

Article development led by **acmqueue**  
queue.acm.org

**Use the database built for your access model.**

BY RICK RICHARDSON

# Disambiguating Databases

THE TOPIC OF data storage is one that does not need to be well understood until something goes wrong (data disappears) or something goes really right (too many customers). Because databases can be treated as black boxes with an API, their inner workings are often overlooked. They are 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 features presented when comparing different technologies.

Benchmarks are often provided in *operations* per second, 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 such as a hard disk, or does it beam it by laser toward 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 data loss.

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. This article enumerates those features at a high level. The intent is to provide readers with a toolset with which they might evaluate databases on their relative merits. Because the topics cannot be covered here in the detail they deserve, references to additional reading have been included. These topics may be the subjects for future articles.

This feature-driven approach should allow readers to assess their own needs and to compare technologies by pairing up like features. When viewed through this lens, comparative benchmarks are valid only on databases that are performing equal work and providing the same guarantees.

Before digging into the features of databases, let's discuss why you would not just take all of the features. The short answer is that each feature typically comes with a performance cost, if not a complexity cost.

Most of the functions performed by a database, as well as the algorithms that implement them, are built to work around the performance bottleneck that is the hard disk. If you have a requirement that your data (and meta data) be durable, then you must pay this penalty one way or another.

## The Hard Disk

The serial ATA (SATA) bus of a typical server (Ivy Bridge Architecture) has a theoretical maximum bandwidth of 750MB per second. That seems high, but compare that with the PCI 3.0 bus, which has a maximum of 40GB per second, or the memory bus, which can do 14.9GB per second per channel (with at least four channels). The SATA bus has the lowest-bandwidth data path within a modern server (excluding peripherals).<sup>5</sup>

In addition to the bandwidth bottleneck, there is latency to consider. The highest-latency operation encountered within a data center is a seek to a ran-



dom location on a hard disk. At present, a 7200RPM disk has a seek time of about four milliseconds. That means it can find and read new locations on disk about 250 times a second. If a server application relies on finding something on disk at every request, it will be capped at 250 requests per second per disk.<sup>4</sup>

Once a location has been found, successive append-to or read-from operations at that same location are 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. Typically, people refer to file operations as either random or sequential, with the understanding the latter comes at a far lower cost than the former.

Solid-state drives (SSDs) have brought massive latency and throughput improvements to disks. A seek on an SSD is about 60 times faster than a hard disk. SSDs bring their own challenges, however. For example, the storage cells within an SSD have a fixed lifetime—that is, they can handle only so many writes to them before they fail. For this reason, they have specialized firmware that spreads writes around the disk, garbage collects, and performs other bookkeeping operations. Thus, they have fewer predictable performance characteristics (though they are predictably faster than hard disks).

### The Page Cache

Because of the high latency and low throughput of hard drives, one optimization found in nearly every operating system is the page cache, or buffer cache. As its name implies, the page cache is meant to transparently optimize away the cost of disk access by storing contents of files in memory pages mapped to the disk by the operating system's kernel. The idea is 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 contents of the page-cache are synchronized with the disk, it will return that content from memory. Conversely, 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.

Assuming that writing a record of data takes five milliseconds, and you have to write 20 different records to disk, performing these operations in the page cache and then flushing to disk would cost only a single disk access, rather than 20. Considering that accessing main memory on a machine is about 40,000 times faster than finding data on disk, the performance savings add up quickly.

Every operating system has a different model for how it flushes its changes to disk, but almost all work with the scheduler to find appropriate points to silently sync the data in memory onto disk. Files and pages can also be manually flushed to disk. This is useful when you need to guarantee that data changes are made permanent.<sup>8</sup>

Be aware the page cache is a significant source of optimization, but it 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. Be mindful of this when analyzing database solutions that leverage the page cache exclusively for their durability operations.

### Database Features

Databases have dozens of classifications. Each of the hundreds of commercially or freely available database systems likely fall into several of these classes. This article skips past the classifications and instead provides a framework through which each database can be evaluated by its features.

The five categories of features explored here are: data model, API, transactions, persistence, and indexing.

**Data model.** There are fundamentally three categories of data models: relational, key value, and hierarchical. Most database systems fall distinctly into one camp but might offer features of the other two.

*Relational model.* Relational databases have enjoyed popularity in recent history. Throughout the 1980s and 1990s, the chief requirement of databases was to conserve a rare and expensive resource: the hard disk. This is where relational databases shine. They allow a database designer to minimize data

duplication within a database through a process called normalization.<sup>7</sup>

Lately, however, the cost of disk storage has fallen considerably,<sup>3</sup> making the economic factor of relational databases less relevant. Despite this, they are still widely used today because of their flexibility and well-understood models. Also, SQL—the lingua franca of relational databases—is commonly known among programmers.

Relational databases work by allowing the creation of arbitrary tables, which organize data into a collection of columns. Each row of the table contains a field from each column. It is customary to organize data into logically separate tables, then relate those tables to one another. This allows constituent parts of a greater whole to be modified independently.

A major downside of relational databases is their storage models do not lend themselves well to storing or retrieving huge amounts of data. Query operations against relational tables typically require accessing multiple indexes and joining and sorting result vectors from multiple tables. These sophisticated schemes work well for 1GB of data but not so well for 1TB of data.

The fundamental trade-off a relational database makes is saving disk space in return for greater CPU and disk load.

The benefits of this model are many: it uses the lowest amount of disk space; it is a well-understood model and query language; it can support a wide variety of use cases; it has schema-enforced data consistency.

The downsides of this model are that it is typically the slowest; its schemas mean a higher programmer overhead for iterating changes; and it has a high degree of complexity with many tuning knobs.

*Key-value model.* Key-value stores have been around since the beginning of persistent storage. They are used when the complexity and overhead of relational systems are not required. Because of their simplicity, efficient storage models, and low runtime overhead, they can usually manage orders-of-magnitude more operations per second than relational databases. Lately, they are being used as event-log collectors. Also, because of their simplicity,

they are often embedded into applications as internal data stores.

Key-value stores operate by associating a key (typically a chunk of bytes) to a value (typically another chunk of bytes). Also, because records are often homogeneous in size and have replicated data, they can be heavily compressed before being stored on disk. This can drastically reduce the bandwidth required across the SATA bus, which can provide performance gains.

Through clever row and column creation, and even schema application, some key-value stores can offer a subset of relational features, but they typically offer far fewer features for data modeling than a relational system. If multiple indexes are needed, they are simulated by using additional key-value lookups.

This is a fast, fairly flexible, and easily understood storage model; however, it often has no schema support, so no consistency checks, and its application logic is more complicated.

*Hierarchical model.* The hierarchical, or document data, model has achieved popularity relatively recently. Its major advantage is ergonomics. The data is stored and retrieved from the database in the way it is stored within objects in an application.


The hierarchical model tends to store all relevant data in a single record, which has delineations for multiple keys and values, where the values could be additional associations of keys and values.

In the general case, all of the data of a real-world object is found within a single record. This means it will necessarily use more storage space than the relational model because it is replicating the data instead of referencing it. It also simplifies the query model since only a single record needs to be retrieved from a single table.


Because the data being stored is heterogeneous in nature, compression can provide limited gains and is typically not used.

Hierarchical databases typically offer some relational features, such as foreign references and multiple indexes. Many such databases do not offer any schema support, as the data structure is arbitrary.

This is the most flexible model. Its arbitrary indexes support easy access



**Because of the high latency and low throughput of hard drives, one optimization found in nearly every operating system is the page cache, or buffer cache.**



to data and it has the highest fidelity between application data structures and on-disk data structures.

On the downside, this model has the highest disk-space usage; and without a schema, data layout is arbitrary, so there are no schema or consistency checks.

#### API

The application programming interface (API) is, in short, how you and your program interact with a database. The interface can be diced in many different dimensions, but let's start with two:

**In process vs. out of process.** If the database is running in the same process (at least partially) as the client application, then typically there is a library of function calls that invoke methods in the database engine directly. This tight coupling results in the lowest possible latency and highest possible bandwidth (memory). It reduces flexibility, however, since it means only a single client application can access the data at one time. It also poses an additional risk: if the client application crashes, so does the database, since they share the same process.

If the database runs in a separate process, a protocol over TCP/IP is typically used. Many RDBMSs (relational database management systems), and recently, other types of databases, support either the ODBC (open database connectivity) or JDBC (Java database connectivity) protocols. This simplifies the creation of client applications, as the libraries that can leverage these protocols are plentiful. A network protocol does drastically improve flexibility of a database, but TCP carries with it latency and bandwidth penalties versus direct memory access.

**SQL vs. not.** SQL is a declarative language that was designed originally as a mechanism to simplify storage and retrieval of relational data. Its usage is ubiquitous, and as such, many developers speak the language fluently. This can aid the adoption of a database.

The biggest “innovation” touted by most NoSQL databases was simply achieving faster operations by removing transactions and relational tables. Many of those databases began to support SQL as an API language, even though they did not use its relational

features. Some SQL features such as querying, filtering, and aggregating were quite useful. Therefore, it was said that NoSQL databases should be re-named NoACID (atomicity, consistency, isolation, and durability) because of their lack of transaction support. In 2015, many of those same databases now have transactional support. These days, NoSQL might be more accurately called NoRelational, but NoSQL sounds better and is close enough.

One challenge of SQL is it must be parsed and compiled by the database engine in order to be used. This imposes a runtime overhead. Most database engines or client APIs work around this by precompiling, or compiling on the first run, the SQL-based function calls into prepared statements. Then the compiled version is saved and used for future calls.

SQL cannot effectively describe all data relationships. For example, hierarchical relationships are difficult to describe in SQL. In addition, because of SQL's declarative nature, iterations or other imperative operations are not describable in the core SQL specification. The specification has been expanded to include recursion to address both iteration and hierarchical relationships. In addition, vendors have provided nonstandard extensions of their own. Support for these extensions is not widely prevalent, however, and neither is the understanding of how to leverage them.<sup>9</sup>

In many cases the features of databases are so sparse, lacking features such as indexing or aggregation, there is simply no reason to support the complexity of a SQL parsing and execution engine. Key-value stores often fall into this category.

## Transactions

A database transaction, by definition, is a unit of work treated in a coherent and reliable way. The most common recipe for database transactions is ACID. Many database systems claim support for transactions or “light-weight” transactions, but they may provide only the features of ACID that are convenient and efficient to support. For example, many distributed databases offer the concept of transactions without the isolation step. This means the data is being modified

in place, and other transactions see that data while it is being modified. You can work around this if this behavior is expected. If not, the results could be disastrous.

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

**Atomicity.** Within a transaction, there could be multiple operations. Atomicity guarantees all operations will either succeed or fail together. An operation could fail for a number of reasons:

- ▶ *Constraints.* A logical constraint is violated, such as foreign keys or uniqueness.

- ▶ *Concurrency.* Another process completes modification of a field that your process was going to modify, and to continue doing so would violate the atomicity guarantee of their transaction.

- ▶ *Failure.* Something in the hardware or software stack fails, causing one of the operations to fail.

In a busy, concurrent database, failures can happen often. Without atomicity, data can get into an inconsistent state very quickly. Thus, atomicity is a key component of the next property of ACID.

**Consistency.** This guarantee means 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 all operations will be processed in the order in which 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 often make an even larger set of consistency guarantees, including foreign-key constraints, cascading operations on dependent types, or triggers that might be executed as part of this operation. In terms of performance, this means all of these operations might be running while rows and/or pages are locked for editing, so no other clients will be able to use those parts of the system during that time. It also, clearly affects the round-trip time of the request.

**Isolation.** Transactions do not happen immediately. They occur in steps, and, as in the atomicity example, if

an outsider were to see a partial set of completed steps, results would range from “amusing” to “horribly wrong.” Isolation is the guarantee that says this will not happen. It hides all of the operations from others until the transaction completes successfully.

**Durability.** An 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.** Six steps are common to ACID transactions:

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 restarted from the log upon startup.

2. Serialize the new values to the index and table data structures in a way that does not 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 possibly the memory page.

4. Move the new values into place.

5. Flush all changes to disk.

6. Record the transaction as completed in the transaction log.

Transactions have performance implications. They can lead to speed-ups over performing the operations piecemeal, since all of the disk operations are batched into a single set of operations. Also, if ACID, transactions are a form of concurrency control. Since they sit at the data itself, they can often be more efficient than custom-built concurrency solutions in the application itself.

On the downside, transactions are not good for highly concurrent applications. Highly contentious operations will generate excessive replays and aborts (which result in more replays). They are also complex—all of the moving parts required to provide transactions add to larger and less maintainable code bases.

## Persistence Models

As previously stated, transactions and even indexing are completely optional

within databases. Persistence, however, is their *raison d'être*.

The performance costs associated with disks (and the risk of data loss associated with the page cache) mean trade-offs with respect to how data is stored and retrieved. A multitude of highly specialized data structures are tailored to different access models, and, typically, if a data structure excels in one area, it will perform poorly in another area. A scheme for inserting large amounts of incoming events in a sequential manner will likely not offer great performance for random updates (or may not even offer that capability at all).

Across all of the potential schemes for storage and retrieval of data, four of the broadest categories are: row based, columnar, memory only, and distributed.

**Row based.** The most common storage scheme is to store data, row by row, in a tree or some other compact data structure on a local hard disk. Although the exact data structures and access models vary, this mechanism is fairly universal.

In row-based storage, the rows themselves are contiguous in memory. This usually means the storage model itself is optimized for fetching regions of entire rows of data at one time.

There are two common data structures for storing rows. The B+ tree is optimized for random retrieval, and the log-structured merge (LSM) tree is optimized for high-volume sequential writes.

**B+ tree.** A B+ tree is a B-tree-style index data structure 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 file systems. The B+ tree is typically a search tree with a high branching factor, and each node is a contiguous chunk of memory containing more than one key. This is specifically designed to maximize the probability that multiple keys can be compared with only a single retrieval of data from disk.<sup>1</sup>

Figure 1 shows how B-tree-based row storage is laid out in memory. Each leaf node has space for four keys, reducing the amount of disk lookups that need to be executed per row. The

key in the tree points to a region on disk or memory that stores the row, which is arranged serially by column. Also note that at each node, not every cell needs to be filled; they can remain free for future values.

**Log structured.** The LSM tree is a newer disk-storage structure optimized for a high volume of sequential writes. It is 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, the LSM tree is beginning

to be used in relational databases as well. It has a major trade-off, however, in that you cannot delete or update in an LSM data structure as part of the standard data path. Such events are recorded as new records in the log. When reading an LSM tree, you typically start from the back to read the newest version of the data.

Periodically, the records that have been made obsolete by subsequent deletes or updates must be garbage collected. This is typically called a compaction process. Some LSM systems compact in separate threads at run-

Figure 1. The layout of a B-tree-based row storage in memory.

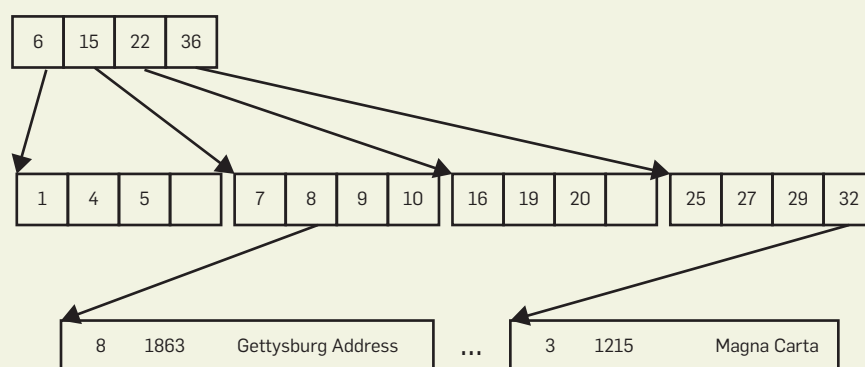
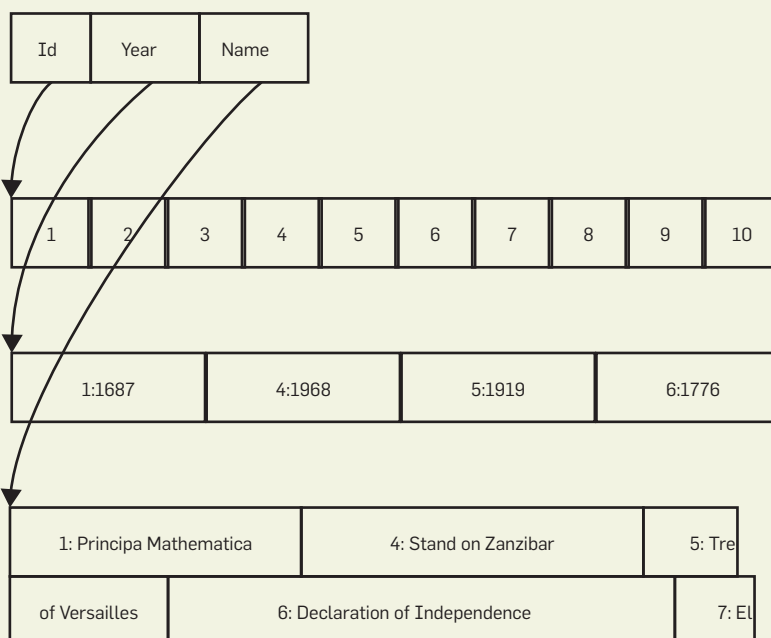


Figure 2. The layout of a columnar data file.



time; other systems attempt to incrementally compact in place.<sup>6</sup>

**Columnar.** Column-based data stores optimize for retrieving regions of the same column of data, rather than rows of data. For this reason, successive columns are stored contiguously in memory.

Because all data types in a column are necessarily the same, compression can have a huge positive impact, thus increasing the amount of data that can be stored and retrieved over the bus. Also, breaking up the data into multiple files, one per column, can take advantage of parallel reads and writes across multiple disks simultaneously.

The downside of column-based databases is they are often inflexible. A simple insert or update requires a significant amount of coordination and calculation. Because data is so typically tightly packed (and compressed) in columns, it is not easy to find and update the data in place.


To help keep “rows” in sync across column files, many times a column field will also contain a copy of the primary key (or row ID, if there are no keys). This aids in reassembly of the data into rows, but it reduces the efficiency of storage and retrieval.<sup>2</sup>

Figure 2 shows a columnar data file. Each data type is laid out in its own contiguous region, whose offset is indicated in the master column. Column files are typically built and rebuilt in batches to serve data-warehousing applications for massive datasets.


**Memory only.** In many cases, durability is simply not a requirement. This is common for systems such as caches, which update frequently and are optimized for nothing other than access speed. Because data in caches is typically short lived, one may not need to persist to disk. This is where in-memory databases shine. Without the requirement to store and retrieve from the disk, a much wider variety of sophisticated trees can be leveraged.

**Distributed.** The topic of distributed databases is vast and would require its own series of articles for proper coverage. In the context of persistence, however, there is one relevant fact: it is faster to copy a dataset across the network within a data center than it is to store it onto a local disk.

Distributed databases can provide an interesting option when establish-



**All of the data of a real-world object is generally found within a single record. This means it will necessarily use more storage space than the relational model, because it is replicating the data instead of referencing it.**



ing the balance between speed and persistence. It might be too risky to store data only in memory on a local machine, as data loss would be complete if the machine crashed. If you copy the data across many machines, then your risk of total data loss is reduced. It is up to the application developer to determine the probability of machine failure and the level of acceptable risk.

#### Page-Cache Considerations

When you opt to store to disk, the page cache is likely to be involved. Accessing the disk directly would be far too cumbersome and would hamper the performance of any other applications running on the system, as you would be monopolizing the disks unnecessarily.

The question of when to flush from the page cache to disk is perhaps the most important of all when designing a database, as it tells you exactly how much risk you have for data loss.

Many databases purport many thousands of operations per second. These databases often operate entirely on data structures on memory-mapped pages in the page cache. That is how they achieve their speed and throughput—by working on in-memory data structures. They do this by deferring all flushing and syncing operations to the operating system itself. This means it is up to the kernel to decide when data in the cache should be persisted to disk. It will likely take into account not only the database, but also all applications running on the system. This means the actual persistence of the data is being left to the operating system, which does not understand the application domain or data reliability requirements.

If durability is not a strong requirement, then deferring to the operating system is probably fine. In most cases, though, it is important to be clear about the behavior of the database with respect to the page cache.

For systems that sync automatically:

- ▶ If it flushes too frequently, it will have poor performance.
- ▶ If it flushes infrequently, it will be faster, but there is a risk of data loss.

A better approach might be to leverage a manual syncing scheme for the database, since that will provide the control to match the guarantees an application requires. This increases the complexity of applications. For highly

concurrent systems, the difficulty level increases, as a disk operation serving one application might interfere excessively with another application.

Systems such as transactions and batch operations that sync at the end can be beneficial. This will reduce the number of disk accesses, but there is a very clear guarantee as to when the data is flushed to disk.

## Indexing

Data is rarely stored as an isolated value. It is typically a heterogeneous collection of fields that make up a record. In relational databases, those fields are called columns, and they are fixed to the schema that defines the tables.

In nonrelational 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 tree, or not to tree.** Since a B-tree is an on-disk data structure as well, it is the tool of choice for most lookup indexes, since it efficiently supports hard disks. Unlike the B+ tree that stores data, a lookup index is optimized for storing references to data. A B-tree can accommodate inserts efficiently without having to allocate storage cells for each operation. It also tends to be flat in structure, reducing the number of nodes that need to be searched, therefore reducing the number of potential disk seeks.

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

A tree-style index grows linearly for the number of items in the index, and search time grows with the depth of the tree (a logarithmic function of the total depth).

A bitmap index, on the other hand, grows with the number of different items in the index. As the name implies, it builds a bitmap that represents the membership of values for all relevant columns. Multiple Boolean operations against bitmap indexes are very fast, and they produce new bitmaps that can be cached efficiently as search results.

One of the other major innovations of the bitmap index is it can be compressed, and it can even perform query operations while compressed. This makes storage retrieval faster. It also makes the bitmap index more CPU cache friendly, which can further reduce latencies. Because of its more complicated update process, the bitmap index tends to be used in read-heavy systems, especially those with multidimensional queries such as OLAP (online analytical processing) cubes.

**Indexing performance summary.** Unless your only data-access model is a full scan of large regions of data, you will probably need indexes. Every additional index that your dataset leverages, however, will add increased disk and CPU load, as well as increasing the latency of inserts.

If your system is read-heavy, and it possesses a relatively low variety of data in the columns (known as low cardinality), you can take advantage of bitmap indexes. For everything else, there are tree indexes.

Many indexes require a unique key to point to a record. If it is the only unique index for that record, it is referred to as the primary key. Even schema-less databases often support such an indexing constraint. They can help ensure consistency and detect errors when loading data.

For performance, there is one basic rule to follow: index as little as possible. Almost every database that supports adding indexes will allow you to add them after the data is loaded. So add them later, once you are sure you need them. Using a unique index can provide a double benefit of ensuring data consistency.

If all of your data is loaded at once (in a data-mart model, perhaps), you might benefit from creating indexes afterward. This can even result in a more efficient index, as many indexes suffer negative effects from fragmentation caused by many inserts and updates.


## Pulling It All Together

The trade-offs between performance and safety revolve around the disk. You might just get the best of both worlds if you choose the database that is built exactly for your access model. Take the time to understand your access model

thoroughly and to know which features you require and which you are willing to forgo in the name of performance.

If you do not need guaranteed and immediate durability for every operation, you can delay persisting the operation to disk by leveraging a memory-mapped data structure. Understand 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.

Regardless of your application, take time to understand the page cache in your operating system. 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 carefree and fast. You should be very clear about how and when your database writes to disk. If it defers to the operating system, then take steps to ensure it behaves correctly for your use case.

It is worth verifying that your expectations match reality. It may just save your data. 

## Related articles on [queue.acm.org](http://queue.acm.org)

### Bridging the Object-Relational Divide

Craig Russell

<http://queue.acm.org/detail.cfm?id=1394139>

### Sentient Data Access via a Diverse Society of Devices

George W. Fitzmaurice et al.

<http://queue.acm.org/detail.cfm?id=966721>

### Distributed Computing Economics

Jim Gray

<http://queue.acm.org/detail.cfm?id=1394131>

## References

1. B-trees; [http://www.scholarpedia.org/article/B-tree\\_and\\_UB-tree](http://www.scholarpedia.org/article/B-tree_and_UB-tree).
2. Column-oriented databases; [http://en.wikipedia.org/wiki/Column-oriented\\_DBMS](http://en.wikipedia.org/wiki/Column-oriented_DBMS).
3. Hard-drive costs; <http://www.mkomo.com/cost-per-gigabyte-update>.
4. Latency numbers; [http://www.eecs.berkeley.edu/~rcs/research/interactive\\_latency.html](http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html).
5. LGA; [http://en.wikipedia.org/wiki/LGA\\_2011](http://en.wikipedia.org/wiki/LGA_2011).
6. LSM trees; <http://dl.acm.org/citation.cfm?id=230826>; and <http://www.eecs.harvard.edu/~margo/cs165/papers/gp-lsm.pdf>.
7. Normalization; [http://en.wikipedia.org/wiki/Database\\_normalization](http://en.wikipedia.org/wiki/Database_normalization).
8. Page cache; <http://www.westnet.com/~gsmith/content/linux-pdflush.htm>.
9. SQL; [http://en.wikipedia.org/wiki/Hierarchical\\_and\\_recursive\\_queries\\_in\\_SQL](http://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL).

**Rick Richardson** is a systems architect for 12Sided Technology where he is helping to reinvent market structure and forge the next generation of trading systems for the financial world.

Copyright held by author. Publication rights licensed to ACM, \$15.00.



Copyright of Communications of the ACM is the property of Association for Computing Machinery and its content may not be copied or emailed to multiple sites or posted to a listserv without the copyright holder's express written permission. However, users may print, download, or email articles for individual use.