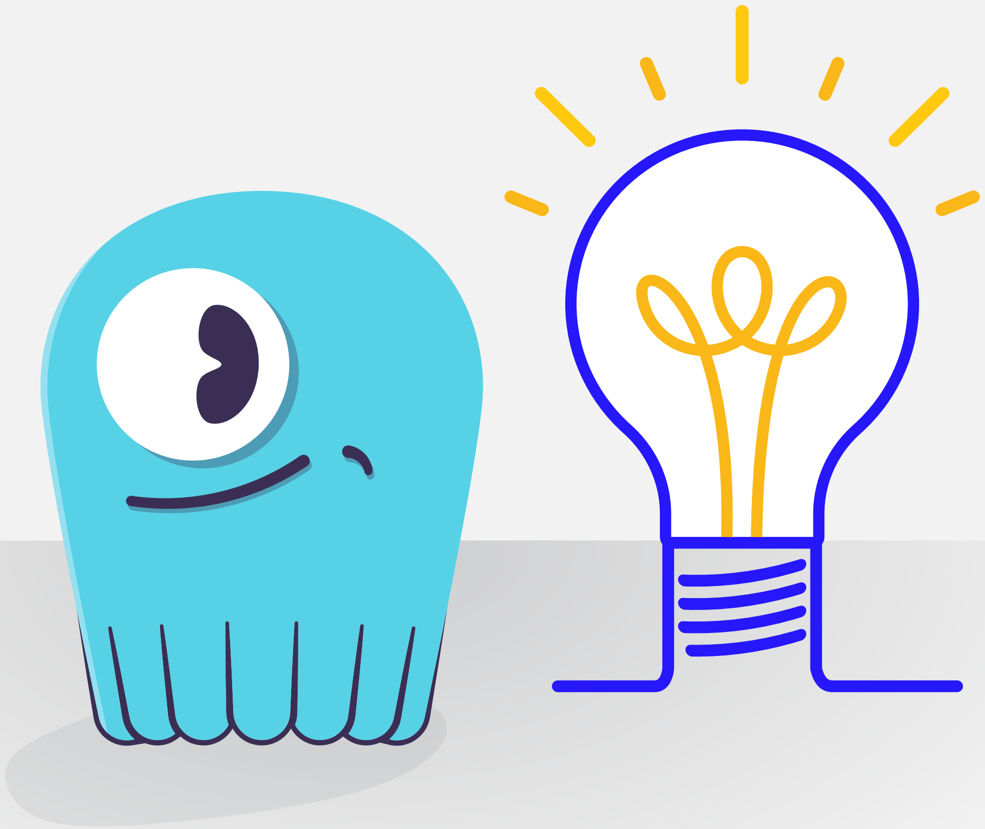
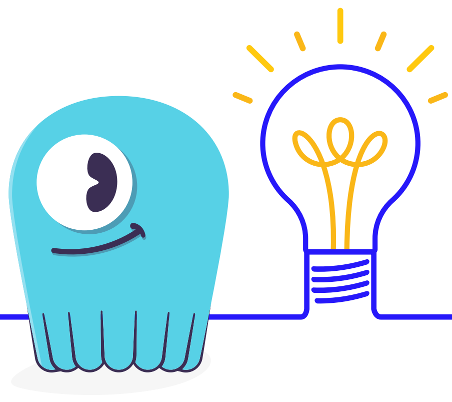


SCYLLADB WHITEPAPER

ScyllaDB

Beyond Legacy NoSQL: 8 Design Principles Behind ScyllaDB



CONTENTS

ScyllaDB is a NoSQL database designed for modern multiprocessor servers, providing high availability, highly elastic scalability and predictable performance at scale. It was designed as a direct alternative to legacy NoSQL and traditional SQL databases based on eight core design principles that guided its architecture and implementation.

In particular, it was designed as an alternative to Apache Cassandra, an early entrant in the NoSQL arena, which became a popular database for non-relational data projects. Cassandra provides IT organizations an easy way to scale data across multiple nodes and datacenters with fault tolerance and data redundancy. Further, developers like the SQL-like API and Cassandra's application-centric data model and implementation approach. Early deployments were successful, but over time, limitations emerged. In practice, Cassandra has proven to be expensive and problematic due to several key limitations of the underlying architecture, most notably its implementation in Java.

ScyllaDB was inspired to specifically address the key issues that hindered Cassandra: node sprawl, inconsistent performance and arduous administration requirements. In part, this is because database software architecture has not kept pace with innovations in hardware over the last decade. While hardware has grown exponentially faster and more sophisticated, 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 were able to translate the full power of the underlying hardware into user-visible performance—in particular modern, multi-core CPUs and fast I/O devices. Soon thereafter, the team chose to apply its expertise to another reengineering effort, this time an API-compatible replacement for Apache Cassandra. Their goal was a NoSQL database with scale-up performance of 1,000,000 IOPS per server with, 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 and CQL.

Since ScyllaDB's inception, the ScyllaDB architects realized that their core database engine can also be applied to other database designs and use cases. After achieving feature parity with Apache Cassandra, the team implemented an API compatible with Amazon DynamoDB, a popular (but costly) document store and key-value database-as-a-service (DBaaS). The same core principles that guided ScyllaDB to replicate the best of Apache Cassandra and yet improve upon it substantially will allow ScyllaDB to extend its capabilities and compatibilities well into the foreseeable future.

In this paper, we will explain the key design decisions that paved the way to a database that is ideally suited to low-latency data-intensive application use cases that require performance at scale.

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 large data volume throughput of different data structure types across globally distributed data centers with high availability SLAs.

Drawing on key concepts behind Amazon Dynamo, 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 project by the non-profit Apache Software Foundation.

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

- **Peer-to-peer architecture:** In Cassandra clusters, 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 and geographical regions, as well as 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, as is the number of replicas to maximize availability and price/ performance.
- **Data Model:** Cassandra provides a simple data model that supports dynamic control over data layout and format.

However, Cassandra's excellent clustering architecture and data model 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 and high-density storage servers
- **Manual Tuning:** Clusters require operators to perform intricate yet unpredictable tuning procedures, while combating compactions and garbage collection storms.
- **Dev Tweaking:** Application developers require database internals knowledge 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 its complexity and stability. Others give up and triple their TCO by going to cloud managed solutions like DynamoDB.

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

In this paper, we share the eight fundamental design decisions we made when architecting a database that's built from the ground up to provide performance at scale for low-latency data-intensive applications, and the results those decisions have had on the ScyllaDB database.

DESIGN DECISION #1: C++ INSTEAD OF JAVA

One of the early and fundamental design decisions behind ScyllaDB is the use of C++ in place of the Java programming language. The Java programming language platform isn't wellsuited 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 in modern cloud infrastructure. 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 this 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. ScyllaDB is built on an advanced, open source C++ framework for high-performance server applications on modern hardware. One example of the way ScyllaDB improves upon Cassandra by using C++ is its kernel-level API enhancement. Such an enhancement would be difficult at best and definitely non-idiomatic in Java.

ScyllaDB developers regularly check the assembly generated code to verify efficiency metrics and uncover potential microarchitectural optimizations, 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.

"So why did we end up moving from Cassandra to ScyllaDB?" Peace of mind, both operationally and expense-wise, was key. We no longer have to worry about 'stop-the-world' GC pauses. Also, we were able to store more data per node, and achieve more throughput per node, thereby saving significant dollars for the company."

Dilip Kolasani, Expedia.

Learn more about his experience in this tech talk.

DESIGN DECISION #2: CASSANDRA AND DYNAMODB 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 an API-compatible alternative that overcomes the performance, latency and resource 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 inventing yet another query language.

ScyllaDB's Cassandra compatibility encompasses:

- **Wire Protocol:** ScyllaDB supports CQL and the full polyglot of client/driver languages. We strive to be fully compliant with CQL and support all functionality, from lists, maps, to counters, UDTs, secondary indexes and lightweight transactions (LWT).

- **Monitoring:** ScyllaDB indirectly supports the Java Management Extensions (JMX) protocol (direct use of the JMX server is not recommended). We add a JVM-proxy daemon that transparently translates JMX into our RESTful API. Common Cassandra dashboards can retrieve the same information from ScyllaDB, with results in JSON format. ScyllaDB provides additional monitoring through the Prometheus API and direct support for REST.

- **Underlying File Format and Algorithms:** ScyllaDB uses Cassandra's Sorted Strings Table (SSTable) format and supports all of Cassandra's compaction strategies. In addition, ScyllaDB uses the Incremental Compaction Strategy (ICS) which delivers a 30% improvement in compression over the other compaction strategies. ScyllaDB also provides support for auxiliary tools for SSTable loading, backup and restore, and repair.

- **Configuration File:** ScyllaDB can directly consume Cassandra's configuration file, cassandra.yaml, allowing for a seamless migration path. ScyllaDB ignores JVM-related options and various other limitations that derive from Cassandra (such as parallel compactors configuration). ScyllaDB always uses maximum parallelism.

- **Command Line Interface:** ScyllaDB parses and executes the same command line tool (nodetool). Its processes—from backup to repair—are identical to Cassandra's. However, ScyllaDB's nodetool code has also been rewritten from Java to C++, making it much faster.

As a result of this fundamental decision, organizations using Cassandra or DynamoDB today can seamlessly migrate to ScyllaDB without rewriting existing applications. They can also continue to leverage their existing ecosystems.

Detailed comparisons of ScyllaDB vs Cassandra, including benchmarks, can be found here.

Similarly, our decision to implement an Amazon DynamoDB-compatible API, ScyllaDB Alternator for DynamoDB applications enables DynamoDB users to easily migrate to a more flexible and price-performant database. ScyllaDB Alternator redirects to ScyllaDB instead of DynamoDB, avoiding significant changes to DynamoDB applications. This approach results in lower costs and latencies at higher throughputs. Additional advantages can be attained for write-heavy use cases or where DynamoDB Global Tables or either DAX or an external cache are required; all of these add-on components can be eliminated with ScyllaDB, further reducing cost and complexity.

- **Wire Protocol:** ScyllaDB Alternator provides a wire protocol converter for applications that use the DynamoDB API by redirecting HTTP or HTTPS traffic to ScyllaDB ports. It is enabled with two easy configuration options in the YAML configuration file.
- **Supported DynamoDB attributes:** Key table and item operations, scans and filters, and all attribute types, including nested documents, are implemented.
- **Underlying file and data formats:** DynamoDB tables are stored in ScyllaDB individually in separate ScyllaDB keyspaces with all standard ScyllaDB options and functionality. DynamoDB's eventual and strong consistency levels for reads are supported through ScyllaDB's CL-level settings.

The ScyllaDB Alternator documentation contains extensive information, and detailed comparisons of ScyllaDB vs. DynamoDB, including benchmarks, can be found here.

"We did not change our application one bit—the same drivers, the same commands we were issuing before with Cassandra worked with ScyllaDB with absolutely no changes. So for a very low technical cost we saved money on infrastructure and got much improved read performance, especially when under load."

David Blythe, Principal Software Engineer, SAS Institute.

Learn more in this tech talk.

DESIGN DECISION #3: ALL THINGS ASYNC

Equipped with dozens of multi-core processors, modern hardware servers are 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 such as Non-Volatile Memory Express (NVMe), we decided not to synchronously wait for either I/O completion or neighboring 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.

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

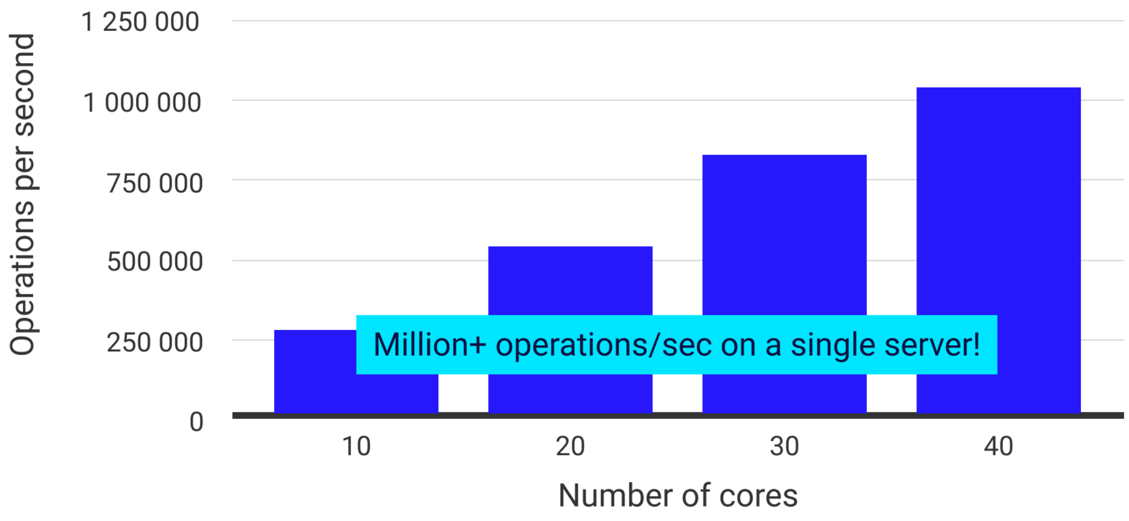


Figure 1: Scale Linearly: ScyllaDB – Operations per Second vs. Number of Cores

DESIGN DECISION #4: SHARD PER CORE

Moore's law went hand-in-hand with Dennard Scaling for decades, doubling single-threaded CPU performance every 18 months. As CPU frequency limits were reached, chip manufacturers began experimenting with multicore CPUs in the late 1990's, and eventually shifting to multicore architectures in the late 2000's with the breakdown of Dennard Scaling. 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 to the data. Other threads, however, are prevented from accessing the data and, as a result, they block. Even a cheap 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 Non-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.

ScyllaDB addresses these scalability issues using a shared-nothing architecture. There are two levels of sharding in ScyllaDB. 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.

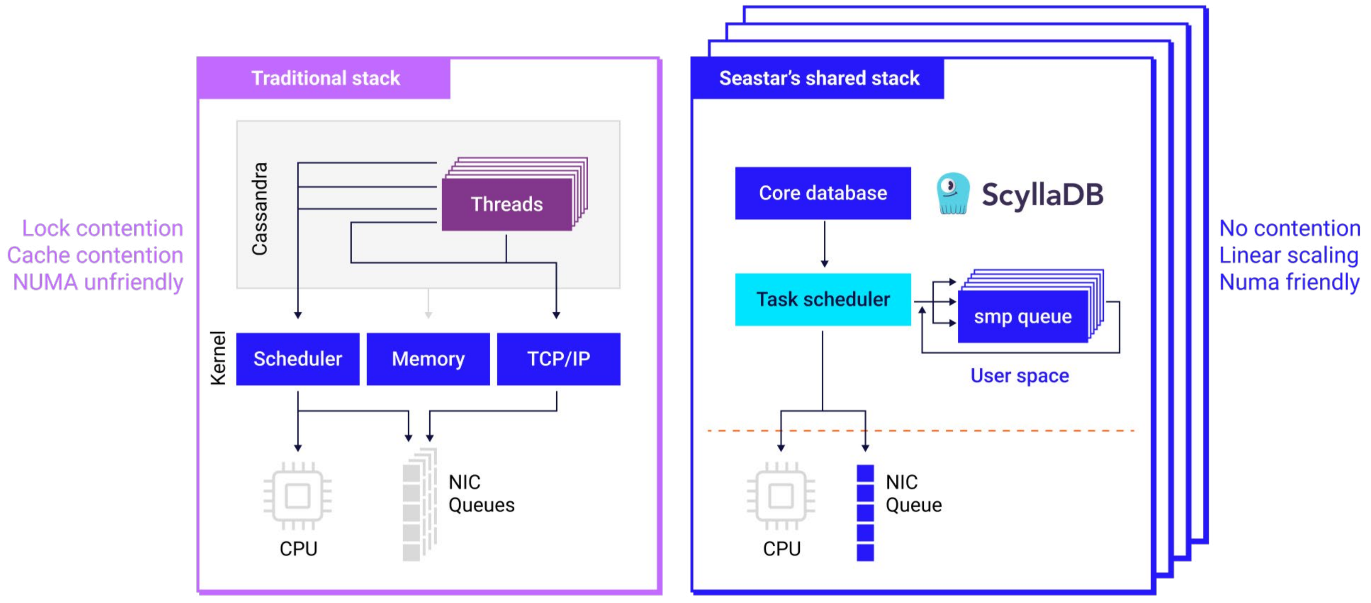


Figure 2: ScyllaDB Network Comparison

The node’s token range is automatically sharded across available CPU cores. (More accurately, datasets are sharded across hyperthreads). Each shard-per-core process 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.

In ScyllaDB, 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 ScyllaDB 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. ScyllaDB’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. So while the Linux page cache is a general-purpose data structure we recognized that a special-purpose cache, intrinsic to the database, would deliver better performance.

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.

To attempt to alleviate read amplification, Cassandra also 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

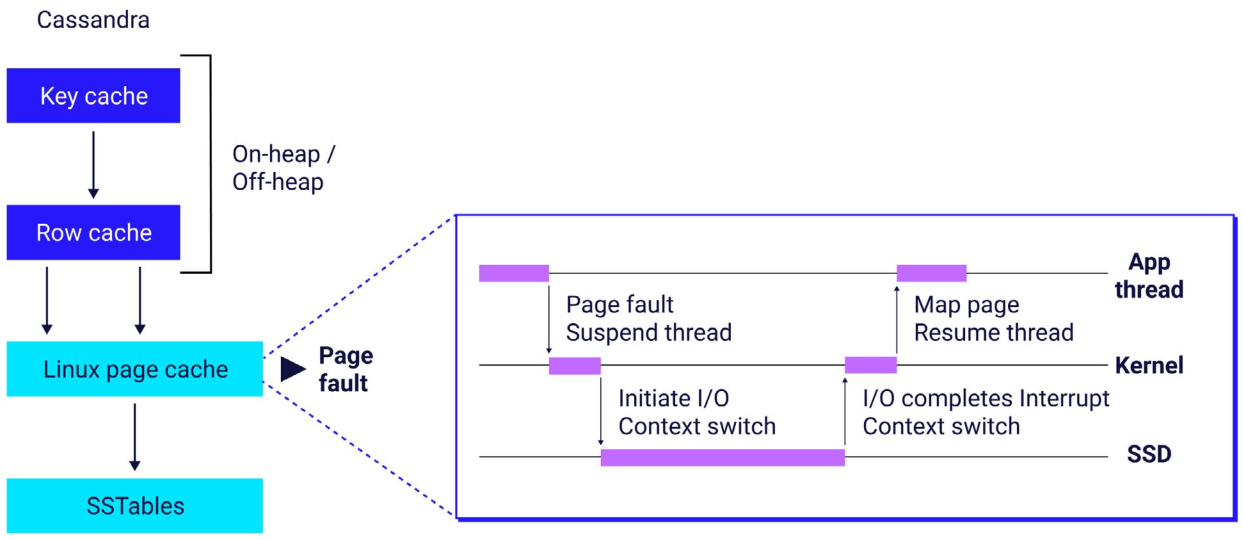


Figure 3: A diagrammatic view of the Linux Page Cache used by Apache Cassandra

are performed at boot time, it’s practically impossible to get it right, especially for dynamic workloads.

The diagram below 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 row-level cache. This cache bypasses the Linux page cache, so does not suffer the same read amplification. As well, a unified cache can dynamically tune itself to the current workload, obviating the need to manually tune multiple different caches. ScyllaDB caches objects itself, and thus always controls their eviction and memory footprint. Moreover, ScyllaDB can dynamically balance the different types of caches stored.

ScyllaDB 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, 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. Seastar will execute it in a µsec (1 million tasks/core/sec) and rush to run the next task. There’s no blocking, heavyweight context switch, waste, or tuning.

For ScyllaDB users, this design decision means higher ratios of disk to RAM, yet also better utilization of your available 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, unpredictable workloads, getting it right is a daunting challenge. An inaccurate cap may cause commensurate 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 ScyllaDB is to provide a database that finds this balance automatically and autonomously, without operator intervention; ScyllaDB should maximize I/O but not overload the drives. This way ScyllaDB has full control of how to prioritize the foreground operations over the background operations. Thus no tuning is required, query operation is always minimized, ScyllaDB maximizes disk throughput while idle time exists and streams gigabytes of data per second, without any limit.

When ScyllaDB 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.

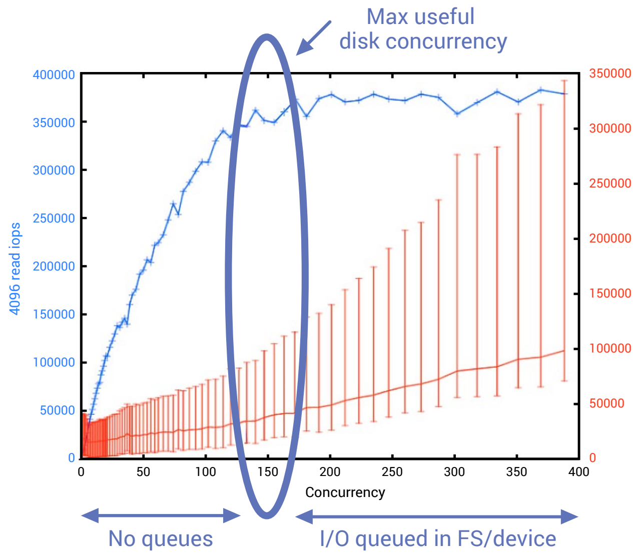


Figure 4: ScyllaDB's iotool benchmarks to ensure the optimal balance between foreground and background operations.

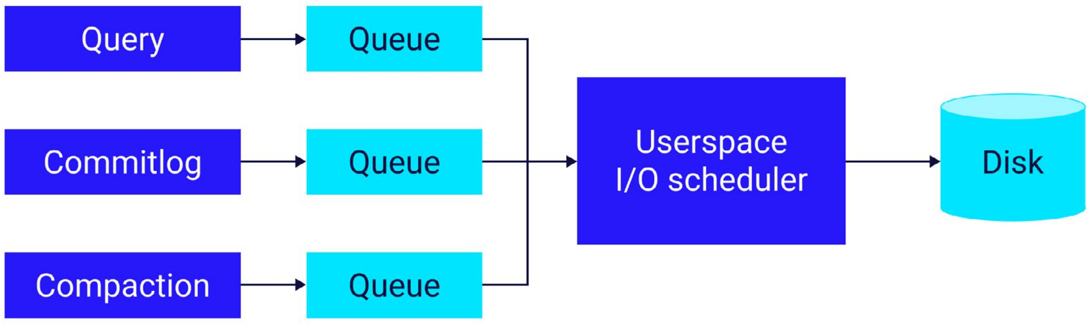


Figure 5: ScyllaDB's I/O Scheduler enforces priority across user-facing workloads and background tasks.

To manage data on disk, both Cassandra and ScyllaDB 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 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 ScyllaDB 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 ScyllaDB 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.

"We've reduced our P99, three 9 and four 9 latencies by 95%."

Phil Zimich, Senior Director of Engineering, Comcast.

Learn more in this tech talk.

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.

Before ScyllaDB, 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.

ScyllaDB 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 ScyllaDB. 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 retune 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.

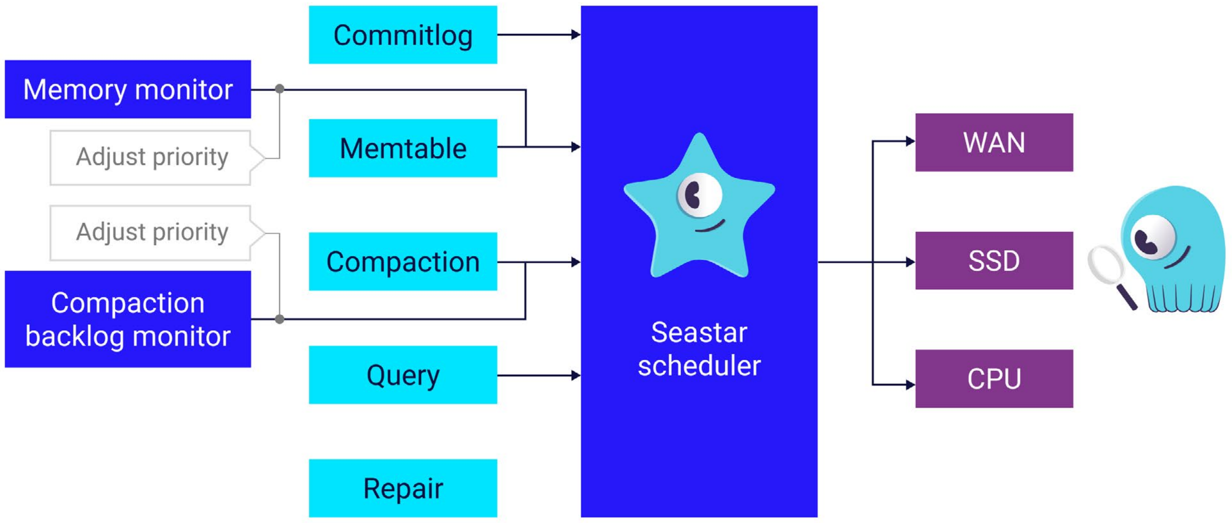


Figure 6: A closer view of ScyllaDB's Scheduler architecture. ScyllaDB dynamically adjusts priority across tasks in user space, enabling self-optimizing operations.

“We needed an aggregation store that is high-throughput, low-latency, low-overhead, low-cost, and low maintenance. That is where ScyllaDB comes in. Low maintenance is extremely important. I have a baby at home. I don’t want another baby at work!”

Aravind Srinivasan, Lead Engineer at Grab.

Read more about his experience in this case study.

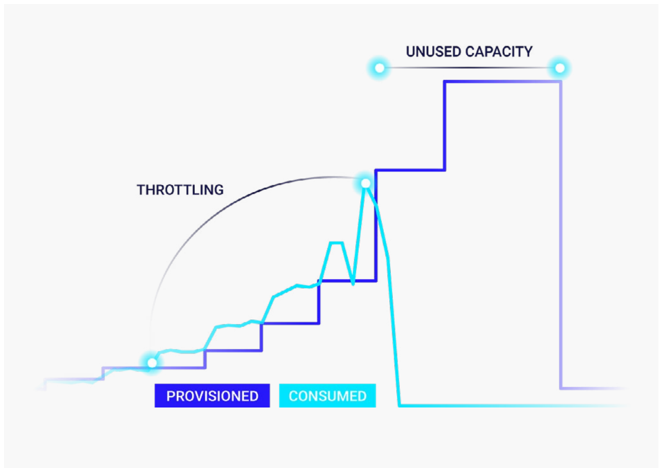
DESIGN DECISION #8: ELASTIC SCALE

Although cloud infrastructure is inherently elastic, most databases don't take full advantage of that elasticity. New nodes can be provisioned fast, but there's a significant lag before they can actually serve traffic – particularly when data distribution is static (e.g., determined exclusively by a hashing function).

The new data distribution must be completed and synchronized across the cluster before new nodes can reliably service reads and writes. Even with "autoscaling," substantially increasing cluster capacity takes time. In addition, moving massive amounts of data is challenging because it may pin too many compute and IO resources, causing a performance bottleneck.

With ScyllaDB's tablets architecture, data is dynamically redistributed as the workload and topology evolve. This design decision, which builds upon ScyllaDB's extension of the Raft consensus protocol, enables new levels of elasticity, speed, simplicity, and efficiency.

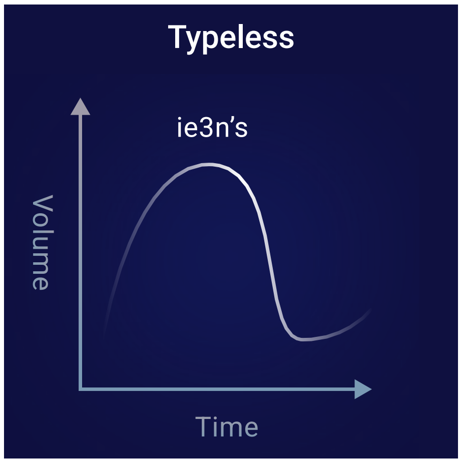
New nodes can be spun up in parallel and start adapting to the load in near real-time. This means teams can quickly scale out in response to traffic spikes – satisfying latency SLAs without needing to overprovision "just in case." Adaptive load rebalancing further optimizes efficiency on top of ScyllaDB's signature shard-per-core architecture.



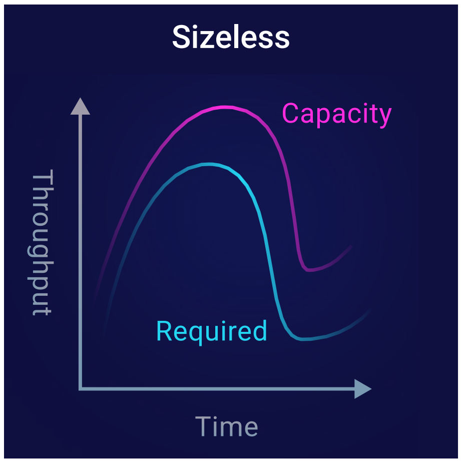
As a result, users can:

- Rely on highly-efficient deployments
- Use what they really need on an hourly or per-minute basis
- Be prepared for any given spike in demand

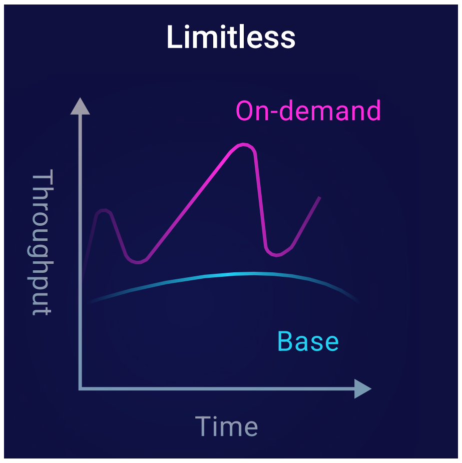
The tablets architecture establishes the foundation for the following use cases:



*You could be **typeless**. You won't have to think about instance type ahead of time. Do you need a volume-based instance like the i3ens, or a throughput-based instance like i4is? What if the volume will vary over time? It won't matter. The instances will adjust automatically.*



*You could be **sizeless**. That means you won't have to worry about capacity planning when you start off. Start small and evolve from there.*



*You could also be **limitless**. You could start off anticipating a high throughput and then reduce it, or you could commit to a base and add on-demand usage if you exceed it.*

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 problematic for many projects.

On the other hand, public cloud users who moved to a NoSQL database-as-a-service (DBaaS) such as Amazon DynamoDB avoided maintenance and administrative tasks, but found their ease-of-use came with a price: operational costs increased dramatically. Plus, once committed to DynamoDB, they were locked into a single cloud vendor solution, limiting their operational flexibility.

Users clearly want a system that provides a best-of-all-worlds solution; high performance, high availability and reliability, ease of administration, operational flexibility as well as significantly lower total cost of ownership (TCO).

ScyllaDB delivers on the original vision of NoSQL—without the architectural downsides associated with Apache Cassandra or the costs of DynamoDB. To achieve this goal, the ScyllaDB team had to rethink many of the foundational architectural choices behind Cassandra. The team behind ScyllaDB 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, ScyllaDB is battle-hardened in production datacenters. Customers use it as an API-compatible replacement for Cassandra or DynamoDB as a replacement for SQL and NoSQL databases such as MongoDB and as a replacement for the Redis cache. Production deployments have shown that ScyllaDB significantly shrinks the datacenter hardware footprint compared to Apache Cassandra clusters, which often run on many small nodes. ScyllaDB was designed from the ground-up to maximize available computing resources and to take advantage of modern multi-core 'commodity' hardware. In this way, ScyllaDB delivers scale-out capabilities along with vastly lower management and administrative overhead compared to other distributed databases on the market today.

NEXT STEPS

Get started with ScyllaDB

Learn more at ScyllaDB University

Explore papers, videos, benchmarks & more

About ScyllaDB

ScyllaDB is engineered to deliver predictable performance at scale. It's adopted by organizations that demand ultra-low latency, even with workloads exceeding 1M ops/sec. Our shard-per-core architecture leverages the power of modern infrastructure – translating to fewer nodes, less admin, and lower costs.

Over 400 game-changing companies like Disney+ Hotstar, Expedia, 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](https://www.scylladb.com)

SCYLLADB.COM

ScyllaDB

United States Headquarters
1309 S Mary Ave
Sunnyvale, CA 94087 USA
Email: info@scylladb.com

Israel Headquarters
11 Galgalei Haplada
Herzelia, Israel