SCYLLA.

# Building the Real-Time Big Data Database: Seven Design Principles behind Scylla

An overview of the close-to-the-hardware design of the Scylla NoSQL database

## CONTENTS

## ABSTRACT

Apache Cassandra, an early entrant in the NoSQL arena, has become a popular database for projects that benefit from non-relational data. Cassandra provides IT organizations with an easy way to scale data across multiple nodes and datacenters with fault tolerance and data redundancy. Early deployments were successful, but over time limitations emerged. In practice, Cassandra has proven to be expensive and risky, due to several key limitations of the underlying architecture.

Scylla was inspired by the key issues that have hindered Cassandra: Node sprawl, inconsistent performance and arduous administration requirements. In part, this is because database technology has not kept pace with innovations in hardware over the last decade. While hardware has grown exponentially faster, neither traditional SQL nor newer NoSQL databases are able to take full advantage of the additional power available to them. The results are wasteful overprovisioning and node sprawl, with high administrative and maintenance costs.

The ScyllaDB team has deep roots in low-level kernel programming, having devised and developed the KVM hypervisor, which now powers most public clouds, including Google, AWS, OpenStack and many others. KVM was a late-entry re-engineered technology that displaced mature technologies like Xen open source and VMware. While trying to extract performance from a Cassandra cluster for a different project, the ScyllaDB team was surprised to discover that neither Apache Cassandra nor any other databases on the market was able to translate the full power of the underlying hardware into user-visible performanc—in particular modern, multi-core CPUs and fast I/O devices. Soon thereafter, the team chose to apply its expertise to another re-engineering effort, this time a drop-in replacement for Apache Cassandra. Their goal was a NoSQL database with scale-up performance of 1,000,000 IOPS, scale-out to hundreds of nodes and 99% latency of less than 1 millisecond—without sacrificing any of the rich functionality, tooling and ecosystem support of Cassandra.

In this paper, we will explain the key design decisions that paved the way to a database that is ideally suited to real-time big data use cases.

## INTRODUCTION

The NoSQL movement was created in response to the requirements of cloud-based, internet- and web-scaled companies. The SQL databases available at the time were not suited to the emerging use cases, which called for managing different types of data structures, as well as high performance and high availability across globally distributed data centers.

Drawing on key concepts behind Amazon DynamoDB, a team at Facebook created Cassandra. The designers originally described it as “a distributed storage system for managing structured data that is designed to scale to a very large size across many commodity servers, with no single point of failure.” Released as an open-source project on Google code in July 2008, by 2010 Cassandra was promoted to a top-level Apache project.

Apache Cassandra has experienced widespread adoption. Its popularity was primarily based on its excellent cluster architecture, which provides the following capabilities:

- **Masterless Replication:** Cassandra clusters have no masters, slaves, or elected leaders. All nodes are symmetric and can serve all reads and writes equally, with no single point of failure.
- **Global Distribution:** Data sets can be replicated across multiple data centers, multiple geographical regions, and across public and private clouds.
- **Linear Scale:** Performance scales along with the number of nodes. Users can increase performance by adding new nodes, without downtime or disruption to running applications.

- **Tunable Data Consistency:** The consistency levels for read and write is tunable and so is the amount of replicas to maximize price/performance.
- **Data Model:** Cassandra provides a simple data model that supports dynamic control over data layout and format.

Cassandra's excellent clustering architecture and data model, however, are undermined by fundamental problems in its node architecture. As a result of these limitations, users of Cassandra have struggled with the following issues:

- **Team Intensive:** Operating Cassandra at scale requires dedicated full-time experts with an increasingly scarce and expensive skill-set.
- **JVM Challenges:** Memory management in the Java virtual machine (JVM) produces unpredictable and unbounded latency.
- **Inefficient Utilization:** Cassandra cannot efficiently exploit modern computing resources, in particular multi-core CPUs.
- **Manual Tuning:** Clusters require operators to perform intricate yet unpredictable tuning procedures, while combating compactions and garbage collection storms.
- **Dev Tweaking:** Application developers require knowledge of database internals to successfully scale applications.

As a result of these issues, architects and developers using Cassandra are forced to choose between availability, simplicity and performance. Some deploy a Redis cache in front of Cassandra and lose simplicity and consistency. Some do not use the full Cassandra feature set due to complexity and stability. Others give up and triple their TCO by going to cloud managed solutions like DynamoDB.

The primary objective behind Scylla is to eliminate this compromise. As an open source alternative to Apache Cassandra, Scylla preserves everything that the Apache community loves about Cassandra, while also delivering higher throughput, consistently low latencies, operational simplicity, and lower TCO.

In this paper, we share the seven fundamental design decisions we made when architecting a database that's built from the ground up to support real-time big data workloads, and the results those decisions have had on the Scylla database.

## DESIGN DECISION #1: C++ INSTEAD OF JAVA

One of the early and fundamental design decisions behind Scylla is the use of C++ in place of the Java programming language. The Java programming language platform 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. C++ serves both purposes 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.

Further, a database written in Java, like Cassandra, 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++ can be considered an infrastructure programming language. It runs as native executable machine code and gives developers complete control over low-level operations. Scylla is built on an advanced, open-source C++ framework for high-performance server applications on modern hardware. One example of the way Scylla improves upon Cassandra by using C++ is its kernel-level API enhancement. Such an enhancement would be impossible in Java.

Scylla developers regularly check the assembly generated code to verify efficiency metrics and to uncover potential optimizations down to the microarchitectural level, as you can see in our published instruction-per-cycle research. By deciding to build in C++, we eliminated the problems associated with garbage collection altogether.

*"When we heard of a Cassandra drop-in-replacement we were skeptics. But very quickly we found it is all true– not only were the latency and GC issues completely gone, better hardware utilization allowed us to shrink the cluster size by half!"*

Gabriel Mizrahi, CTO, *Investing.com*.

Read more about his experience in the case study.

## DESIGN DECISION #2: CASSANDRA COMPATIBILITY

With more than a decade of technology investment, the Cassandra user community isn't eager to learn a new database or data model. Yet they would benefit from a drop-in alternative that overcomes the performance and latency issues that have often put their projects at risk. This takes advantage of the tremendous value in the accumulated body of institutional knowledge and expertise generated by the Cassandra community. For these reasons, we chose to leverage the existing work in drivers, query language, and the ecosystem surrounding Cassandra, rather than building a new suite of drivers or inventing yet another query language.

Scylla's Cassandra compatibility encompasses :

- **Wire Protocol:** Scylla supports Thrift, CQL, and the full polyglot of languages. We strive to be fully compliant with CQL and support all functionality, from lists, maps, to counters, UDTs, secondary indexes and, soon, lightweight transactions (LWT).
- **Monitoring:** Scylla supports the JMX protocol. We add a JVM-proxy daemon that transparently translated JMX into our RESTful API. Common Cassandra dashboards can retrieve the same information from Scylla. Scylla provides additional monitoring in the form of the Prometheus API and direct support for REST.
- **Underlying File Format and Algorithms:** Scylla adopted Cassandra's SSTable format and supports all Cassandra compaction strategies. Auxiliary tools for SSTable loading, backup and restore, and repair are also supported.
- **Configuration File:** Scylla is able to directly consume Cassandra's configuration file, `cassandra.yaml`, allowing for a seamless migration path. Scylla ignores JVM-related options, as well as various other limitations that derive from Cassandra (such as parallel compactors configuration). Scylla always uses maximum parallelism.
- **Command Line Interface:** Scylla uses the same command line tool (`nodetool`). Its processes —from backup to repair —are identical to Cassandra's.

As a result of this fundamental decision, organizations using Cassandra today can seamlessly migrate to Scylla without impacting existing applications. They can also continue to leverage their existing ecosystems. As Scylla embraces and extends Cassandra, we guarantee ongoing compatibility with Cassandra.

*"Scylla was the natural choice for us since it's fully compatible with Cassandra. We realized we could just put it where Cassandra was and everything would keep working. We started our transition to Scylla without rewriting any of our stack."*

- David Haguenauer, Software Engineer, *AdGear*.

Read more about his experience in this case study.

## DESIGN DECISION #3: ALL THINGS ASYNC

Equipped with multi-core processors, modern hardware is capable of performing millions of I/O operations per second (IOPS). To take advantage of this speed, software needs to be asynchronous, to drive both I/O and CPU processing in a way that scales linearly with the number of CPU cores.

In any database, many different operations execute simultaneously. Multiple queries executing on different cores may be waiting for data from other cores, or even from the network or storage media. It is already common practice not to wait for slow devices such as HDD disks through the use of thread pools and other similar architectures. Yet as storage technology improves, the cost of dispatching one I/O operation approaches the cost of a thread context switch. As the common core count increases, context switches can also appear as a result of locking. In order to scale with the core count and extract maximum performance from newer, faster storage technology, we decided not to synchronously wait for either I/O completion or neighbouring CPUs, even for nanoseconds.

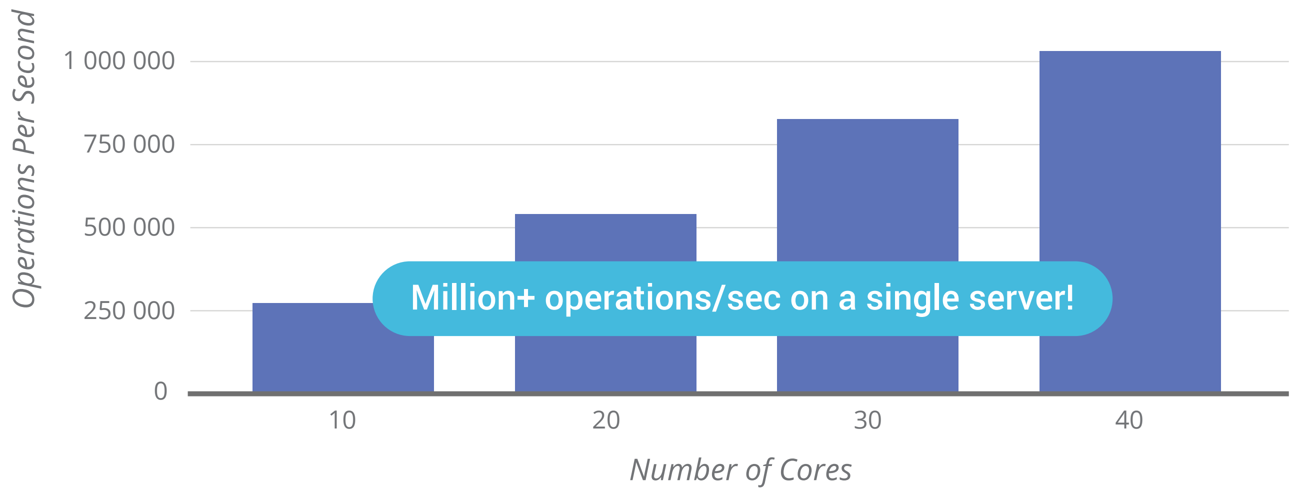
To accommodate this reality, we made the fundamental decision to adopt a completely asynchronous architecture. While building an asynchronous framework meant more upfront work for our team, once it was complete, all of the traditional problems associated with concurrency were eliminated. By taking this approach, we implemented a system in which the number of concurrent queries is constrained only by system resources, not by the framework itself. In other words, the framework drives both I/O and CPU processing, enabling it to scale linearly with the number of cores available.

Scylla relies on a dedicated asynchronous engine (known as Seastar.) Here you can read how Scylla accesses the disk with async and direct memory access (DMA) using the Linux API.

## DESIGN DECISION #4: SHARD PER CORE

Moore's law was upheld for decades through increasingly higher CPU clock cycles.
As frequency limits were reached, chip manufacturers began experimenting with multi-core CPUs in the late 90's. Since the typical programming model involves many threads, increasing cores-per-CPU virtually guarantees scalability problems. A threaded programming model lets the application focus on the problem at hand, relying on locks to ensure exclusive access the data. Other threads, however, are prevented from accessing the data and, as a result, they block. Even a cheap

*Figure 1: Scale Linearly: Scylla and Qualcomm Centriq™ 2400 – Operations per Second vs. Number of Cores*



locking technique, such as busy waiting, must lock the CPU bus and invalidate caches, adding overhead even when the lock is not contended. Application-level locks are even more expensive, since they cause the contended thread to sleep and issue context switches. As the core-count grows, the chances of hitting the contended case grow as well, thereby limiting the scalability of threaded applications.

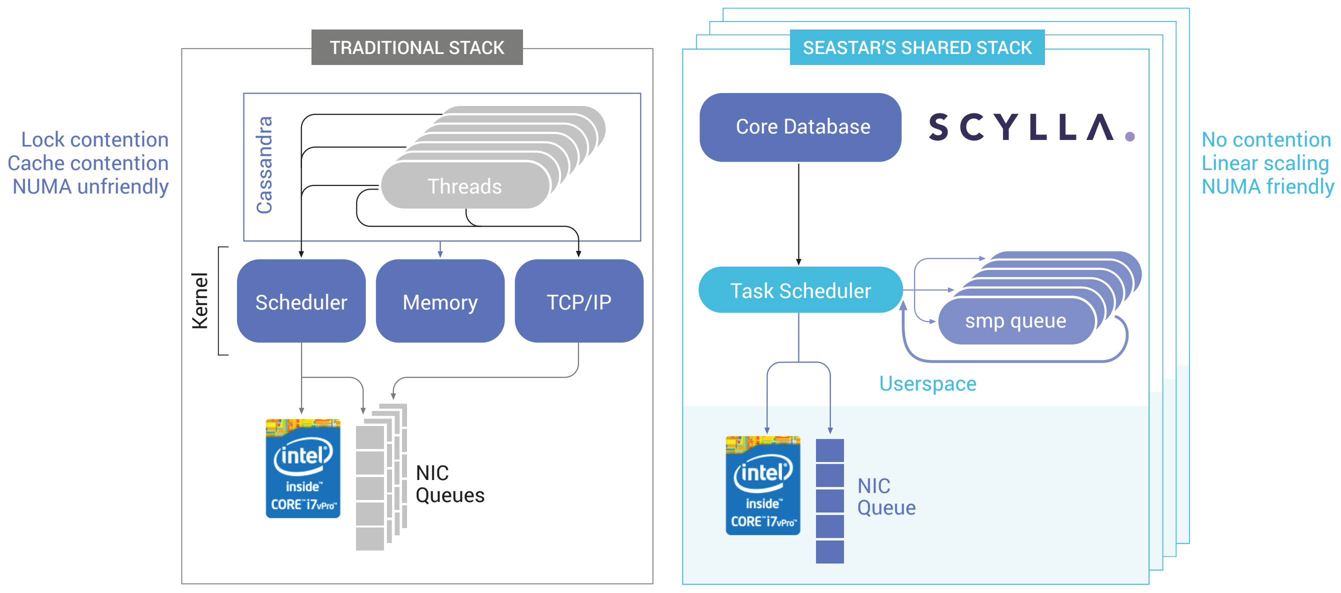
The complexity of the traditional threaded model goes beyond locking. Where more than a single socket is involved, memory access may spread across multiple sockets. Access to a memory bank with a remote socket imposes twice the cost as access to a local socket. Threaded applications are usually agnostic about memory location, and can be moved around to different CPUs in different sockets. This doubles response times for applications that are not None Uniform Memory Access (NUMA) friendly.

Finally, there are the I/O devices. Modern NICs and storage devices have multi-queue modes where every CPU can drive I/O. The standard model typically lacks enough handlers or, being threaded by nature, will require context switches to service I/O.

Scylla addresses these scalability issues using a shared-nothing architecture. There are two levels of sharding in Scylla. The first, identical to Cassandra, the entire dataset of the cluster is sharded into individual nodes. The second level, transparent to users, is within each node. The node's token range is automatically sharded across available CPU cores. (More accurately, datasets are sharded across hyperthreads). Each shard-per-core is an independent unit, fully responsible for its own dataset. A shard has a thread of execution, a single OS-level thread that runs on that core and a subset of RAM. The thread and the RAM are pinned in a NUMA-friendly way to a specific CPU core and its matching local socket memory. Since a shard is the only entity that accesses its data structures, no locks are required and the entire execution path is lock-free. In the journey towards pure shared-nothing and lock-free operation, our engineering team developed its own memory allocation (malloc) library so each thread will have its own pool with no hidden OS-level locking. Even exception handling, derived from the GCC compiler, was optimized since the GCC library used to acquire spin-lock created contentions during exceptional paths

Each shard issues its own I/O, either to the disk or to the NIC directly. Administrative tasks such as compaction, repair and streaming are also managed independently by each shard.

*Figure 2: ScyllaDB Network Comparison*



In Scylla, shards communicate using shared-memory queues. Requests that need to retrieve data from several shards are first parsed by the receiving shard and then distributed to the target shard in a scatter/gather fashion. Each shard performs its own computation, with no locking and therefore no contention.

Each Scylla shard runs a single OS-level thread, leveraging an internal task scheduler to allow the shards to perform a range of different tasks, such as network exchange, disk I/O, compaction, as well as foreground tasks such as reads and writes. Scylla's task scheduler selects from low-overhead lambda functions, which we refer to as *continuations*. By taking this approach, both the overhead of switching tasks and the memory footprint are reduced, enabling each CPU core to execute a million continuation tasks per second.

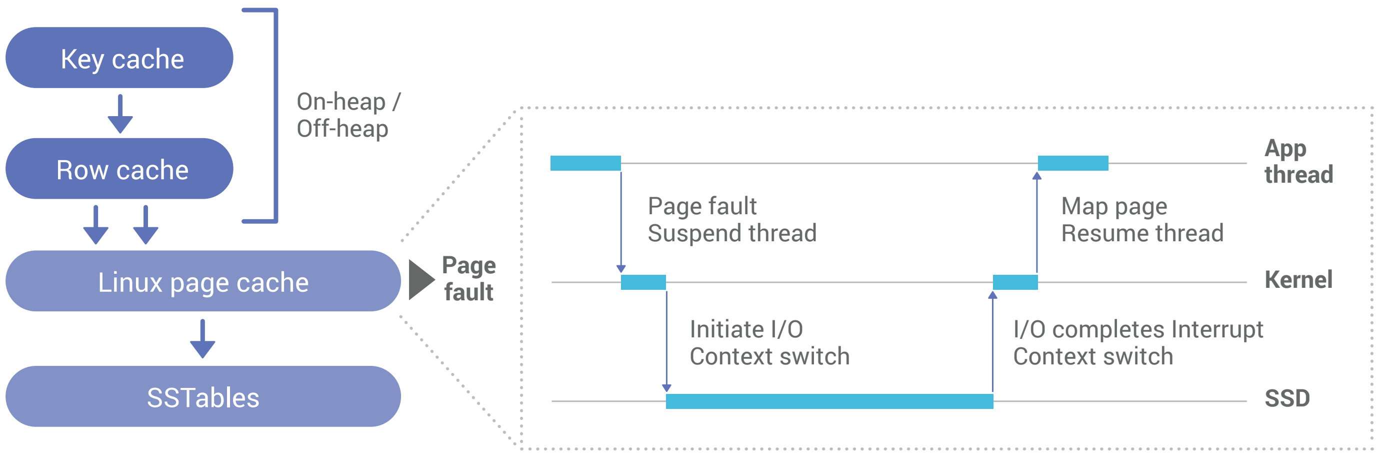
## DESIGN DECISION #5: UNIFIED CACHE

The page cache, sometimes also called disk cache, improves operating system performance by storing page-size chunks of files in memory, saving expensive disk seeks. In Linux, by default the kernel treats files as 4KB chunks. This speeds performance, unless data is smaller than 4KB, as is the case with many common database operations. In those cases, a 4KB minimum leads to high read amplification. Having very poor spatial locality, that extra data is rarely useful for subsequent queries. It's just wasted bandwidth. Specifically, the Linux page cache is a general-purpose data structure. We recognized that a special-purpose cache would deliver better performance.

To alleviate read amplification, Cassandra employs a key cache and a row cache, which directly store frequently-used objects. However, the added caches increase overall complexity. The operator allocates memory to each cache; different ratios produce varying performance characteristics, and each workload will benefit from a different setting. The operator also has to decide how much memory to allocate to the JVM's heap and its off-heap structures. Since the allocations are performed at boot time, it's practically impossible to get it right, especially for dynamic workloads.

The Linux page cache also performs synchronous blocking operations under the hood, decreasing 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: A diagrammatic view of the Linux Page Cache used by Apache Cassandra*



The diagram above displays the architecture of Cassandra's caches, with layered key, row, and underlying Linux page caches.

In light of this complexity, it was a straightforward decision to implement our own unified cache. A unified cache can dynamically tune itself to the current workload, and obviates the need to manually tune multiple different caches. Scylla caches objects itself, and thus always controls their eviction and memory footprint. Moreover, Scylla can dynamically balance the different types of caches stored. Scylla has controllers such as a memtable controller, compaction controls, and cache controller and can dynamically adjust their sizes. Once the data isn't cached in memory, Scylla 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. Seastar will execute it in a usec (1 million tasks/core/sec) and rush to run the next task. There's no blocking, heavyweight context switch, waste, or tuning.

For Scylla users, this design decision means higher ratios of disk to RAM. Each node can serve more data, enabling the use of smaller clusters with larger disks. The unified cache simplifies operations, since it eliminates multiple competing caches and is capable of dynamically tuning itself at runtime to accommodate varying workloads.

## DESIGN DECISION #6: I/O SCHEDULER

Database users want their data to be committed to disk as quickly as possible. For distributed databases, this requirement is nontrivial. I/O producers have to compete for bandwidth. If too much data is submitted at once, the data will be queued in the underlying device. The filesystem and the disk are ignorant of the content and purpose of the data, and they cannot judge whether the blocks originated from latency sensitive, real-time workloads or from batch background tasks.

It's difficult to reach a compromise between low latencies for sensitive tasks while maximizing throughput and preserving SLAs. Cassandra controls I/O submission by capping background operations, such as compaction, streaming, and repair. Cassandra users are required to carefully tune it and obtain a detailed knowledge of database internals. With spiky workloads, getting it right is a daunting challenge. An inaccurate cap may cause spiky latency when the cap is too high, starving foreground operations of compute and I/O resources. Set the cap too low and streaming terabytes of data between nodes might take days—rendering autoscaling a nightmare.

The goal behind Scylla is to provide a database that finds this balance automatically and autonomously, without operator intervention; Scylla should maximize I/O but not overload the drives. This way Scylla has full control of how to prioritize the foreground operations over the background operations. Thus no tuning is required, query operation is alway minimized, Scylla maximizes disk throughput while idle time exists and streams gigabytes of data per second, without any limit.

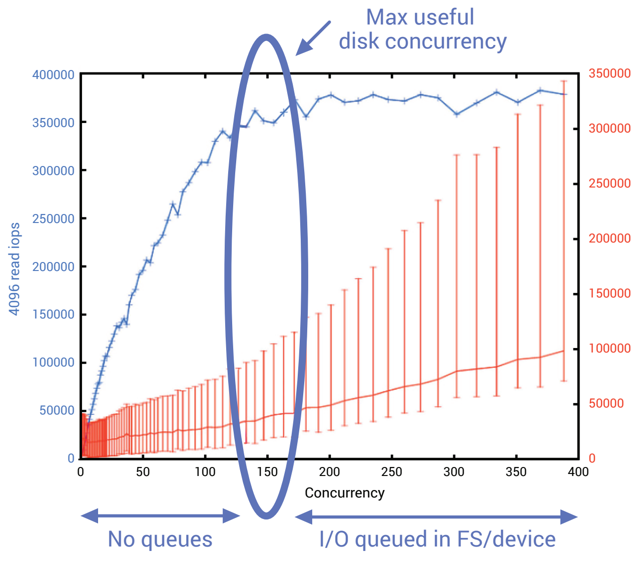


*Figure 4: Scylla's iotool benchmarks to ensure the optimal balance between foreground and background operations.*

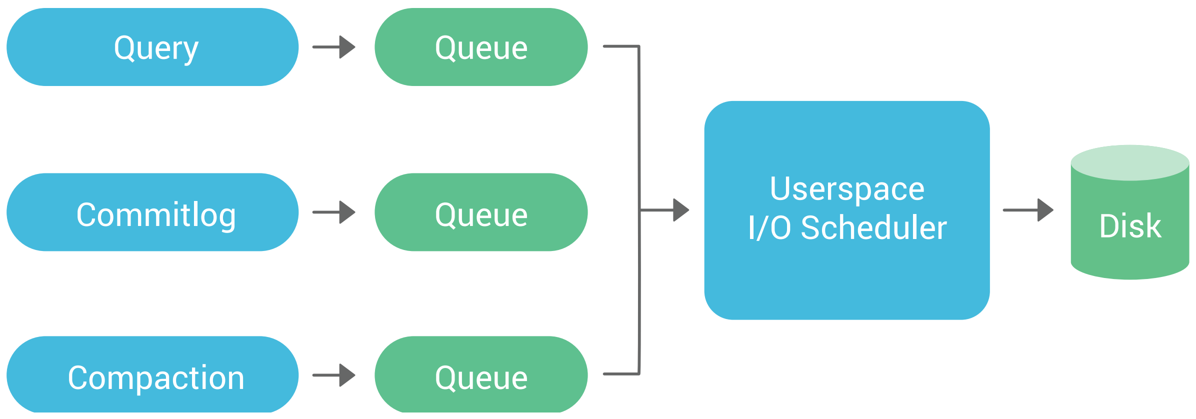


*Figure 5: Scylla's I/O Scheduler enforces priority across user-facing workloads and background tasks.*

When Scylla is installed, a tool called `scylla_io_setup` runs under the hood. `scylla_io_setup` is a benchmark that automatically determines the maximum useful disk concurrency, defined as the point at which maximum throughput is achieved while latency is good, and no data is queued by the disk or the file system.

To manage data on disk, both Cassandra and Scylla use an algorithm called Log Structured Merge Tree (LSM tree). LSM trees create immutable files during data insertion with sequential I/O (the "Log-Structured" part of LSM)—yielding great initial throughput. But that penalizes future reads, which may now have to deal with data for its queries coming from multiple SSTable files. To alleviate that, a background operation called compaction goes through the existing files and merges them (the "Merge" part in LSM) into fewer files. Issues arise when these background operations compete with user queries, reducing throughput in the process, which is, in general, an undesirable scenario.

To address this we saw the need to control all of the I/O going through the system. Our approach relies on a scheduler that allows types of requests—both foreground requests and background requests—to be tagged according to their the origin of the I/O operation. Once tagged as such, these requests can be metered and the scheduler can decide which priority should be applied to each class of operations.

The I/O scheduler guarantees that the disk will always be in its sweet spot and thus latency remains low while bandwidth is maximized.

This design decision helps Scylla users meet SLAs while maintaining system stability. Operators spend less time tuning and can be confident that background operations will complete as quickly as possible without impacting performance.

Another example of a background operation is the commissioning of new nodes and decommissioning of old ones. For example, to commission or decommission nodes, an operator can simply instruct Scylla to perform the operation, without having to take the speed of the operation into consideration. The operation will simply run mediated by the I/O Scheduler at the fastest speed that won't impact system throughput.

## DESIGN DECISION #7: AUTONOMOUS CAPABILITIES

Developing a database with self-optimizing capabilities is an overarching design decision that informs many of the design principles that we have described to this point. Cassandra users have told us time and again that they waste significant time and energy not only tuning the database, but also dealing with the fallout of complex and tricky tuning mechanisms.

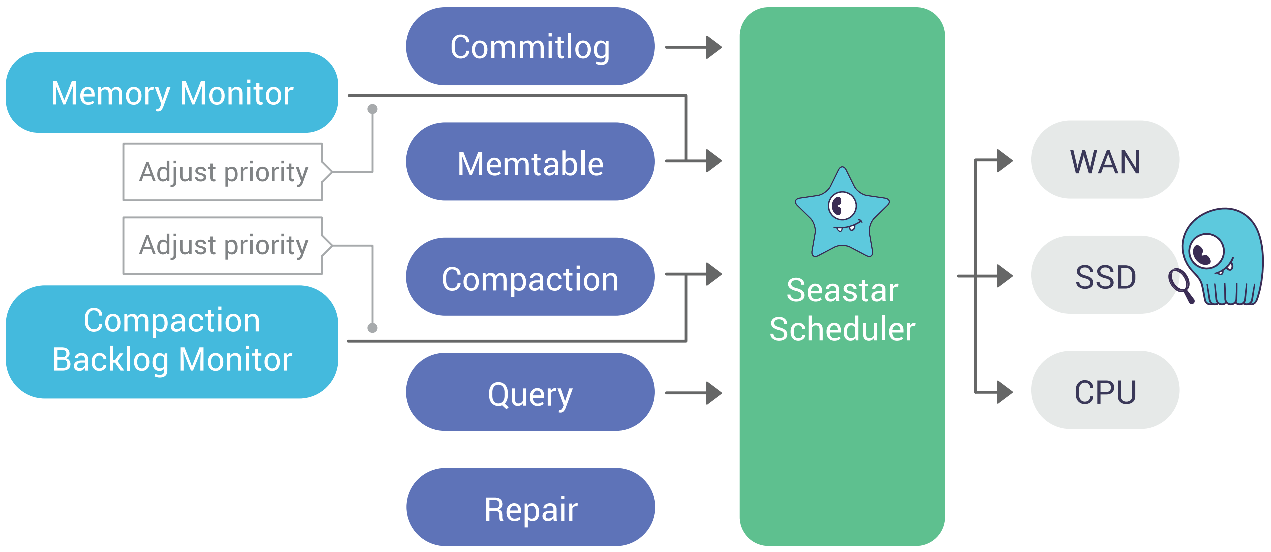


*Figure 6: A closer view of Scylla's Scheduler architecture. Scylla dynamically adjusts priority across tasks in usespace, enabling self-optimizing operations.*

Before Scylla, the only solutions available to this problem were to become an expert in Cassandra internals or to hire expensive consultants. To alleviate this burden and to help IT groups focus on more pressing issues, we committed to building a database that continually monitors all operations and dynamically adjusts internal processors to optimize performance.

Scylla takes advantage of control theory to achieve autonomous operations. Control theory is routinely employed in other fields like industrial plants and automotive cruise control systems. It works by setting a particular level for user-visible properties that the system has to maintain, leaving the tuning of the component parts that will yield the desired behavior up to the control algorithm. This is used in many parts of Scylla. It is present in background operations like compactions (where we make sure that the uncompacted backlog never grows out of control), in the caches (so we automatically move memory to where it is needed), and many other parts of the system.

This self-optimizing approach made much more sense to us than saddling users with elaborate guides and explanations of how to tune and re-tune for changing workloads and corner cases. Not only does an autonomous database greatly reduce administrative burden, it also means that operators can achieve 100% resource utilization while maintaining SLAs, and optimize infrastructure budgets, all at the same time.

*"Scylla hasn't failed or had any hiccups. To be honest, I've spent zero time trying to figure out what's going on. Scylla just works."*

- Ophir Horn, Head of Engineering at *TellusLabs*

> Read more about his experience in this case study.

## 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 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 administration due to high latencies, complex performance tuning, and inefficient memory management. Ultimately, Cassandra has proven to be too expensive and risky for many projects.

Scylla delivers on the original vision of NoSQL — without the downsides associated with Apache Cassandra. To achieve this goal, the ScyllaDB team had to rethink many of the foundational architectural choices behind Cassandra. The team behind Scylla applied their experiences with the hypervisor infrastructure underpinning many production cloud platforms. They relied on this practical experience with modern distributed systems to reinvent Apache Cassandra for the modern datacenter, finally releasing a NoSQL database capable of 1 million operations per second on a single node.

Today, Scylla is battle-hardened in production datacenters. Customers use it as a drop-in replacement for Cassandra or as a replacement for SQL and NoSQL databases such as MongoDB and as a replacement for the Redis cache. Production deployments have shown that Scylla significantly shrinks the datacenter hardware footprint compared to Apache Cassandra clusters, which often run on many small nodes. Scylla was designed from the ground-up to maximize available computing resources and to take advantage of modern multi-core 'commodity' hardware. In this way, Scylla delivers scale-out capabilities along with vastly lower management and administrative overhead compared to other distributed databases on the market today.

## RECOMMENDED READING

- Learn more about Scylla from our product page.
- See what our users are saying about ScyllaDB.
- Download Scylla. Check out our download page to run Scylla on AWS, install it locally in a Virtual Machine, or run it in Docker.
- Take Scylla for a Test Drive. Our Test Drive lets you quickly spin-up a running cluster of Scylla so you can see for yourself how it performs.

## ABOUT SCYLLADB

Scylla is the real-time big data database. Fully compatible with Apache Cassandra, Scylla embraces a shared-nothing approach that increases throughput and storage capacity as much as 10X that of Cassandra. AppNexus, Samsung, Mogujie, CERN, Grab, Olacabs, Investing.com, Eniro, IBM's Compose and many more leading companies have adopted Scylla to realize order-of-magnitude performance improvements and reduce hardware costs. ScyllaDB was founded by the team responsible for the KVM hypervisor and is backed by Bessemer Venture Partners, Innovation Endeavors, Wing Venture Capital, Qualcomm Ventures, TLV Partners, Magma Venture Partners, Western Digital Capital and Samsung Ventures.  For more information: ScyllaDB.com

SCYLLADB.COM

SCYLLA.

United States Headquarters
1900 Embarcadero Rd
Palo Alto, CA 94303 U.S.A.
Phone: +1 (650) 332-9582
Email: info@scylladb.com

Israel Headquarters
11 Galgalei Haplada
Herzelia, Israel