Note that a fault is not the same as a failure [2]. A fault is usually defined as one com‐ ponent of the system deviating from its spec, whereas a *failure* is when the system as a whole stops providing the required service to the user. It is impossible to reduce the probability of a fault to zero; therefore it is usually best to design fault-tolerance mechanisms that prevent faults from causing failures. In this book we cover several techniques for building reliable systems from unreliable parts.

*head-of-line blocking*.

For example, percentiles are often used in *service level objectives* (SLOs) and *service level agreements* (SLAs), contracts that define the expected performance and availa‐ bility of a service. An SLA may state that the service is considered to be up if it has a median response time of less than 200 ms and a 99th percentile under 1 s (if the response time is longer, it might as well be down), and the service may be required to be up at least 99.9% of the time. These metrics set expectations for clients of the ser‐ vice and allow customers to demand a refund if the SLA is not met.

In order to figure out how bad your outliers are, you can look at higher percentiles: the *95th*, *99th*, and *99.9th* percentiles are common (abbreviated *p95*, *p99*, and *p999*). They are the response time thresholds at which 95%, 99%, or 99.9% of requests are faster than that particular threshold. For example, if the 95th percentile response time is 1.5 seconds, that means 95 out of 100 requests take less than 1.5 seconds, and 5 out of 100 requests take 1.5 seconds or more.

Even if only a small percentage of backend calls are slow, the chance of getting a slow call increases if an end-user request requires multiple back‐ end calls, and so a higher proportion of end-user requests end up being slow (an effect known as *tail latency amplification*

1. Whether you store an ID or a text string is a question of duplication. When you use an ID, the information that is meaningful to humans (such as the word *Philanthropy*) is stored in only one place, and everything that refers to it uses an ID (which only has meaning within the database). When you store the text directly, you are duplicating the human-meaningful information in every record that uses it.

The advantage of using an ID is that because it has no meaning to humans, it never needs to change: the ID can remain the same, even if the information it identifies changes. Anything that is meaningful to humans may need to change sometime in the future—and if that information is duplicated, all the redundant copies need to be updated. That incurs write overheads, and risks inconsistencies (where some copies of the information are updated but others aren’t). Removing such duplication is the key idea behind *normalization* in databases.ii

ii. Literature on the relational model distinguishes several different normal forms, but the distinctions are of little practical interest. As a rule of thumb, if you’re duplicating values that could be stored in just one place, the schema is not normalized.

**Which data model leads to simpler application code?**

If the data in your application has a document-like structure (i.e., a tree of one-to- many relationships, where typically the entire tree is loaded at once), then it’s proba‐

page60image38881728

**38 | Chapter 2: Data Models and Query Languages**

bly a good idea to use a document model. The relational technique of *shredding*— splitting a document-like structure into multiple tables (like positions, education, and contact\_info in Figure 2-1)—can lead to cumbersome schemas and unnecessa‐ rily complicated application code.

The document model has limitations: for example, you cannot refer directly to a nes‐ ted item within a document, but instead you need to say something like “the second item in the list of positions for user 251” (much like an access path in the hierarchical model). However, as long as documents are not too deeply nested, that is not usually a problem.

The poor support for joins in document databases may or may not be a problem, depending on the application. For example, many-to-many relationships may never be needed in an analytics application that uses a document database to record which events occurred at which time [19].

However, if your application does use many-to-many relationships, the document model becomes less appealing. It’s possible to reduce the need for joins by denormal‐ izing, but then the application code needs to do additional work to keep the denor‐ malized data consistent. Joins can be emulated in application code by making multiple requests to the database, but that also moves complexity into the application and is usually slower than a join performed by specialized code inside the database. In such cases, using a document model can lead to significantly more complex appli‐ cation code and worse performance [15].

It’s not possible to say in general which data model leads to simpler application code; it depends on the kinds of relationships that exist between data items. For highly interconnected data, the document model is awkward, the relational model is accept‐ able, and graph models (see “Graph-Like Data Models” on page 49) are the most natural.

A graph consists of two kinds of objects: *vertices* (also known as *nodes* or *entities*) and *edges* (also known as *relationships* or *arcs*). Many kinds of data can be modeled as a graph. Typical examples include:

*Social graphs*

Vertices are people, and edges indicate which people know each other.

*The web graph*

Vertices are web pages, and edges indicate HTML links to other pages.

*Road or rail networks*

Vertices are junctions, and edges represent the roads or railway lines between them.

Well-known algorithms can operate on these graphs: for example, car navigation sys‐ tems search for the shortest path between two points in a road network, and PageRank can be used on the web graph to determine the popularity of a web page and thus its ranking in search results.

In the examples just given, all the vertices in a graph represent the same kind of thing (people, web pages, or road junctions, respectively). However, graphs are not limited to such *homogeneous* data: an equally powerful use of graphs is to provide a consis‐ tent way of storing completely different types of objects in a single datastore. For example, Facebook maintains a single graph with many different types of vertices and edges: vertices represent people, locations, events, checkins, and comments made by users; edges indicate which people are friends with each other, which checkin hap‐ pened in which location, who commented on which post, who attended which event, and so on [35].

**Property Graphs**

In the property graph model, each vertex consists of:

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

Each edge consists of:  
• A unique identifier

page72image39122128

**50**

• The vertex at which the edge starts (the *tail vertex*) **| Chapter 2: Data Models and Query Languages**

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

You can think of a graph store as consisting of two relational tables, one for vertices and one for edges, as shown in Example 2-2 (this schema uses the PostgreSQL json datatype to store the properties of each vertex or edge). The head and tail vertex are stored for each edge; if you want the set of incoming or outgoing edges for a vertex, you can query the edges table by head\_vertex or tail\_vertex, respectively.

**The Cypher Query Language**

*Cypher* is a declarative query language for property graphs, created for the Neo4j graph database [37]. (It is named after a character in the movie *The Matrix* and is not related to ciphers in cryptography [38].)

Example 2-3 shows the Cypher query to insert the lefthand portion of Figure 2-5 into a graph database. The rest of the graph can be added similarly and is omitted for readability. Each vertex is given a symbolic name like USA or Idaho, and other parts of the query can use those names to create edges between the vertices, using an arrow notation: (Idaho) -[:WITHIN]-> (USA) creates an edge labeled WITHIN, with Idaho as the tail node and USA as the head node.

*Example 2-3. A subset of the data in Figure 2-5, represented as a Cypher query*

**CREATE**

(NAmerica:Location {name:'North America', type:'continent'}),

(USA:Location {name:'United States', type:'country' }),

(Idaho:Location {name:'Idaho', type:'state' }),

(Lucy:Person {name:'Lucy' }),

(Idaho) -[:WITHIN]-> (USA) -[:WITHIN]-> (NAmerica),

(Lucy) -[:BORN\_IN]-> (Idaho)

When all the vertices and edges of Figure 2-5 are added to the database, we can start asking interesting questions: for example, *find the names of all the people who emigra‐ ted from the United States to Europe*. To be more precise, here we want to find all the vertices that have a BORN\_IN edge to a location within the US, and also a LIVING\_IN edge to a location within Europe, and return the name property of each of those verti‐ ces.

Example 2-4 shows how to express that query in Cypher. The same arrow notation is used in a MATCH clause to find patterns in the graph: (person) -[:BORN\_IN]-> ()

page74image38952880

**52 | Chapter 2: Data Models and Query Languages**

matches any two vertices that are related by an edge labeled BORN\_IN. The tail vertex of that edge is bound to the variable person, and the head vertex is left unnamed.

*Example 2-4. Cypher query to find people who emigrated from the US to Europe*

**MATCH**

(person) -[:BORN\_IN]-> () -[:WITHIN\*0..]-> (us:Location {name:'United States'}),

(person) -[:LIVES\_IN]-> () -[:WITHIN\*0..]-> (eu:Location {name:'Europe'}) **RETURN** person.name

The query can be read as follows:  
Find any vertex (call it person) that meets *both* of the following conditions:

1. person has an outgoing BORN\_IN edge to some vertex. From that vertex, you can follow a chain of outgoing WITHIN edges until eventually you reach a vertex of type Location, whose name property is equal to "United States".
2. That same person vertex also has an outgoing LIVES\_IN edge. Following that edge, and then a chain of outgoing WITHIN edges, you eventually reach a vertex of type Location, whose name property is equal to "Europe".

For each such person vertex, return the name property.

There are several possible ways of executing the query. The description given here suggests that you start by scanning all the people in the database, examine each per‐ son’s birthplace and residence, and return only those people who meet the criteria.

But equivalently, you could start with the two Location vertices and work backward. If there is an index on the name property, you can probably efficiently find the two vertices representing the US and Europe. Then you can proceed to find all locations (states, regions, cities, etc.) in the US and Europe respectively by following all incom‐ ing WITHIN edges. Finally, you can look for people who can be found through an incoming BORN\_IN or LIVES\_IN edge at one of the location vertices.

As is typical for a declarative query language, you don’t need to specify such execu‐ tion details when writing the query: the query optimizer automatically chooses the strategy that is predicted to be the most efficient, so you can get on with writing the rest of your application.

An index is an *additional* structure that is derived from the primary data. Many data‐ bases allow you to add and remove indexes, and this doesn’t affect the contents of the database; it only affects the performance of queries. Maintaining additional structures incurs overhead, especially on writes. For writes, it’s hard to beat the performance of simply appending to a file, because that’s the simplest possible write operation. Any kind of index usually slows down writes, because the index also needs to be updated every time data is written.

**Hash Indexes**

Let’s start with indexes for key-value data. This is not the only kind of data you can index, but it’s very common, and it’s a useful building block for more complex indexes.

Key-value stores are quite similar to the *dictionary* type that you can find in most programming languages, and which is usually implemented as a hash map (hash table). Hash maps are described in many algorithms textbooks [1, 2], so we won’t go into detail of how they work here. Since we already have hash maps for our in- memory data structures, why not use them to index our data on disk?

1. However, the hash table index also has limitations:
   * The hash table must fit in memory, so if you have a very large number of keys, you’re out of luck. In principle, you could maintain a hash map on disk, but unfortunately it is difficult to make an on-disk hash map perform well. It requires a lot of random access I/O, it is expensive to grow when it becomes full, and hash collisions require fiddly logic [5].
   * Range queries are not efficient. For example, you cannot easily scan over all keys between kitty00000 and kitty99999—you’d have to look up each key individu‐ ally in the hash maps
2. Now we can make a simple change to the format of our segment files: we require that the sequence of key-value pairs is *sorted by key*. At first glance, that requirement seems to break our ability to use sequential writes, but we’ll get to that in a moment.
3. We call this format *Sorted String Table*, or *SSTable* for short. We also require that each key only appears once within each merged segment file (the compaction process already ensures that). SSTables have several big advantages over log segments with hash indexes:
4. 1. Merging segments is simple and efficient, even if the files are bigger than the available memory. The approach is like the one used in the *mergesort* algorithm and is illustrated in Figure 3-4: you start reading the input files side by side, look at the first key in each file, copy the lowest key (according to the sort order) to the output file, and repeat. This produces a new merged segment file, also sorted by key.
5. You still need an in-memory index to tell you the offsets for some of the keys, but it can be sparse: one key for every few kilobytes of segment file is sufficient, because a few kilobytes can be scanned very quickly.i
6. 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

Fine so far—but how do you get your data to be sorted by key in the first place? Our incoming writes can occur in any order.

Maintaining a sorted structure on disk is possible (see “B-Trees” on page 79), but maintaining it in memory is much easier. There are plenty of well-known tree data structures that you can use, such as red-black trees or AVL trees [2]. With these data structures, you can insert keys in any order and read them back in sorted order.

We can now make our storage engine work as follows:

* When a write comes in, add it to an in-memory balanced tree data structure (for

example, a red-black tree). This in-memory tree is sometimes called a *memtable*.

* When the memtable gets bigger than some threshold—typically a few megabytes —write it out to disk as an SSTable file. This can be done efficiently because the tree already maintains the key-value pairs sorted by key. The new SSTable file becomes the most recent segment of the database. While the SSTable is being written out to disk, writes can continue to a new memtable instance.
* In order to serve a read request, first try to find the key in the memtable, then in the most recent on-disk segment, then in the next-older segment, etc.
* From time to time, run a merging and compaction process in the background to combine segment files and to discard overwritten or deleted values.

This scheme works very well. It only suffers from one problem: if the database crashes, the most recent writes (which are in the memtable but not yet written out to disk) are lost. In order to avoid that problem, we can keep a separate log on disk to which every write is immediately appended, just like in the previous section. That log is not in sorted order, but that doesn’t matter, because its only purpose is to restore the memtable after a crash. Every time the memtable is written out to an SSTable, the corresponding log can be discarded.

Lucene, an indexing engine for full-text search used by Elasticsearch and Solr, uses a similar method for storing its *term dictionary* [12, 13]. A full-text index is much more complex than a key-value index but is based on a similar idea: given a word in a search query, find all the documents (web pages, product descriptions, etc.) that mention the word. This is implemented with a key-value structure where the key is a word (a *term*) and the value is the list of IDs of all the documents that contain the word (the *postings list*). In Lucene, this mapping from term to postings list is kept in SSTable-like sorted files, which are merged in the background as needed [14].

Bloom filter <https://www.geeksforgeeks.org/bloom-filters-introduction-and-python-implementation/>

As always, a lot of detail goes into making a storage engine perform well in practice. For example, the LSM-tree algorithm can be slow when looking up keys that do not exist in the database: you have to check the memtable, then the segments all the way back to the oldest (possibly having to read from disk for each one) before you can be sure that the key does not exist. In order to optimize this kind of access, storage engines often use additional *Bloom filters* [15]. (A Bloom filter is a memory-efficient data structure for approximating the contents of a set. It can tell you if a key does not appear in the database, and thus saves many unnecessary disk reads for nonexistent keys.)

B trees:

Like SSTables, B-trees keep key-value pairs sorted by key, which allows efficient key- value lookups and range queries. But that’s where the similarity ends: B-trees have a very different design philosophy.

The log-structured indexes we saw earlier break the database down into variable-size *segments*, typically several megabytes or more in size, and always write a segment sequentially. By contrast, B-trees break the database down into fixed-size *blocks* or *pages*, traditionally 4 KB in size (sometimes bigger), and read or write one page at a time. This design corresponds more closely to the underlying hardware, as disks are also arranged in fixed-size blocks.

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.

The number of references to child pages in one page of the B-tree is called the *branching factor*. For example, in Figure 3-6 the branching factor is six. In practice, the branching factor depends on the amount of space required to store the page refer‐ ences and the range boundaries, but typically it is several hundred.

If you want to update the value for an existing key in a B-tree, you search for the leaf page containing that key, change the value in that page, and write the page back to disk (any references to that page remain valid). If you want to add a new key, you need to find the page whose range encompasses the new key and add it to that page. If there isn’t enough free space in the page to accommodate the new key, it is split into two half-full pages, and the parent page is updated to account for the new subdi‐ vision of key ranges—see Figure 3-7.ii

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

The basic underlying write operation of a B-tree is to overwrite a page on disk with new data. It is assumed that the overwrite does not change the location of the page; i.e., all references to that page remain intact when the page is overwritten. This is in stark contrast to log-structured indexes such as LSM-trees, which only append to files (and eventually delete obsolete files) but never modify files in place.

You can think of overwriting a page on disk as an actual hardware operation. On a magnetic hard drive, this means moving the disk head to the right place, waiting for the right position on the spinning platter to come around, and then overwriting the appropriate sector with new data. On SSDs, what happens is somewhat more compli‐ cated, due to the fact that an SSD must erase and rewrite fairly large blocks of a stor‐ age chip at a time [19].

Moreover, some operations require several different pages to be overwritten. For example, if you split a page because an insertion caused it to be overfull, you need to write the two pages that were split, and also overwrite their parent page to update the references to the two child pages. This is a dangerous operation, because if the data‐ base crashes after only some of the pages have been written, you end up with a cor‐ rupted index (e.g., there may be an *orphan* page that is not a child of any parent).

In order to make the database resilient to crashes, it is common for B-tree implemen‐ tations to include an additional data structure on disk: a *write-ahead log* (WAL, also known as a *redo log*). This is an append-only file to which every B-tree modification must be written before it can be applied to the pages of the tree itself. When the data‐ base comes back up after a crash, this log is used to restore the B-tree back to a con‐ sistent state [5, 20].

An additional complication of updating pages in place is that careful concurrency control is required if multiple threads are going to access the B-tree at the same time —otherwise a thread may see the tree in an inconsistent state. This is typically done by protecting the tree’s data structures with *latches* (lightweight locks). Log- structured approaches are simpler in this regard, because they do all the merging in the background without interfering with incoming queries and atomically swap old segments for new segments from time to time.

1. Linearizability:Highest level of consistency. Force ordering. Causing request to wait. No concurrency.

Head of line blocking(single point of failure). High latency .Low availability.How to achive this:Single thread ,Raft

Raft algorithm for consensus.

<https://www.geeksforgeeks.org/raft-consensus-algorithm/>

Eventual Consistency: May not get committed data

Causal Consistency:Cause and effect consistency. Lock is applied to key. Lock should happen sequentially rather than concurrently since we might get inconsistent data

**Facebook** is an online Social networking service. After registering to use the site, users can create a user profile, add other users as friends, exchange messages, post status updates and photos, share videos and receive notifications when others update their profiles.

When you log into your account on Facebook, the server will show your own status messages and your friends’ status messages at that point in time. Status messages on Facebook may contain pictures, shared links and stories or your own messages. Naturally, your account data requires a strong consistency, but for status data the weaker consistency models are acceptable. During the time the user is online, the status updates of a user’s friends and of the user do not need to be strictly ordered, and the causal ordering is enough.

Thus when a user A sends a status update and a user B replies to that update, there is a causal order on the two updates. However, when users C and D do a totally unrelated update, the order these updates appear to users A and B is not relevant. This is because users A and B do not know in which order updates are performed.

The reason why the eventual consistency is not enough for Facebook status updates is that the eventual consistency does not require any ordering between writes. Consider a case, where the user A first sends a status update, and after few seconds A updates the first status update. With the eventual consistency, all friends of A could see only the first update, because the eventual consistency does not guarantee that first update is performed before the second one. In the causal consistency, as there is a read (by user A) of first update and then write (updated status from user A), these are causally related and all user A’s friends will naturally see second update.

Although the causal consistency is the possible consistency model for Facebook status updates and several similar distributed services containing status updates like LinkedIn, Twitter and Yahoo, to author’s knowledge there is not scientific or other literature that would show the causal consistency being really used.

https://cloud.google.com/datastore/docs/articles/balancing-strong-and-eventual-consistency-with-google-cloud-datastore#h.w3kz4fze562t

1. QUORUM: Problem with causal is with aggregations. R+w<=n (eventual consistency) ,R+w>N(string consistencey)

Data consistency models:

Non-repeatable reads are when your transaction reads committed UPDATES from another transaction. The same row now has different values than it did when your transaction began.

Phantom reads are similar but when reading from committed INSERTS and/or DELETES from another transaction. There are new rows or rows that have disappeared since you began the transaction.

| **Isolation Level** | **Dirty Read** | **Non-Repeatable Read** | **Phantom Read** |
| --- | --- | --- | --- |
| **Read uncommitted** | **Possible** | **Possible** | **Possible** |
| **Read committed** | **Not possible** | **Possible** | **Possible** |
| **Repeatable read** | **Not possible** | **Not possible** | **Possible** |
| **Serializable** | **Not possible** | **Not possible** | **Not possible** |
| Read uncommitted | Transactions are not isolated from each other. If the DBMS supports other transaction isolation levels, it ignores whatever mechanism it uses to implement those levels. So that they do not adversely affect other transactions, transactions running at the Read Uncommitted level are usually read-only. | | |
| Read committed | The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.  The transaction holds a read lock (if it only reads the row) or write lock (if it updates or deletes the row) on the current row to prevent other transactions from updating or deleting it. The transaction releases read locks when it moves off the current row. It holds write locks until it is committed or rolled back. | | |
| Repeatable read | The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.  The transaction holds read locks on all rows it returns to the application and write locks on all rows it inserts, updates, or deletes. For example, if the transaction includes the SQL statement **SELECT \* FROM Orders**, the transaction read-locks rows as the application fetches them. If the transaction includes the SQL statement **DELETE FROM Orders WHERE Status = 'CLOSED'**, the transaction write-locks rows as it deletes them.  Because other transactions cannot update or delete these rows, the current transaction avoids any nonrepeatable reads. The transaction releases its locks when it is committed or rolled back. | | |
| Serializable | The transaction waits until rows write-locked by other transactions are unlocked; this prevents it from reading any "dirty" data.  The transaction holds a read lock (if it only reads rows) or write lock (if it can update or delete rows) on the range of rows it affects. For example, if the transaction includes the SQL statement **SELECT \* FROM Orders**, the range is the entire Orders table; the transaction read-locks the table and does not allow any new rows to be inserted into it. If the transaction includes the SQL statement **DELETE FROM Orders WHERE Status = 'CLOSED'**, the range is all rows with a Status of "CLOSED"; the transaction write-locks all rows in the Orders table with a Status of "CLOSED" and does not allow any rows to be inserted or updated such that the resulting row has a Status of "CLOSED".  Because other transactions cannot update or delete the rows in the range, the current transaction avoids any nonrepeatable reads. Because other transactions cannot insert any rows in the range, the current transaction avoids any phantoms. The transaction releases its lock when it is committed or rolled back. | | |

<https://www.geeksforgeeks.org/difference-between-shared-lock-and-exclusive-lock/>

If the database option READ\_COMMITTED\_SNAPSHOT is OFF, SQL Server uses a **locking** implementation of the read committed isolation level, where shared locks are taken to prevent a concurrent transaction from concurrently modifying the data, because modification would require an exclusive lock, which is not compatible with the shared lock.

The key difference between SQL Server locking read committed and locking repeatable read (which also takes shared locks when reading data) is that read committed **releases the shared lock as soon as possible**, whereas repeatable read holds these locks to the end of the enclosing transaction.

When locking read committed acquires locks at row granularity, the shared lock taken on a row is **released** when a shared lock is taken on the **next row**. At page granularity, the shared page lock is released when the first row on the next page is read, and so on. Unless a lock-granularity hint is supplied with the query, the database engine decides what level of granularity to start with. Note that granularity hints are only treated as suggestions by the engine, a less granular lock than requested might still be taken initially. Locks might also be escalated during execution from row or page level to partition or table level depending on system configuration.

The important point here is that shared locks are typically held for only a **very short time** while the statement is executing. To address one common misconception explicitly, locking read committed **does not** hold shared locks to the end of the statement.

1. Serializbility:

SQL Server happens to use a locking implementation of the serializable isolation level, where physical locks are acquired and **held** to the end of the transaction (hence the deprecated table hint HOLDLOCK as a synonym for SERIALIZABLE).

This strategy is not quite enough to provide a technical guarantee of full serializability, because new or changed data could appear in a range of rows previously processed by the transaction. This concurrency phenomenon is known as a phantom, and can result in effects which could not have occurred in any serial schedule.

To ensure protection against the phantom concurrency phenomenon, locks taken by SQL Server at the serializable isolation level may also incorporate key-range locking to prevent new or changed rows from appearing between previously-examined index key values. Range locks are not always acquired under the serializable isolation level; all we can say in general is that SQL Server always acquires sufficient locks to meet the logical requirements of the serializable isolation level. In fact, locking implementations quite often acquire more, and stricter, locks than are really needed to guarantee serializability, but I digress.

Locking is just one of the possible physical implementations of the serializable isolation level. We should be careful to mentally separate the specific behaviours of the SQL Server locking implementation from the logical definition of serializable.

As an example of an alternative physical strategy, see the PostgreSQL implementation of [serializable snapshot isolation](http://wiki.postgresql.org/wiki/SSI), though this is just one alternative. Each different physical implementation has its own strengths and weaknesses of course. As an aside, note that Oracle still does not provide a fully compliant implementation of the serializable isolation level. It has an isolation level named serializable, but it does not truly guarantee that transactions will execute according to some possible serial schedule. Oracle instead provides snapshot isolation when serializable is requested, in much the same way PostgreSQL did before serializable snapshot isolation (**SSI**) was implemented.

Snapshot isolation does not prevent concurrency anomalies like write skew, which is not possible under truly serializable isolation. If you are interested, you can find examples of write skew and other concurrency effects allowed by snapshot isolation at the SSI link above. We will also discuss the SQL Server implementation of snapshot isolation level later in the series.

1. Snapshot isolation
2. Reading your own writes
3. Write skew and phantoms
4. Synchornised clocks
5. Sloppy quoroms and hinted handoffs
6. “Serializable Snapshot Isola‐ tion (SSI)”
7. Total order broadcast
8. Search engine working

TODO techs

* Caching memcache
* Solr or elastic search
* Kafka

1. Chaos monkey

In [computer science](https://en.wikipedia.org/wiki/Computer_science), the **log-structured merge-tree** (also known as **LSM tree**, or **LSMT**[[1]](https://en.wikipedia.org/wiki/Log-structured_merge-tree#cite_note-1)) is a [data structure](https://en.wikipedia.org/wiki/Data_structure) with performance characteristics that make it attractive for providing [indexed](https://en.wikipedia.org/wiki/Database_index) access to files with high insert volume, such as [transactional log data](https://en.wikipedia.org/wiki/Transaction_log). LSM trees, like other [search trees](https://en.wikipedia.org/wiki/Search_tree), maintain key-value pairs. LSM trees maintain data in two or more separate structures, each of which is optimized for its respective underlying storage medium; data is synchronized between the two structures efficiently, in batches.

One simple version of the LSM tree is a two-level LSM tree.[[2]](https://en.wikipedia.org/wiki/Log-structured_merge-tree#cite_note-2) As described by [Patrick O'Neil](https://en.wikipedia.org/wiki/Patrick_O%27Neil), a two-level LSM tree comprises two [tree-like](https://en.wikipedia.org/wiki/Tree_(data_structure)) structures, called C0 and C1. C0 is smaller and entirely resident in memory, whereas C1 is resident on disk. New records are inserted into the memory-resident C0 component. If the insertion causes the C0 component to exceed a certain size threshold, a contiguous segment of entries is removed from C0 and merged into C1 on disk. The performance characteristics of LSM trees stem from the fact that each component is tuned to the characteristics of its underlying storage medium, and that data is efficiently migrated across media in rolling batches, using an algorithm reminiscent of [merge sort](https://en.wikipedia.org/wiki/Merge_sort).

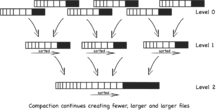
[](https://en.wikipedia.org/wiki/File:LSM_Tree.png)

Diagram illustrating compaction of data in a log-structured merge tree

Most LSM trees used in practice employ multiple levels. Level 0 is kept in main memory, and might be represented using a tree. The on-disk data is organized into sorted *runs* of data. Each run contains data sorted by the index key. A run can be represented on disk as a single file, or alternatively as a collection of files with non-overlapping key ranges. To perform a query on a particular key to get its associated value, one must search in the Level 0 tree and also each run. The Stepped-Merge version of the LSM tree[[3]](https://en.wikipedia.org/wiki/Log-structured_merge-tree#cite_note-3) is a variant of the LSM tree that supports multiple levels with multiple tree structures at each level.

A particular key may appear in several runs, and what that means for a query depends on the application. Some applications simply want the newest key-value pair with a given key. Some applications must combine the values in some way to get the proper aggregate value to return. For example, in [Apache Cassandra](https://en.wikipedia.org/wiki/Apache_Cassandra), each value represents a row in a database, and different versions of the row may have different sets of columns.[[4]](https://en.wikipedia.org/wiki/Log-structured_merge-tree#cite_note-4)

In order to keep down the cost of queries, the system must avoid a situation where there are too many runs.

Extensions to the 'leveled' method to incorporate [B+ tree](https://en.wikipedia.org/wiki/B%2B_tree) structures have been suggested, for example bLSM[[5]](https://en.wikipedia.org/wiki/Log-structured_merge-tree" \l "cite_note-5) and Diff-Index.[[6]](https://en.wikipedia.org/wiki/Log-structured_merge-tree#cite_note-6) LSM-tree was originally designed for write-intensive workloads. As increasingly more read and write workloads co-exist under an LSM-tree storage structure, read data accesses can experience high latency and low throughput due to frequent invalidations of cached data in buffer caches by LSM-tree compaction operations. To re-enable effective buffer caching for fast data accesses, a Log-Structured buffered-Merged tree (LSbM-tree) is proposed and implemented.[[7]](https://en.wikipedia.org/wiki/Log-structured_merge-tree#cite_note-7)