Disambiguating Databases

The topic of data storage is one that doesn't need to be well understood until something goes wrong (data disappears), or something goes really right (too many customers). Because databases can be treated like black boxes with an API, their inner workings are often overlooked. They're often treated as magic things that just take data when offered, and supply it when asked. Since these two operations are the only understood activities of the technology, they are often the only feature points presented when one is comparing different technologies.

But what exactly is an *operation*? Within the realm of databases, this could mean any number of things. Is that operation a transaction? Is it an indexing of data? A retrieval from an index? Does it store the data to a durable medium like a hard disk, or does it beam it by laser towards Alpha Centauri?

It is this ambiguity that causes havoc in the software industry. Misunderstanding the features and guarantees of a database system can cause, at best, user consternation due to slowness or unavailability. At worst, it could result in fiscal damage, or even jail time due to loss data.

The scope of the term "database" is vast. Technically speaking, anything that stores data for later retrieval is a database. Even by that broad definition, there is functionality that is common to most databases. The scope of this article is to enumerate those features at a high level. The intent is to provide the reader with a tool-set with which they might evaluate databases on their relative merits.

Applying this feature driven approach, the reader should see two benefits. Firstly, it will allow them to accurately assess their own needs. Secondly, it will allow them to compare technologies by pairing up like features. When viewed with with this lens, comparative benchmarks will only be valid on databases that are performing equal work and providing the same guarantees.

The best way to illustrate the features of a database is step through a database, from query through storage and back, one operation at a time.

A datum's trip through a database

We will briefly what happens to data from the moment it arrives at the database's query processing engine through its resting place in RAM or on Disk in a database. The intent is to provide the overall process before drilling down into its constituent parts in subsequent sections.

Insert Magnificent Flow Chart Here which shows the steps enveloped by a transaction

Step 1: Query Processing
At the very least, a query engine must parse a query into a request data structure which reflects which sections of which tables need to be retrieved. In relational or distributed database management systems this process features a query planner. It is usually much more advanced.

The query planner looks at the state of the indices that it must traverse in order to complete the query. It might also compare statistics such as table length, record size, or cardinality. It also might change its search algorithm, or parallelize its operations when possible. For complex queries, the query itself may go through simplification and optimization steps.

The few microseconds it spends assessing this information that it keeps handy can shave off seconds or minutes of table and index traversal.

Step 2: Transaction
After the request has been unpacked by the query processing engine, it might possibly begin a transaction. Transactions are intended to provide guarantees regarding data consistency and durability, but can incur a significant overhead if misunderstood.

Step 3: Persistence
If the point of a database is to store data for later retrieval, then we have reached the entire reason for its existence. How (and even when) data is persisted can have the single biggest impact on the speed and success of data storage and retrieval operations.

Step 4: Indexing
Typically, storing the data isn't enough, it must also be retrieved. Indexing is a way of calculating and storing additional pathways with which persisted data might be quickly found and read.

Relevant Hardware and Software Components

This article is about comparing the performance characteristics of different database features. It is rather important then, to note the relative performance of the hardware and software systems leveraged by databases.

The Hard Disk

The highest latency operation you will encounter within a data center is a seek to a random location on a hard disk. For the purposes of this article, all timings presented will be for that of a 7200 RPM drive, which is typical of a multi-terabyte commodity drive found in data intensive applications.

> *SSDs have brought massive latency and throughput improvements to disks. A seek on an SSD is about 60 times faster than a hard disk. They bring their own challenges, however, one interesting one is that the storage cells within an SSD have a fixed lifetime, that is, they can only handle so many writes to them before they fail. For this reason, they have specialized firmware that spreads writes around the disk, garbage collects, and does other bookkeeping operations. Because of this, they have less predictable performance characteristics (though they are predictably faster than hard disks)*

It takes 4.17 milliseconds to seek to a random location on a hard disk. This is because the spindle on which the data is stored actually has to start rotating, spin to the correct location, then stop accurately to within a nanometer. Frankly it is amazing that they are as fast as they are.

Once a location has been found, successive appends-to, or reads-from, that location is significantly cheaper. This is called a sequential read or write.

Algorithms regarding data storage and retrieval have been optimized against this fact since magnetic rotating disks were invented.

The network

The only other operation with latencies within 3 orders of magnitude of a disk seek is a network round trip. It takes about 0.5ms for a packet to go from machine A to machine B, and then for Machine B to turn around to send one back.

To establish a TCP connection via the 3 way handshake it would take about 0.75 ms. Subsequent requests (that fit into a packet) would then take 0.5ms (for the data and the ack)

With this understanding, many data systems have taken to distributing data across a network, in the memory of other computers, as a low-latency replacement for local disk based storage.

Memory

Finding a byte of data in RAM on a machine can be done in approximately 100 nanoseconds, or 41 *thousand* times faster than a hard disk seek.

This means that data structures in memory can be more complex and nuanced than data structures on disk. The design of every database has taken this into careful consideration.

The Page Cache

Because of the massive latency costs that hard disks impose, one optimization that is found in nearly every operating system is the page cache, or buffer cache.

As its name implies, the purpose of the page cache is to optimize disk access by transparently storing contents of files in memory pages in the kernel. The idea is that the same local parts of a disk or a file will be read or written many times in a short period of time. This is usually true for databases.

When a read occurs, if the file contents are not in the cache, it will simultaneously load the data into the cache, and return the data. A write will modify the contents of the cache, but not necessarily write to the hard disk itself. This is to eliminate as many disk accesses as possible. If, in the course of updating an index in a transaction you have to modify 20 locations on disk, modifying the cache 20 times and then flushing to disk will only once will only take 5ms instead of 100ms.

> CAUTION : The page cache is a significant source of optimization, but can also be a source of danger. If writes to the page cache are not flushed to disk, and a power, disk or kernel failure occurs, YOU WILL LOSE YOUR DATA.

The features of a database

We now know how the database operates at a high level, and we understand the relative cost of using the tools a computer provides for us. It's time to do a deep-dive into the core features while using our performance knowledge to attribute latency characteristics to each operation.

Transaction

Transactions are typically used in DBMS's as a boundary to the beginning and end to the set of operations on data. In a busy, concurrent database, many things can go wrong, even when things are going well, writes can still fail simply because an earlier operation has locked or changed a cell. Transactions provide many guarantees as well as a standard programmatic interface to eliminate the hassle of having to handle all potential failure modes in code.

ACID When transactions are provided, they are generally used to provide a set of guarantees commonly known as ACID. ACID stands for Atomicity, Consistency, Isolation and Durability.

Let's briefly look at the ACID guarantees, and then what a DB might do to provide them.

Atomicity Within a transaction, there could be multiple operations. Atomicity guarantees that all operations will either succeed or fail together. Why is this necessary? Let's look at a simple social network database example:

```
BEGIN TRANSACTION;
    DELETE "http://pics.com/pic_of_dead_cat.jpg" from cover_page;
    INSERT "http://pics.com/pic_of_cheeseburger.jpg" into cover_page;
    INSERT "Hey all. Check out tonight's dinner!" into wall;
END TRANSACTION;
```

In this not at all contrived example. We see a problem here if some of these instructions succeed without the others.

If the insert instruction on line 4 succeeds, but the previous two instructions fail, then we might be giving viewers of this page some very incorrect information.

The atomicity guarantee doesn't allow this to happen, if any one instruction fails, then they all fail together.

Consistency This guarantee states that the state of the database will be valid to all users before, during and after the transaction. Databases may make certain guarantees about the data itself. Basic guarantees such as serializability mean that all operations will be processed in the order that they are applied. This might sound easy, but when many applications with many threads are operating on a system concurrently, (expensive) steps must be taken to ensure this is possible.

Relational databases will often make an even larger set of consistency guarantees. This includes foreign key constraints, cascading operations on dependent types, or triggers that might be executed as part of this operation.

What this means, in terms of performance, is that all of these operations might be running while rows and tables are locked for editing, so no other clients will be able to use those parts of the system during that time. It also, clearly effects the round-trip-time of the request.

Isolation Transactions don't happen immediately, they occur in steps, and, like in the Atomicity example, if an outsider were to see a partial set of completed steps, results would range from "amusing" to "horrible wrong". Isolation is the guarantee that say that this won't happen. It hides away all of the operations from others until the transaction completes successfully.

Durability A very important trait indeed. Durability simply promises that when the transaction completes, the results of the operations will be successfully persisted on the specified storage medium (typically the hard disk).

Implementation of Transactions There are 4 steps that are common to an ACID transaction:

1. Log the incoming request to persistent storage in a transaction log (also known as a Write Ahead Log) This will protect the data in case of a system failure. In the worst case scenario, this transaction will be able to be re-started from the log upon start-up.
2. Serialize the new values to the index and table data structures in a way that doesn't interfere with existing operations.
3. Obtain write locks on all cells that need to be modified. Depending on the operation in question and the database. This might mean locking the entire table, the row, or possible the memory page.
4. Move the new values into place.
5. Record the transaction as completed in the transaction log.
6. Flush all changes to disk.

Note that these 6 steps are where every transactional database system keeps its secret sauce. Transactions come at a significant cost. To be able to optimize

this process even a little bit will provide customers with an advantage. Every DBMS will execute steps 1-6 in many different ways. They might try to execute all or some in parallel. They might leverage highly specialized data structure systems, such as MVCC, which reduce the need for locking. In future articles, we will survey prevailing technologies and their approaches.

> *A note on NoSQL:*
> *The biggest "innovation" touted by most NoSQL databases was simply achieving faster operations by removing transactions. It has been stated that NoSQL should more correctly be termed NoACID. One thing to note, when a system supports locking on its data structures, it creates an overhead for every operation on every data structure that might be blocked on a lock. Speedups can be achieved if a database can provide atomic updates to rows without locking, but then transactions become difficult or impossible.*

Transaction Performance Summary Where N is the number of fields written, and M is the number of fields read.

###This should all be a table### - Store transaction to transaction log - 1 disk write, 1 disk flush. 5ms - Lock various rows - N+M disk seeks and reads then write. - (Optionally) Consistency and Integrity checks - M disk reads. (hopefully cached) - Move new structures into place - N memory seeks, possibly disk seeks depending on transaction style. - Mark transaction as completed in log. 1 disk write - Flush cache changes to disk.

Total cost: 2 guaranteed disk seeks. 2 guaranteed disk writes. 0 to max(N+M)/2 additional disk seeks depending on page cache hits. —— 10ms or more

End table

Transactions can actually improve performance of large amounts of serial reads and writes. This is because a transaction only guarantees durability at the end of a transaction. This means that even if there are thousands of rows read/written to disk, it only needs a single flush from page-cache to disk. This fact of course depends on two things: An operating system which supports a write-back instead of a write-through page cache. It also requires having a large enough page cache to contain all of the writes.

Persistence

As was stated above, transactions and even indexing are completely optional within databases. Persistence, however, is the raison d'etre of a database.

Since the performance game in a database is to minimize disk seeks, we must devise a plan to find the home of a piece of data quickly. This includes minimizing

disk seeks, and also storing data with maximum locality so as to minimize cache load times, and maximize chances for cache hits.

To do this, we'll need to store the record data in its own tree based data structure. This is often referred to as a table index. It is technically an index, because trees are tools for efficient look-up. But we'll use a data structure designed to store large amounts of data while minimizing disk seeks.

There are two tree style on disk data structures that form the basis of almost all database storage.

B+ Tree A B+ Tree is a B-tree style index data structure that is optimized for, you guessed it, minimizing disk seeks. It is one of the most common storage mechanisms in databases for table storage. It is also the data structure of choice for almost all modern filesystems.

Log Structured Merge Tree The LSM-tree is a newer disk storage structure that is optimized for a high volume of sequential writes. It was designed to handle massive amounts of streaming events, such as for receiving web server access logs in real-time for later analysis.

Despite its origins in log style event collection, it is beginning to be considered for relational databases as well. There is a major trade-off in an LSM tree, you cannot delete or update in an LSM data structure. Such events are recorded as new records in the log. When reading an LSM tree, one typically starts from the back in order to read the newest version of the data.

Periodically, the records which have been made obsolete by subsequent deletes or updates must be garbage collected.

Persistence Performance Summary To find a location in the tree, it is log base B of N number of searches, where B is the branching factor of the tree and N is the number of items in the tree. For instance, if it had a branching factor of 10, it would take 6 searches, worst case, to find a specific node in a tree. This could mean 6 disk seeks, compare that with 19.9 for a binary tree, or 2.8 for for a tree with a branching factor of 128!. The default branching factor in many on-disk database b-trees is 128, so we'll assume that for the rest of this article.

In addition to the insert itself, the tree nodes might need to be split and rebalanced, which could result in additional seeks, although the items in question are likely now in the cache.

If the database uses MVCC, the tree might be a bit more complex, as it would store and replace nodes which represent versions of the data, as well as the data itself.

For LSM trees, lets assume that the write to the memory-log causes it to pass a threshold and it needs to be merged to the on-disk log. The write to disk is

similar to the B+Tree write above, but it is done in large batches, and since all of the writes are ordered, it is typically done in a small set of sequential writes.

The LSM tree has two background tasks which must occur frequently which can put additional load on the server and the disk. It has to frequently flush the data in the memory portion onto the disk. Secondly, it must perform its compacting step against the on-disk portion. The second step can invoke significant overhead.

B+Tree lookup or update : log base 128 of N disk seeks rebalance on update : log base 128 of N disk seeks

LSM Tree lookup+rebalance every M operations, where M is the maximum population of the in-memory structure.

Indexing

Data is rarely stored as isolated values. Typically heterogeneous collections of fields which make up a record. In relational databases, those fields are called columns, and they are fixed to the schema that defines the tables.

In non-relational databases, heterogeneous fields are still often accommodated and even indexed. When you want to look up a table by specifying one of the fields in a record, that field needs to be part of an index.

An index is just a data structure for performing random lookups given a specified field (or, where supported, a tuple of specified fields). To avoid ambiguity with the table index described in Persistence, we will call these lookup indices.

To Tree, or not to Tree Since these are on-disk data structures as well, a B-tree style index is the the tool of choice for most lookup indices since they efficiently support hard disks. B-trees can accommodate inserts efficiently without having to (re)allocate storage cells for each operation. They also tend to be flat in structure, which reduces the number of nodes which need to be searched, therefor reducing the number of potential disk seeks.

There are other options, however. For instance, a bitmap index is a data-structure that provides very efficient join queries of multiple tables.

Tree style indexes grow linearly for the number of items items in the tree, and search time grows with the depth of the tree (a logarithmic function of the total depth)

Bitmap indices, on the other hand, grow with the number of *different* items in a column. As the name implies, they build bitmaps which represent the membership of values for all relevant columns. Multiple boolean operations against bitmap indexes are very fast, they themselves produce new bitmaps which can be cached efficiently as search results.

One of the other major innovations of bitmap indices is that they can be compressed, and they can even perform query operations while compressed. This makes storage, retrieval faster. It also makes them more CPU cache friendly, which can further reduce latencies.

Lookup Index to Table Index Referencing The value found at the key in an index is actually a pointer back to where the row is physically stored. This can be done two ways:

1. Storing the physical offset directly. The advantage here is incredibly fast lookups. The downside is that write speed is reduced. Whenever more data is inserted into the table index, all effected storage locations must have their new offsets updated within the lookup index.
2. Storing the ID of the row. This means that the database must, in turn, look up the row in the table index by ID. The upside of this extra layer of indirection is no modification of the lookup index is necessary when new data is added to the table index.

Index Performance Summary The cost of B-Tree style indices are discussed above. In the case of lookup-indices, they store less data, and are often optimized to better fit into the page cache, and even the CPU cache.

Bitmap indices often have a much smaller memory footprint, and can sit in memory. They have one major drawback, however, and that is that they cannot be efficiently updated. Since the data is stored, quite literally as a matrix of bits packed into memory, large updates would require a full scan and modification of the index.

costs: lookup or update : log base 128 of N disk seeks re-balance on update : log base 128 of N disk seeks

Odds are that if a re-balance is required, the initial traversal through the tree loaded all relevant data into the page cache, so there would be no additional disk seeks.

Pulling it all together

If there is one take-away here, it is that there are many knobs to turn on a database and an operating system to optimize performance. The trade-offs all revolve around the disk. If you don't need guaranteed, immediate durability for every operation, you can delay persisting the operation to disk and leverage an in-memory data structure temporarily. This can be accomplished either manually in your own batches, such as the in-memory store of the LSM tree, or automatically, via the page cache.

Understand of course that the risk of data-loss is present any time you rely on memory to speed things up. If a failure occurs, those pending writes can disappear. If you are operating in an environment where data loss is acceptable, then by all means use in-memory structures, or don't bother flushing the page cache to disk.

Regardless of your application, you should take time to understand the page cache in your OS. Writes that you think are safe may not be. It, too, has many settings for fine-tuning performance. It can be set to be highly paranoid, but busy, or care-free and fast. It is worth verifying that your expectations match reality.

We now know that write benchmarks for an ACID, b-tree style database should come nowhere close to those of an LSM style log event collector. It would be silly to compare such things. We also know that it would be folly to use an in-memory database to run bank transactions.

As with all software systems, check to see if a database's features match your requirements before comparing its speed.

It may just save your data.

References

1. 3 way handshake
2. Page Cache
3. B+ Trees
4. Bitmap Index
5. Bitmap Index in GPU 1
6. Bitmap Index in GPU 2
7. Log Structured Merge Tree
8. Latency Visualization