SCYLLA

7 Reasons Not to Put an External Cache in Front of Your Database

Caches can be one of the more problematic components of a distributed application architecture. This paper identifies the seven primary problems behind using external caches with a NoSQL database.

CONTENTS

WHY CACHE A DATABASE IN THE FIRST PLACE?

You have a database. Why would you need to add a cache? Traditionally, caches have been added to databases because they provided a performance boost. Caches temporarily hold data in memory, making data access much faster by side-stepping persistent storage media, like hard or solid-state drives (SSD). RAM access (measured in tens of nanoseconds) is about three orders of magnitude faster than SSD access (measured in tens of microseconds).

Simple enough, right? Add a cache and improve application performance. If it were that simple, we'd all just deploy caches and be done with it. However, there is a famous dictum:

There are only two hard things in Computer Science: cache invalidation and naming things.
— Phil Karlton, c. 1995

Caches are not as simple as they are often made out to be. In fact, they can be one of the more problematic components of a distributed application architecture. If your data infrastructure relies on caches, your overall approach may be subject to the downsides of caching. To help you assess these downsides, we've identified the seven primary problems behind using external caches with a NoSQL database. We also explain how the Scylla NoSQL database's native caching alleviates these downsides and provides a simple, scalable solution.

BACKGROUND

Myriad database caching solutions have been developed over the years. They range from simple application caches to in-memory data grids that span global computer clusters. Amazon Web Services, for example, offers a managed cache solution, known as Amazon DynamoDB Accelerator (DAX), that runs in front of its database. Redis, as another example, popularized the idea that a system can be both a store and a cache, all at once.

Before we dive into the problems with these solutions, let's examine a few generic approaches to database caching.

Pre-caching and Caching

Pre-caching is a process that loads data ahead of time in anticipation of its use. For example, when a web page is retrieved, the pages that users typically jump to when they leave that page might be pre-cached in anticipation. An application might pre-cache files or records that are commonly called for at some time during a session. Pre-caching differs from web and browser caching, in that pre-caching implies storing files that are expected to be used, whereas regular caching deals with files already requested by the user.

Pre-caching requires some method to determine what should be cached ahead of time (i.e., time and expertise needed to make manual or procedural decisions on what to pre-cache). However, pre-caching might contain data that never actually gets used. If you are caching in an all-RAM instance, that means you're paying money on expensive resources for pre-caching info of low-value.

Caching, on the other hand, is populated as needed. Since the cache is reactive, the first transaction to it is likely to be much slower than subsequent actions on the same data. Since nothing is primed in the cache, it has to be populated as requests are served. This could saturate certain requests until the cache is fully populated.

Side Cache versus Transparent Cache

A further distinction exists between side caches and transparent caches. External cache deployments are typically implemented in the form of a "side cache." The cache is independent of the database, and it places a heavy burden on the application developer: The application itself is responsible for maintaining cache coherency. The application performs double writes, both for the cache and for the database. Reads are done first from the cache, and, if the data isn't found, a separate read is dispatched to the database.

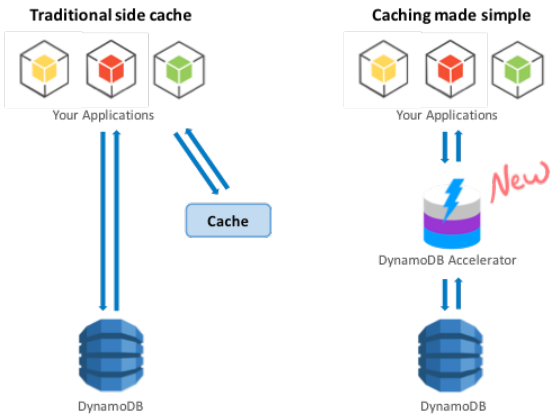


Figure 1: side caching and transparent caching

Transparent caching, such as that used by Amazon DAX, improves on this situation somewhat by unburdening the application developer. Only a single database query needs to be issued. Figure 1 above shows the advantages of a transparent cache like the DAX: It doesn't require application changes and it is automatically coherent. DAX eliminates the major data coherency problem.

THE SEVEN FAILS OF EXTERNAL CACHES

Still, many believe that external data caching is the most effective strategy to improve overall application performance. But is this the best industry standard practice? Are databases doomed to have a cache in front of them? We at ScyllaDB believe these external caches are inefficient and that users should refrain from using caches unless they have no other choice.

With this in mind, we present seven reasons why it's in your best interest to avoid external caches:

1. **An external cache adds latency**
   A separate cache means another hop on the way. When a cache surrounds the database, the first access occurs at the cache layer. If the data is in the cache then performance may indeed improve. However, if the data isn't in the cache, then you will see even slower performance than if you did not have a cache at all. This is because the request is sent first to the cache, and then, upon failure at the cache, to the database. The result is additional latency to an already slow path of uncached data. One may claim that when the entire data set fits in the cache (quite an expensive proposition; see below) the additional latency doesn't come into play, but most of the time there is more than a single workload/pattern that hits the database and some of it will carry the extra hop cost.

2. **An external cache adds needless expense**
   Caching means expensive DRAM, which translates to a higher cost per gigabyte than solid state drives. Even when RAM can store frequently accessed objects, it is best to use the existing database RAM, and even increase it for internal caching rather than provision entirely separate infrastructure on RAM-oriented instances. Provisioning a cache to be the same size as the entire persistent dataset may be prohibitively expensive. In other cases, the working set size can be too big, often reaching petabytes, making an SSD-friendly implementation the preferred, and cheaper, option.

3. **External caching decreases availability**
   No cache's high availability (HA) solution can match that of the database itself. Modern distributed databases have multiple replicas; they also are topology-aware and speed-aware and can sustain multiple failures without data loss.

   For example, a common replication pattern in Scylla is three local replicas, and a quorum is required for acknowledged replies. In addition, copies reside in remote data centers, which can be consulted.

   Caches often lack high-availability properties and can easily fail. Partial failures, which are more common, are even worse in terms of consistency. When the cache inevitably fails, the database will get hit by the unmitigated firehose of queries and likely wreck your SLAs. In addition, even if a cache itself may have some HA features, it can't coordinate handling such failure with the persistent database

it is frontending. Bottom line: Rely on the database, rather than exposing yourself to data loss via the cache.

4. **Application complexity — your application needs to handle more cases**
   Application and operational complexity are problems for external caches. Once you have an external cache, you need to keep the cache up-to-date with the client and the database. For instance, if your database runs repairs, the cache needs to be synced or invalidated. Your client retry and timeout policies need to match the properties of the cache but also need to function when the cache is done. Usually, such scenarios are hard to test.

5. **External caching ruins database caching**
   Modern databases have embedded caches and complex policies to manage them. When you place a cache in front of the database, most read requests will reach only the external cache and the database won't keep these objects in its memory. As a result, the database cache is rendered ineffective, and when requests eventually reach the database, its cache will be cold and the responses will come primarily from the disk.

6. **External caching complicates data security**
   An external cache adds a whole new attack surface to your infrastructure. Encryption, isolation and access control on data placed in the cache are likely to be different from the ones at the database layer itself.

7. **External caching ignores the database's intelligence and resources**
   Databases are very complex and impose high disk I/O workloads on the system. Any of the queries access the same data, and some amount of the working set size can be cached in memory in order to save disk accesses. A good database should have multiple sophisticated logical processes to decide which objects and indexes it should cache, and who should have access to them.

The database also should have various eviction policies (with the least recently used policy as a straightforward example) that determine when new data should replace existing older cached objects.

When scanning a large dataset, say a large range or a full-table scan, a lot of objects are read from the disk. The database can realize this is a scan and not a regular query and chose to leave these objects outside its internal cache, but an external cache would treat this query like any other and attempt to cache the results. The database automatically synchronizes the content of the cache with the disk and with the incoming requests, and thus the user and the developer do not need to do anything to make it happen. If, for some reason, your database doesn't respond fast enough, it means that...

- The cache is misconfigured
- It doesn't have enough RAM for caching
- The working set size and request pattern don't fit the cache
- The database cache implementation is poor

A BETTER WAY: SCYLLA'S EMBEDDED CACHE

The Scylla NoSQL database offers a better approach to caching, one that addresses the significant problems covered above, while also delivering the performance gains that caching promises. To understand how Scylla's cache implementation addresses these problems, it's important to first examine how its cache works.

Scylla is designed to be fully compatible with Apache Cassandra. Yet, unlike Cassandra, Scylla does not rely on the default cache offered by Linux. Linux caching is inefficient for database implementations for the following reasons.

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 off-heap 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.

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.

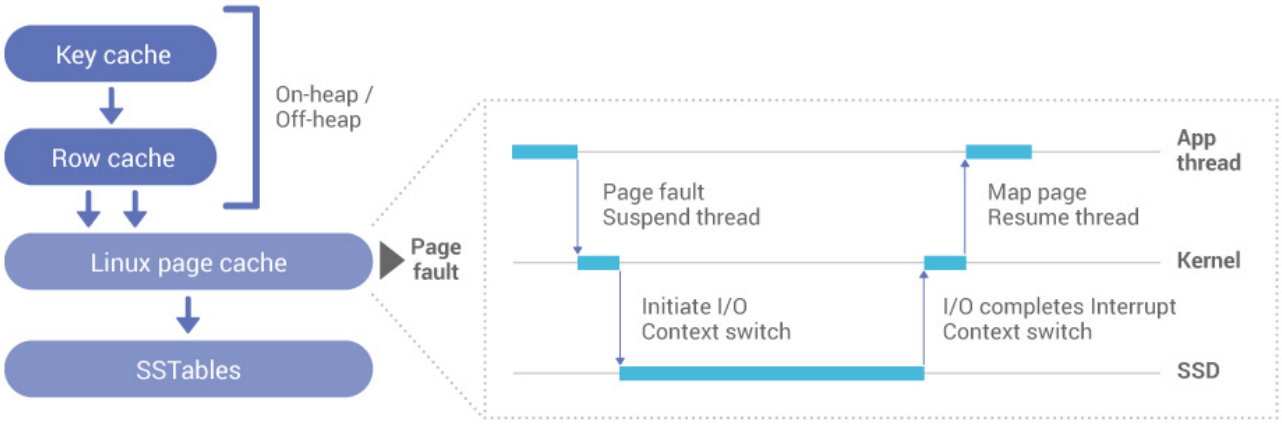
The diagram below displays the architecture of Cassandra's caches, with layered key, row, and underlying Linux page caches.

The architects who designed Scylla recognized that a special-purpose cache 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 Scylla caches objects itself, it always controls their eviction and memory footprint.

More importantly though, Scylla can dynamically balance the different types of caches stored. Scylla 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, 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.

The C++ Seastar framework on which Scylla 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 Scylla users, this design means higher ratios of (cheap) disk to (expensive) RAM.

This cache design enables each Scylla node to serve more data, which in turn lets operators



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

run smaller clusters of more powerful nodes with larger disks. Scylla's unified cache also simplifies operations, since it eliminates multiple competing caches and dynamically tunes itself at runtime to accommodate varying workloads.

Finally, because Scylla has a very efficient internal cache it obviates 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, Scylla allows filtering results to bypass the cache.

PERFORMANCE CHARACTERISTICS OF SCYLLA'S CACHE

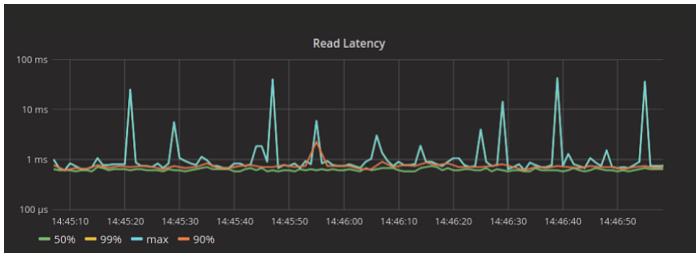
Scylla's cache is much faster than Cassandra's. To prove this, we ran the same workload on Google Cloud to compare the performance of Scylla against Cassandra.

A single-node cluster was running on n1-standard-32 and the loaders were running on n1-standard-4. Both Scylla and Cassandra were run using default configurations.

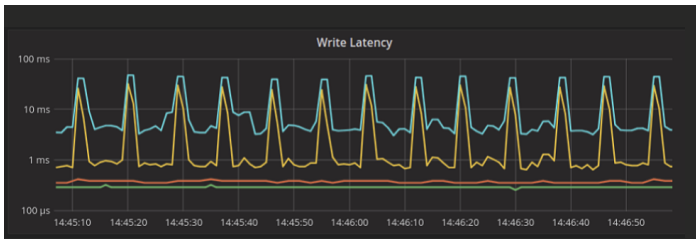
The latency graphs below display the performance characteristics that were recorded during the test.

As you can see, these latency tests show noticeable differences for the cacheable workload. In a 2-minute span Cassandra showed 5 spikes of read latencies greater than 10ms. Scylla never had more than single-digit read latencies. For write latencies, Cassandra regularly suffered spikes far greater than 10ms every nine seconds, whereas Scylla generally kept write latencies under 10ms.

Cassandra

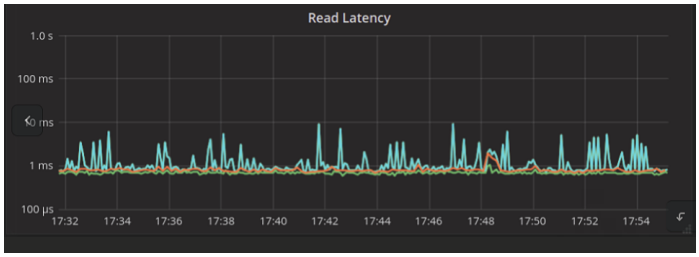


Read Latency

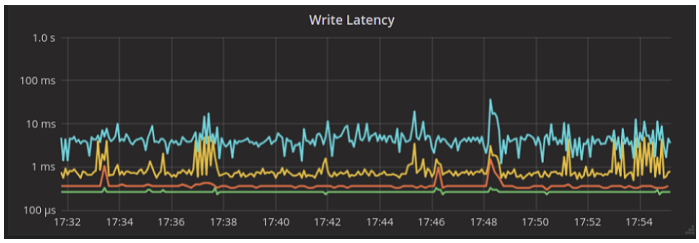


Write Latency

Scylla



Read Latency



Write Latency

SCYLLA'S CACHE IN THE REAL-WORLD

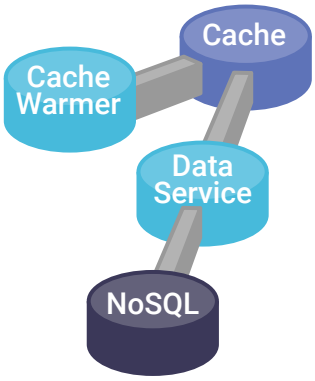
COMCAST ADOPTS SCYLLA, IMPROVES PERFORMANCE AND SIMPLIFIES DEPLOYMENTS

Comcast, the second-largest broadcasting and cable television company in the world, provides a real-world example of the benefits of moving to Scylla's embedded cache. Comcast's X1 platform supports more than 30 million devices used on a monthly basis. Their X1 Scheduler processes over 2 billion RESTful calls daily.

Scylla helped achieve that level of scale in large part by enabling Comcast to eliminate external caches from their data architecture. In one instance, Comcast uses Scylla as an internal data store that experiences very heavy writes. Although the service supports a low read volume in general, it encounters traffic spikes for a variety of reasons, including top and bottom of the hour, weekends, and national events, such as the Super Bowl and the Nationwide Emergency Alert system. The team's goal was to achieve consistency across data centers and to be able to read the data as needed.

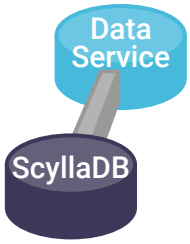
Before Scylla, Comcast had been using Apache Cassandra, which often suffered from long-tail latency spikes. Thus they decided to place an HTTP cache in front of the data store. The data service was a RESTful API connected to their NoSQL data store (Cassandra). Since the cache and related infrastructure had to be replicated across data centers, Comcast needed to keep caches warm. Even though the data was being populated to one data center, the cache warmer ensured that data would also be available to be read from other data centers.

While such a distributed architecture is easy to set up with a NoSQL database, it required additional cache infrastructure. To meet their performance requirements, Comcast implemented a cache warmer that examined write volumes, and then replicated the data across data centers.



Comcast architecture using Cassandra plus a cache and cache warmer (pre-cache)

While this solution solved Comcast's immediate problems, it was not ideal and the team searched for other options. Scylla's embedded cache seemed to be an attractive alternative. Designed to minimize latency spikes through its internal caching mechanism, Scylla enabled Comcast to completely eliminate the external caching layer, providing a simple framework in which the data service connected directly to the data store.



Comcast architecture using Scylla

The result was reduced complexity and higher performance, with a much simpler deployment model. Ultimately, Comcast boosted performance while also reducing its footprint from 962 nodes of Apache Cassandra to just 78 of Scylla.

This resulted in two key benefits:

- Dramatic reduction in TCO and operational overhead
- Flexibility to add new customers without adding new infrastructure

IMVU REINS IN COSTS BY MOVING FROM REDIS TO SCYLLA

A popular social community, IMVU enables people all over the world to interact with each other using 3D avatars on their desktops, tablets, and mobile devices. In order to meet growing requirements for scale, IMVU decided it needed a more performant solution than their previous database architecture of Memcached in front of MySQL and Redis. They looked for something that would be easier to configure, easier to extend, and, if successful, easier to scale.

They decided on using Scylla. Redis worked well in terms of features, but when IMVU actually rolled it out to a hundred thousand concurrent users, they found the expense difficult to justify. Scylla allowed IMVU to maintain that same responsiveness for a scale ten to a hundred times as large as what Redis could handle.

“Redis was fine for prototyping features, but once we actually rolled it out to a hundred thousand concurrent users, the expenses started getting hard to justify,” said Ken Rudy, a senior software engineer at IMVU. “Scylla is optimized for keeping the data you need in memory and everything else in disk. Scylla allowed us to maintain the same responsiveness for a scale a hundred times what Redis could handle.”

CONCLUSION

Modern applications rely on memory architectures that are both extremely fast and globally distributed. The ideal architecture finds a balance between memory (RAM) and storage (SSD) to deliver high performance along with reliability and consistency.

Many organizations use in-memory caching alongside a NoSQL database. As we have shown in this paper, caching solutions that are not integral to the database can cause enormous headaches. Among many problems, they impose unnecessary cost, complexity, and operational overhead.

If you are using an external cache or in-memory database for high-speed operational performance, you should take a closer look at the Scylla NoSQL database.

NEXT STEPS

Visit scylladb.com to...

- **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. A drop-in alternative to Apache Cassandra and Amazon DynamoDB, Scylla embraces a shared-nothing approach that increases throughput and storage capacity as much as 10X that of Cassandra. AdGear, AppNexus, Comcast, Fanatics, FireEye, Grab, IBM Compose, MediaMath, Ola Cabs, Samsung, Starbucks and many more leading companies have adopted Scylla to realize order-of-magnitude performance improvements and reduce hardware costs. Scylla is available in Open Source, Enterprise and fully managed Cloud editions. ScyllaDB was founded by the team responsible for the KVM hypervisor and is backed by Bessemer Venture Partners, Eight Roads Ventures, Innovation Endeavors, Magma Venture Partners, Qualcomm Ventures, Samsung Ventures, TLV Partners, Western Digital Capital and Wing Venture Capital.

For more information: ScyllaDB.com

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