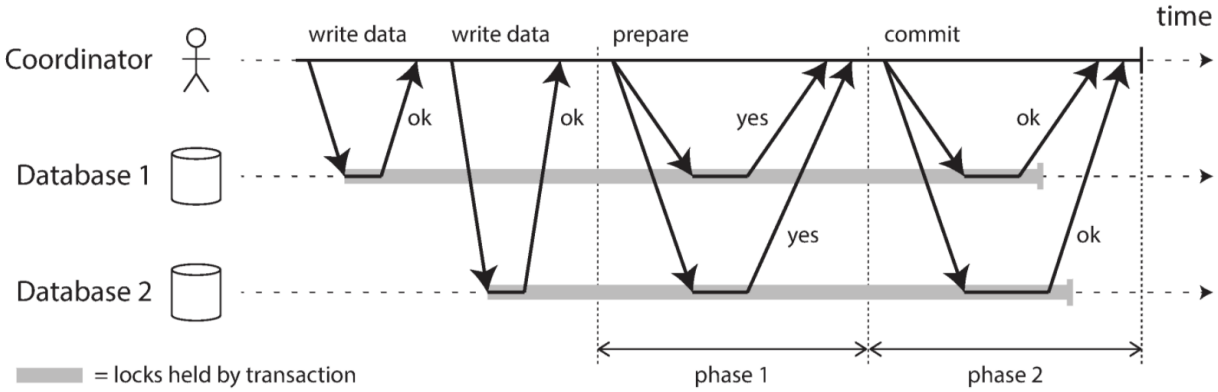


Figure 9-9. A successful execution of two-phase commit (2PC).

- *transaction manager*
- *participants*
- *application*

1. 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.
2. 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:

3. 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.
4. If any of the participants replies “no,” the coordinator sends an *abort* request to all nodes in phase 2.

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

1. When the application wants to begin a distributed transaction, it requests a transaction ID from the coordinator. This transaction ID is globally unique.
2. 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.
3. 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.
4. When a participant receives the prepare request, **it makes sure that it can definitely commit the transaction under all circumstances**.
    1. 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. In other words, the participant surrenders the right to abort the transaction, but without actually committing it.
5. 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*.

6. 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. There is no more going back: 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

We have discussed what happens if one of the participants or the network fails during 2PC:

- if any of the prepare requests fail or time out, the coordinator aborts the transaction;
- if any of the commit or abort requests fail, the coordinator retries them indefinitely. However, it is less clear what happens if the coordinator crashes.

1. If the coordinator fails before sending the prepare requests, a participant can safely abort the transaction.
2. 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.

3. 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*.

The situation is illustrated in Figure 9-10. In this particular example, the coordinator actually decided to commit, and database 2 received the commit request. However, the coordinator crashed before it could send the commit request to database 1, and so database 1 does not know whether to commit or abort. Even a timeout does not help here: if database 1 unilaterally aborts after a timeout, it will end up inconsistent with database 2, which has committed. Similarly, it is not safe to unilaterally commit, because another participant may have aborted.

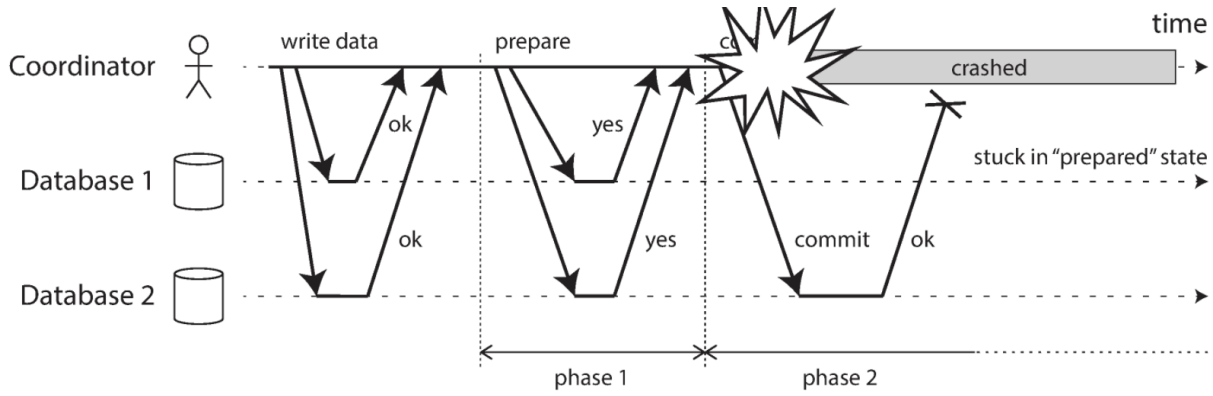


Figure 9-10. The coordinator crashes after participants vote "yes." Database 1 does not know whether to commit or abort.

WAL in coordinator:

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.

- nonblocking atomic commit requires a *perfect failure detector*

Distributed Transactions in Practice:

- Many cloud services choose not to implement distributed transactions due to the operational problems they engender.
- Some implementations of distributed transactions carry a heavy performance penalty
    - E.g. distributed transactions in MySQL are reported to be over 10 times slower than single-node transactions.
- **Two quite different types of distributed transactions**:
    - **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: e.g. two databases from different vendors, or even non-database systems such as message brokers.
    -
- **Exactly-once message processing**:
        - Heterogeneous distributed transactions allow diverse systems to be integrated in powerful ways.
        - E.g. Message Queue work with DB.
        - by atomically committing the message and the side effects of its processing, we can ensure that the message is effectively processed exactly once, even if it required a few retries before it succeeded.
    - **XA transactions**:
        - X/Open XA (short for eXtended Architecture)

        - 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. (e.g. JTA → JDBC, JMS)
        - XA is supported by many traditional relational databases (including PostgreSQL, MySQL, DB2, SQL Server, and Oracle) and message brokers (including ActiveMQ, HornetQ, MSMQ, and IBM MQ).
    - **Holding locks while in doubt**:
        - database transactions usually take a row-level exclusive lock on any rows they modify, to prevent dirty writes.
    - **Recovering from coordinator failure**:
        - In theory, if the coordinator crashes and is restarted, it should cleanly recover its state from the log and resolve any in-doubt transactions.
        - However, in practice, orphaned in-doubt transactions do occur — that is, transactions for which the coordinator cannot decide the outcome for whatever reason.
            - The only way out is for an administrator to manually decide whether to commit or roll back the transactions. (potentially requires a lot of manual effort)
        - Many XA implementations have an emergency escape hatch called **heuristic decisions**. (a euphemism for probably breaking atomicity)
- **Limitations of distributed transactions**:
    - **Transaction coordinator** is itself a kind of database (in which transaction outcomes are stored), and so it needs to be approached with the same care as any other important database. (It like Single Point of Failure)

- **Distributed transactions** thus have a **tendency of amplifying failures**, which runs counter to our goal of building fault-tolerant systems.