

5 Signs Your Cache + Database Architecture May Be Obsolete

Table of Contents

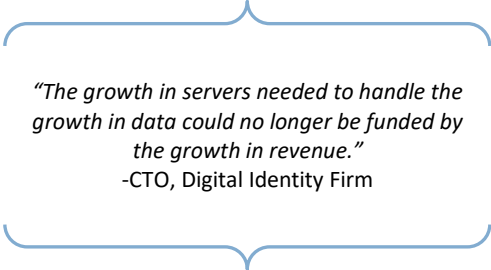
Executive Overview.....	3
5 Signs Your Cache + Database Architecture May Be Obsolete	4
What Is Hybrid Memory Architecture and Why Is It important?	4
Five Signs Your Cache + Database Architecture Is Obsolete.....	5
Sign #1: Your caching nodes are growing at an uncontrolled pace and you’re preparing yet another unplanned budget variance request.....	5
Sign#2: It takes many hours - or even days - to repopulate your cache after a disruption, and dealing with stale data is now part of your daily operations	7
Sign #3: You have implemented a cache-first architecture, - but you still can’t meet your SLAs.	8
Sign #4: Your data has grown to where you need clustering - and that scares you.....	9
Sign #5: You’ve “solved” your cache stampede problem by code revision - again – and you now have to update code libraries; or push a cache proxy update – again.....	10
Reevaluate the need for an external caching layer altogether	11
Let’s review true hybrid memory systems and how Aerospike works	13
Hybrid memory systems store indices in memory (DRAM) and data on SSD storage, reducing server count by as much as a factor of 10.	13
Hybrid Memory Systems are multi-threaded, massively parallel systems.....	14
Hybrid Memory Architecture significantly improves uptime and availability, without manual DevOps processes.....	15
Hybrid Memory Architecture automatically distributes both data and traffic to all the nodes in a cluster.	17
Lower TCO, creating a competitive advantage and enhanced business value.....	18
The Path Forward.....	19
From Cache + RDBMS	19
From Cache + First-Generation NoSQL	19
Summary	20

Executive Overview

For years using an external caching layer with an RDBMS was the accepted conventional wisdom for attaining performance and reliability. Data was kept in memory by the cache to reduce the amount of time spent querying the underlying data store. As data volumes grew, NoSQL databases were substituted for the RDBMS to provide horizontal scaling to clusters and to keep latency down.

Then “Digital Transformation” happened.

In the blink of an eye, simple web applications have become systems of engagement (SoE) that are now serving billions of objects, with millions of contextual data points and enabling rich interactions and engagement - all in a few milliseconds. Data has grown in both velocity and volume - the more data, delivered faster, the richer the engagement and the better the decision. Larger datasets of up to 40TB to 100TB are now common, according to Forrester Research¹, and there is no end in sight to data growth or to transaction velocity. The only way to support modern velocities and simultaneously deal with growing cache data volumes is to subscribe to a never-ending cycle of adding cache nodes and deploying increasingly more complex cache management systems and strategies. This approach ignores the high cost of external DRAM caching layers. As data volumes expand external caching solutions are un-fundable over time and require a significant investment in managing complex data lifecycle issues such as cache consistency and correctness. Translation: cache is unaffordable, un-trustable, and unstable at high volumes.



“The growth in servers needed to handle the growth in data could no longer be funded by the growth in revenue.”
-CTO, Digital Identity Firm

It’s time to challenge conventional wisdom.

Even though it’s still common practice today, the harsh reality is that using an external cache layer as a fundamental component of a System of Engagement (SoE), where huge scale, ultrafast response, and rock-solid reliability are critical success factors, is a last decade approach for solving next decade problems. Instead of more memory, or a better cache, perhaps what’s needed is a better data architecture - one which uses the fact that systems of engagement typically don’t require; and doesn’t rely on an external caching layer for performance, reliability, or scale: *Hybrid Memory Architecture*. Forrester defines hybrid memory architecture as a database architecture that combines the use of DRAM and SSD and combines DRAM’s parallelism and SSD’s fast, random bulk storage capability to provide near DRAM performance with SSD reliability and lower cost, including integrated internal DRAM caches. A hybrid memory database provides automation to simplify the access and processing of data, and support for transactional, operational and analytical workloads. It leverages SSD and flash natively in an optimized and engineered manner.

Fortunately, the Aerospike database, built with hybrid memory architecture, allows you to eliminate that external caching layer while simultaneously handling internet-scale data volumes at sub-millisecond response times and with significantly fewer servers. It is used in production and trusted by industry-leading organizations to power their Systems of Engagement.

¹ “Forrester Study on Hybrid Memory NoSQL Architecture for Mission-Critical, Real-Time Systems of Engagement” 2017 Forrester Research, Noel Yuhanna

5 Signs Your Cache + Database Architecture May Be Obsolete

What are the 5 signs your external cache + database architecture may be obsolete?

- 1 Your caching nodes are growing at an uncontrolled pace and you're preparing yet another unplanned budget variance request**
 - Is your growth in data volume or access patterns driving ever-larger caching clusters (30 nodes or more)?
 - Are you frequently requesting budget for more memory or servers, over and above your original projections?

- 2 It takes hours or even days to repopulate your cache after a disruption, and that is now part of your daily operations**
 - Do you frequently hold new nodes or servers out of production while the cache gets repopulated? For how long?
 - Are you worried that your data will be out of date while the cache gets rebuilt?

- 3 You have implemented a cache-first architecture, but you still can't meet your SLAs.**
 - What's causing you to miss your SLAs? At what cost?
 - Even if you are meeting your SLAs today, will you be able to meet them as your data and customer size grows?

- 4 Your cache has grown to where you need clustering - and that scares you**
 - Do you have the time, talent, and experience to rearchitect your new system for clustering?
 - Are you worried that the new architecture may not support further - especially unexpected - growth?

- 5 You've "solved" your cache stampede problem by code revision - again – and you now have to update code libraries; or push a cache proxy update – again**
 - Do you experience cache stampede often?
 - Have you tried different algorithms to solve the problem?
 - Is it becoming hard to convince your development groups to use the "latest" fix for this problem?

What Is Hybrid Memory Architecture and Why Is It important?

The cache + database architecture was never designed to handle the volume and latency requirements for today's systems of engagement. An RDBMS just doesn't scale horizontally and time-consuming complex joins and queries are rarely necessary. First-generation NoSQL databases are built for horizontal scale but can't maintain sub-millisecond response times.

Why is hybrid memory architecture important?

“Hybrid memory architecture is a new approach that leverages both volatile memory (DRAM) and nonvolatile memory such as SSD and flash to deliver consistent, trusted, reliable and low-latency access to support existing and new generations of transactional, operational, and analytical applications. Implementing this type of architecture allows organizations to move from a two-tier in-memory architecture to a single-tier structure that simplifies the movement and storage of data without requiring a caching layer. Early-adopters are seeing several benefits to their business like a lower cost of ownership, huge reductions in their server footprint, simplified administration and improved scalability.”²

Hybrid memory architecture has the following characteristics:

1. Stores indices in memory (DRAM) and data on SSD storage, reducing server count by as much as a factor of 10.
2. Has multi-threaded, massively parallel systems for predictable performance.
3. Significantly improves uptime and availability, without complex and error-prone manual DevOps processes.
4. Automatically distributes both data and traffic to all the nodes in a cluster. Clients automatically do load balancing for improved performance and correctness.
5. Provides better data consistency, because external cache unreliability is no longer an issue.³

Aerospike Database hybrid memory architecture eliminates the need for the out-dated and costly external cache layer; it’s the best way to architect systems of engagement. The result is significantly better performance, reliability and uptime, as well as simplified datacenter operations.

“The business benefits of making this leap (to hybrid memory architecture) can be far-reaching and position an organization for the next wave of digital business growth in a world of instant analytic insights on real-time data.”⁴

Five Signs Your Cache + Database Architecture Is Obsolete

Sign #1: Your caching nodes are growing at an uncontrolled pace and you’re preparing yet another unplanned budget variance request

As your business grows, so does your data and the size of your cache, at a rate that’s directly proportional. As the value of engagement increases, new applications and projects clamor for database access, increasing transaction volumes and cache working set sizes. Server counts need to keep up, even though budgets were set at the beginning of the year. When growth happens beyond expectations it *must* be addressed or your System of Engagement won’t be able to keep up.

² “Hybrid Memory Architecture Drives Real-Time Systems of Engagement” Forrester Research, May 2017

³ Ibid.

⁴ “Delivering Digital Business Value Using Practical Hybrid Transactional/Analytical Processing” Gartner, Inc. Published: 27 April 2017 ID: G00314780

As noted above, our customers who were using the old cache + database architecture found that funding this growth was simply unsustainable. Business growth means a non-linear growth in data as more information is compiled or queried per customer and per transaction. Add to that the use of additional data sources for better analysis (more data is always considered better when it comes to transactional analytics) and your cache will rapidly and happily expand to consume the budget available - and then some.

“When your cache runs out of memory...”

-(overheard from multiple presenters at RedisConf2017)

The operative word here in the quote to the left is “when” and not “if.” The companies at the marquee 2017 Redis conference who presented “solutions” to their cache management issues ranged in size from a few dozen employees to some of the largest, well-known companies on the planet. They *all* acknowledged that the issues and strategies for dealing with cache management were serious and common enough to warrant

presenting them to other users. No one using an external cache, it seems, is immune from the challenge of running out of memory. The only companies whose caches won’t run out of memory are those that aren’t growing, which doesn’t sound like a successful business outcome.

There’s a direct relationship between data growth and server growth that’s easy to calculate in node count, memory, and cost. As of the time of this writing, the maximum memory a typical DRAM cache node can have is 768GB of DRAM.⁵ It’s common to allocate ~512GB to the cache per node, leaving room for local processing, etc. Even if your current database is small by today’s standards (- say 20TB) and with modern transaction volumes, requires an 80% cache hit rate over a 10TB working set, you will need to allocate about 16 servers for your cache layer. As this system of engagement is core to the company’s business function, best practices would mandate spares and over-provisioning of either 2x (in a primary-secondary design), or 50% (in a complex sharded cache design), meaning you need between 24 and 36 nodes. We’ve spoken with one customer who had an internal requirement of 4 redundancies, so the number of nodes jumps to 128 - without factoring in any growth to the data or their business.

Key Points

Data growth has made an external caching layer virtually unaffordable.

Rebalancing cached data to SSD may lower your costs in the near term but adds to the complexity of managing that data.

⁵ Intel, Samsung, Micron; et al, are investing billions of dollars into SSD technology; thus we realize that it’s only be a matter of time before 1TB+ servers hit the market. When it is available, memory will still be, on average, many times the price of storage. In the above example, having a 1TB node doesn’t alter the math appreciably. A 20TB database would still require at least 14 (- and more like 20 nodes) - with a zero replication factor. And at what cost?

Sign#2: It takes many hours - or even days - to repopulate your cache after a disruption, and dealing with stale data is now part of your daily operations

Nodes stop working. Switches break. The power goes out. Packets get dropped. Storage fails. Disruptions are a fact of life. The larger your datacenter or cloud, the more frequent the disruptions. Provisioning a new caching server in the cloud and DevOps era is now a matter of minutes for most companies but, sadly, that doesn't apply to the data in your caching layer. It has to be "rehydrated" to a level where the hit rate is acceptable before it has its intended impact of reducing database load. For most data-heavy companies this process can take hours or even days; this forces them to deal with limited performance, inaccurate data, the unnecessary cost of even greater over-provisioning, and application complexity - and somehow that has become acceptable. Why?

If your cache has crashed, what do you do while it repopulates? If you had a single-layer cache, you would be forced to query the database directly. That's not acceptable because your performance would be too slow, and you would run the risk of causing cache stampede if enough users were to access the same item, making for an even bigger disaster. (As we discuss later on). The second option would be to anticipate losing the cache and set up a "hot" standby system from the beginning. Your standby would be accurate but now you've just doubled the cost of your cache and greatly increased the amount of network traffic in your system. The third - and most common option - is to set up a consistently hashed sharding system and overprovision it by some factor, depending on your cache hit rate sensitivity. You're not carrying as much extra cost as a hot-standby system, but now you've now added significant software complexity to the mix, as well as operational complexity and growing and shrinking a sharded cache. However, there can still be serious issues. Let's take a deeper look at a real-world example.

The sixth-largest brokerage firm in the world was running their intraday trading system on a traditional cache + RDBMS architecture. Their customers trade multiple times throughout the day and it is imperative that the customer's "position" - the amount of stock and money in their account - must be accurate all the time. At the brokerage's daily transaction volume they had challenges with both the cache as well and the database itself failing during the day. If a customer makes a trade just as the cache or the database fails, what should they see? What happens if either the standby cache or the database has stale data and the customer cannot make an additional trade because their position shows they don't have the available funds? Eventually, the customer's position will be correct but how long will that take and at what cost? At best, the customer has a scary experience while they refresh their screen a few times. And at worst, the brokerage could be held liable for missed or inaccurate trades. Imagine being the customer service representative getting a call from a client that is wondering why some of their money disappeared for a bit before reappearing in their account.

Key Point

If a best practice forces you to sacrifice customer experience or puts your business at risk, perhaps it is no longer a best practice.

Sign #3: You have implemented a cache-first architecture, - but you still can't meet your SLAs.

When you're the fraud detection system for a major online payments system, not meeting SLAs can mean millions of dollars in lost - or fraudulent - transactions per day. "Everyone knows" that neither an RDBMS nor a first-generation NoSQL database is ever going to be fast enough on its own to meet sub-millisecond response times so you have to put a cache in front of it. But it's rarely as simple as that, and this architecture does nothing to guarantee meeting SLAs in the face of growth.

Take the case of one of the largest payment processing firms in the world. Their transaction flow and architecture (highly simplified) looked like the following:

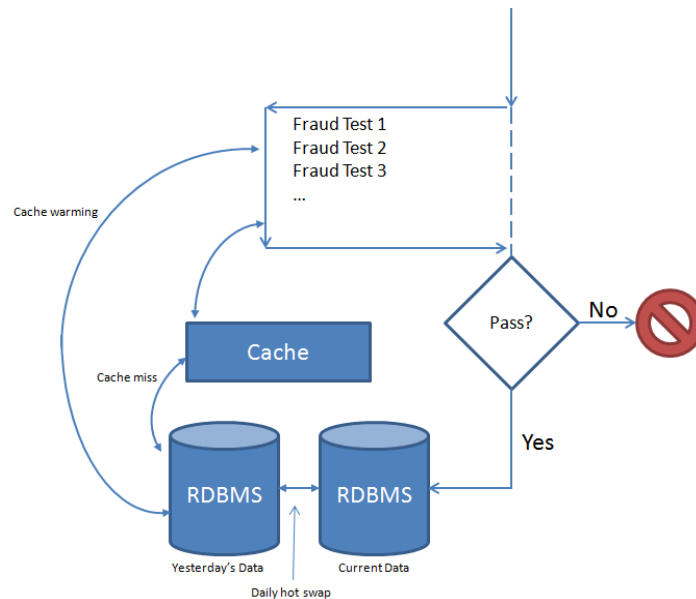


Figure 1. Architecture and Transaction Data of a Large Global Payment Processing Company

As each transaction occurred, they would run hundreds of queries to determine its validity. Each request would refer to the cache. The cache would return a value if it had it, or, in the case of a cache miss, read from the database itself and return that result, adding to the latency. If the transaction was valid, it would be written to the current data instance of the database. Because of the transaction rates and volume, they chose to use a second instance of the database, populated with yesterday's data, behind the cache. Each day, they would perform a hot swap of the databases to try to keep as current as possible. If the cache failed for any reason, the time to rewarm it grew as their business grew. In the meantime, the fraud algorithm was continuously querying stale data, further adding to latency, as well as generating both false positives and negatives.

Despite the complexity and inherent flaws of the above system they were able to meet their performance SLAs of 375 milliseconds for each payment transaction and 175 milliseconds for the fraud detection process – but only barely – as the data took nearly an entire day to load.

One day, an executive announced that the company needed to plan for 10x growth. The payments landscape was about to rapidly change, and they needed to be ready for it. Scaling the architecture to meet such a drastically increased load meant, at best, going from 300 servers to 3000 servers. At worst, it meant scaling from 1,000 servers to 10,000.

Switching to a hybrid memory architecture had dramatic results for the company:

- The server count went from 300 to 30
- Fraud algorithm performance went from 175 milliseconds to less than 80 milliseconds – at peak loads
- They eliminated the caching layer and the dual RAC database clusters, saving millions of dollars in operating costs

Key Point
Scaling the cache, while technically possible, is not the best use of time, money, or staff and provides no assurance of success.

Sign #4: Your data has grown to where you need clustering - and that scares you

Congratulations! Your business is finally starting to grow, perhaps significantly. Whether an unplanned event caused the growth, or all that hard work is finally starting to pay off, your cache size has morphed into a distributed in-memory database requiring the additional burden of sharding, clustering and other new techniques. Typically, personnel with these skill sets demand expert practitioner rates.

You may already be using sharding inside your application to create additional capacity - it *is* a best practice, after all. However, your strategy has problems that get progressively bigger with scale, including one or more of the following:

- The cache size itself can be problematic, as we've discussed above. While it's easy to think that the cache will hold the "most valuable data;" this is often very difficult to predict. Choosing a cache that's too small renders it useless. Some use cases may require caching all the data and using the relational database as a backup. The hardware costs of storing this much data in memory can be prohibitively high, as discussed earlier.
- Re-sharding becomes increasingly difficult. Making changes to the sharding algorithm as you add nodes is disruptive, invalidating the existing data cache and forcing a rebuild. During this time, performance will at best degrade, but may also run into service outages for a time. You may find that some shards have a grossly disproportionate amount of data or traffic, forcing you to spend a lot of time manually tuning them.
- Automated fail-over to another cache is not without risk. As discussed above, allowing for cache failure requires developers to code their application to go back to the relational database at a performance cost, produce an error which can compromise customer experience, or use stale data (which can have financial consequences).
- There is no persistence, so when a cache instance dies all the data in that node goes with it. As a result when a host goes down due to a hardware fault, the cache on the host must be rebuilt (if possible). As noted above, this takes time and introduces risk that most users no longer tolerate.
- Developers must think about and manage the data between the underlying database and the cache separately. This often results in making multiple reads or writes – one for the cache and one for the relational database. Although caching is not a difficult concept, this gets extremely complicated.

With enough growth, however, sharding may no longer be sufficient, which leads to cluster management. Clustering is a more advanced alternative to sharding with a new set of limitations and trade-offs that must be understood; for instance, some commands may not be supported in a clustering environment, multiple databases may not be supported, programmers need to know which node serves which subset of keys, and key lookup across clusters can become an issue. In addition, the process of moving to clustering is an exercise in operational planning. Instead of a simple, one-time project, moving to clustering often turns into an on-going activity requiring a dedicated team of experts doing non-core operations work.

The fully-burdened cost of a scaling engineer in Silicon Valley as of the time of this writing starts at over \$250,000 per year⁶. Experts in the field, if you can find them and lure them away from their current engagement, cost closer to \$500,000 each and it rarely takes only one person to manage a complex system.

Key Points
Advanced management techniques like sharding and cluster management are best left to the experts but this comes at a price.
Even experts can't eliminate the inherent stability, accuracy, and management issues of a cache-first architecture.

Sign #5: You've "solved" your cache stampede problem by code revision - again – and you now have to update code libraries; or push a cache proxy update – again

Wikipedia defines a [cache stampede](#) as "...A type of [cascading failure](#) that can occur when massively [parallel computing](#) systems with [caching](#) mechanisms come under very high load. This behavior is sometimes also called *dog-piling*."

Under particularly heavy load when a cached item expires, multiple processes may all independently try to access that item simultaneously from the database. When all shared resources are exhausted, this can lead to congestion collapse. Once this happens, the item may never be completely recomputed or re-cached. The cache hit rate goes to zero and stays there until the load reduces to a point where resources can be recovered and the cache can be rebuilt.

From your users' standpoint, a cache stampede means that the item they want to see or buy just won't load, and will eventually time out. Users, in their impatience, will either abandon their request or try to refresh the page, exacerbating the problem. In either case, the results can be disastrous for your reputation or revenue stream.

There are three methods for dealing with cache stampede; all involve code changes at the application level, which then must be "sold" to development groups for them to incorporate into their code bases to hopefully prevent a

⁶ Interview with Jeff Tavanger, CEO of The Armada Group, On Demand Talent Solutions

recurrence. At the end of the day, as the Wikipedia article notes, none of these methods solve the problem permanently.

So why have your application developers bear the burden of cache and database administration in the first place? It creates needless complexity, impacts quality and time to market, and places the customer experience at greater risk. Why not eliminate the caching layer altogether and rely on a distributed database that manages all these problems natively, without requiring the application to get involved?

Key Points
Cache stampede doesn't go away completely, no matter what strategy you use.
A cache adds needless complexity and risk to your business. Get rid of it.

Reevaluate the need for an external caching layer altogether

The fact that the above issues of server growth, architectural complexity, instability, and cache stampede exist in the first place indicates that an external caching layer is not always the best strategy for success, especially for systems with spiky or heavy, continuously growing data loads. The fact that companies have entire teams or elaborate software applications devoted to managing the cache is another indication that an external caching layer is not the best strategy for success. The fact that a poorly-architected cache can impede business is yet another obvious indicator that it's probably not the best strategy for success. And finally, when both Gartner and Forrester say that hybrid memory architecture is better than an external caching layer for Systems of Engagement, it's time to challenge current thinking about best practices and accepted architectures.

The left side of *Figure 2* is an example of a typical cache + database architecture for a System of Engagement. Transaction servers interact with users and run applications for use cases like fraud detection, real-time bidding, and payment processing. The caching layer provides data to those applications (and users) at speed. As transactions complete, data is populated back to the underlying operational database, and somehow the cache gets updated. A data integration layer (on the right hand of the figure) is then responsible for updating the enterprise data store to support batch analytics and a System of Record. In addition to the problems outlined above, such an architecture puts a natural barrier between the transactions and the advanced analytics; this can drive better decisions and business outcomes. With the time and analysis pressures of digital transformation, this architecture is now obsolete.

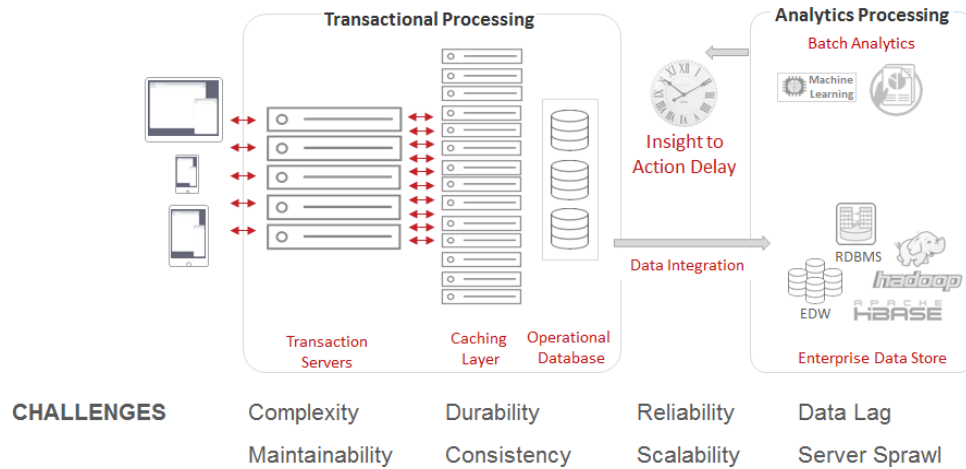
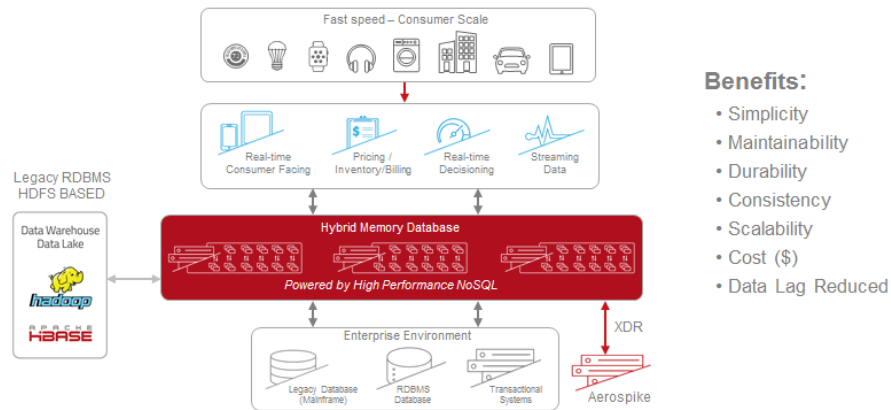


Figure 2. Traditional Caching Layer Architecture

Figure 3 below shows Aerospike's simplified hybrid memory architecture. The external caching layer and all its incumbent problems is completely eliminated, providing one-hop access to the data, at scale, with far fewer resources. It also eliminates the wall with your systems of record, creating new possibilities for action and insights.

Hybrid Memory Systems - Enabling a New Class of Real-time Applications



Aerospike Delivers Predictable Performance, Highest Availability, and Lowest TCO

Figure 3. Aerospike Database's Hybrid Memory Architecture

Let's review true hybrid memory systems and how Aerospike works

Hybrid memory systems store indices in memory (DRAM) and data on SSD storage, reducing server count by as much as a factor of 10.

Aerospike implements a hybrid memory architecture wherein the index is purely in-memory (not persisted), and data is stored only on a persistent storage (SSD) and read directly from the disk. Disk I/O is not required to access the index, which enables predictable performance. Such a design is possible because the read latency characteristic of I/O in SSDs is the same, regardless of whether it is random or sequential. For such a model, optimizations described are used to avoid the cost of a device scan to rebuild indexes.

This ability to do random read I/O comes at the cost of a limited number of write cycles on SSDs. In order to avoid creating uneven wear on a single part of the SSD, Aerospike does not perform in-place updates. Instead, it employs a copy-on-write mechanism using large block writes. This wears the SSD down evenly, which, in turn, improves device durability. Aerospike bypasses the Operating System's file system and instead uses attached flash devices directly as a block device using a custom data layout.

When a record is updated, the old copy of the record is read from the device and the updated copy is written into a write buffer. This buffer is flushed to the storage when completely full.

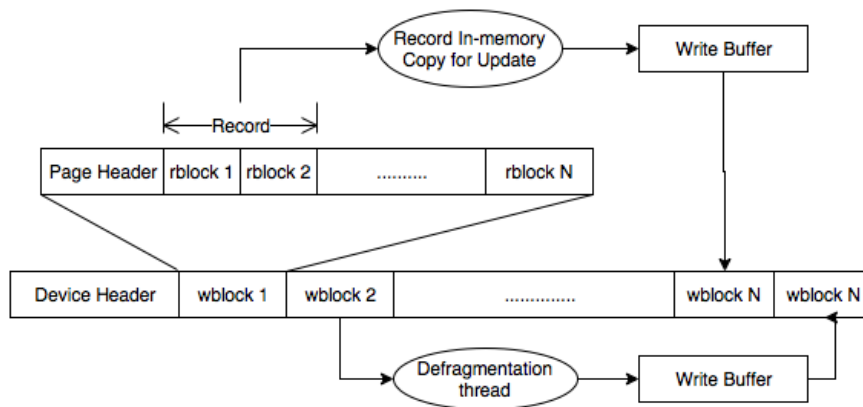


Figure 4. Aerospike Storage Layout

The unit of read, RBLOCKS, is 128 bytes in size. This increases the addressable space and can accommodate a single storage device of up to 2TB in size. Writes in units of WBLOCK (configurable, usually 1MB) optimize disk life.

Aerospike operates on multiple storage units of this type by striping the data across multiple devices based on a robust hash function; this allows parallel access to the data while avoiding any hot spots.

Note that SSDs can store an order of magnitude more data per node than DRAM. The IOPS supported by devices keep increasing; for instance, NVMe drives can now perform 100K IOPS per drive. Many 20-30 node Aerospike clusters use this setup and run millions of operations/second 24x7 with sub-millisecond latency.

Hybrid Memory Systems are multi-threaded, massively parallel systems.

Aerospike's real-time engine can scale up to millions of transactions per second at sub-millisecond latencies per node.

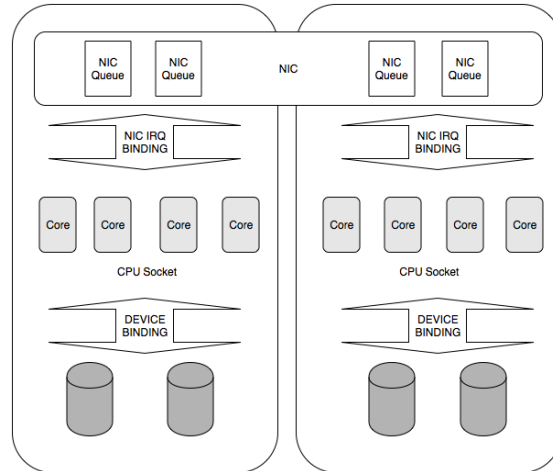


Figure 5. Aerospike Multi-Core Architecture

For data structures that need concurrent access such as indexes and global structures, there are three potential design models:

1. Multi-threaded data structures with a complex nested locking model for synchronization, e.g., step lock in a B+tree
2. Lockless data structures
3. Partitioned, single-threaded data structures

Aerospike adopts the third approach in which all critical data structures are partitioned, each with a separate lock, reducing contention across partitions. Access to nested data structures like index trees does not involve acquiring multiple locks at each level; instead, each tree element has both a reference count and its own lock. This allows for safe and concurrent read, write, and delete access to the tree without holding multiple locks. These data structures are carefully designed to make sure that frequently and commonly accessed data has locality and falls within a single cache line in order to reduce cache misses and data stalls. For example, the index entry in Aerospike is exactly 64 bytes, the same size as a cache line.

In addition to basic key value store operations, Aerospike supports batch queries, scans, and secondary index queries. Scans are generally slow background jobs that walk through the entire data set. Batch and secondary index queries return a matched subset of the data and, therefore, have different levels of selectivity based on the particular use case. Balancing throughput and fairness with such a varied workload is a challenge, but it can be achieved by following three major principles:

1. Partition jobs based on their type: Each job type is allocated its own thread pool and prioritized across pools. Jobs of a specific type are further prioritized within their own pool.

2. Effort-based unit of work: The basic unit of work is the effort needed to process a single record, including lookup, I/O, and validation. Each job is composed of multiple units of work, which define its effort.
3. Controlled load generation: The thread pool has a load generator that controls the rate of generation of work. The threads in the pool perform the actual work.

Aerospike uses cooperative scheduling whereby worker threads yield CPU for other workers to finish their job after x units of work. These workers have CPU core and partition affinity to avoid data contention when parallel workers access certain data.

In Aerospike, concurrent workloads of a certain basic job type are generally run on a first-come, first-served basis to ensure low latency for each request. The system also needs the ability to make progress in workloads that are long-running and sometimes guided by user settings and/or the application's ability to consume the result set, such as scans and queries. For such cases, the system dynamically adapts and shifts to round-robin task scheduling, in which many tasks that are run in parallel are paused and re-scheduled dynamically based on the progress they can make.

Rather than depend on the programming language or on a runtime system, Aerospike handles all its memory allocation natively. To this effect, Aerospike implements various special-purpose slab allocators to handle different object types within the server process. Aerospike's in-memory computing solution effectively leverages system resources by keeping the index packed into RAM.

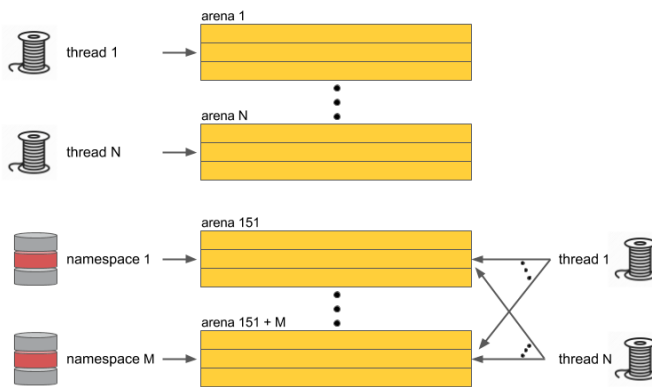


Figure 6. Aerospike's Memory Arena Assignment

Specifically, as shown in the above figure, by grouping data objects by namespace into the same arena, the long-term object creation, access, modification, and deletion pattern is optimized, and fragmentation is minimized.

Hybrid Memory Architecture significantly improves uptime and availability, without manual DevOps processes.

Aerospike's dynamic cluster management handles node membership and ensures that all the nodes in the system come to a consensus on the current membership of the cluster. Events such as network faults and node arrival or

departure trigger cluster membership changes. Such events can be both planned and unplanned. Examples of such events include randomly occurring network disruptions, scheduled capacity increments, and hardware/software upgrades. The specific objectives of the cluster management subsystem are to:

1. Arrive at a single consistent view of current cluster members across all nodes in the cluster,
2. Automatically detect new node arrival/departure and seamless cluster reconfiguration,
3. Detect network faults and be resilient to such network flakiness,
4. Minimize time to detect and adapt to cluster membership changes.

Aerospike distributes data across nodes as shown in *Figure 7* below. A record's primary key is hashed into a 160-byte digest using the RipeMD160 algorithm, which is extremely robust against collisions. The digest space is partitioned into 4096 non-overlapping "partitions". A partition is the smallest unit of data ownership in Aerospike. Records are assigned partitions based on the primary key digest. Even if the distribution of keys in the key space is skewed, the distribution of keys in the digest space—and therefore in the partition space—is uniform. This data partitioning scheme is unique to Aerospike. It significantly contributes to avoiding the creation of hotspots during data access, which helps achieve high levels of scale and fault tolerance.

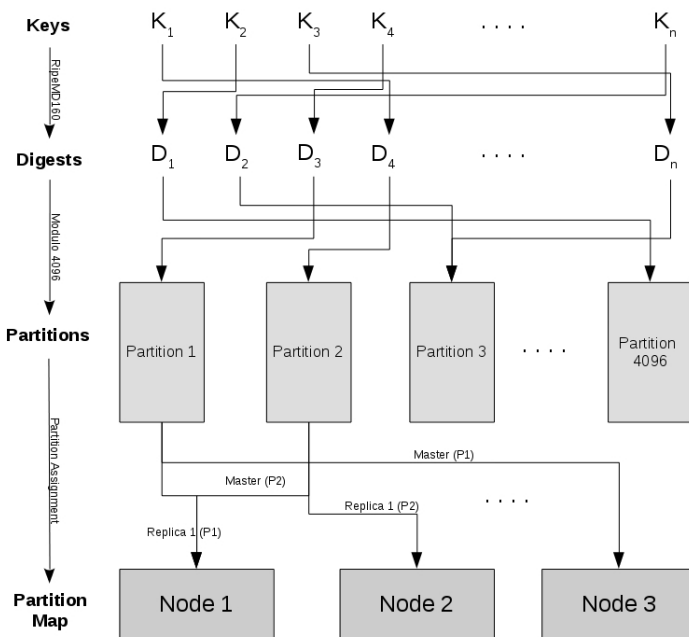


Figure 7. Aerospike's Data Distribution

Aerospike indexes and data are colocated to avoid any cross-node traffic when running read operations or queries. Writes may require communication between multiple nodes based on the replication factor. Colocation of index and data, when combined with a robust data distribution hash function, results in uniformity of data distribution across nodes. This, in turn, ensures that:

1. Application workload is uniformly distributed across the cluster,
2. The performance of database operations is predictable,
3. Scaling the cluster up and down is easy, and
4. Live cluster reconfiguration and subsequent data rebalancing is simple, non-disruptive and efficient.

The Aerospike partition assignment algorithm generates a replication list for every partition. The replication list is a permutation of the cluster succession list. The first node in the partition's replication list is the master for that partition, the second node is the first replica, the third node is the second replica, and so on. The partition assignment algorithm has the following objectives:

1. To be deterministic so that each node in the distributed system can independently compute the same partition map
2. To achieve uniform distribution of master partitions and replica partitions across all nodes in the cluster
3. To minimize movement of partitions during cluster view changes.

Hybrid Memory Architecture automatically distributes both data and traffic to all the nodes in a cluster.

Clients automatically do load balancing for improved performance and correctness.

The client needs to know about all the nodes of the cluster and their roles. In Aerospike, each node maintains a list of its neighboring nodes. This list is used for the discovery of the cluster nodes. The client starts with one or more seed nodes and discovers the entire set of cluster nodes. Once all nodes are discovered, the client needs to know the role of each node. Each node owns a master or replica for a subset of partitions out of the total set of partitions. This mapping from partition to node (partition map) is exchanged and cached with the clients. Sharing of the partition map with the client is critical in making client-server interactions extremely efficient. This is why, in Aerospike, there is single-hop access to data from the client. In steady state, the scale-out ability of the Aerospike cluster is purely a function of the number of clients or server nodes. This guarantees the linear scalability of the system as long as other parts of the system-like network interconnect-can absorb the load.

Each client process stores the partition map in its memory. To keep the information up to date, the client process periodically consults the server nodes to check if there are any updates. It does this by checking the version that it has stored locally against the latest version of the server. If there is an update, it performs requests for the full partition map.

For each of the cluster nodes, at the time of initialization, the client creates an in-memory structure on behalf of that node and stores its partition map. It also maintains a connection pool for that node. All of this is torn down when the node is declared to be down. The setup and tear-down is a costly operation. Also, in case of failure, the client needs to have a fallback plan to handle the failure by retrying the database operation on the same node or on a different node in the cluster. If the underlying network is flaky and this happens repeatedly, it can end up degrading the performance of the overall system. This leads to the need to have a balanced approach to identifying cluster node health. Aerospike uses the following strategies to achieve this balance.

The client's use of transaction response status code alone as a measure of the state of the DBMS cluster is a suboptimal scheme. The contacted server node may temporarily fail to accept the transaction request. Or it could be that there is a transient network issue, while the server node itself is up and healthy. To discount such scenarios, clients track the number of failures encountered by the client on database operations at a specific cluster node. The client drops a cluster node only when the failure count (a.k.a. "happiness factor") crosses a particular threshold. Any successful operation to that node will reset the failure count to zero.

Inconsistent networks are often tough to handle. One-way network failures (A sees B, but B does not see A) are even tougher. There can be situations where the cluster nodes can see each other but the client is unable to see some cluster nodes directly (say, X). In these cases, the client consults all the nodes of the cluster visible to itself and sees if any of these nodes has X in their neighbor list. If a client-visible node in the cluster reports that X is in their neighbor list, the client does nothing. If no client-visible cluster nodes report that X is in their neighbor list, the client will wait for a threshold time and then permanently remove the node by tearing down the data structures referencing the removed node. Over many years of deployments, we have found that this scheme greatly improved the stability of the overall system.

Lower TCO, creating a competitive advantage and enhanced business value.

A hybrid memory architecture, by virtue of eliminating the external caching layer, reduces server footprints by a factor of three or more, and decreases hardware costs by a factor of six or more.

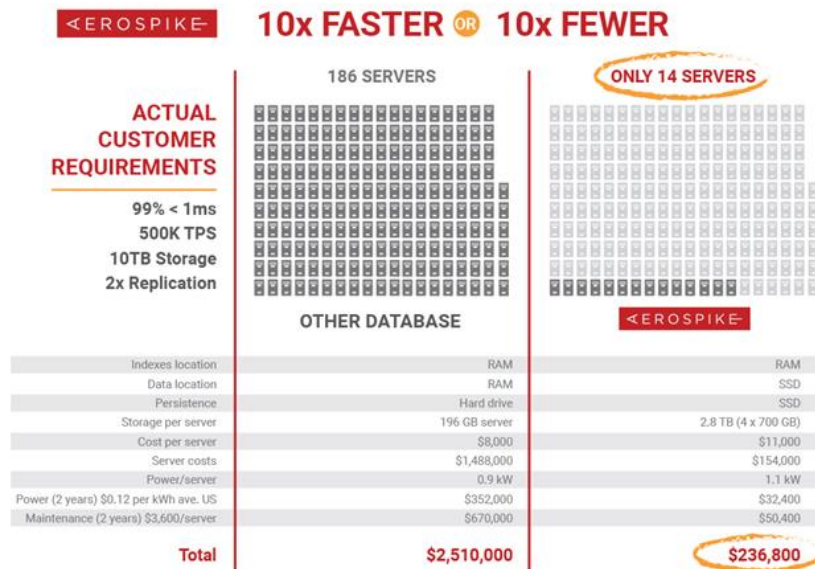


Figure 8. The Aerospike Difference, based on a typically priced 10 TB system (with 2x replication) handling a total of 500K transactions per second (TPS).

The above ROI was developed in conjunction with a large financial services firm. They were running Gemfire cache on top of a DB2 database. As their customer base, dataset, and data sources mushroomed, they forecasted the need to grow their server farm to over 1,000 servers just to keep up. By switching to Aerospike and its hybrid memory architecture, they were able to run the system - and accommodate their growth - on only 14 servers in a single cluster. That's a saving of a minimum of \$2 million in the first year. This figure does not take into account the savings from not having to manage the cache, recovering from cache stampede, or reacting to other unplanned outages.

The Path Forward

If you've reached the conclusion that your current architecture is in need of an upgrade, then it's time to investigate how hybrid memory architecture may be right for you. The good news is that you don't have to do this investigation on your own. Our solution architects will be happy to analyze your current architecture, as well as your use case, data volumes, growth forecasts, and SLAs to determine if Aerospike is a good fit for you; if it is, they'll help you plot a path forward.

We realize that change, even to better systems, is non-trivial. The real question is how to effect that change to optimize resources with the least impact on operations. Happily, having taken many customers through the process we are well-versed in how to ensure your success. Our solutions architects will have more technical detail on whether Aerospike and hybrid memory architecture are right for your use cases. In the meantime, following are two scenarios we commonly see.

From Cache + RDBMS

Some of our customers have chosen the big step of replacing both their external caching layer *and* their RDBMS. Here, the application simply reads and writes all data from an Aerospike cluster. Rather than maintaining separate connections to the cache and a relational database, applications simply write to an Aerospike cluster. This gives you all the benefits of an improved cache, plus:

- Simpler application development. Developers need to only worry about interfacing with an Aerospike cluster, rather than reading and writing from both a cache and a relational database.
- Simpler operations of a single (self-managing) Aerospike cluster as opposed to a relational database and many independent cache instances.
- Aerospike provides the same performance and latency of an external caching layer, while being much easier to manage on significantly fewer nodes.

If you have a cache and an RDBMS, one set of our customers have simply replaced their cache with Aerospike and kept their relational database. Using Aerospike as a better cache is an entirely legitimate use case and eliminates the maintenance and uptime issues you are currently experiencing with a cache.

Without changing your current architecture, this gives you:

- Immediate failover in the event of a lost node.
- Persistence of the data on disk.
- A choice of RAM or low-cost SSD to store the data.

From Cache + First-Generation NoSQL

In the quest for scalability and performance, many customers took the interim step of replacing their RDBMS with a first-generation NoSQL database along the way. Sadly, they too, ran into the issues of uncontrolled server growth, application complexity, cache stampede, *et al.* They, too, chose to replace both their external caching layer *and* their NoSQL database. Again, the application simply reads and writes all data from Aerospike clusters while keeping the index in memory.

- Simpler application development. Developers need to only concern themselves with interfacing with an Aerospike cluster rather than reading and writing from both a cache and a relational database.

- Simpler operations of a single (self-managing) Aerospike cluster as opposed to a NoSQL database and many independent cache instances.
- Aerospike provides the same performance and latency of an external caching layer and the scalability of the NoSQL database, while being much easier to manage on significantly fewer nodes.

For additional insight on the challenges of managing a first-generation NoSQL database, please read our white paper titled: [“Five Signs You’ve Outgrown Cassandra \(And What to Do About It\).”](#)

Summary

The days of the external caching layer as the *de facto* architecture for performance and scale are now long gone. The growth in data and the continued downward pressure on response times have rendered this technology obsolete for many use cases. Analyst groups from Gartner and to Forrester; to 451 Group all agree that hybrid memory architecture is the key to current and future success for businesses going through digital transformation.

If you’re experiencing any of the above symptoms and are concerned that your current cache architecture may be obsolete, contact Aerospike at info@aerospike.com. Ask for a free one-on-one consultation with our Solution Architect staff to evaluate your specific situation.

About Aerospike

Aerospike is the world's leading enterprise-grade, internet scale, key-value store database whose patented Hybrid Memory Architecture™ enables digital transformation by powering real-time, mission critical applications and analysis. Only Aerospike delivers strong consistency, predictable high performance and low TCO with linear scalability. Serving the financial services, banking, telecommunications, technology, retail/ecommerce, adtech/martech and gaming industries, Aerospike has proven customer deployments with zero downtime for seven years running. Recognized by industry analysts as a visionary and leader, Aerospike customers include Nielsen, Williams Sonoma, Kayak, Neustar, Bharti Airtel, ThreatMetrix, InMobi, Applovin and AppNexus. Aerospike is based in Mountain View, CA, and is backed by New Enterprise Associates, Alsop Louie Partners, Eastward Capital Partners, CNTP and Silicon Valley Bank.