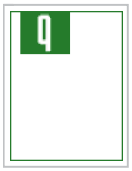
Concurrency
Vol. 12 No. 11 – November 2014



Internal Access Controls
Trust, but Verify

Every day seems to bring news of another dramatic and high-profile security incident, whether it is the discovery of longstanding vulnerabilities in widely used software such as OpenSSL or Bash, or celebrity photographs stolen and publicized. There seems to be an infinite supply of zero-day vulnerabilities and powerful state-sponsored attackers. In the face of such threats, is it even worth trying to protect your systems and data? What can systems security designers and administrators do?

by Geetanjali Sampemane

Scalability Techniques for Practical Synchronization Primitives
Designing locking primitives with performance in mind

In an ideal world, applications are expected to scale automatically when executed on increasingly larger systems. In practice, however, not only does this scaling not occur, but it is common to see performance actually worsen on those larger systems.

by Davidlohr Bueso

Too Big to Fail
Visibility leads to debuggability.

Our project has been rolling out a well-known, distributed key/value store onto our infrastructure, and we've been surprised - more than once - when a simple increase in the number of clients has not only slowed things, but brought them to a complete halt. This then results in rollback while several of us scour the online forums to figure out if anyone else has seen the same problem. The entire reason for using this project's software is to increase the scale of a large system, so I have been surprised at how many times a small increase in load has led to a complete failure. Is there something about scaling systems that's so difficult that these systems become fragile, even at a modest scale?

by George Neville-Neil

Disambiguating Databases
Use the database built for your access model.

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 as 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 features presented when comparing different technologies.

by Rick Richardson

acmqueue

Internal Access Controls

Trust but verify

Geetanjali Sampemane, Google

Every day seems to bring news of another dramatic and high-profile security incident, whether it is the discovery of longstanding vulnerabilities in widely used software such as OpenSSL or Bash, or celebrity photographs stolen and publicized. There seems to be an infinite supply of zero-day vulnerabilities and powerful state-sponsored attackers. In the face of such threats, is it even worth trying to protect your systems and data? What can systems security designers and administrators do?

While these threats are very real, they are not the biggest ones faced by most organizations. Most organizations do not face targeted attacks from hostile governments or criminals intent on stealing users' data; their systems are more likely to be unavailable because of ill-timed software updates or misconfiguration.[2,3,4]

People tend to overreact to dramatic events like terrorist attacks, but they underestimate mundane threats. This is made worse by the fact that the threat landscape is evolving; security advice that was once reasonable becomes obsolete. For example, users are routinely advised to use long, complex passwords, but account compromise caused by password reuse is probably a bigger threat these days than brute-force password cracking, so choosing different passwords for different sites is a better strategy than creating a complex password, memorizing it, and using it everywhere.

In a former life, I helped organizations connect to the Internet, and, as part of that process, warned administrators of new threats they now faced. Those conversations convinced me that practical systems security was still too hard for most people to get right. In the years since, Internet connectivity has become more routine, but methods for securing systems have not kept pace.

This article argues in favor of relatively mundane tools that systems security designers and administrators can use to protect their systems and detect attacks. The principles proposed here are good *internal* access controls: regular automated monitoring and verifying of access configurations, and auditing user access to data. At Google we use these techniques as part of our security strategy, but the principles are applicable to any organization with data to protect.

THE PROBLEM

Systems security administrators, who have more incentive than the average user to get security right, have a hard job. With the increasing proliferation of mobile devices, and increased expectation of anytime/anywhere access, there are only a few high-security environments where users can be prohibited from bringing their personal phones or devices into the corporate environment. Keyboard loggers and malware on personal machines can thus be a path to attack enterprise systems. These devices can be used to exfiltrate data, deliberately or accidentally.

Even when users are restricted to using corporate-owned and -managed devices for work, they still tend to reuse passwords on different systems, and this can provide a vector of attack. Stashes of username/passwords stolen from compromised servers can be retried on other sites, so users

who have reused a username/password on multiple sites can contribute to a bigger problem. People remain vulnerable to social engineering or phishing attacks. Improved authentication systems, such as having a second factor or one-time passwords, help some, but the vast majority of systems do not use those yet.

It is therefore reasonable to assume that some user accounts will get compromised, and it is important to design a system to be resilient to that. Such a system also offers the benefit of providing some protection against malicious insiders. Insider attacks have the potential to cause great damage, since they are caused by people with authorized access and, often, knowledge of systems and processes. Designing protections against insider attacks, however, can be difficult without making the system very cumbersome to use or making users feel untrusted and, therefore, uncooperative with security measures.

Users of the system often do not understand the threat models, so they end up viewing security measures as hoops they have to jump through. Better explanations of the rationale for restrictions may make users more cooperative and dissuade them from looking for ways around the hoops.

Another common problem is misconfigured security controls. As systems and security software grow more complex, the chance of administrators misunderstanding them increases. This can lead to an increase in successful attacks based on such flaws as overlooked default passwords or misconfigured firewall rules.

WHY HAVE INTERNAL ACCESS CONTROLS?

The case for good internal access controls, also called defense in depth, is easy to understand but surprisingly hard to get right in practice. Internal access controls make it harder for attackers to break in (it's not just the firewall that needs to be breached) and limit damage if a system *is* attacked (one phished password will at most get the attackers what that user has access to, not everything on the internal network). Given that a common way systems are attacked is via compromised legitimate user accounts, limiting the damage that a single compromised (or malicious) user can get away with undetected is a useful goal.

The problem is that systems typically start out small, with little or no valuable data, and internal access controls seem like overkill. A good firewall and unrestricted access for (the small number of) authorized users seems like more than enough. People get used to that unrestricted internal access, and processes and tools are developed under that assumption, so adding internal security barriers as the system grows can be disruptive and meet with resistance from users. Removing permissions can also break systems, often in unexpected ways. Retrofitting security into systems is hard.

Most organizations have different kinds of valuable information that needs protecting — company-confidential code and documents, customer information, or data entrusted to them by their users (in the case of cloud service providers). Different employees need access to different subsets of this information, either for development and debugging services, or to provide customer service, or for routine activities such as indexing or backup. How does the organization ensure that people have the right level of access they need and no more?

ACHIEVING THE RIGHT GRANULARITY OF PERMISSIONS

Administrative usability is often overlooked while designing access schemes. Very fine-grained permissions seem like a good idea, since they can grant exactly the necessary access, but they can

easily become too much work to manage. Too many or too low-level permissions can also result in clutter and can be hard to understand and reason about.

The problem with access that is too coarse-grained, on the other hand, is that it can grant too much access. One of the bigger problems with granting too much access is not malicious use but accidental use. Many systems don't enable permissions on an as-needed basis but, rather, have all the permissions a user is granted; this is the equivalent of always running as a superuser rather than as a regular user. Again the problem is one of granularity—having to specify every permission needed becomes tedious, so the tendency is just to leave permissions enabled.

Role-based access control systems[1] help with this by grouping related sets of permissions, but people who perform different roles still end up with a lot of access and not-always-great ways of using the least-privileged access possible.

What can be done about this? Try to understand the system well enough to set up access controls at the right places, but also recognize that you will sometimes get this wrong and will grant more or less access than is needed. This may be because you want to simplify administration or because your mental model of permissions and usage is wrong. It is thus useful to have a system in place to review and monitor permissions, and correct the access configuration as appropriate.

MONITORING ACCESS CONFIGURATIONS

Too often, access requests are reviewed at grant time and never again. People in an organization move across roles and projects, but old permissions do not always expire. Removing unused permissions rarely seems that urgent, and guessing wrong about whether something is unused can break running systems. Unused permissions are not dangerous as long as they remain unused, but they do make the access configuration harder to understand.

At Google we use regular monitoring of access configurations to identify unexpected or unwanted permission behavior. The principle of access-configuration monitoring is much like unit testing for code. Like any type of verification, this is most useful if the verification uses a different approach from the configuration—for example, viewing the permissions in the live production configuration rather than just viewing them as configured.

Administrators specify invariants about the access configuration that should be maintained, and automated test infrastructure periodically verifies that these invariants hold. Preconfigured alerts can be raised if any problems are detected.

Access-configuration monitoring is useful for a few different purposes:

- **Catching differences between static and live configurations.** Some access systems require configuration changes to be reviewed by administrators and then "pushed" to take effect. Occasionally, changes are pushed to live systems without changing the static configuration, or the configuration is changed and not pushed. This sort of situation can lead to unpleasant surprises when long-running systems are restarted.
- **Verifying that the configuration is behaving as expected.** Most configuration languages have their quirks, so it's good to have tests to confirm that they're doing what you expect them to do. A common example is firewall rules that block too much or too little traffic.
- **Tripwire-like monitoring to notify people of changes.** Typically, these are expected changes, but this can catch unauthorized or unexpected changes. It's important that these notices not be too noisy, or people who receive them will tune them out.

• **Catching drifts such as sudden (or even gradual) increases in the number of authorized people.** People often create an ACL (access-control list) for a particular reason, and, over time, tend to use it for other reasons, and the size grows. This sort of monitoring can be useful for recognizing when a group has grown too large, contains too many permissions, and should be split.
• **Verifying that separation of permissions holds.** For example, you may want to prevent any one person from having certain combinations of permissions (like being able to make changes to code and push them to production without review).

AUDITING TO UNDERSTAND ACCESS

Audit logs are a common part of systems security. Typically, all configuration changes and any access to sensitive data generate audit logs, which are hard to subvert. These are often a requirement for regulatory compliance.

Many systems, however, stop at generating the audit logs, using them only for postmortem analysis when something goes wrong. An "audit" in these systems is a sign of trouble. Therefore, access audits should be much more routine, and not a hostile process. Whenever an employee performs a nonroutine access, perhaps for troubleshooting or debugging, the access will be audited. In most cases, this may involve just documenting the reason for access. This develops a culture of accountability, where users expect to have to justify access to sensitive data.

Knowing that all accesses are audited makes granting permissions a little easier. Restricting access to very few people can make a system fragile. It would be more robust if more people were granted emergency access but did not have to use it. Having overbroad permissions, however, is generally a problem. Users could accidentally or maliciously misuse their accesses or become targets for social-engineering attacks because of it. Having good audit logs at the time of *use* of permissions mitigates this risk somewhat, since inappropriate access is unlikely to go undetected.

Routine access audits also help identify access patterns and can help tune access configuration. If all access is logged, it becomes possible to identify unused permissions reliably and prune them safely if needed. This catches the cases where people move jobs or roles without explicitly giving up permissions.

Auditing accesses that are actually used provides visibility into which accesses are needed for people to do their jobs. This allows for the development of better tools, sometimes reducing the amount of access that needs to be granted for a particular task.

Good tools are needed to prevent access audits from becoming bureaucratic nightmares. Routine access can be recognized, based on job roles or access history, and only unusual access patterns can be flagged for extra or manual review.

It is also worth noting that auditing accesses is not a substitute for good access controls; audits can recognize inappropriate access only after it has happened, unlike access controls, which prevent it. As just described, however, auditing all accesses can help tune access configurations. Having to justify access also helps prevent inappropriate access by authorized users. Further, in the unfortunate event of inappropriate access, audit logs can help administrators assess the damage.

CONCLUSION

While high-profile targeted attacks will continue, organizations can do a lot to protect their systems. Internal access controls at the right granularity, combined with access logging and auditing, can

help detect and prevent unwanted access. Access configurations suffer from "bit rot," and users often accumulate unnecessary permissions over time; therefore, regular monitoring, à la unit tests for code, can help detect unwanted situations.

Making security goals and threats clear to system users may encourage their cooperation, rather than leaving them to view security as a nuisance to be worked around. Making the system and security configuration easy for administrators to understand will likely lead to fewer configuration errors, and well-designed monitoring can catch any remaining ones. Finally, making access audits routine can help system administrators understand access patterns and notice unusual access, whether it is a result of some nonroutine event or because a user account has been compromised.

REFERENCES

1. Computer Security Resource Center. 2014. Role based access control (RBAC) and role based security. National Institute of Standards and Technology, Computer Security Division; http://csrc.nist.gov/groups/SNS/rbac/.
2. Hockenson, L. Facebook explains the cause behind its early Thursday downtime. Gigaom; https://gigaom.com/2014/06/19/facebook-explains-the-cause-behind-its-early-thursday-downtime/.
3. Moscaritolo, A. 2014. Verizon billing system hit by major outage. *PC Mag UK*; http://uk.pcmag.com/news/33726/verizon-billing-system-hit-by-major-outage.
4. Wikipedia. 2012 RBS Group computer system problems; http://en.wikipedia.org/wiki/2012_RBS_Group_computer_system_problems.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

GEETANJALI SAMPEMANE (geta@google.com) belongs to the Infrastructure Security and Privacy group at Google. She started her career administering India's first connection to the Internet and then spent a few years working for the United Nations Development Program, helping developing countries connect to the Internet. She has a Ph.D. in computer science from the University of Illinois.

acmqueue

Scalability Techniques for Practical Synchronization Primitives

Designing locking primitives with performance in mind

Davidlohr Bueso, SUSE Labs

In an ideal world, applications are expected to scale automatically when executed on increasingly larger systems. In practice, however, not only does this scaling not occur, but it is common to see performance actually worsen on those larger systems.

While *performance* and *scalability* can be ambiguous terms, they becomes less so when problems present themselves at the lower end of the software stack. This is simply because the number of factors to consider when evaluating a performance problem decreases. As such, concurrent multithreaded programs such as operating-system kernels, hypervisors, and database engines can pay a high price when misusing hardware resources.

This translates into performance issues for applications executing higher up in the stack. One clear example is the design and implementation of synchronization primitives (locks) for shared memory systems. Locks are a way of allowing multiple threads to execute concurrently, providing safe and correct execution context through mutual exclusion. To achieve serialization, locks typically require hardware support through the use of atomic operations such as CAS (compare-and-swap), fetch-and-add, and arithmetic instructions. While details vary across different cache-coherent architectures, atomic operations will broadcast changes across the memory bus, updating the value of the shared variable for every core, forcing cache-line invalidations and, therefore, more cache-line misses. Software engineers often abuse these primitives, leading to significant performance degradation caused by poor lock granularity or high latency.

Both the correctness and the performance of locks depend on the underlying hardware architecture. That is why scalability and hardware implications are so important in the design of locking algorithms. Unfortunately, these are rare considerations in real-world software.

With the advent of increasingly larger multi- and many-core NUMA (non-uniform memory access) systems, the performance penalties of poor locking implementations become painfully evident. These penalties apply to the actual primitive's implementation, as well as its usage, the latter of which many developers directly control by designing locking schemes for data serialization. After decades of research, this is a well-known fact and has never been truer than today. Despite recent technologies such as lock elision and transactional memory, however, concurrency, parallel programming, and synchronization are still challenging topics for practitioners.[10]

Furthermore, because a transactional memory system such as TSX (Transactional Synchronization Extensions) still needs a fallback path with a regular lock when the transaction does not succeed, these challenges are not going to disappear any time soon. In addition, a transactional system does not guarantee specific progress beyond starvation freedom; therefore, this is not always a viable option for regular locking schemes. As a result of the complexities, not only can systems lack in scalability, but also their overall performance can be brought to its knees. Previous work has demonstrated that the cost of a poor, nonscalable, locking implementation grows as the number

of cores in the system increases.[2] The onset of the performance collapse can, in fact, happen very suddenly, with just a few more cores added to the equation. The issues with performance and scalability go well beyond synthetic workloads and benchmarks, affecting real-world software.

There have recently been significant efforts to address lock-scaling issues in the Linux kernel on large high-end servers.[3] Many of the problems and solutions apply to similar system software. This article applies general ideas and lessons learned to a wider systems context, in the hope that it can be helpful to people who are encountering similar scaling problems. Of course, locks are important on any shared memory system, but optimizing them does not imply ignoring the more important aspects: how those locks are used and what is being serialized.

HYBRID LOCKING MODELS AND OPTIMISTIC SPINNING

Locking primitives are traditionally classified as busy-waiting or blocking, depending on what occurs when a lock is not immediately available. If lock hold times are small, such as only serializing a reference count operation, then it makes more sense to avoid all blocking overhead and just burn CPU cycles for a short period of time. This is typically implemented by looping CAS calls until an unlocked condition is met. The alternative is to block until that condition becomes true. Apart from the higher overhead and latency, blocking primitives can commonly have a strict dependency on the operating-system kernel's scheduler. As such, the thread-scheduling policies will increasingly depend on the previous level of execution in the software stack—for example, at hardware, kernel, or multiple user-space layers. Multiple scheduling policies can impact both lock fairness and determinism.

Thrashing is another factor to keep in mind when building sleeping locks. Practitioners must consider the system consequences under heavy lock contention. Popular examples of busy-wait primitives are memory barriers and spinlocks. Blocking mechanisms normally include semaphores and monitors. The Linux kernel, for example, has three primary kinds of sleeping semaphores: mutexes (binary) and two counting semaphores, which includes a widely used reader/writer variant.

The use of a hybrid model is a common way of dealing with the tradeoffs of each lock type. The goal is to delay blocking as much as possible and optimistically busy-wait until the lock is available. Determining when the lock sleeps, however, has an effect on performance. The algorithm needs to perform equally well for workloads that might benefit differently. In other words, users who explicitly want the characteristics of sleeping locks cannot pay performance penalties when a CPU will first spin on a lock for a determined period of time—often, not even at all. In addition, users of general-purpose locking primitives should never be allowed to influence their algorithmic behavior. Misdesigning locking APIs can lead to unexpected consequences later, such as when the lock becomes heavily used. Simplicity is very much a virtue—and a consequence of a well-designed locking interface.

By using the notion of lock ownership, the Linux kernel keeps a pointer to the task that is currently holding the lock. The benefits of knowing the lock owner are twofold: it is a key piece of data when determining when to stop spinning; and it serves a debugging purpose—for example, deadlock detection. Similar strategies date back to 1975, when the notion of lock ownership in databases was first proposed.[8] Because of the overhead of maintaining lock ownership, implementers can decide against reentrant locks, which also typically use a counter field.[15]

The rationale behind optimistic spinning is that if the thread that owns the lock is running, then

it is likely to release the lock soon. In practice, a Linux kernel mutex or rw_semaphore (reader-writer semaphore), the two most commonly used locks throughout the system, can follow up to three possible paths when acquiring the lock, depending on its current state:[12]

• **Fastpath**. It tries to acquire the lock atomically by modifying an internal counter such as fetch_and_add or atomic decrementing. This logic is architecture-specific. As shown here, with x86-64, the mutex-locking fastpath has only two instructions for lock and unlock calls:

```
0000000000000e10 <mutex_lock>:
e21:   f0 ff 0b                lock decl (%rbx)
e24:   79 08                   jns    e2e <mutex_lock+0x1e>

0000000000000bc0 <mutex_unlock>:
bc8:   f0 ff 07                lock incl (%rdi)
bcb:   7f 0a                   jg     bd7 <mutex_unlock+0x17>
```

• **Midpath** (aka optimistic spinning). It tries to spin for acquisition while the lock owner is running and there are no other higher-priority tasks ready to run, needing to reschedule. Spinner threads are queued up using an MCS lock so that only one spinner can compete for the lock.

• **Slowpath.** As a last resort, if the lock still cannot be acquired, the task is added to the wait queue and sleeps until the unlock path wakes it up.

Because hybrid locks can still block, these primitives need to be used safely in a sleeping context. Optimistic spinning has proved its value in operating-system kernels such as Linux and Solaris. Even to this day, simply delaying any kind of blocking overhead can have important impacts on system software. On a Linux VFS (Virtual File System) create+unlink microbenchmark to stress mutexes, optimistic spinning gave a boost in throughput of about 3.5 times on a commodity desktop system, as opposed to immediately sleeping. Similarly, for rw_semaphores on AIM7 workloads, hybrid locking provided an increase in throughput of about 1.8 times.[3]

One notable difference between rw_semaphores and mutexes in the Linux kernel is how they deal with lock ownership. When a lock is shared, the notion of ownership becomes more ambiguous than with exclusive locks. When workloads combine both readers and writers, there is a chance that optimistic spinning can make writers spin excessively since there is no owner to spin on when readers hold the lock. Strategies to overcome this problem exist, such as using heuristics and magic numbers to determine when to stop spinning for readers. Reader ownership can be particularly tricky, however, and tends to add extra complexity and overhead in optimistic-spinning fastpaths. Furthermore, the use of magic numbers in locking primitives can bring unexpected consequences, and they must not be used lightly. By their very nature, heuristics can help in particular scenarios while actually hurting performance in the common cases. Scalability is not about optimizing for the 1 percent and not thinking of the other 99 percent, but quite the opposite. In addition, solving the actual source of contention can be a valid alternative before considering overcomplicated primitives.

CONTRIBUTING FACTORS TO POOR LOCK SCALING

It is not uncommon to find multiple factors contributing to poor lock performance at any given time. Naturally, the impact will vary depending on each system and workload. The factors and lock

properties described here can be divided at a software-engineering level: between implementers, who typically design and implement the locking primitives, and users, who strictly apply them to their parallel or concurrent workloads and algorithms.

• **Length of critical region.** Reducing the length of a critical region can certainly help alleviate lock contention. In addition, the type of primitive used to serialize concurrent threads in lock implementations can play a key role in performance. When holding a lock in a slowpath, such as those that handle contended scenarios when acquiring or releasing a lock, practitioners often need to manage internal wait queues for threads that are waiting to take some action. In these cases, implementers must ensure that the critical regions are short enough to not cause unnecessary internal contention. For example, prechecks or wake-ups (for sleeping primitives) can easily be issued asynchronously, without any additional serialization. Most recently, in the Linux kernel, efforts to shorten critical regions in mutexes and SysV semaphores (as well as other forms of inter-process communication) provided important performance benefits.[3,13]

• **Lock overhead.** This is the resource cost of using a particular lock in terms of both size and latency. Locks embedded in data structures, for example, will bloat that type. Larger structure sizes mean more CPU cache and memory footprint. Thus, size is an important factor when a structure becomes frequently used throughout the system. Implementers also need to consider lock overhead when enlarging a lock type, after some nontrivial modification; this can lead to performance issues in unexpected places. For example, Linux kernel file-system and memory-management developers must take particular care of the size of VFS struct inode (index node) and struct page, optimizing as much as possible.[4] These data structures represent, respectively, information about each file on the system and each of the physical page frames. As such, the more files or memory present, the more instances of these structures are handled by the kernel. It is not uncommon to see machines with tens of millions of cached inodes, so increasing the size of the inode by 4 percent is significant. That's enough to go from having a well-balanced workload to not being able to fit the working set of inodes in memory. Implementers must always keep in mind the size of the locking primitives.

As for latency, busy-wait primitives, because of their simplicity, have better latency than more complex locks. Calls to initialize, but particularly to acquire or release the lock, need to be cheap, incurring the fewest CPU cycles. The internal logic that governs the primitive must not be confused with other factors that affect call latency, such as hold times and contention. Blocking (or sleeping) locks are expected to be more expensive, as any algorithm must at least take into account events such as putting threads to sleep and waking them up when the lock becomes available. The tradeoff, of course, is that users choose blocking locks only when dealing with large critical regions or executing in a context that requires sleeping, such as when allocating memory. Thus, if the average time that a thread expects to wait is less than twice the context-switch time, then spinning will actually be faster than blocking.[15] The quality-of-service guarantee is another factor to consider when choosing between spinning and sleeping locks, particularly in realtime systems. Blocking on larger NUMA systems can ultimately starve the system of resources.

In addition, reader-writer primitives may have different latencies when dealing with shared or exclusive paths. This is not an ideal situation, as users will have to consider the extra overhead penalties when, for example, sharing the lock. Making matters worse, users might not even realize this difference, and, thus, sharing the lock will in fact result in poorer performance than using an exclusive lock. The read:write ratio (mentioned later) is an important factor when determining

whether sharing the lock is actually worth the extra cost of using a reader lock. In general, choosing the wrong type of lock can impact performance, incurring unnecessary overhead.

• **Lock granularity.** This refers to the amount of data a lock protects, normally with a tradeoff between complexity and performance. Coarse granularity tends to be simpler, using fewer locks to protect large critical regions. Fine granularity, on the other hand, can improve performance at the cost of more involved locking schemes, particularly beneficial when a lock is contended. When designing concurrent algorithms, coarse-grained locks might be misused only because they can be simpler and, initially, more obvious than the finer-grained alternatives. Because bugs related to synchronization—such as deadlocks, race conditions, and general corruption—can be particularly hard to debug, programmers might prefer them only because of fear and uncertainty, assuming that protecting too much is better than not enough.

Making matters worse, these issues can easily make entire software systems unreliable—thus, practically useless. Even experienced lock practitioners can overlook the potential performance benefits of fine-grained locking, not noticing scaling problems until they are reported. Perhaps the most famous coarse-grained lock in the Linux kernel was the now-replaced BKL (Big Kernel Lock), serializing threads that enter kernel space to service system calls. Because the lock protects so much data, such primitives are also referred to as giant locks. Futexes are another area in the kernel that can suffer from coarse-grained locking schemes. With a chained hash-table architecture, spinlocks protecting the chain can become heavily contended only because of collisions. Finer graining and parallelizing these paths can be done by simply increasing the size of the hash table. This approach improved the hashing throughput by up to a factor of eight.[3] This fine-grained technique is also known as lock stripping, improving concurrency by having multiple locks to serialize different, nondependent parts of an array, or list-based data structure.

Coarse-grained locks certainly do have their place, nonetheless. One important factor to consider when addressing coarse-grained locks is the overhead of an individual lock. In some cases, the extra memory footprint of embedding additional locks will overshadow the benefits of alleviating the contention. Abusing fine-grained locking can have just as negative an effect on performance as abusing coarse granularity. Furthermore, coarse granularity can be particularly good when contention is not a problem and critical regions and hold times are short, albeit rare.

Both practice and research have shown the performance benefits of combining coarse and fine granularity, although it's usually more complex. Hybrid strategies have been proposed for locking in the Hurricane kernel, combining the benefits of both kinds of granularity.[16] For more than 20 years, the Linux kernel's SysV semaphore implementation suffered from coarse-grained locking when dealing with `semtimedop(2)` system calls. When a finer-grained locking model was introduced for these calls to handle the common case of a task waiting to manipulate a single semaphore within an array containing multiple semaphores, benchmark throughput improved by more than nine times.[13] This hybrid strategy also directly impacts important RDBMS (relational database management system) workloads that rely heavily on these semaphores for internal locking, with contention decreasing from approximately 85 percent to 7 percent. Coarse-grained locking in IPC (inter-process communication) can still occur when manipulating more than a single semaphore. Choosing lock granularity must be a well-informed decision. Changing the granularity later in a program life cycle can be an expensive and error-prone task.

• **Read:write ratios.** This is the ratio between the number of read-only critical regions and the

number of regions where the data in question is modified. Reader/writer locks will take advantage of these scenarios by allowing multiple readers to hold the lock while acquiring it exclusively when modifying protected data. Multiple research and development efforts have tried to optimize primitives for read-mostly situations, making the cost of reader synchronization as minimal as possible—normally at a higher cost, or overhead, for writer threads. Examples include variations of the RCU (read-copy update) mechanism, sequence locks (seqlocks), and read-mostly locks (rmlocks) in FreeBSD.

It is well known that the Linux kernel makes heavy use of RCU, allowing lockless readers to coexist with writers. Because readers do not actually hold a lock, it is a particularly fast mechanism that avoids the overhead and hardware implications that regular reader/writer locks incur. RCU handles updates by: (1) making them visible to readers by single-pointer read and write, ensuring that readers execute before or after the modification, depending on whether they see the update in time; and (2) delaying the reclaiming of the data structure until all readers are done with their critical regions, as it is guaranteed that no readers hold references to the data structure, similar to garbage-collection approaches. Introduced in the 2.5 era, in the early 2000s, it is no coincidence that the adoption of RCU within the kernel has grown substantially, including, among many examples, scaling of the dentry (directory entry) cache, NMI (nonmaskable interrupt), and process ID handling.[11] Most recently, converting the epoll control interface from using a global mutex to RCU permitted significant performance improvements by allowing file-descriptor addition and removal to occur concurrently. Specifically, important Java-based workloads were boosted with throughput improvements of up to 2.5 times on large HP and SGI NUMA systems.

• **Fairness.** Most importantly, fairness avoids lock starvation by using strict semantics to choose which task is next in line to acquire the lock in contended scenarios. A common example of unfair spinlocks is any thread acquiring the lock without respecting if other threads were already waiting for it. This includes the same task constantly reacquiring the lock. Unfair locks tend to maximize throughput but incur higher call latencies. If such a scenario becomes pathological, it is a real concern for lock starvation and lack of progress—unacceptable in real-world software. A widely used solution to this is different variations of ticket spinlocks. Fair locks address starvation issues at the cost of increased preemption sensitivity,[1,6] making them at times unsuitable when preemption cannot be controlled, such as in user-space applications. If the kernel's CPU scheduler preempts a task that is next to acquire the lock and the lock is released during this time, then it will cause the rest of the contending threads to wait until the preempted task is rescheduled. Similarly, the greater the cost of lock acquisition or release, the greater the chance of excessive queuing contributing to poor performance.

Experiments conclude that unfair locks are particularly useful when running more than one thread per core, as they can outperform fair alternatives in highly contended scenarios.[7] Of course, since threads have to wait longer to acquire the lock, this problem can become significantly more pronounced when dealing with NUMA systems, particularly if the fair lock leads to expensive cache-line issues, as described later.

Statistically, it is possible to have different degrees of fairness in NUMA systems. For example, because of CPU node locality, threads can have a better chance of acquiring a lock if it is on the local memory node. By transforming any kind of busy-wait lock into a NUMA-aware primitive, cohort locks were developed to address some of these issues. In this scheme, writer locks are passed among

contending threads within the same NUMA node, while readers maintain shared resources within the same node.

Fairness takes a different turn when dealing with read/write locks, depending on the context, workload, and reader:writer ratios. There will be occasions where it is more suitable to give preference to readers, and vice versa. Either way, practitioners should take special care that the primitive does not starve reader or writer threads to the point that it makes the lock perform poorly in some use cases. One alternative is implementing different variations of the same read/write primitive with particular fairness preferences, but this can also lead to developer misuse in different contexts. For performance reasons, the Linux kernel uses the concept of writer-lock stealing for rw_semaphores, breaking the strict FIFO (first-in first-out) fairness for writers. Writers can atomically acquire the lock, even if other writers are already queued up.

• **Cache-line mishandling.** Cache-line bouncing and contention are probably the two worst forms of performance degradations on large NUMA systems when it comes to low-level locking primitives. Tasks spinning on a contended lock will try to fetch the lock cache line repeatedly in some form of tight CAS loop. For every iteration, usually in an atomic context, the line containing the lock is moved from one CPU cache to another. It is easy to see how this bouncing will degrade performance with increasing CPU counts, incurring expensive memory-bus and interconnect usage penalties. Furthermore, if the lock-protected data structure is in the same cache line, it can significantly slow down the progress of the lock holder, leading to much longer lock hold times.

A straightforward way of addressing cache-line bouncing was proposed 30 years ago[14] with the CCAS (compare compare-and-swap) technique. The idea is to do a simple read of the lock state and incur the read-modify-write CAS operation only when the lock becomes available. The Linux kernel now relies on CCAS techniques for the internal counter checks of both mutexes and read/write semaphores when attempting to acquire a lock exclusively (note that sharing the lock does not require such CAS storms). Concretely, for mutexes, Java-based benchmarks experienced up to a 90 percent throughput increase[3] on a 16-socket, 240-core system. AIM7 benchmark workloads that are designed to execute mostly in kernel space also saw noticeable improvements on eight-node, 80-core Westmere systems with throughput increases of up to three times. As for read/write semaphores, CCAS results from pgbench on smaller, quad-core desktop systems showed improvements of up to 40 percent on 1-GB PostgreSQL databases. Most noticeably improving the cache-line bouncing that occurs when the mmap_sem is heavily contended, this lock is designed, among other things, to serialize concurrent address-space operations.

In general, experimentation shows that CCAS techniques will help on large high-end systems with four or more sockets, or NUMA nodes. Normally, the overhead of checking the counter is so low in noncontended cases that it will not impact performance negatively on smaller systems. Performance work must always ensure that no regressions are introduced in lower-end machines, particularly in the Linux kernel, which spans an enormous user base.

Alternatively, the use of backoff algorithms, which address expensive CAS operations when spinning on a lock, can help alleviate cache-line bouncing and memory-interconnect overhead. As with CCAS techniques, the idea is to delay read-modify-write calls in tight spinning loops, the main difference being the delay factor when the lock is not immediately available. Significant research has attempted to estimate the optimal delay factor across multiple systems and workloads; the different algorithms and heuristics can be classified as static and dynamic. Delaying for a static amount of

time can be optimized for a particular workload but, of course, won't necessarily perform well when generic locking solutions are needed. To this end, dynamic delays are far more realistic, but at the cost of extra overhead in the heuristics and the risk of miscalculating the backoff strategy: threads should not back off for too long, as this can eliminate any performance benefits that these strategies try to introduce.

Good heuristics for backoff algorithms are based on the number of CPUs trying to acquire the lock, such as proportional and exponential, with analogies drawn from CSMA (carrier sense multiple access) networks such as Ethernet. Delaying for the optimal time requires estimating the length of the critical region and the latency of the lock holder to release the lock so another thread can acquire it.[6,15] Needless to say, this sort of information is rarely available for real-world workloads. In addition, backoff algorithms do not address the fact that, just as with regular CAS and CCAS spinlocks, all threads spin on the same shared location, causing cache-coherence traffic on every successful lock acquisition. With these caveats, it is not surprising that backoff algorithms are not suitable for the Linux kernel, even though promising results have been seen with ticket spinlocks and proportional backoff strategies.[5]

DEMONSTRATING THE EFFECTS OF CACHE-LINE CONTENTION

The sole purpose of locking and concurrency is improving system performance by allowing correct parallel executions of threads. Even on uniprocessor systems, concurrency permits the illusion of multitasking in preemptive scheduling environments. If locking primitives get in the way of performance, then they are simply broken. Of course, unlike correctness attributes, locks that perform poorly will matter more on larger systems.

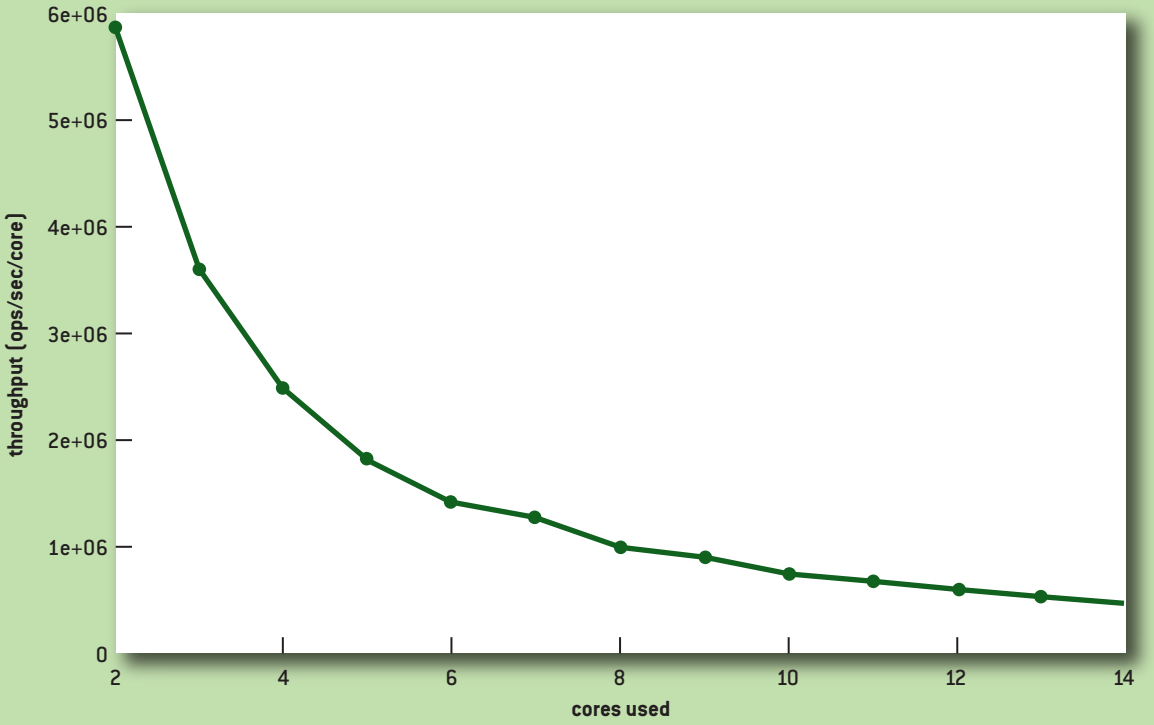
In a modern large-scale, multicore NUMA system, the effects of cache-line contention can go from incurring moderately acceptable overhead to being the culprit for severe system issues. Table 1 shows three high-end HP x86-64 servers used as test systems.[3] All these machines follow a normal NUMA topology in which cores and memory resources are evenly distributed among nodes. There are, therefore, no exotic NUMA use cases.[9] Each socket, or node, has 15 cores, and memory is equally divided as well. A simple spinlock microbenchmark is enough to measure the cost of cache-line contention in large NUMA systems—in this case, using a Linux 3.0-based spinlock implementation, iterating a million times in a tight loop just acquiring and releasing the lock, similar to what a torturing program such as rcutorture would do.

While this pathological behavior doesn't fall into the real-world category, it nicely exemplifies theoretical issues such as lock contention. Furthermore, real software with very similar behavior does exist out there, leading to horrendous contention. Based on the number of threads and the loop iteration count, the average number of operations per second per CPU can be calculated when *N* CPUs are involved in the cache-line contention. This serves as the benchmark throughput.

TABLE 1 **Testing three high-end HP x86-64 servers**

System	Sockets/Cores	Memory
HP DL580 Gen8 (Intel Xeon E7 -4890 v2)	4 sockets/60 cores	2 TB
HP CS900 (Intel Xeon E7 - 4890 v2)	8 sockets/120 cores	6 TB
HP CS900 (Intel Xeon E7 - 4890 v2)	16 sockets/240 cores	12 TB

FIGURE 1 **Effects of Cache-Line Contention within a Single Socket**



For a full understanding of the effects of cache-line contention, the examination must include a single socket. Otherwise, factors such as NUMA awareness and interconnect cost couldn't be differentiated from the results. Furthermore, it should provide the least performance impact and therefore can provide a level ground to compare against other results that involve multisocket communication. Figure 1 shows the microbenchmark throughput regression by increasing core counts within a single socket.

It's easy to see how performance degrades smoothly as more cores contribute to cache-line contention. By maximizing resources and using all cores in a single socket (that is, by a factor of seven), the microbenchmark throughput decreases by 91.8 percent when compared with using only two cores. The situation ends up, therefore, being the exact opposite of what naive developers will normally intuit, expecting performance at least to improve "some," by blindly throwing in more cores. Furthermore, the 91.8 percent decrease can be considered a maximum performance penalty when evaluating systems that will logically partition their software architecture around NUMA topologies. This is commonly done to avoid latency penalties when dealing with remote memory, particularly on large-end machines.

Of course, the situation doesn't get any better. In fact, it gets a great deal worse. Table 2 compares the costs of cache-line contention in a single socket versus two sockets. Most noticeably, there's more than an 80 percent decrease in throughput when going from 15 to 30 cores.

Furthermore, the cost of having the lock's memory location on a remote node is clearly worse

TABLE 2 **Comparison of cache-line contention costs in a single socket with two sockets**

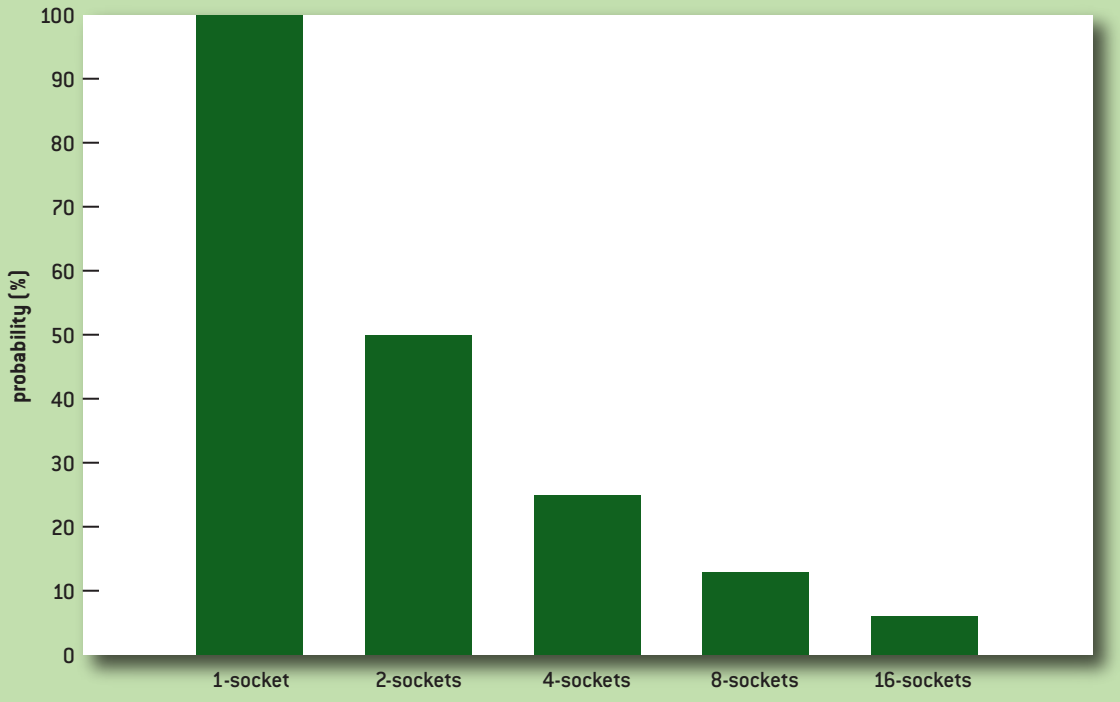
Execution Nodes	Memory Node	Sockets Used	Cores Used	ops/sec/core
Node 0	Node 1	1 socket	15 cores	474,519
-	-	-	-	-
Nodes 0-1	Node 1	2 sockets	30 cores	69,200
Nodes 1-2	Node 1	2 sockets	30 cores	67,126
Nodes 2-3	Node 1	2 sockets	30 cores	45,993

when using two sockets, reducing throughput by more than 90 percent when compared with a single socket. In this case, benchmark runtimes are even more concerning, going from two to 21 seconds to complete the million iterations when the memory is completely remote.

The number of cores is not the only factor to consider when measuring this sort of contention. The distribution method plays a key role in performance: cache-line contention is always better when all the contention is contained in as few sockets as possible.[3] As core counts increase, the FF (first fill) method will always try to use the cores available in the same socket, while an RR (round robin) distribution will use cores from increasing socket counts.

Figure 2 exhibits the probability of inter- versus intra-cache-line contention, estimated by

FIGURE 2 **Chances of Two Random Cores Participating in the Cache-Line Contention**



computing 100/num_sockets. Increasingly large multisocket systems will thus have a much lower chance that two random cores participating in the same cache-line contention will reside on the same socket.

This can have catastrophic effects on performance and demonstrates that contention in a two-socket system is extremely different from that in a 16-socket system. As such, applications tuned for smaller systems cannot be blindly executed on larger systems and immediately expect good results without previous careful thought and analysis. As previously seen, scaling based only on the number of CPUs is likely to introduce significant lock and cache-line contention inside the Linux kernel.

QUEUED/MCS: SCALABLE LOCKING

As already discussed, even busy-wait locks with CCAS or backoff strategies can incur enough cache-line contention to impact overall system performance significantly on large machines. As such, their inherently unfair nature exposes them to an irregular number of cache misses when the lock becomes contended. On the other hand, a truly scalable lock is one that generates a constant number of cache misses, or remote accesses, per acquisition,[2] thus avoiding the sudden performance collapse that occurs with nonscalable alternatives. Queued locks achieve this by ensuring that each contending thread spins for a lock on the CPU's local memory, rather than the lock word. As a consequence of this determinism, queued locks are fair, commonly granting lock ownership to waiting threads in FIFO order. Another attractive property of these locks is the overhead, requiring only O(n+j) space for n threads and j locks,[15] as opposed to O(nj) for nonscalable primitives. Most forms of queued locks—MCS, K42, and CLH, to name a few popular ones—maintain a queue of waiters, each spinning on its own queue entry. The differences in these locks are how the queue is maintained and the changes necessary for the lock's interfaces.

A common experiment is to replace a regular spinlock with an MCS lock,[2,3] always with quite superb results in kernel space (normally using the Linux kernel). Figure 3 compares a vanilla 2.6 Linux spinlock with a prototype MCS implementation on an AIM7 file-system benchmark, serializing concurrent linked-lists operations with a single lock. Contention on this global spinlock is the cause of all the cache-line contention.

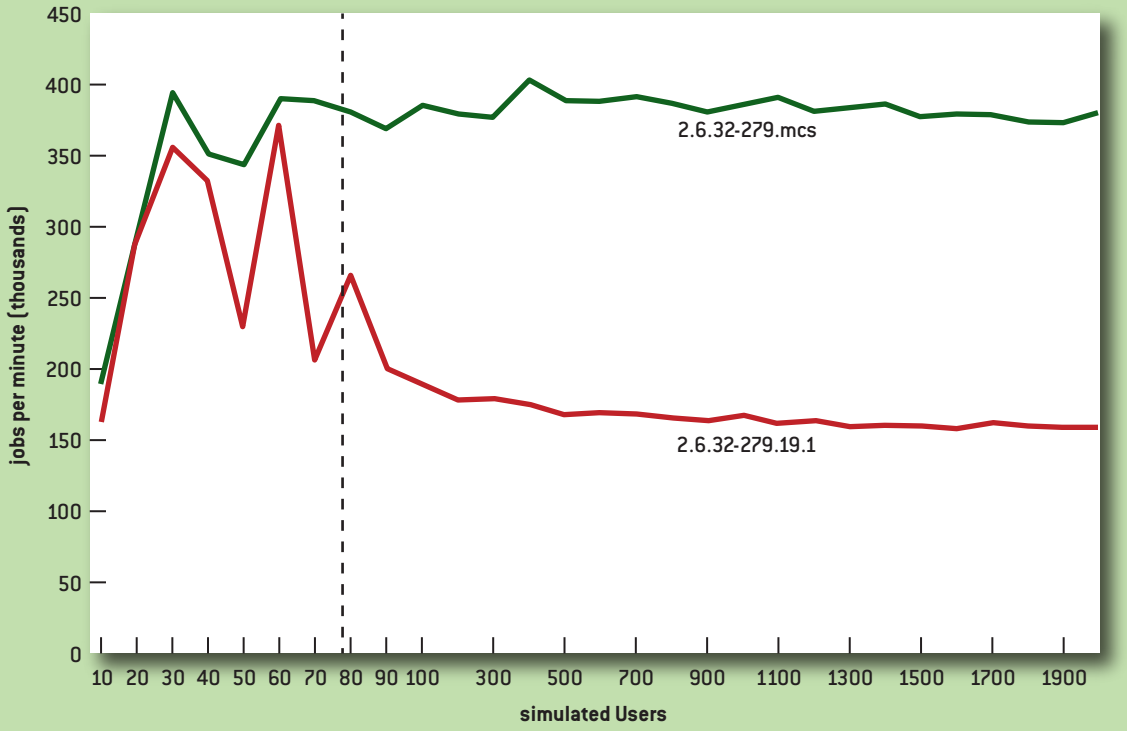
As more users are added to the workload, throughput (jobs per minute) in the spinlock case quickly flatlines, while with the MCS lock it steadily improves by a factor of 2.4. Similarly, system time drops from 54 percent to a mere 2 percent; thus, the bottleneck is entirely removed. As no additional optimization efforts are made, such as finer-graining the lock, it is safe to conclude that the MCS lock alone makes a huge impact by minimizing only the intersocket cache-line traffic.

To this end, MCS locks have been added to the Linux kernel,[2] particularly in the spinning logic of sleeping semaphores. In the case of mutexes, this provides a throughput boost of around 248 percent in a popular Java workload on a 16-socket, 240-core HP Converged-System 900 for SAP HANA. The following code shows the primitive's interfaces:

```
struct mcs_spinlock {
        struct mcs_spinlock *next;
        int locked;
};
```

FIGURE 3

Eight-Socket/80-Core HT-Enabled Xeon




```
void mcs_spin_lock(struct mcs_spinlock **lock, struct mcs_spinlock *node);
void mcs_spin_unlock(struct mcs_spinlock **lock, struct mcs_spinlock *node);
```

The idea is that when a thread needs to acquire the contended lock, it will pass its own local node as an argument, add itself to the queue in a wait-free manner, and spin on its own cache line (specifically on node->locked) until its turn comes to acquire the lock. The FIFO fairness comes naturally as the current lock holder will pass the lock along to the next CPU in the linked list when it releases the lock. One drawback of this algorithm is that, in order to add itself to the wait queue, a task must store the pointer of its local node in the ->next field of its predecessor, potentially incurring extra overhead if the predecessor task needs to release the lock.

Unfortunately, using MCS locks in regular ticket spinlocks cannot be done without enlarging the size of the lock. Spinlocks are used throughout the kernel and cannot be larger than a 32-bit word size. To this end, other queued locking approaches[17] offer valid alternatives to the current ticketing system.

CONCLUSIONS

Designing locking primitives with performance in mind is not only good practice in general, but also can mitigate problems in the future. This article was born from experience with larger-scale servers and on real issues that impacted real clients using Linux. While Donald E. Knuth stated that

"premature optimization is the root of all evil," it is quite the opposite when implementing locking primitives. Empirical data and experiments have shown the cost of misusing locks or ignoring the underlying hardware implications of locking. In addition, both users and implementers must take special care in how these locks are used and the consequences of particular decisions during the life cycle of a lock, such as different degrees of contention.

In the future, as computer systems evolve and increase their processing capabilities, the practice and theory of locking primitives will need to adapt accordingly, ultimately making better use of the hardware architecture. Even today, very specialized locks are available, such as cohort and hierarchical primitives like HCLH that optimize for memory locality, addressing the fairness issues of backoff-based locks, while keeping the benefits of queued-based locks.

Of course, there is no single recipe for locking performance and scalability, and there are many more related topics that the reader can follow up on, including lockless data structures and addressing particular constraints for specialized systems such as priority inversion and inheritance for realtime environments. An article of this nature should, at least, serve as a way of creating awareness among practitioners when dealing with severe scaling problems on large systems.

ACKNOWLEDGMENTS

The scaling work in the Linux kernel has been a team effort, both internally in the Hewlett-Packard Linux performance group and the upstream kernel community for invaluable feedback and entertaining discussions. Many thanks to Scott J. Norton. Without his analysis and empirical data, it would have been very hard to realize the deep architectural implications of poorly implemented primitives. I am in debt to Paul E. Mckenney for his support and Samy Al Bahra for careful review and suggestions, leading to a better article. Thanks to Jim Maurer and the rest of the *ACM Queue* team for their support and feedback.

REFERENCES

1. Al Bahra, S. 2013. Nonblocking algorithms and scalable multicore programming. *ACM Queue* 11(5).
2. Boyd-Wickizer, S. Kaashoek, F. M., Morris, R. Zeldovich, N. 2012. Non-scalable locks are dangerous. *Proceedings of the Linux Symposium*. Ottawa, Canada.
3. Bueso, D., Norton, S. J. 2014. An overview of kernel lock improvements. LinuxCon North America, Chicago, IL; http://events.linuxfoundation.org/sites/events/files/slides/linuxcon-2014-locking-final.pdf.
4. Corbet, J. 2013. Cramming more into struct page. LWN.net; http://lwn.net/Articles/565097/.
5. Corbet, J. 2013. Improving ticket spinlocks. LWN.net; http://lwn.net/Articles/531254/.
6. Crummey-Mellor, J. M., Scott, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions of Computer Systems* 9(1): 21-65.
7. Fuerst, S. 2014. Unfairness and locking. Lockless Inc.; http://locklessinc.com/articles/unfairness/.
8. Gray, J. N., Lorie, R. A., Putzolu, G. R., Traiger, I. L. 1975. Granularity of locks and degrees of

consistency in a shared data base. San Jose, CA: IBM Research Laboratory.
9. Lameter, C. 2014. Normal and exotic use cases for NUMA features. Linux Foundation Collaboration Summit, Napa, CA.
10. McKenney, P. E. 2014. Is parallel programming hard, and, if so, what can you do about it?; https://www.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.
11. McKenney, P. E., Boyd-Wickizer, S., Walpole, J. 2013. RCU usage in the Linux kernel: one decade later; http://www2.rdrop.com/users/paulmck/techreports/RCUUsage.2013.02.24a.pdf.
12. Molnar, I. Bueso, D. 2014. Design of the generic mutex subsystem. Linux kernel source code: documentation/mutex-design.txt.
13. van Riel, R., Bueso, D. 2013. ipc,sem: sysv semaphore scalability. LWN.net; http://lwn.net/Articles/543659/.
14. Rudolph, L., Segall, Z. 1984. Dynamic decentralized cache schemes for MIMD parallel processors. *Proceedings of the 11th Annual International Symposium on Computer Architecture*: 340-347.
15. Scott, M. L. 2013. *Shared-memory synchronization. Synthesis Lectures on Computer Architecture*. San Rafael, CA: Morgan & Claypool Publishers.
16. Unrau, R. C. Krieger, O. Gamsa, B., Stumm, M. 1994. Experiences with locking in a NUMA multiprocessor operating system kernel. Symposium on Operating Systems Design and Implementation; https://www.usenix.org/legacy/publications/library/proceedings/osdi/full_papers/unrau.a.
17. Zijlstra, P., Long, W. 2014. locking: qspinlock. LWN.net; http://lwn.net/Articles/590189/.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DAVIDLOHR BUESO is a performance engineer at SUSE Labs. He is an active Linux kernel contributor in areas such as synchronization primitives, memory management, and IPC—all with a special focus on scalability. His interest in performance goes back to graduate school where he explored ways of improving hypervisor memory management for his M.Sc. thesis. More recently, as a software engineer at Hewlett-Packard, he participated in important Linux kernel scaling efforts on large high-end x86-64 systems.

acmqueue

Too Big to Fail

Visibility leads to debuggability

Dear KV,
Our project has been rolling out a well-known, distributed key/value store onto our infrastructure, and we've been surprised—more than once—when a simple increase in the number of clients has not only slowed things, but brought them to a complete halt. This then results in rollback while several of us scour the online forums to figure out if anyone else has seen the same problem. The entire reason for using this project's software is to increase the scale of a large system, so I have been surprised at how many times a small increase in load has led to a complete failure. Is there something about scaling systems that's so difficult that these systems become fragile, even at a modest scale?

Scaled Back

Dear Scaled,
If someone tells you that scaling out a distributed system is easy they are either lying or drunk, and possibly both. Anyone who has worked with distributed systems for more than a week should have this knowledge integrated into how they think, and if not, they really should start digging ditches. Not to say that ditch digging is easier but it does give you a nice, focused task that's achievable in a linear way, based on the amount of work that you put into it. Distributed systems, on the other hand, react to increases in offered load in what can only politely be referred to as non-deterministic ways. If you think programming a single system is hard, programming a distributed system is a nightmare of Orwellian proportions where you almost are forced to eat rats if you want to join the party.

Non-distributed systems fail in much more predictable ways. Tax a single system and you run out of memory, or CPU, or disk space, or some other resource, and the system has little more than a snow ball's chance of surviving a Hawaiian holiday. The parts of the problem are so much closer together and the communication between those components is so much more reliable that figuring out "who did what to whom" is tractable. Unpredictable things can happen when you overload a single computer, but you generally have complete control over all of the resources involved. Run out of RAM? Buy more. Run out of CPU, profile and fix your code. Too much data on disk? Buy a bigger one. Moore's law is still on your side in many cases, giving you double the resources every 18 months.

The problem is that eventually you will probably want a set of computers to implement your target system. Once you go from one computer to two, it's like going from a single child to two children. To paraphrase an old comedy sketch, if you only have one child, it's not the same has having two or more children. Why? Because when you have one child and all the cookies are gone from the cookie jar, YOU KNOW WHO DID IT! Once you have two or more children, each has some level of plausible deniability. They can, and will, lie to get away with having eaten the cookies. Short of slipping your kids a truth serum at breakfast every morning, you have no idea who is

telling the truth and who is lying. The problem of truthfulness in communication has been heavily studied in computer science, and yet we still do not have completely reliable ways to build large distributed systems.

One way that builders of distributed systems have tried to address this problem is to put in somewhat arbitrary limits to prevent the system from ever getting too large and unwieldy. The distributed key store Redis had a limit of 10,000 clients that could connect to the system. Why 10,000? No clue, it's not even a typical power of 2. One might have expected 8,192 or 16,384, but that's probably another article. Perhaps the authors had been reading the *Tao Te Ching* and felt that their universe only needed to contain 10,000 things. Whatever the reason, this seemed like a good idea at the time.

Of course, limiting the number of clients is only one way of protecting a distributed system against overload. What happens when a distributed system moves from running on 1Gbps network hardware to 10Gbps NICs? Moving from 1Gbps to 10Gbps doesn't "just" increase the bandwidth by an order of magnitude, it also reduces the request latency. Can a system with 10,000 nodes move smoothly from 1G to 10G? Good question, you'd need to test or model that, but it's pretty likely that a single limitation—such as number of clients—is going to be insufficient to prevent the system from getting into some very odd situations. Depending on how the overall system decides to parcel out work, you might wind up with hot spots, places where a bunch of requests all get directed to a single resource, effectively creating what looks like a denial of service attack and destroying a node's effective throughput. The system will then fail out that node and redistribute the work again, perhaps picking another target, and taking it out of the system because it looks like it, too, has failed. In the worst case, this continues until the entire system is brought to its knees and fails to make any progress on solving the original problem that was set for it.

Distributed systems that use a hash function to parcel out work are often dogged by this problem. One way to judge a hash function is by how well-distributed the results of the hashing function are, based on the input. A good hash function for distributing work would parcel out work completely evenly to all nodes based on the input, but having a good hash function isn't always good enough. You might have a great hash function, but feed it poor data. If the source data fed into the hash function doesn't have sufficient diversity (that is, it is relatively static over some measure, such as requests) then it doesn't matter how good the function is, as it still won't distribute work evenly over the nodes.

Take, for example, the traditional networking 4-tuple, source and destination IP address, and source and destination port. Together this is 96 bits of data, which seems like a reasonable amount of data to feed the hashing function. In a typical networking cluster, the network will be one of the three well-known RFC 1918 addresses (192.168.0.0/16, 172.16.0.0/12, or 10.0.0.0/8). Let's imagine a network of 8,192 hosts, because I happen to like powers of 2. Ignoring subnettting completely, we assign all 8,192 hosts addresses from the 192.168.0.0 space, numbering them consecutively 192.168.0.1-192.168.32.1. The service being requested has a constant destination port number (e.g., 6379) and the source port is ephemeral. The data we now put into our hash function are the two IPs and the ports. The source port is pseudo-randomly chosen by the system at connection time from a range of nearly 16 bits. It's nearly 16 bits because some parts of the port range are reserved for privileged programs, and we're building an underprivileged system. The destination port is constant, so we remove 16 bits of change from the input to the function. Those nice fat IPv4 addresses that

should be giving us 64 bits of data to hash on actually only give us 13 bits, because that's all we need to encode 8,192 hosts. The input to our hashing function isn't 96 bits, but is actually fewer than 42. Knowing that, you might pick a different hash function or change the inputs, inputs that really do lead to the output being spaced evenly over our hosts. How work is spread over the set of hosts in a distributed system is one of the main keys to whether that system can scale predictably, or at all.

An exhaustive discussion of how to scale distributed systems is a topic for a book far longer than this piece, but we can't leave the topic until we talk about what debugging features exist in the distributed system. "The system is slow" is a poor bug report—in fact, it is useless. However, it is the one most often uttered in relation to distributed systems. Typically the first thing that users of the system notice is that the response time has increased and that the results they get from the system take far longer than normal. A distributed system needs to express, in some way, its local and remote service times so that the systems operators, such as the devops or systems administration teams, can track down the problem. Hot spots can be found through the periodic logging of the service request arrival and completion on each host. Such logging needs to be lightweight and not directed to a single host, which is a common mistake. When your system gets busy and the logging output starts taking out the servers, that's bad. Recording system level metrics, including CPU, memory and network utilization will also help in tracking down problems, as will the recording of network errors. If the underlying communications medium becomes overloaded, this may not show up on a single host, but will result in a distributed set of errors, with a small number at each node, which lead to chaotic effects over the whole system. Visibility leads to debuggability; you cannot have the latter without the former.

Coming back around to your original point, I am not surprised that small increases in offered load are causing your distributed system to fail, and, in fact, I am most surprised that some distributed systems work at all. Making the load, hot spots, and errors visible over the system may help you track down the problem and continue to scale it out even further. Or, you may find that there are limits to the design of the system you are using, and you'll have to either choose another or write your own. I think you can see now why you might want to avoid the latter at all costs.

KV

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

KODE VICIOUS, known to mere mortals as George V. Neville-Neil, works on networking and operating system code for fun and profit. He also teaches courses on various subjects related to programming. His areas of interest are code spelunking, operating systems, and rewriting your bad code (OK, maybe not that last one). He earned his bachelor's degree in computer science at Northeastern University in Boston, Massachusetts, and is a member of ACM, the Usenix Association, and IEEE. He is an avid bicyclist and traveler who currently lives in New York City.

acmqueue

Disambiguating Databases

Use the database built for your access model

Rick Richardson

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 as 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 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 subject of 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 wouldn't 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 metadata) be durable, then you must pay this penalty one way or another.

THE HARD DISK

The SATA (serial ATA) bus of a typical server (Ivy Bridge Architecture) has a theoretical maximum bandwidth of 750 MB per second. That seems high, but compare that with the PCI 3.0 bus, which has a maximum of 40 GB per second, or the memory bus, which can do 14.9 GB per second per channel (with at least four channels). The SATA bus has the lowest-bandwidth data path within a modern server (excluding peripherals).[1]

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 random location on a hard disk. At present, a 7200-RPM 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.[2]

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 for 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 that the latter comes at a far lower cost than the former.

SSDs (solid-state drives) 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 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 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.[3]

Be aware that 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 '80s and '90s, 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.[4]

Lately, however, the cost of disk storage has fallen considerably,[5] making the economic advantage 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 that their storage models don't 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 1 GB of data but not so well for 1 TB of data.

The fundamental tradeoff a relational database makes is saving disk space at the cost of 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 that this model 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 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

API, which stands for application programming interface, 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 VERSUS 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 that only a single client application can access the data at a 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 libraries that can leverage these protocols are plentiful. A network protocol does drastically improve the flexibility of a database, but TCP carries with it latency and bandwidth penalties versus direct memory access.

SQL VERSUS NOT

SQL is a declarative language that was designed originally as a mechanism to simplify storage and retrieval of relational data. It 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 didn't 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 renamed NoACID (atomicity, consistency, isolation, and durability) because of their lack of transaction support. In 2014 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 that it must be parsed and compiled by the database engine in order to be used. This imposes a runtime overhead. Most database engines and 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 widespread, however, and neither is the understanding of how to leverage them.[6]

In many cases the features of databases are so sparse, lacking features such as indexing or aggregation, that 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 "lightweight" 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 that 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 that 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 the other 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 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 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 that 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 don't 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 won't 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 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 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.

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 the moving parts required to provide transactions add to larger and less maintainable code bases.

PERSISTENCE MODELS

As previously stated here, 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 tradeoffs 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.

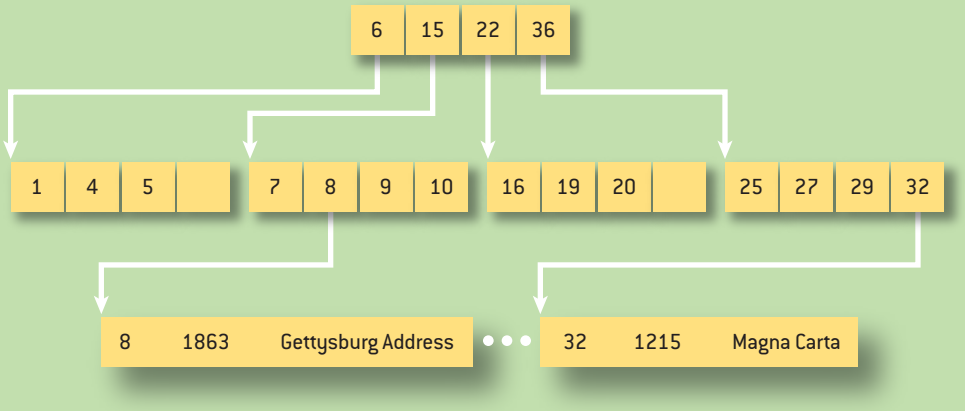
In row-based storage, the rows themselves are contiguous in memory. This usually means that the storage model itself is optimized for fetching regions of entire rows of data at one time.

There are two common data structures for storing rows. The B+ tree is optimized for random retrieval, and the LSM (log-structured merge) 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.[7]

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

FIGURE 1 **The Layout of a B-Tree-Based Row Storage in Memory**



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 realtime 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 tradeoff, however, in that you cannot delete or update in an LSM data structure as part of the standard data path. Deletions and updates 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 runtime; other systems attempt to incrementally compact in place.[8]

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 that they are often inflexible. A simple insert or update requires a significant amount of coordination and calculation. Because data is typically so 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, often 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.

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 data sets.

MEMORY ONLY

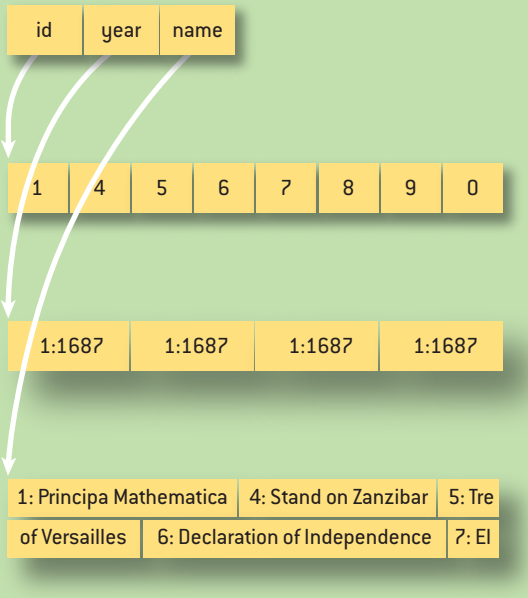
In many cases, durability is simply not a requirement. This is common for systems such as caches, which update frequently and are optimized for nothing other than access speed. Because data in caches is typically short-lived, it may not need to persist to disk. This is where in-memory databases shine. Without the requirement to store and retrieve from 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 data set across the network within a data center than it is to store it onto local disk.

Distributed databases can provide an interesting option when establishing 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,

FIGURE 2 **The Layout of a Columnar Data File**



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 of 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 that 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 with 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 that 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 data set leverages, however, will increase 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. It 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're 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 tradeoffs 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 don't need guaranteed and immediate durability for every operation, you can delay persisting the operation to disk by leveraging a memory-mapped data structure. Understand 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.

Regardless of your application, take time to understand the page cache in your operating system. Writes that you think are safe may not be. The cache, 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 that it behaves correctly for your use case.

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

REFERENCES

1. LGA; http://en.wikipedia.org/wiki/LGA_2011.
2. Latency numbers; http://www.eecs.berkeley.edu/~rcs/research/interactive_latency.html.
3. Page cache; http://www.westnet.com/~gsmith/content/linux-pdflush.htm.
4. Normalization; http://en.wikipedia.org/wiki/Database_normalization.
5. Hard-drive costs; http://www.mkomo.com/cost-per-gigabyte-update.
6. SQL; http://en.wikipedia.org/wiki/Hierarchical_and_recursive_queries_in_SQL.
7. B-trees; http://www.scholarpedia.org/article/B-tree_and_UB-tree.
8. LSM trees; http://dl.acm.org/citation.cfm?id=230826; and http://www.eecs.harvard.edu/~margo/cs165/papers/gp-lsm.pdf.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

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. His passion lies in massive distributed systems and databases. He firmly believes we can make the world a better place through data literacy.