SCYLLA

SCYLLADB WHITE PAPER

Where Apache Cassandra Falls Short, and Why

APACHE CASSANDRA: GREAT IN THEORY, PROBLEMATIC IN PRACTICE

Apache Cassandra, an early entrant in the NoSQL arena, quickly became a popular database for non-relational data projects. Cassandra provides IT organizations with an easy way to scale data across multiple nodes and datacenters with fault tolerance and data redundancy. However, in practice, Cassandra has proven to be expensive and problematic due to several key limitations of the underlying architecture.

Cassandra users commonly struggle with issues such as:

- **Inefficient Utilization:** Cassandra cannot efficiently exploit modern computing resources, in particular multi-core CPUs and high-density storage servers.
- **Unpredictable and Unbounded Latency:** Memory management in the Java virtual machine (JVM) produces unpredictable and unbounded latency.
- **Team Intensive:** Operating Cassandra at scale requires dedicated full-time experts with an increasingly scarce and expensive skillset.
- **Manual Tuning:** Clusters require operators to perform intricate yet unpredictable tuning procedures while combating compactions and garbage collection storms.

Cassandra 4.0 does offer some performance improvements vs. previous Cassandra releases, but it doesn't solve these fundamental issues. The root cause of these frustrations ultimately lies in Cassandra's design, and that has not changed substantially since its introduction in 2008.

This paper explores four core issues at the root of common problems encountered running Cassandra in production. It also introduces how ScyllaDB, a Cassandra-compatible database for data-intensive apps that require high performance and low latency, addresses these shortcomings by taking a fundamentally different architectural approach than Cassandra. Explanation follows.

HARDWARE UTILIZATION

The NoSQL revolution in database management systems kicked off over a decade ago. Since then, organizations of all sizes have benefitted from a key feature that the NoSQL architecture introduced: massive scale using relatively inexpensive commodity hardware. Thanks to this innovation, organizations have been able to deploy architectures that would have been prohibitively expensive and impossible to scale with traditional relational database systems.

Over the same decade, 'commodity hardware' itself has undergone a transformation. However, most modern software doesn't take advantage of modern computing resources. Most frameworks that scale out for data-intensive applications don't scale up. They aren't able to take advantage of the resources offered by large nodes, such as the added CPU, memory, and solid-state drives (SSDs), nor can they store large amounts of data on disk efficiently.

Because of its inability to take full advantage of high density multicore multi-CPU architectures, Cassandra is limited to the density of storage it can manage. A rule of thumb for many Cassanda and Cassandra-like databases such as DataStax Enterprise is to keep data capacity requirements at or below 1-2 terabytes per server. Compare that to modern cloud architecture, such as the AWS EC2 I3en "meganode" which can scale to 60 terabytes of storage per server.

Cassandra's architecture prevents it from efficiently exploiting modern computing resources—in particular, multi-core CPUs. Even as cloud platforms offer bigger and bigger machine instances with massive amounts of memory and ever denser storage options, Cassandra is constrained. It's constrained by many factors, primarily its use of Java, but also its caching model and its lack of an asynchronous architecture. As a result, Cassandra is often deployed on clusters of small instances that run at low levels of utilization. This low hardware utilization rate results in system sprawl, which equates to operational overhead, with a far larger footprint to keep managed and secure. All of this directly impacts staffing and infrastructure spend and ROI.

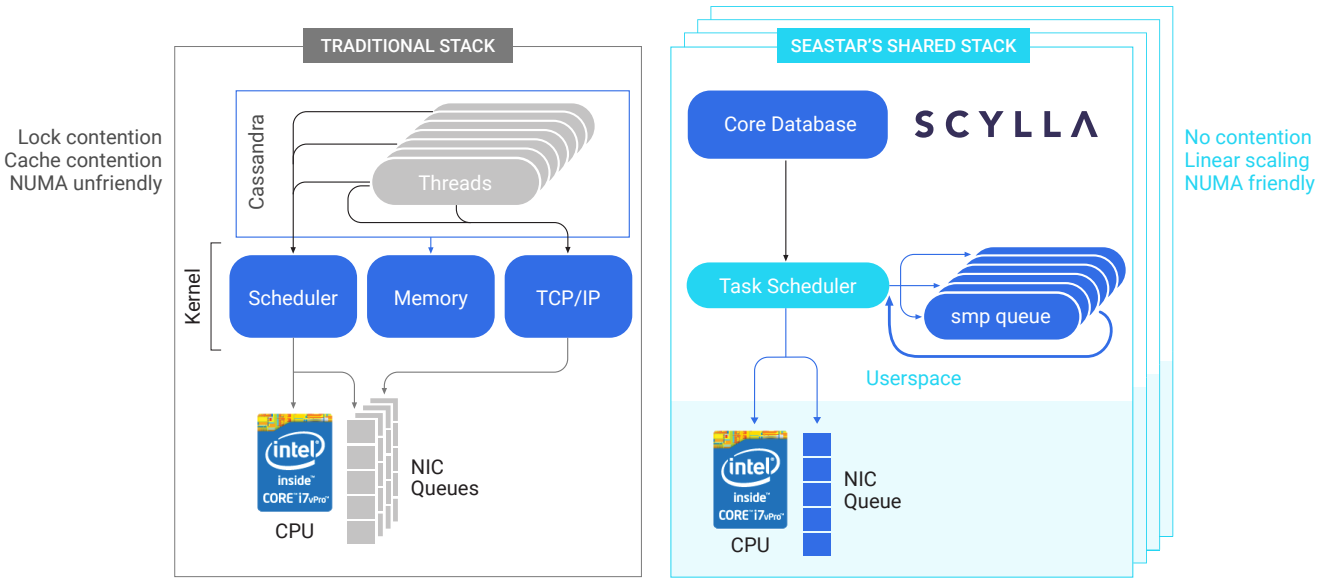


Figure 1: Cassandra's traditional design vs. ScyllaDB's modern design

ScyllaDB set out to fix these deficiencies with its close-to-the-hardware database design. Written in C++ instead of Java, and optimized to make full utilization of the Linux operating systems found ubiquitously across modern cloud environments, ScyllaDB implements a number of low-level architectural techniques, such as thread-per-core, async everywhere, shared-nothing, and automatic sharding. Those techniques enable ScyllaDB to scale linearly with the available resources. This means ScyllaDB can run massive workloads on smaller clusters of larger, denser nodes – an approach that vastly reduces all of the inputs to total cost of ownership (TCO).

A few common concerns are that large nodes won't be fully utilized, that they have a hard time streaming data when scaling out and, finally, that they might have a catastrophic effect on recovery times. ScyllaDB breaks these assumptions, resulting in improved TCO and reduced maintenance.

In light of these concerns, we ran real-world scenarios against ScyllaDB to demonstrate that the skepticism towards big nodes is misplaced. In fact, with ScyllaDB, big nodes are often best for data-intensive applications.

BALANCING BANDWIDTH, DISK I/O, AND CPU

Databases have long struggled with a conundrum: exponential growth in storage volume has been accompanied by linear growth in disk I/O bandwidth. As infrastructure has become capable of storing ever more data, the associated networks (internal to the chips) have struggled to keep pace. The results are unpredictable performance and bottlenecks.

In any distributed database, many actors compete for available disk I/O bandwidth. Cassandra controls I/O submission by statically capping background operations such as compaction, streaming, and repair. Getting that cap right requires painstaking, trial-and-error tuning combined with expert knowledge of specific database internals. Spiky workloads, with wide variance between reads and writes, or unpredictable end-user demand, are risky. Set the cap too high and foreground operations will be starved of bandwidth, creating erratic latency and bad customer experiences. Set it too low and it might take days to stream data between nodes, making auto-scaling a nightmare. What works today may be disastrous tomorrow.

ScyllaDB is designed to autonomously balance the two ideals of simultaneously maintaining system stability while preserving customer-facing Service Level Agreements (SLAs). ScyllaDB tackles this challenge by managing requests inside the database itself.

Most databases delegate requests to lower-level kernel space queues. Queueing requests in user space does not improve latency by itself; it merely swaps one queue for another. However, by queuing requests in user space, ScyllaDB gains control over the processing of I/O requests. For instance, ScyllaDB can:

- Provide metrics that enable application-level throttling
- Prioritize requests and process them selectively
- Cancel requests before they impact a lower layer's software stack.

Figure 2 shows this architecture at a high level. In ScyllaDB, requests bypass the kernel for processing and are sent directly to ScyllaDB's user space disk I/O scheduler. The I/O scheduler applies rich processing to simultaneously maintain system stability and meet SLAs.

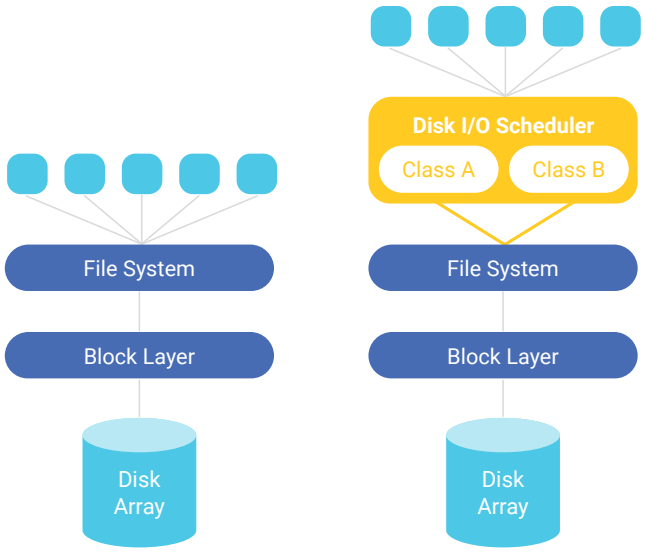


Figure 2: ScyllaDB's I/O scheduler vs traditional database request processing

On the left, requests generated by the userspace process are thrown directly into the kernel and lower layers. On the right, ScyllaDB's disk I/O scheduler intermediates requests. ScyllaDB classifies requests into semantically meaningful classes (A and B), then tracks and prioritizes them – guaranteeing balance while ensuring that lower layers are never overloaded.

CACHING

Linux caching is inefficient for database implementations. The Linux page cache, also called disk cache, improves operating system performance by storing page-size chunks of files in memory to save on expensive disk seeks. The Linux kernel treats files as 4KB chunks by default. This speeds up performance, but only when data is 4KB or larger. The problem is that many common database operations involve data smaller than 4KB. In those cases, Linux's 4KB minimum leads to high read amplification. Adding to the problem, the extra data is rarely useful for subsequent queries (since it usually has very poor 'spatial locality'). For most cases, it's just wasted bandwidth.

Cassandra attempts to alleviate read amplification by adding a key cache and a row cache, which directly store frequently used objects. However, Cassandra's extra caches increase overall complexity and are very difficult to configure properly. The operator allocates memory to each cache; different ratios produce varying performance characteristics. Different workloads benefit from different settings. The operator also has to decide how much memory to allocate to the JVM's heap as well as the offheap memory structures. Since the allocations are performed at boot time, it's practically impossible to get it right – especially for dynamic workloads that can change dramatically over time.

Moreover, there is another problem. Under the hood, the Linux page cache also performs synchronous blocking operations, which decrease the performance and predictability of the system. Since Cassandra is unaware that a requested object does not reside in the

Linux page cache, accesses to non-resident pages will cause Linux to issue a page fault and context switch to read from disk. Then it will context switch again to run another thread. The original thread is paused and its locks are held. Eventually, when the disk data is ready (yet another interrupt context switch), the kernel will schedule in the original thread.

Figure 3 displays the architecture of Cassandra's caches, with layered key, row, and underlying Linux page caches.

The architects who designed ScyllaDB recognized that a special-purpose cache, intrinsic to the database, would deliver better performance than Linux's default cache. A unified cache can dynamically tune itself to any workload and obviates the need to manually tune multiple different caches as one is forced to do with Apache Cassandra. Since ScyllaDB caches the actual objects, it always controls their eviction and memory footprint.

More importantly though, ScyllaDB can dynamically balance the different types of caches stored. ScyllaDB does this using a set of controllers, including a memtable controller, compaction controls, and a cache controller, which enable it to dynamically adjust their sizes. Once data is no longer cached in memory, ScyllaDB will generate a continuation task to read the data asynchronously from the disk using direct memory access (DMA), which allows hardware subsystems to access main system memory independent of CPU.

The C++ Seastar framework on which ScyllaDB is built will execute the continuation task in a µsec (1 million tasks/core/sec) and will rush to run the next task. There's no blocking, heavyweight context switch, waste, or tuning. For ScyllaDB users, this design means higher ratios of (cheap) disk to (expensive) RAM.

This cache design enables each ScyllaDB node to serve more data, which in turn lets operators run smaller clusters of more powerful nodes with larger disks. ScyllaDB's unified cache also simplifies operations, since it eliminates multiple competing caches and dynamically tunes itself at runtime to accommodate varying workloads.

Finally, because ScyllaDB has a very efficient internal cache, it removes the need for a separate external cache, making for a more efficient, reliable, secure, and cost-effective unified solution. There are also times when you want to make a query that avoids hitting the cache at all. For instance, if you anticipate making a broad ad hoc query, you might wish to avoid the cache, read directly from disk, and thus avoid having any returned results needlessly take up space in the cache. To support this functionality, ScyllaDB allows filtering results to bypass the cache.

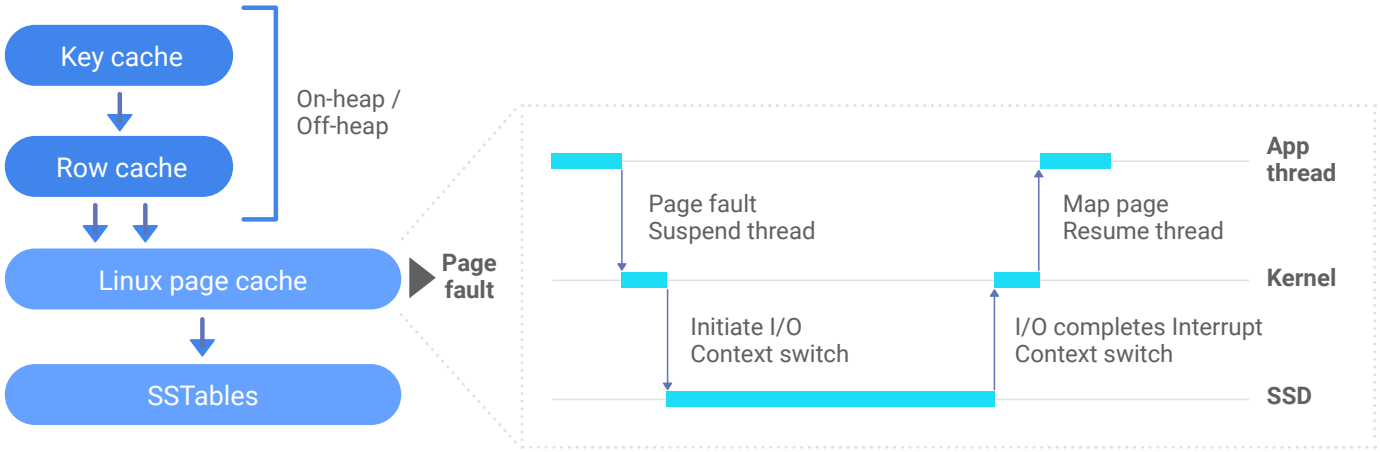


Figure 3: A diagrammatic view of the Linux page cache used by Apache Cassandra

JAVA IMPLEMENTATION

Cassandra is implemented in Java. Java isn't well-suited for I/O and compute-intensive workloads because it deprives developers of control. A modern database requires the ability to use large amounts of memory and have precise control over what the server is doing at any time. Java isn't well suited to either of these requirements.

Further, a database written in Java is unable to fully optimize low-level operations against the available hardware. Cassandra's reliance on the Java Virtual Machine (JVM) makes it susceptible to performance and latency issues caused by garbage collection. Cassandra users can side-step garbage collection by using off-heap data structures, but that fragments memory and ultimately defeats the purpose of managed memory entirely.

In contrast, C++ serves I/O and compute-intensive workloads well. It provides very precise control over everything a database does, along with abstractions that enable database developers to create code that's both complex and manageable.

One of the early and fundamental design decisions behind ScyllaDB was using C++ instead of Java. ScyllaDB is built on an advanced, open source C++ framework for high-performance server applications on modern hardware. Unlike Java, C++ can be considered an infrastructure programming language. It runs as native executable machine code and gives developers complete control over low-level operations. By eliminating Java, it eliminates the problems associated with garbage collection altogether.

PERFORMANCE

Of course, all of these architectural differences impact performance... but to what extent?

To measure it, ScyllaDB engineers tested the latencies and throughputs measured for various workloads, as well as the speed of common administrative operations such as expanding clusters and running major compactions. They measured performance with respect to:

- Throughput and latency with various distributions of data
- Adding a new node
- Doubling the cluster size
- Replacing a single node
- Performing a major compaction

ScyllaDB consistently and significantly outperformed both Apache Cassandra 3.11 and Cassandra 4.0.

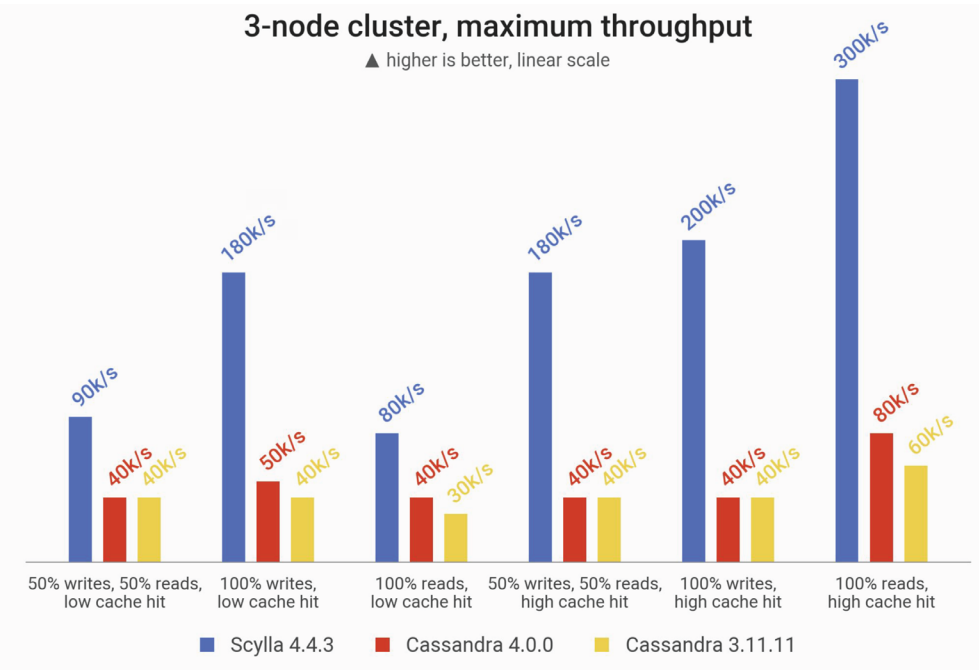


Figure 4: The maximum throughput (measured in operations per second) achieved on 3 x i3.4xlarge machines (48 vCPUs)

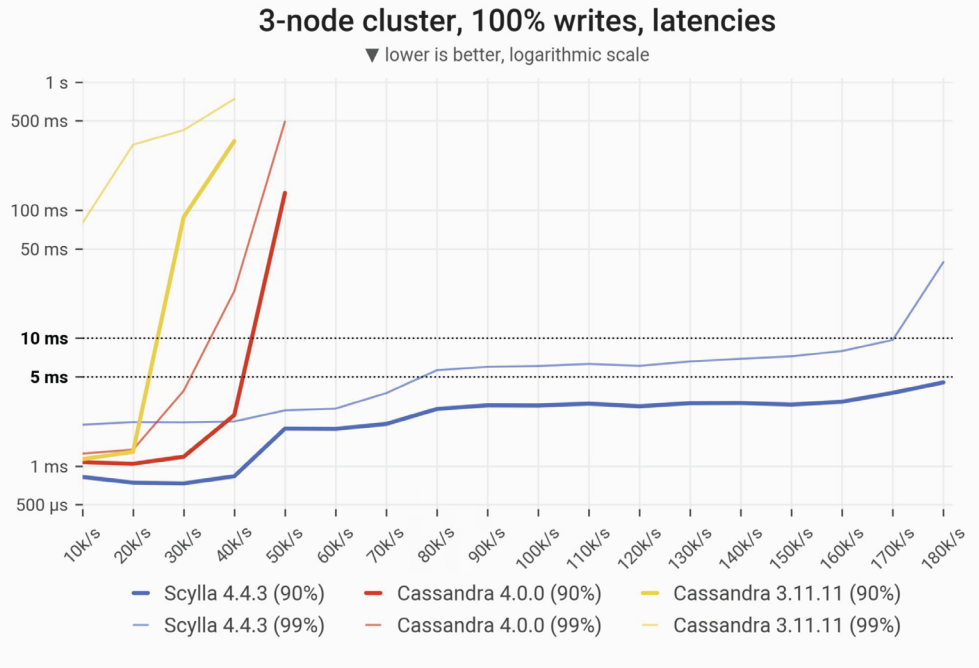


Figure 5: The 90- and 99-percentile latencies of UPDATE queries, as measured on three i3.4xlarge machines (48 vCPUs in total) in a range of load rates

On identical hardware, ScyllaDB withstood up to 5x greater traffic and offered lower latencies than Apache Cassandra 4.0 in almost every tested scenario. ScyllaDB also completed admin tasks 2.5 to 4 times faster than Cassandra 4.0.

To evaluate how well ScyllaDB can utilize very large nodes, they also compared a setup of 4 i3.metal machines (288 vCPUs in total) vs. 40(!) i3.4xlarge Cassandra machines (640 vCPUs in total — almost 2.5x ScyllaDB's resources). This test showed that a ScyllaDB cluster can be 10x smaller in node count and run on a cluster 2.5x less expensive, yet maintain the equivalent performance of Cassandra 4.0.

GET BENCHMARK DETAILS HERE

CONCLUSION

IT organizations of all sizes have embraced NoSQL in general and Apache Cassandra in particular based on the promise of greater flexibility and cost-effective scale. As a first-generation NoSQL solution, Apache Cassandra delivered on that early promise of NoSQL. Over time, however, it has become clear that limitations in the fundamental architecture of Cassandra render it unable to fully leverage the computing resources in the modern datacenter.

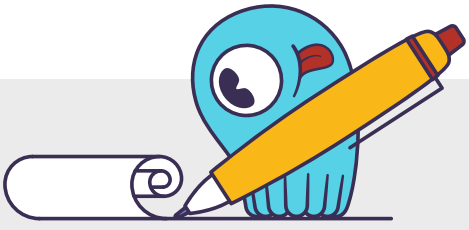
Organizations that adopted Cassandra now struggle with costs, maintenance, and administrative overhead. This is due to a number of reasons, from node sprawl due to poor hardware utilization, throughput bottlenecks, inconsistent and high latencies, complex performance tuning, and inefficient memory management. Ultimately, Cassandra has proven to be too expensive and too problematic for many projects.

Cassandra was revolutionary when it first debuted in 2008, leading to its broad adoption. However, more than a decade later, many companies have recognized its underlying limitations and have now moved on. Leading companies such as Discord, Comcast, Fanatics, Expedia, Samsung, and Rakuten have replaced Cassandra with ScyllaDB. ScyllaDB delivers on the original vision of NoSQL — without the architectural downsides associated with Apache Cassandra (or the costs at volume of databases like Amazon DynamoDB). ScyllaDB is built with deep knowledge of the underlying Linux operating system and architectural advancements that enable consistently high performance at extreme scale.

If your team is thinking of replacing Cassandra, our experts can help you assess whether your use case and requirements might be better supported by ScyllaDB.

ABOUT SCYLLADB

ScyllaDB is the database for data-intensive apps that require high performance and low latency. It enables teams to harness the ever-increasing computing power of modern infrastructures – eliminating barriers to scale as data grows. Unlike any other database, ScyllaDB is built with deep architectural advancements that enable exceptional end-user experiences at radically lower costs. Over 400 game-changing companies like Disney+ Hotstar, Expedia, FireEye, Discord, Crypto.com, Zillow, Starbucks, Comcast, and Samsung use ScyllaDB for their toughest database challenges. ScyllaDB is available as free open source software, a fully-supported enterprise product, and a fully managed service on multiple cloud providers. For more information: ScyllaDB.com



SCYLLA

United States Headquarters
2445 Faber Place, Suite 200
Palo Alto, CA 94303 U.S.A.
Email: info@scylladb.com

Israel Headquarters
11 Galgalei Haplada
Herzelia, Israel