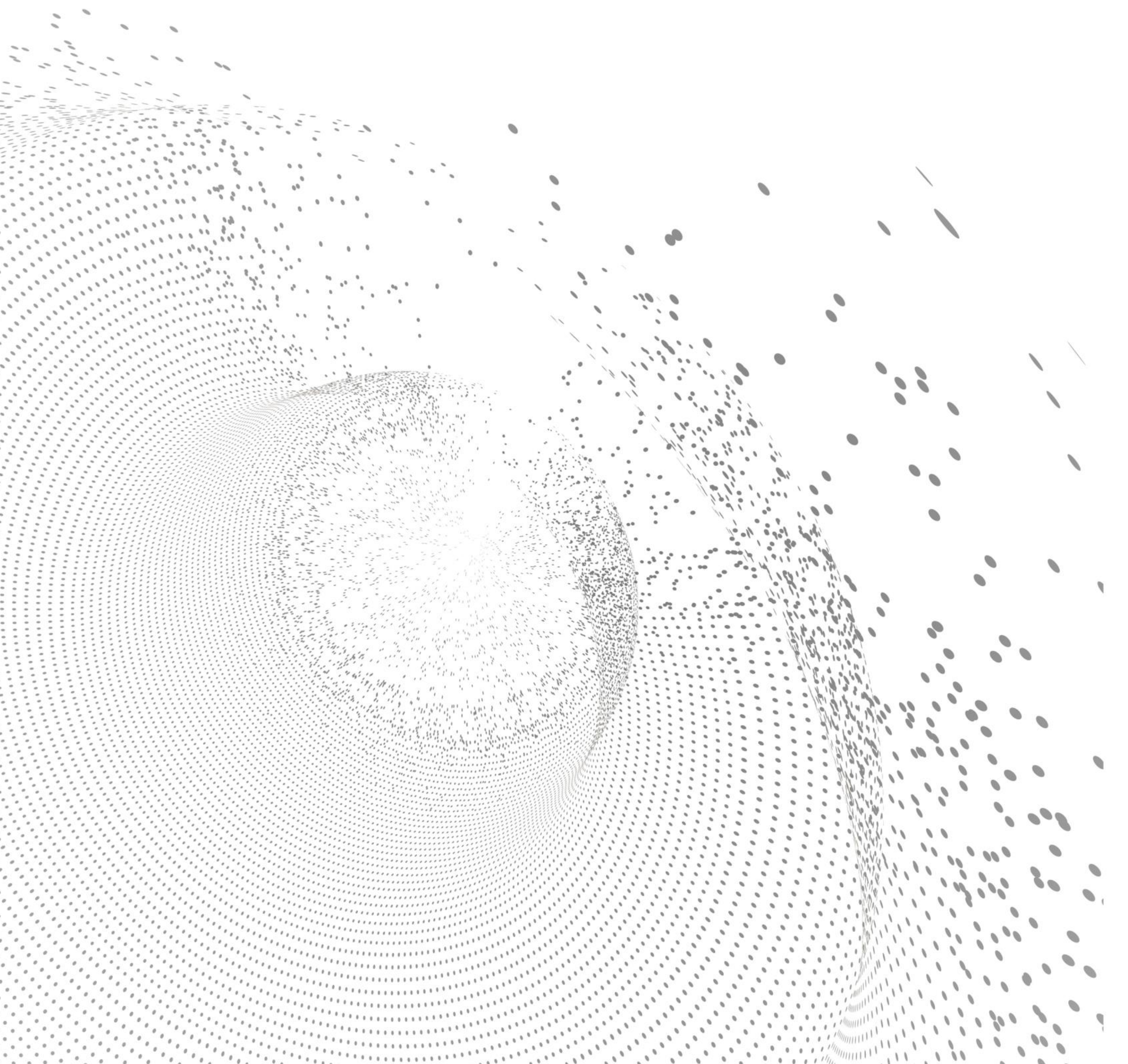
AEROSPIKE

Five Signs You Have Outgrown Redis



Contents

Executive Summary 3
Five Signs 4
Sign #1: You're worried about TCO 4
Sign #2: You need scalability and elasticity 4
Sign #3: You need persistence with high performance 4
Sign #4: You need strong data consistency 4
Sign #5: You need manageability and operational ease at scale 4
Exploring the five signs in depth 4
Sign #1: You're worried about TCO 5
Comparative scenario: Redis and Aerospike 5
Comparative scenario: RocksDB and Aerospike 7
Sign #2: You need scalability and elasticity 9
Redis on Flash (ROF) 9
Compactions and write amplification 9
Elasticity 10
Sign #3: You need persistence with high performance 12
Snapshots 12
Append-only files (AOF) 12
Users weigh in 13
Sign #4: You need strong consistency 14
Eventual consistency can lead to data loss and stale reads 15
Eventual consistency increases development complexity and operational risk 16
Sign #5: You need manageability and operational ease as your data volumes grow 18
Are you ready for the future with Redis? 18
Scalability strategies with Redis 19
Coping with failure 19
Synchronizing data across a multi-layered architecture 20
Aerospike: The Enterprise-Grade NoSQL Database 21
Low total ownership costs (TCO) 21
Scalability 22
Persistence 22
Strong consistency 23
Operational simplicity at scale 23
Summary 24
Appendix A: TCO configuration details 25
Amazon EC2 instances 25
Redis sizing 25
Aerospike sizing 27
About Aerospike 28

Executive Summary

It's an all-too familiar situation: at first, Redis users find the system easy to deploy and use. But then their workloads grow and their data volumes increase, and things start to change quickly. That's when many organizations discover that the ownership costs, scalability, and operational complexity of Redis to be much worse than they'd ever imagined. Faced with IT budget overruns, service-level agreement (SLA) violations, and delayed application rollouts, many Redis users end up looking for a better alternative. That's when Aerospike can help.

Aerospike is a NoSQL key-value store that offers exceptional runtime performance for mixed read/write workloads at scale. Aerospike also provides extremely high availability, near-linear scalability, and strong data consistency. But don't take our word for it -- consider what our customers have to say. For example, explore how one firm[1] replaced Redis with Aerospike for a critical production application so that it could "flawlessly" handle its massively growing data set. That company now considers its Aerospike-based infrastructure to be one of its "most stable and resilient systems." But perhaps even more remarkable is the cost reduction many firms enjoy after switching to Aerospike. Indeed, one organization[2] specializing in real-time fraud detection cut costs by 85% after migrating from Redis to Aerospike. Surprised? Don't be. Substantial savings in total cost of ownership (TCO) is common, thanks to Aerospike's highly efficient distributed database that features a Hybrid Memory Architecture™ (HMA).

Yet cost efficiency is only one reason why firms in financial services, telecommunications, technology, retail, and other industries have come to rely on Aerospike to power their mission-critical operational applications. Extreme uptime, support for self-forming and self-healing clusters, and predictable, ultra-fast access to billions of records in databases of 10s - 100s of TB are a few others. You'll have a chance to learn more about Aerospike later. But for now, let's turn back to the primary topic of this paper and explore five common signs that you may have outgrown Redis.

[1] https://medium.com/appsflyer/how-we-built-the-worlds-biggest-real-time-db-of-mobile-devices-33c1c3921a48

[2] https://www.aerospike.com/resources/videos/adjust-scaling-platform-for-real-time-fraud-detection-without-breaking-bank/

AEROSPIKE

Five Signs

What are five signs that your firm may have outgrown Redis?

Sign #1: You're worried about TCO
- Are growing business demands stressing your Redis cluster?
- Are you concerned about excessive DRAM, tuning, and operational expenses?
- Can your IT budget accommodate the rapid increase in total cost of ownership (TCO)?

Sign #2: You need scalability and elasticity
- Do you need to expand your cluster to manage 10s - 100s of TBs or more?
- Have you struggled to maintain high performance over rapidly growing data sets with Redis on Flash (ROF) and other options?
- Are seasonal workload peaks hard to handle and causing service interruptions?
- Do you need a simple way to grow or shrink your cluster as business needs change?

Sign #3: You need persistence with high performance
- Do you need to move beyond a purely in-memory solution?
- Are snapshots proving impractical as your data volumes grow?
- Are append-only files (AOF) slowing runtime performance?
- Do you wish persistence was fast, reliable, and straightforward?

Sign #4: You need strong data consistency
- Do you need to prevent applications from reading stale data?
- Are you concerned about data loss?
- Is eventual consistency insufficient for your applications?

Sign #5: You need manageability and operational ease at scale
- Are you worried about operational complexity as you scale your database to match growing business needs?
- Are you struggling with system stability?
- Are you having problems with high availability and automatic failovers?
- Is your multi-layered architecture introducing data synchronization and system management issues?

Exploring the five signs in depth

As data volumes increase at an unprecedented rate, firms face ever-greater challenges: deliver new applications faster, apply analytical technologies (such as machine learning) to hundreds of terabytes data (or more), provide a reliable and engaging user experience, drive digital transformations, and more. How do these map to the signs that you've outgrown Redis?

Sign #1: You're worried about TCO

It's an all-too common scenario: you got off to a quick start with Redis. It was easy to use when your data volumes were modest, and your workloads weren't too demanding. But as your needs grew, things got more complicated. You found yourself deploying more nodes, more memory, and more manpower to keep up with evolving business requirements. And that translated into higher and higher total cost of ownership (TCO).

It doesn't have to be that way. Consider the experience of Adjust,[3] a firm that specializes in verifying billions of advertising impressions. Originally a Redis user, the firm faced growing data volumes, demanding performance requirements, and high availability needs that stressed its Redis infrastructure. After migrating from Redis to Aerospike, the firm shrank its footprint from 40 nodes to 6 and reduced its costs by 85%.

That's not an anomaly. Aerospike users routinely save 5 times or more in TCO compared with alternate solutions. How's that possible? In short, Aerospike leverages cost-efficient hardware resources in ways that other systems -- like Redis -- don't. Furthermore, Aerospike automates or simplifies many administrative tasks with self-managing features, and it reduces development complexity with support for strong, immediate data consistency and other traditional database capabilities.

But let's return to the original topic of TCO. Understandably, when any vendor claims to deliver dramatic cost savings, you might get a little skeptical. So, let's walk through a sample use case derived from a firm using Aerospike in production and compare costs for Redis and Aerospike.

Comparative scenario: Redis and Aerospike

Consider a 3-year scenario involving a 40TB database. In the first year, the database needs to store 17 billion primary objects of 500 bytes each as well as 17 billion replica objects. Assuming a data growth rate of 25% per year, the database will manage 21.3 billion primary objects in the second year and 26.6 billion primary objects in the third year. Furthermore, during this 3-year period, the workload will be evenly split (50/50) between reads and writes with a random distribution of operations across the key set. The writes will be based on the reads so batch operations cannot be used.

Given that scenario, we compared the costs of using Amazon Elastic Compute Cloud (EC2) for Redis and Aerospike. We even configured Redis three different ways to explore possible cost savings of different approaches. The upshot is that Aerospike delivered substantial cost savings beginning in Year 1 regardless of the Redis configuration used. By Year 3, Aerospike's cumulative savings ranged from $2.2 million to $8.7 million depending on the Redis configuration. And those savings don't reflect hard-to-quantify benefits such as greater operational ease and scalability, lower failure rates, and more.

Perhaps that sounds too good to be true, so let's step through the details. We included three different Redis configurations to reflect solutions commonly considered by Redis users:

- in-memory processing with persistence using append-only files (AOF),
- in-memory processing only,
- Redis on Flash (ROF) with 50% of data in memory and 50% of data on flash drives.

These Redis configurations each required different combinations of high-memory capacity servers and non-volatile memory express solid state drives (NVMe SDDs), so the underlying Amazon EC2 footprints naturally differed. For example, deploying Redis with an in-memory only configuration eliminated the need for SSDs but increased volatile memory and CPU requirements. We used only one Aerospike configuration in this study

[3] https://www.aerospike.com/resources/videos/adjust-scaling-platform-for-real-time-fraud-detection-without-breaking-bank/

-- a common production configuration with indexes in volatile memory and user data on NVMe SSDs. Full details on the configurations and sizings for this TCO analysis are included in Appendix A.

Table 1 summarizes the costs for each configuration of Redis as well as for Aerospike. The bottom three rows calculate the savings in operational expenses (OpEx) achieved with Aerospike when compared with the Redis configurations.

	EC2 instance/ Yearly $	Year 1 servers	Year 1 $	Year 2 servers	Year 2 $	Year 3 servers	Year 3 $	Total $
Aerospike	I3. 8xlarge / $13,928	12	$167,136	15	$208,920	19	$264,632	$640,688
Redis with AOF persistence	R5d. 12xlarge / $17,833	138	$2,460,954	172	$3,067,276	214	$3,816,262	$9,344,492
Aerospike savings			$2,293,818		$2,858,356		$3,551,630	$8,703,804
Redis memory only	R5. 12xlarge / $15,576	69	$1,074,744	86	$1,339,536	107	$1,666,632	$4,080,912
Aerospike savings			$907,608		$1,130,616		$1,402,000	$3,440,224
Redis on Flash (50% memory / 50% SSD)	I3. 8xlarge / $13,928	54	$752,112	68	$947,104	85	$1,183,880	$2,883,096
Aerospike savings			$584,976		$738,184		$919,248	$2,242,408

Table 1: Cost comparison of Redis and Aerospike

As you'll note, Aerospike requires a substantially smaller footprint than any of the Redis configurations. This drives considerable cost savings. Aerospike's smaller footprint also helps reduce operational complexity and node failure rates -- topics we'll cover later in this paper. For now, though, let's take a moment to discuss each Redis configuration further.

Redis with AOF persistence

We'll start with the Redis configuration that used append-only files (AOF) for persistence. This is arguably the Redis configuration that's most comparable to Aerospike, calling for both systems to store user data on SDDs and use volatile memory for data processing (albeit in different ways). As you can see, Aerospike reduced cluster size by 91% and cut costs by a whopping $8,703,804 compared to Redis with persistence. As Table 1 indicates, this Redis configuration used Amazon R5d.12xlarge instances at a yearly upfront cost of $17,833 per instance. We calculated the number of required servers (Amazon instances) based on Redis's recommendations[4] that the minimum size of persistent storage should be 3x the node's RAM and the recommended size should be >= 6x the node's RAM. Redis needed 138 instances for the first year, 172 for the second, and 214 for the third. Compare that to what Aerospike required: 12, 15, and 19 Amazon

[4] https://docs.redislabs.com/latest/rs/administering/designing-production/hardware-requirements/

I3.8xlarge instances (at $13,928 yearly per instance) for these same years. The differences in cluster sizes -- and associated costs -- is remarkable. . . . In case you're wondering if using a different EC2 instance type for Aerospike yielded an unfair cost advantage, think again. If we used I3.8xlarge instances for Redis, its costs would have soared to more than $10 million over the 3-year period due to the number of instances of that type that would need to be deployed to support the workload.

Redis memory only

Given how expensive Redis with persistence turned out to be for this scenario, perhaps you're curious about an in-memory only configuration of Redis. After all, many Redis users deploy the system in this manner. As it turns out, the 3-year cumulative TCO of using Redis *was* lower without persistence than with it. That's the good news. The bad news (for Redis users) is that it was still $3.4 million higher than Aerospike's. Referring again to Table 1, you'll see that we used R5.12xlarge memory-optimized AWS EC2 instances for this Redis configuration. This type of EC2 instance is cheaper than the type we used in the previous configuration, which employed SSDs for persistence. However, even with Redis configured for in-memory processing only, our analysis revealed that Aerospike delivered nearly $1 million in savings in the Year 1 and more than $3.4 million in cumulative savings over 3 years. And that's with Aerospike still providing full data durability with its built-in persistence -- a capability outside the scope of this Redis configuration.

Redis on Flash (50% memory / 50% SSD)

The last configuration we'll consider is Redis on Flash (ROF). Although ROF is not recommended for random workloads,[5] we included it in this TCO analysis because firms seeking high scalability with Redis often evaluate it. (We'll discuss the scalability challenges of ROF in a later section; this section just focuses on its TCO implications.) As Table 1 indicates, both Aerospike and Redis used EC2 I3.8xlarge instances. To simplify our comparison, we omitted the additional memory required for ROF meta-data and its underlying storage engine (RocksDB) from our Redis sizing estimates. Of course, production users wouldn't have this luxury. Finally, the amount of memory needed for ROF is determined at configuration time by administrators, who must specify how much data to store in memory and how much data to store on flash (in RocksDB). Our study specified a 50 / 50 split. The upshot? Aerospike delivered a 3-year cost savings of more than $2.2 million. Of course, you could alter the ratio of data kept in memory versus flash with ROF. But Redis Enterprise documentation[6] recommends keeping at least 20% in RAM. Even if we presumed a configuration with 20% of the data in memory and 80% in flash, Aerospike would still lower costs by 45%. And that's without taking ROF meta-data and RocksDB memory needs into account. Of course, performance is another issue. With a random workload and 80% of data in RocksDB, performance would suffer greatly, which is why we used a more realistic 50/50 split for our ROF configuration.

Comparative scenario: RocksDB and Aerospike

Since we just brought up the subject of RocksDB performance in our TCO analysis, let's explore that topic a bit more here. We'll discuss RocksDB and ROF in more detail when we examine scalability issues in the next section. But for now, let's turn to the results of a benchmark we ran on Aerospike and RocksDB using a single Dell PowerEdge R730xd with 512GB of memory and 4 Micron 9100 3.2TB drives. This benchmark illuminates the performance of RocksDB running a random mixed workload of 50% reads and 50% writes.

As Table 2 shows, Aerospike's throughput was 4 times that of RocksDB at the point the drives were saturated.

[5] https://redislabs.com/redis-enterprise/technology/redis-on-flash/
[6] https://redislabs.com/redis-enterprise/technology/redis-on-flash/

	Obj Size	Keys	Workload	TPS
RocksDB	1200 B	4.7 billion	50/50	203K
Aerospike	1200 B	4.7 billion	50/50	802K

Table 2. Transactions per second (TPS) comparison of RocksDB and Aerospike

With this random heavy read/write workload RocksDB processed 203K operations a second while Aerospike processed 802K operations a second, demonstrating the power of Aerospike's Hybrid Memory Architecture™ (HMA) over databases that use log-structured merge (LSM) trees like RocksDB. In short, RocksDB uses computing resources less efficiently than Aerospike. For example, it periodically initiates a compaction process when storing new or changed user data; this amplifies write activities, leading to slower performance and unpredictable latencies. As a result, if you want your RocksDB environment to achieve comparable performance to Aerospike, you'll need to provision significantly more computing resources -- and pay nearly 3 times more, as shown in Table 3.

	Server cost	Memory	Total Cost
RocksDB	2x4300	4.5 TB($679x18)	$20,822
Aerospike	1x4300	512 GB($199x16)	$7,484

Table 3. Cost comparison of RocksDB and Aerospike

And that's just part of the story. As your data set grows larger and your workload becomes more demanding, your server costs will increase more quickly with RocksDB. Furthermore, Redis on Flash (ROF), which uses RocksDB, introduces additional costs that we haven't even included in the above comparison.

So how might you cope with this situation? Minimizing the workload on RocksDB is one option. For example, you could reduce the amount of data managed by RocksDB or attempt to access your data less frequently. Indeed, the recommendations for sizing ROF shards minimizes the amount of data stored in -- and, therefore, the load placed on -- RocksDB. Specifically, the documentation[7] recommends storing 50GB of data per shard. This size reduces the amount of compaction and write amplification per instance of RocksDB and keeps a larger portion of the RocksDB data in page cache. But this approach comes at a price. Managing a large data set requires many shards, which will drive you to deploy more and more server nodes as the number of shards you need expands beyond the number of CPUs on a node.

It's worth noting that ROF doesn't provide persistence. By contrast, Aerospike (as configured in this example) delivers comparable performance to RocksDB *and* persistence. This is significant, because adding Redis persistence can almost double the need for RAM on heavy write workloads. More memory is needed for the persistence process, and more storage is needed for your data. Indeed, to reach your performance targets with ROF, you may need to double memory and triple your storage beyond what RocksDB needs. To put this another way, we can reasonably extrapolate that ROF will perform much worse -- perhaps 6 times worse -- than Aerospike on the same server footprint.

[7] https://docs.redislabs.com/latest/rs/concepts/high-availability/clustering/

Sign #2: You need scalability and elasticity

The business mandate for super-fast -- and highly targeted -- decisions is driving many organizations to capture and analyze increasingly large data volumes to maintain (or gain) competitive advantage. Workloads that once spanned hundreds of gigabytes or even a few terabytes often prove too modest for modern-day business needs that now require processing tens to thousands of terabytes without running up exorbitant costs or sacrificing strong runtime performance. Many Redis users find such requirements to be extremely challenging, in part because Redis is a single-threaded system.

A common approach to scaling Redis is to add more nodes and, therefore, more dynamic RAM. On the surface, this makes sense -- after all, Redis was originally designed as an in-memory data store. However, DRAM is expensive, and creating large clusters increases administrative complexity, operational costs, and failure rates. It can even lead to some extreme engineering efforts, as the case of one particularly ambitious Redis user -- Twitter[8] -- illustrates. Several years ago, Twitter described its Redis deployment of 105TB of DRAM and 10,000+ instances and detailed how it forked Redis's code base to add features it needed for its production environment.

Very few firms can afford to make such an engineering investment and commit to years of retrofitting open source upgrades onto a custom code base. It's simply beyond their means. Indeed, it wasn't long after Twitter's disclosure that Redis on Flash (ROF) emerged in the market. We mentioned ROF briefly in the previous section on TCO, so let's take a closer look at ROF here with an emphasis on scalability issues.

Redis on Flash (ROF)

What is ROF? Simply put, it's Redis Enterprise integrated with RocksDB as a memory extender. And what is RocksDB? It's an embedded database often used as a storage engine for other databases. However, ROF data stored in RocksDB is ephemeral. If Redis Enterprise is restarted, RocksDB is restarted empty. Perhaps that's acceptable to you -- maybe you're willing to leverage other Redis options for persistence (which we'll discuss shortly). Let's consider, then, how ROF impacts memory requirements and performance.

Depending on how ROF is configured, "hot" data is kept in memory through page caches. Furthermore, ROF keeps all metadata and indexes in memory. So even with ROF, Redis users often find themselves buying additional memory to meet predictable performance and linear scalability goals. Consider the case of Whitepages, a firm that used ROF to implement its Global Identity Graph. A Whitepages blog makes it clear that memory is key to performance:

> *Redis on Flash bumps up the keys to RAM (Standard LRU) and hence after a good amount of traffic we did reach a stable state of over 97% RAM hit rate.*[9]

The latency achieved from the Whitepages example is due to memory -- not from RocksDB and flash. To scale performance, firms still have to scale memory, which drives up cost and mitigates the price/performance advantages that many clients associate with flash. Although we discussed TCO in the previous section, it's worth exploring some of the technical aspects of RocksDB to get some clues about how its design drives up costs. . . . .

Compactions and write amplification

RocksDB is based on the log-structured merge (LSM) tree algorithm. Performance of LSM-tree DBMSs like RocksDB depend heavily on memory usage, as its user base confirms. For example, in a talk at Carnegie

[8] http://highscalability.com/blog/2014/9/8/how-twitter-uses-redis-to-scale-105tb-ram-39mm-qps-10000-ins.html

[9] https://www.whitepages.com/blog/scaling-whitepages-with-redis-on-flash/

Mellon University[10], Igor Canadi, a Facebook RocksDB engineer, noted that Facebook configured its servers with quite a bit of memory to reduce the number of times a disk is read. Depending on the configuration, RocksDB uses memory for indexes, memtables, block cache, and page cache. This yields higher performance -- and higher costs.

Furthermore, LSM trees have inherent performance problems with mixed read/write workloads. To achieve reasonable read performance, nearly the entire tree needs to be kept in memory. Achieving predictable write performance can be challenging as well due to mandatory compactions of user data that occur periodically. After RocksDB flushes data from full memtables to immutable SST files on disk, a compaction process removes the older duplicate data from the SST files. This compaction process generates additional write operations behind the scenes, causing performance to slow.

A Github post[11] from Facebook's RocksDB engineering team explains how *leveled compaction* works in greater detail. It's worth noting that compacting files at the base level of the LSM tree (Level 0) must be done at the same time with a single thread, creating a bottleneck for write-intensive workloads and delaying compactions at other levels. Large sets of hot keys or random workloads can cause frequent compactions, resulting in more direct disk accesses, higher latency and slower performance.

In a talk at XLDBconf[12], Facebook engineer Siying Dong discussed write amplification, which can be as high as 40 with RocksDB. As write loads increase, RocksDB incurs more write amplification from compaction. This, in turn, can stall -- or even stop -- write activities. Tuning RocksDB to maintain consistent write performance isn't a simple task, as the RocksDB team at Facebook indicated:

> *Unfortunately, configuring RocksDB optimally is not trivial. Even we as RocksDB developers don't fully understand the effect of each configuration change.*[13]

Elasticity

Many industries experience some form of seasonality. Retailers and advertising technology vendors encounter bursts of business from mid-November through the end of December. Car dealers and manufacturers see sales spike during the summer. Online tax services see demand skyrocket from the end of January until April 15 in the USA. These peak seasons place additional load and stress on data systems. To stay competitive and profitable, firms need such systems to be scalable and flexible. As one DBMS vendor put it:

> *If a big chunk of annual sales happens during a peak season, profitability will depend on how quickly you scale up—to meet demand—and then scale down (just as quickly) in order to not incur extra expenses when sales return to 'normal'.*[14]

To put it another way, modern-day DBMSs must be *elastic*. As database consultant Craig Mullins wrote in a Dzone article,

> ***Elasticity** is the degree to which a system can adapt to workload changes by provisioning and deprovisioning resources in an on-demand manner, such that at each point in time the available resources match the current demand as closely as possible.*[15]

[10] https://youtu.be/jGCv4r8CJEI?t=1154
[11] https://github.com/facebook/rocksdb/wiki/Leveled-Compaction
[12] https://www.youtube.com/watch?v=Wbq6E71I97A
[13] https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide
[14] https://blogs.oracle.com/smb/scaling-up-scaling-down-how-to-cope-with-seasonal-sales
[15] https://dzone.com/articles/what-do-we-mean-by-database-scalability

Unfortunately, Redis requires upfront configuration decisions that inhibit elasticity. Consider these requirements cited in the Redis Enterprise documentation:

> *When enabling database clustering you can set the number of database shards. The minimum number of shards per database is 2 and the maximum depends on the subscription you purchased.*
>
> *Once database clustering has been enabled and the number of shards has been set, you cannot disable database clustering or reduce the number of shards. You can only increase the number of shards by a multiple of the current number of shards. For example, if the current number of shards was 3, you can increase to 6, 9, 12 and so on.*[16]

As a result, scaling Redis Enterprise for a peak period like Black Friday can be costly. Because the cluster can only be scaled out by a multiple of the current number of shards, this can lead to excess database provisioning and associated costs. Alternatively, users can recreate the database at a cost-appropriate level and incur the risk -- and overhead -- of running the existing database while creating and switching over to the new database.

After the peak period is over, deprovisioning resources can be problematic, too. Users can only add shards to a cluster -- not remove them. Therefore, nodes that remain after scaling down must handle the additional load from shards added during the scale out process. Keep in mind that an additional shard isn't just additional data. A shard consists of a process for the Redis instance. With persistence, an extra process for each shard will be spawned periodically to write data to an RDB file or compact data in an append-only file. If Redis on Flash (ROF) is used, there are even more processes for RocksDB and RocksDB compaction. As you might imagine, these added processes can adversely affect the performance of the smaller cluster.

Furthermore, the optional RediSearch module for secondary indexes imposes even more severe scalability restrictions:

> *An exception to this is a RediSearch Enterprise enabled database. Once you configure the number of shards, you can no longer change that value. If you need to scale your RediSearchEnterprise enabled database, you must create a new database at the new required size and replicate the current database to the new one.*[17]

For firms that need to readily scale their databases to support rapidly changing business demands, the approach just described seems unworkable.

Indeed, challenges with elasticity and cost-effective scalability have led various firms to evaluate Aerospike as an alternative to Redis. Consider Travel Audience, a digital advertising firm in Germany that needed to "stay flexible in order to scale our system based on the load" and began noticing "more and more issues while using Redis." The firm benchmarked Aerospike and Redis:

> *From both Read and Write benchmarks you can see a significant gap between Redis TPS (transactions per second) and Aerospike TPS:*
> ***430.000 writes per sec for Aerospike vs 130.000 writes per sec for Redis***
> ***and***
> ***480.000 reads per sec for Aerospike vs 133.000 reads per sec for Redis.*** [18]

Not surprisingly, the firm decided to move from Redis to Aerospike. In doing so, it deployed one Aerospike cluster of 3 nodes to manage data previously stored in 5 Redis databases.

[16] https://docs.redislabs.com/latest/rs/concepts/high-availability/clustering/
[17] https://docs.redislabs.com/latest/rs/concepts/high-availability/clustering/
[18] https://tech.travelaudience.com/aerospike-vs-redis-da49ee50d4b8

Sign #3: You need persistence with high performance

Published benchmarks for Redis are typically based on environments that maintain a single copy of user data and employ full DRAM instances, a configuration that differs from what many clients need in their production environments. If you need persistence, you may find that performance degrades significantly. As Brian Ip, a software engineer on the Square online database team, said during his 2017 Redisconf session:

> *The two Redis provided methods don't really meet up to our performance standards so we turn them off by default on all of our active masters.* [19]

Append-only files (AOF) and snapshots are Redis's two options for persistence, which Redis Enterprise documentation explains this way:

Redis Enterprise is a fully durable database. It supports the following data persistence mechanisms:

- *AOF (Append-Only File) data persistence – Every shard of a Redis database appends new lines to its persistent file in one of the following manners:*
  - *every second (fast but less safe)*
    or
  - *every write (safer and slower)*
- *Snapshot – The entire point-in-time view of the dataset is written to persistent storage, across all shards of the database. The snapshot time is configurable.* [20]

We'll explore each of these a bit further, starting with the snapshot method, which is the simpler of the two.

Snapshots

The snapshot method periodically writes data from memory to an RDB file by doing a copy-on-write fork. As noted earlier, snapshots persist the full data set at a given point in time.
This post on the Packt website describes some practical concerns about snapshots:

- *The periodic background save can result in significant loss of data in case of server or hardware failure.*
- *The fork() process used to save the data might take a moment, during which the server will stop serving clients. The larger the data set to be saved, the longer it takes the fork() process to complete.*
- *The memory needed for the data set might double in the worst-case scenario, when all the keys in the memory are modified while snapshotting is in progress.* [21]

If the prospect of data loss, slow performance, and increased memory requirements gives you pause, perhaps you're wondering about AOF . . . .

Append-only files (AOF)

The AOF method maintains a log of all the write operations for Redis. These writes are kept in a buffer and are periodically flushed to disk based on configuration settings. AOF also periodically performs a copy-on-write fork to compact the growing AOF file.

[19] https://youtu.be/8nyJPpbt50I?t=493
[20] https://redislabs.com/redis-enterprise/technology/durable-redis-2/
[21] https://hub.packtpub.com/implementing-persistence-redis-intermediate/

However, AOF isn't without its drawbacks. For example, the fsync rate (which essentially determines how frequently buffers are written to disk) can significantly slow performance. Didier Spezia, a software developer and Redis community contributor, commented on AOF with systematic fsync in this post on Stackoverflow*:*

> *The effect on performance is catastrophic. Any RDBMS or NoSQL engine with a write ahead log will perform much better than Redis in this configuration. This is because Redis is single-threaded. In this configuration, fsync are done directly in the event loop.* [22]

Another potentially troublesome area involves the fork() process for saving data. The server stops serving clients while fork() is active, restricting data availability and system responsiveness. As you might imagine, larger data sets require longer fork() execution times, exacerbating scalability issues.

Redis outlines various advantages and disadvantages of its persistence options in its documentation, ultimately offering this advice:

> *The general indication is that you should use both persistence methods if you want a degree of data safety comparable to what PostgreSQL can provide you. If you care a lot about your data, but still can live with a few minutes of data loss in case of disasters, you can simply use RDB alone. There are many users using AOF alone, but we discourage it since to have an RDB snapshot from time to time is a great idea for doing database backups, for faster restarts, and in the event of bugs in the AOF engine.* [23]

Users weigh in . . . .

Performance and persistence have created challenges for many Redis users. Consider the case of Wayfair, an online retailer firm that required "minimal" read latency and persistence for its customer data graph.

> *. . . we netted out adopting Aerospike when we took into account a number of features (reduced maintenance overhead, substantially lower hardware costs, and performance comparable to a pure in-memory cache) that made it virtually a drop in upgrade for persistent Redis company-wide.* [24]

And Wayfair isn't alone. IronSource, an adware company headquartered in Israel, migrated from Redis to Aerospike. As developer Gil Shoshan explained in a 2019 conference presentation,

> *Redis didn't have a real cluster (and) very soon we reached a point that Redis couldn't scale any more. We moved to Aerospike. Aerospike is developer friendly. The scale problem was the first to be solved.* [25]

But that's not all. Shoshan emphasized that Aerospike "enabled us to offer new business features to our product."

Finally, the chief architect of Hewlett Packard Enterprise's AI-Driven Big Data Solutions team, Theresa Melvin, presented in 2019 on AI at Hyperscale - How to Go Faster with a Smaller Footprint. Faced with the need to plan for emerging "exascale" (extreme scale) requirements -- e.g., ingesting more than 85PB (petabytes) of data per day, storing 2.6EB (exabytes) of raw data monthly for 5 - 10 years -- Melvin evaluated various NoSQL databases. Her team ultimately benchmarked several "pmem-aware" (persistent memory aware) offerings: Aerospike, Redis, RocksDB, and Cassandra. While Aerospike successfully completed the load test, Melvin observed that:

---

[22] https://stackoverflow.com/questions/14682470/redis-and-data-integrity
[23] https://redis.io/topics/persistence
[24] https://tech.wayfair.com/2019/04/ad-tech-customer-intelligence-engineering-at-wayfair/
[25] https://www.aerospike.com/resources/videos/summit19/company/ironsource/

Redis could not 'load' 500M records in the required timeframe. . . . (Redis) would have taken >2.5 days to load 500M records. [26]

After evaluating the full suite of test results, Melvin offered this "take away":

Aerospike's pmem-aware performance for a single node
- *300% increase over RockDB*
- *Between 270% and 1667% over Cassandra's*
  - *270% if Cassandra was able to sustain its 92K ops/sec*
  - *1667% per Cassandra's current 15K sustained r/u performance*

- *Aerospike's Cluster Footprint*
  - *2-nodes required for Strong Consistency*
  - *33% footprint reduction over nearly any other K/V store* [27]

Sign #4: You need strong consistency

Consistency defines the ordering of operations allowed in a distributed system during normal operations as well as during periods with system anomalies. Contemporary DBMSs typically offer different degrees of consistency (i.e., different guarantees) to satisfy the needs of different applications.

To illustrate how different users may require different levels of data reliability, Microsoft researcher Doug Terry provided an example in his paper on Replicated Data Consistency Explained Through Baseball.[28] Specifically, Terry contrasted the data reliability needs of a baseball umpire and a radio announcer: an umpire needs strong consistency to call the game correctly, but a radio announcer only needs eventual consistency to inform fans about the game.

In some situations, lack of strong consistency can be catastrophic for businesses. A prime example is Flexcoin[29], a Bitcoin bank that closed after $600,000 worth of Bitcoin was stolen because of consistency problems.

So, what does it really mean to have strong consistency? Here's a reasonable definition from an analysis of NoSQL databases:

Strong consistency means that all processes connected to the database will always see the same version of a value and a committed value is instantly reflected by any read operation on the database until it is changed by another write operation. . . .

Eventual Consistency is weaker and does not guarantee that each process sees the same version of the data item. Even the process which writes the value could get an old version during the inconsistency window. [30]

Eventually consistent systems follow multiple paths and may take time to converge on a value, as this Stackoverflow post explains:

---

[26] https://www.aerospike.com/resources/videos/summit19/company/hpe/
[27] https://www.aerospike.com/resources/videos/summit19/company/hpe/
[28] https://www.microsoft.com/en-us/research/wp-content/uploads/2011/10/ConsistencyAndBaseballReport.pdf
[29] https://www.businessinsider.com/r-bitcoin-bank-flexcoin-shuts-down-after-theft-2014-04
[30] https://wwwmatthes.in.tum.de/file/ikcuitkq8cpm/Publications/2010/Or10/Or10.pdf

Conflicts can arise, but nodes communicate [with] each other their changes to solve those conflicts, so in time they agree upon the definitive value. Thus, if no more changes are applied to the data for a certain period, then all nodes will agree in the data value (i.e. they will eventually agree) so readers of data will eventually see the same value that at some unspecified point in the future converge on the on an agreed upon value. [31]

For some applications, that's sufficient. But for others, it clearly isn't.

Eventual consistency can lead to data loss and stale reads

Redis with asynchronous replication is eventually consistent. Computer researcher and consultant Kyle Kingsbury evaluated Redis asynchronous replication, concluding in his blog that the Redis implementation allows for data loss:

Out of 2000 writes, Redis claimed that 1998 of them completed successfully. However, only 872 . . . were present in the final set. Redis threw away 56% of the writes it told us succeeded. [32]

Later, Redis released a WAIT command to provide safer operations through synchronous replication. Salvatore Sanfilippo, the creator of open source Redis who joined Redis Labs in 2015, explained the capability this way on Stackoverflow:

WAIT implements synchronous replication for Redis. Synchronous replication is required but not sufficient in order to achieve strong consistency.

WAIT does not make Redis linearizable, what it does is to make sure the specified number of slaves will receive the write, that in turn makes failover more robust, but without any hard guarantee. [33]

Exploring this topic further, we see WAIT described on the Redis open source website as follows:

This command blocks the current client until all the previous write commands are successfully transferred and acknowledged by at least the specified number of replicas. If the timeout, specified in milliseconds, is reached, the command returns even if the specified number of replicas were not yet reached. [34]

A key point is that the command does not block all activity on the affected shard. It only blocks the client that sent the command. Other clients can still act on the data being replicated and potentially be exposed to data inconsistencies.

Given what we know about open source Redis, what does that say about consistency for Redis Enterprise? Redis Enterprise uses the same mechanisms as open source Redis to implement replication, so it has similar behaviors. However, one difference with Redis Enterprise's support for replication and consistency involves the process that monitors and handles failovers. Redis Labs created their own Sentinel-like process called the *cluster_wd*. The documentation indicates that only one replica is used per primary shard or instance. This helps eliminate some potential data loss described in Kingsbury's blog on open source Redis. What does the Redis Labs documentation say about consistency guarantees?

---

[31] https://stackoverflow.com/questions/29381442/eventual-consistency-vs-strong-eventual-consistency-vs-strong-consistency
[32] https://aphyr.com/posts/283-jepsen-redis
[33] https://stackoverflow.com/questions/33629339/can-the-wait-command-provide-strong-consistency-in-redis
[34] https://redis.io/commands/wait

> *Redis Enterprise Software (RES) comes with the ability to replicate data to another slave for high availability and persist in-memory data on disk permanently for durability. With the new WAIT command, you can control the consistency and durability guarantees for the replicated and persisted database in RES . . . .* [35]
>
> *With the WAIT command, applications can have a guarantee that even under a node failure or node restart, an acknowledged write will be present in the system and will not be lost.*

However, if you dig a bit further, you'll find that the WAIT command can be used to create various levels of eventual consistency but not strong consistency. Redis Enterprise documentation cites the Redis open source site for details on the WAIT command, which is explained as follows:

> *Note that WAIT does not make Redis a strongly consistent store: while synchronous replication is part of a replicated state machine, it is not the only thing needed. However, in the context of Sentinel or Redis Cluster failover, WAIT improves the real world data safety.*
>
> *Specifically if a given write is transferred to one or more replicas, it is more likely (but not guaranteed) that if the master fails, we'll be able to promote, during a failover, a replica that received the write: both Sentinel and Redis Cluster will do a best-effort attempt to promote the best replica among the set of available replicas.* [36]

One topic missing from both the open source and the Redis Labs websites is information about the guarantees for reads -- an important consideration. Stale reads can occur with the WAIT command because WAIT blocks operations only for the client that issued the WAIT. All other clients can still issue writes and reads on the key that was accessed by the blocked client. Given that a network failure can happen while the primary is waiting for a response from the replica, a read operation could be executed on the primary that would return the value from the last write that was blocked and waiting on the replica. Even with the *min-slaves-to-write* option that improves the write safety during a network partition, stale reads can still occur because the slave will have been promoted to a primary. Consider Kingsbury's conclusion about how the WAIT command can cause consistency problems for reads:

> *This case demonstrates that reads are a critical aspect of linearizability. Redis WAIT is not transactional. It allows clients to read unreplicated state from the primary node, which is just as invalid as reading stale data from secondaries.* [37]

Finally, as of this writing, we see no evidence that open source Redis or Redis Enterprise has passed Kingbury's Jespen tests, an open source suite that helps programmers explore the safety of distributed systems.

Eventual consistency increases development complexity and operational risk

The eventual consistency approach implemented through the WAIT command forces developers to account for more paths in their design to get all the edge cases correct. Yiftach Schoolman, Redis Labs CTO and co-founder, discussed this point in the comment section of a blog post on the WAIT command by Sanfilippo:

IMO an extremely important feature for Redis, well done Salvatore.

---

[35] https://docs.redislabs.com/latest/rs/concepts/data-access/consistency-durability
[36] https://redis.io/commands/wait
[37] https://aphyr.com/posts/309-knossos-redis-and-linearizability

My 2 cents - based on my experience/knowledge, many of the app developers who use an eventually consistent database like Riak/Dynamo/Cassandra would prefer not to deal with the complexity involved in writing code that can deal with eventually consistent situations, like mentioned in this post.

I would therefore suggest to add an option to launch Redis as a completely synchronous system with default & configurable WAIT parameters . From the developers point of view, this can ease the implementation, i.e. critical data that is less sensitive to latency can be stored in the synchronous Redis, whereas less critical data that is performance sensitive will be stored in a regular async Redis. [38]

So, what can developers do? Software engineer and researcher Martin Kleppman blogged about distributed locking with Redis' Redlock algorithm:

I think the Redlock algorithm is a poor choice because it is "neither fish nor fowl": it is unnecessarily heavyweight and expensive for efficiency-optimization locks, but it is not sufficiently safe for situations in which correctness depends on the lock.

In particular, the algorithm makes dangerous assumptions about timing and system clocks (essentially assuming a synchronous system with bounded network delay and bounded execution time for operations), and it violates safety properties if those assumptions are not met. Moreover, it lacks a facility for generating fencing tokens (which protect a system against long delays in the network or in paused processes). [39]

In addition, Kingsbury offered several conclusions (in *italics*) in his Jespen post on Redis. (Note that "CP" refers to the CAP theorem[40] involving consistency, availability, and partition tolerance in distributed systems.)

- Distributed locks like Redlock need strong consistency.
  Bottom line: distributed lock services must be CP.

- Distributed queues need strong consistency.
  Most distributed queue services can provide reliable at-most-once or at-least-once delivery. CP queue systems can provide reliable exactly-once delivery with higher latency costs. Use them if message delivery is important.

- Even your database needs strong consistency.
  If you use Redis as a database, be prepared for clients to disagree about the state of the system. Batch operations will still be atomic (I think), but you'll have no inter-write linearizability, which almost all applications implicitly rely on. [41]

Even simple applications like locks or distributed queues need to be strongly consistent, particularly when an application or service is performing millions of operations per second. The costs of inconsistent data can run into huge losses, as illustrated by Flexcoin.

[38] http://antirez.com/news/66#comment-1152009384
[39] https://martin.kleppmann.com/2016/02/08/how-to-do-distributed-locking.html
[40] https://en.wikipedia.org/wiki/CAP_theorem
[41] https://aphyr.com/posts/283-jepsen-redis

Sign #5: You need manageability and operational ease as your data volumes grow

Finally, consider the operational impact of managing a large Redis cluster. Even if your firm's data volumes are relatively modest today, what will happen when that changes? It's no secret that the volume of operational data has soared in recent years. And that's not all. Demand for extremely fast -- and predictable -- access to that data is turning once-serviceable data management solutions into operational nightmares. This begs a question: from an operational standpoint, are you sure you can meet your firm's future scalability and performance needs with Redis or ROF?

Are you ready for the future with Redis?

As we've already discussed, scaling Redis requires substantial memory and leads to large clusters. Operating and managing such clusters can get complicated quickly. Imagine that you need to manage 30TB of user data with Redis. Assuming 5 nodes per TB and a replication factor of 3, you'd need to deploy 450 nodes:

5 nodes * 30 TB = 150 nodes
150 nodes * 3 replication factor = 450 nodes

Now consider the operational support required to manage a cluster of that size. How many full-time engineers would you need to monitor, tune, and maintain that cluster? Perhaps 3 engineers (1 per 150 nodes)? Even if you don't anticipate having any problems budgeting for -- and training and retaining -- such staff, consider how the increased failure rates of such a large cluster can wreak havoc with attempts to achieve SLAs that demand predictable performance and 24x7 availability. . . .

So, ask yourself: what if you managed that same 30TB of user data on a much smaller footprint using Aerospike? How much simpler would it be to staff and operate such an environment? If you're like many former Redis users who are now running Aerospike, the answer is simple: it's a lot easier. Sound too good to be true? IronSource, a Redis user now running Aerospike, recently characterized its daily maintenance with Aerospike as "minimal to none." As one IronSource staff member put it,

> *I barely touch the cluster; it simply works.* [42]

Perhaps you're thinking that ROF (with RocksDB) would solve the problem in a more manageable way. But, as an earlier section explained, configuring and tuning ROF to deliver performance at scale isn't trivial. Indeed, even experienced users admit to having problems fully understanding the impact of various configuration changes. To get some idea of the expertise required to tune RocksDB effectively, browse through the tuning guide[43] created by RocksDB specialists at Facebook; you'll see that it discusses multiple amplification factors, different types of compactions, various statistics to monitor, and more.

Finally, durability and data consistency requirements introduce further operational considerations. With Aerospike, persistence is fast, straightforward, and built in. So is strong, immediate data consistency. The same isn't true for Redis.

[42] https://www.aerospike.com/resources/videos/summit19/company/ironsource/

[43] https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide

Scalability strategies with Redis

Among the critical design points of Redis impacting its ability to scale easily and efficiently is its single-threaded nature. Here's an excerpt from the Redis FAQ:

> Question: *Redis is single threaded. How can I exploit multiple CPU / cores?*
>
> Answer: *To maximize CPU usage you can start multiple instances of Redis in the same box and treat them as different servers. At some point a single box may not be enough anyway, so if you want to use multiple CPUs you can start thinking of some way to shard earlier.* [44]

Didier Spezia, a Redis community contributor, compared scaling of Redis open source with that of an RDBMS in a post on Stackoverflow:

> *A unique Redis instance is not scalable. It only runs on one CPU core in single-threaded mode. To get scalability, several Redis instances must be deployed and started. Distribution and sharding are done on client-side (i.e. the developer has to take care of them). If you compare them to a unique Redis instance, most RDBMS provide more scalability (typically providing parallelism at the connection level). They are multi-processed (Oracle, PostgreSQL, ...) or multi-threaded (MySQL, Microsoft SQL Server, ...), taking benefits of multi-cores machines.* [45]

So how can users scale Redis? Options include client-side sharding, the open source Redis Cluster, and proxy-based solutions. Let's consider each in turn.

Client-side sharding may be a simple starting point for a small datastore. But as a cluster grows in size and complexity, the effort to upgrade and maintain a client-side solution becomes significant. Furthermore, many firms require other features like replication, automated failover and re-sharding to create a reliable, performant datastore at scale.

The **Redis Cluster** solution was developed for partitioning and scaling. The cluster shards the data across multiple nodes, and the cluster client maintains a map of the location of keys on each node. But Redis pipelining and the MULTI/EXEC command are limited when using the open source cluster.

The **proxy solution** has garnered the highest interest of the open source solutions and include offerings such as Twemproxy, Corvus Proxy, Codis Proxy, and Dynomite. These solutions tend to take significant efforts to scale and maintain, and each proxy has its own unique limitations. Furthermore, not all Redis commands work with these proxy solutions. Pipelining works with limited data capacity and higher latency.

In an attempt to provide scalability, Redis Labs created Redis Enterprise and ROF (ROF). However, as an earlier section of this paper explained, ROF often carries a high cost to achieve desired scalability and performance targets.

Coping with failure

Failures happen. Nodes, networks, disks, and other critical components go offline or get corrupted. What's critical is how your DBMS platform copes with inevitable failures.

Achieving high availability with open source Redis can be elusive. As one user noted in his 2018 blog,

[44] https://redis.io/topics/faq
[45] https://stackoverflow.com/questions/10906246/what-is-the-disadvantage-of-just-using-redis-instead-of-an-rdbms/10913413#10913413

Recently, at the place where I work, . . . a question about Redis high availability arose. Turns out, for Redis there is no ready-made solution. . . . . Choosing the right solution for Redis high availability is full of tradeoffs. Nobody knows your situation better than you, so get to know how Redis works – there is no magic here – in the end, you'll have to maintain the solution. [46]

What about Redis Enterprise? Its failover mechanism has some serious limitations for real-life deployments, as noted in this documentation excerpt:

The Redis Enterprise cluster has out-of-the-box HA profiles for noisy (public cloud) and quiet (VPC, on-premises) environments. We have found that triggering failovers in too aggressive of a manner can potentially create stability issues. On the other hand, in a quiet network environment, a Redis Enterprise cluster can be easily tuned to support a constant single digit (<10 sec) failover time in all failure scenarios. [47]

Put simply: failover has stability issues in production environments and requires tuning even with a "quiet" network.

Compare that with the experience of So-Net Media Networks, a Japanese internet service provider operated by Sony Network Communications. After evaluating various key-value data stores (including Redis and Aerospike), the firm selected Aerospike. Chief Architect Takahiro Yasuda recently presented on its 8 Years of Operation without a Single Stop. He cited Aerospike's speed, scalability, stability, and low overall costs as key advantages.

So-Net Media Networks isn't the only Aerospike user to enjoy near 100% uptime. Consider Wayfair, which one of its data architects (Ken Bakunas) recently described as "a technology company that just happens to sell furniture." The online retailer offers more than 14 million products from 11,000+ suppliers and enjoys high YTY growth rates. Challenged to find a replacement for its "outdated" Redis environment, the firm sought a database supplier that could deliver low latency, reliability, scalability, manageability, and a strong partnership. After encountering performance problems, operational complexity, and other issues with Cassandra, the firm turned to Aerospike. As of this writing, Wayfair uses a 7-node Aerospike cluster with 180TB disk space to manage 6 billion master objects. Aerospike services an average of 100,000 reads and 20,000 writes per second around the clock. Latencies for 99.9% of both read and write operations are under 1 ms. The team's success with Aerospike hasn't gone unnoticed:

Other groups [in our firm] hate us since nothing goes wrong, no downtime. [48]

Synchronizing data across a multi-layered architecture

Combining an in-memory data cache with a back-end persistent store is an architectural pattern that's been deployed in the IT industry for years, particularly with Redis. Quite often, firms turn to this pattern in a desperate attempt to "speed up" access to data managed by a SQL or NoSQL store that simply can't deliver the desired runtime performance.

Extensive deployments of this pattern have yielded some hard truths. For example, it's no secret that synchronizing data across applications, the cache, and the back-end store requires careful planning to ensure that applications don't end up working with stale data or waiting longer than expected to access persistent data due to cache misses. To get an idea of the effort and trade-offs involved, browse through a

[46] https://alex.dzyoba.com/blog/redis-ha/
[47] https://redislabs.com/redis-enterprise/technology/highly-available-redis/

[48] https://www.aerospike.com/resources/videos/summit19/company/wayfair/

RedisConf18 user presentation[49], read Amazon's documentation on Database Caching Strategies with Redis[50], or even skim through a blog[51] on generic caching strategies. Although different techniques can be effective for different situations, the fact remains that data architects and application developers must undertake extra work to understand, design, implement, and test one or more approaches to address data synchronization and latency issues. Mistakes can be costly and hard to detect.

Furthermore, a multi-layered architecture implies more work to configure, monitor, tune, and manage the overall environment than an efficient single-layer architecture requires. Diagnosing and resolving performance, availability, and other issues becomes more complex. These are problems that can't be readily solved with hardware upgrades and expanded server footprints even if you're willing to bear the added costs, which can be considerable.

Firms that replace a multi-layered architecture with Aerospike often realize a simplified operational environment, significant TCO savings, and ultra-fast performance for data at scale. In the next section, you'll learn more about Aerospike's technology so you can understand how such remarkable results are possible.

Aerospike: The Enterprise-Grade NoSQL Database

Aerospike was built to deliver extremely fast – and predictable – response times for accessing large data sets with high cost efficiency. Indeed, worldwide firms use Aerospike in production to fulfill SLAs that require sub-millisecond data access speeds over billions of records in databases of 10s - 100s of TB. Other hallmarks of Aerospike's unique architecture include high scalability, strong consistency for both in-memory and persistent data, and operational ease.

Low total ownership costs (TCO)

If you think that sounds too good to be true, consider the experience of Adjust[52], a firm that specializes in verifying billions of advertising impressions. Data correctness and low latency are critical for fraud detection work and Adjust initially used Redis to support its work. But, in time, that environment became impractical due to "weird" latency spikes, increased average response times, long and disruptive failovers, and more. When the firm migrated from Redis to Aerospike, it drastically reduced its server footprint and cut costs by 85%. Adjust has since expanded its use of Aerospike to 3 locations and relies on Aerospike to process 1.1 million reads and .5 million writes per second.

That's not just an isolated case. Appsflyer[53], another former Redis user, migrated to Aerospike to fulfill its aggressive scalability and performance needs. When its initial Redis-based architecture created "too much weight" on the network and raised scalability concerns, Appsflyer tried other alternatives, ultimately settling on an architecture that uses Aerospike, Apache Kafka, and Apache Spark for its real-time operational needs. Appsflyer now enjoys a stable, resilient system that has "flawlessly" managed a three-fold data set growth and

---

[49] https://www.slideshare.net/RedisLabs/redisconf18-techniques-for-synchronizing-inmemory-caches-with-redis

[50] https://d0.awsstatic.com/whitepapers/Database/database-caching-strategies-using-redis.pdf

[51] https://codeahoy.com/2017/08/11/caching-strategies-and-how-to-choose-the-right-one/

[52] https://www.aerospike.com/resources/videos/adjust-scaling-platform-for-real-time-fraud-detection-without-breaking-bank/

[53] https://medium.com/appsflyer/how-we-built-the-worlds-biggest-real-time-db-of-mobile-devices-33c1c3921a48

substantially improved runtime performance for certain activities (e.g., writing times of Spark's slave decreased from 30 minutes to 5 minutes).

Firms often turn to Aerospike when their efforts to deploy other NoSQL solutions fall short. It's worth briefly exploring several key aspects of Aerospike's technology that makes it the NoSQL system of choice for companies that demand high performance, scalability, strong consistency, operational ease, and more.

Scalability

Aerospike's Hybrid Memory Architecture™ (HMA) can reliably handle millions of transactions per second and large data volumes at low latency. Because Aerospike optimizes hardware efficiency, customers often reduce production hardware footprints by an order of magnitude. This contributes to substantial cost savings. Indeed, many Aerospike clients cut ownership costs by $1 - 10 million per application over a 3-year period compared with other alternatives. And thanks to Aerospike's near-linear scalability, firms can readily add or remove nodes to adjust throughput or storage capacity even while running production loads. The ability to seamlessly move from a dataset of only hundreds of gigabytes to hundreds of terabytes or even petabytes is one of the most desired and critical requirements of a modern DBMS.

Aerospike provides horizontal scalability via dynamic cluster management. The cluster management system automatically moves data as the cluster size changes, re-sharding on the fly if needed. Additionally, network communication is minimized through the use of optimized, compressed management protocols. With Aerospike, cluster sizes of hundreds of nodes, each with potentially hundreds of terabytes of flash that support millions of transactions per second, are now practical. Furthermore, Aerospike deployments have achieved extraordinary multi-year uptime in production situations. Being able to scale horizontally as well as vertically opens up the potential to explore new and expanded business opportunities.

Aerospike's All Flash feature extends Aerospike's HMA to support a broader set of use cases. Aerospike can be configured to use storage (often NVMe flash drives) for indexing, radically reducing costs for applications that can tolerate nominally increased latencies compared with Aerospike's traditional configuration, which maintains indexes in DRAM. Yet even with indexes persisted to flash, Aerospike still delivers very high performance. In testing[54] with 150-byte objects, Aerospike has seen < 2 ms latencies for 99% of the workload on Amazon i3 instances – and much higher performance on more modern hardware. As a result, Aerospike can be deployed instead of HBase or the Hadoop Distributed File System (HDFS) in many situations.

Contrast Aerospike's proven scalability features with what we've discussed about Redis's scalability challenges. If you want to avoid serious cost and operational challenges as your need to scale increases, doesn't it make sense to investigate Aerospike?

Persistence

Aerospike's typical configuration keeps indexes purely in memory, while data is persisted to and read directly from SSDs. This helps drive predictable performance. Such a design is possible because the read latency characteristic of I/O in SSDs is the same whether it's random or sequential. For such a model, the optimizations described are used to avoid the cost of a device scan to rebuild indexes.

The ability to randomly read data comes at the cost of a limited number of write cycles on SSDs. To avoid 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 evenly, which, in turn, improves device durability. Aerospike bypasses the operating system's file system and instead uses attached flash

[54] https://www.aerospike.com/blog/aerospike-4-3-all-flash-uniform-balance/

storage 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 to a buffer that's flushed to storage when full.

Starting with Aerospike Enterprise Edition 4.5, Aerospike supports both traditional DRAM memory and Intel Optane DC persistent memory. Optane DC pmem is a new class of memory and storage technology architected specifically for data-intensive applications that require extremely low-latency, high durability and strong data consistency.

Strong consistency

Aerospike's support for strong, immediate consistency guarantees that all writes to a single record will be applied in a specific sequential order and writes will not be re-ordered or skipped. In particular, the guarantee ensures that writes acknowledged as having been committed have been applied and exist in the transaction timeline. This guarantee applies even with network failures. Aerospike's strong consistency guarantee is per-record; each record's write operation is atomic and isolated, and ordering is guaranteed using a hybrid clock.

For reads, Aerospike supports both full linearizability, which provides a single linear view among all clients that can observe data, as well as a more practical session consistency, which guarantees an individual process sees the sequential set of updates. These policies can be chosen on a read-by-read basis, thus allowing the few transactions that require a higher guarantee pay the extra synchronization price.

In the case of a "timeout" return value -- which could be generated due to network congestion, external to any Aerospike issue -- the write is guaranteed to be written or not written.

For more information about Aerospike's strong consistency support, see the white paper on Exploring data consistency in Aerospike Enterprise Edition[55] or read about the Jepsen report.[56]

Operational simplicity at scale

Complex technology becomes truly valuable when it's easy to use, particularly in challenging production environments. Clients can't afford to be burdened with labor-intensive efforts whenever they need to expand a cluster, refresh hardware, or migrate to a new data center. In such situations, the DBMS should behave more like a utility, enabling clients to increase capacity as needed without extensive planning or maintenance windows.

Scaling Aerospike is simple. There's no planning required. That's because Aerospike supports self-forming and self-healing clusters that are both rack-aware and data center-aware. No downtime is required to add or remove nodes from a cluster; Aerospike automatically re-distributes partitions of data to ensure new resources are efficiently used. Rack awareness also ensures that data is correctly separated to avoid compounding failures. Using a proprietary algorithm, nodes can be restarted -- for example, to perform a software upgrade -- but the memory contents are preserved, allowing the process to restart without the traditional delays needed to warm memory buffers and caches. How's that possible? With Aerospike, the data is never evicted from DRAM on restart.

Finally, Aerospike's ability to handle high data volumes and demanding workloads on relatively small server footprints further simplifies operations. Fewer nodes reduces monitoring and maintenance efforts required to fulfill SLAs.

---

[55] https://www.aerospike.com/lp/exploring-data-consistency-aerospike-enterprise-edition/
[56] https://jepsen.io/analyses/aerospike-3-99-0-3

Summary

While firms often find Redis easy to use for small-scale projects that require the versatility of a NoSQL DBMS, that can change quickly as needs evolve. Common trouble spots for Redis clients include high ownership costs, scalability difficulties, operational challenges, and more. Indeed, here are five signs that your firm may have outgrown Redis:

1. You're worried about TCO (total cost of ownership), particularly as your needs grow.
2. You need scalability and elasticity.
3. You need persistence with high performance.
4. You need strong data consistency.
5. You need manageability and operational ease at scale.

Aerospike offers firms a compelling alternative. It's a NoSQL key-value store that delivers ultra-fast runtime performance for read/write workloads, high availability, near-linear scalability, and strong data consistency. And it does that at a fraction of the cost of other alternatives. Indeed, many global firms enjoy substantial TCO savings thanks to Aerospike's highly efficient distributed database platform and patented architecture. For example, one of its clients is a former Redis user that cut its server footprint from 40 to 6 nodes and its overall costs by 85% by migrating to Aerospike. Another former Redis user turned to Aerospike to handle its massively growing data set with ease -- a feat Aerospike achieved while providing greater stability and resiliency than Redis.

So why wait? If you're struggling to achieve what you want with Redis, why not investigate what Aerospike can do for you? Plenty of firms have already realized concrete business benefits by deploying Aerospike for their mission-critical applications. Contact Aerospike to arrange a technical briefing or discuss potential pilot projects. It might be the best move you'll make.

Appendix A: TCO configuration details

An earlier section of this paper explored a sample use case to analyze the total cost of ownership (TCO) of various Redis configurations, comparing each with Aerospike. This appendix explains the details of these configurations so you can better understand the TCO analysis.

Amazon EC2 instances

Figure 1 outlines the different Amazon EC2 instances used for each Redis configuration and the common Aerospike configuration. As you'd expect, prices differ based on the type of EC2 instance deployed. In all cases, we selected the most reasonably priced instance suitable for the workload and DBMS configuration. Subsequent sections on Redis and Aerospike sizings explain how we arrived at these selections.

Redis in-memory with AOF persistence:

R5d.12xlarge

CPUs - 48
Memory - 384 G.
Network - 10 GbE.
Storage - 2 x 900 NVMe SSD
Yearly upfront cost of one server - $17,833

Redis in-memory only:

R5.12xlarge

CPUs - 48
Memory - 384 G.
Nework - 10 GbE.
Storage - EBS-only. Drive.
Yearly upfront cost of one server - $15,576

Aerospike and Redis on Flash (ROF):

i3.8xlarge

CPUs - 32
Memory - 244 G
Network - 10 GbE
Local Storage – 4 x 1,900 NVMe SSD
Yearly upfront cost of one server - $13,928

Figure 1: Amazon EC2 instances for comparative study

Redis sizing

How did we arrive at these specific configurations? Let's start with Redis. Table 4 includes the statistics we used to arrive at total memory for a cluster. We followed these steps:

1. Multiply the number of keys by the average object size (with overhead) to calculate the unique data footprint.
2. Double the size of the unique data footprint to get the replicated data set (because the replication factor is 2). Note that the replicated data size is the amount of memory needed just for the data.
3. Divide the replicated data set by 0.7 to determine the total RAM needed for the whole data set. Why .7? Redis Enterprise documentation[57] recommends a 70% RAM utilization rate. You can think of the

[57] https://docs.redislabs.com/latest/rs/administering/designing-production/hardware-requirements/

replicated data size as the 70% value of RAM. (Note that if persistence is used, up to 2x memory is needed for the data set. The persistence 2x value may vary depending on the workload. Higher write workloads with larger data sets will most likely need 2x the memory needed.)

	Year 1	Year 2	Year 3
# of keys (in billions)	17	21.3	26.6
Object size, avg (bytes)	500	500	500
Object with overhead	580	580	580
Unique total data footprint (TB)	8.97	11.24	14.03
Replication factor	2	2	2
Replication size (TB)	17.94	22.48	28.06
Recommend memory usage limit	70%	70%	70%
Total Memory for cluster	25.63	32.14	40.06
Memory factor for persistence	2	2	2
Total Memory with persistence	51.26	65.28	80.12

Table 4: Sizing Redis

With this sizing information, we then derived server counts required for each Redis configuration based on the capacities of different EC2 instances. To do so, we divided the total memory needed and the total memory with persistence from Table 4 by the memory of the appropriate EC2 instance type shown in Fig. 1. This yielded Redis server counts shown in Table 5.

	EC2 instance	Year 1	Year 2	Year 3
# of servers with AOF persistence	R5d.12xlarge	138	172	214
# of servers in-memory only	R5.12xlarge	69	86	107
# of servers with ROF (50% memory / 50% flash)	I3.8xlarge	54	68	85

Table 5: Redis server counts based on configuration type

Aerospike sizing

Now let's turn to Aerospike. By default, Aerospike keeps indexes in memory and user data on SSDs. For this scenario, 1.01 TB of memory was needed for indexes and 40TB of SSDs to store the primary and replica data set. Additional storage space was allocated for handling the defragmentation process, yielding total SSD requirements shown in Table 6.

	Year 1	Year 2	Year 3
# of Keys (in billions)	17	21.3	26.6
Object size, avg (bytes)	500	500	500
Object storage size	640	640	640
Unique total data footprint(TB)	9.8	12.4	15.5
Replication factor	2	2	2
Replication size (TB)	18.8	24.7	30.9
Reserve factor for defrag plus overprovisioning.	2.1	2.1	2.1
Total storage SSD size (TB)	41.6	52.0	64.9

Table 6: Sizing Aerospike

To determine the number of servers needed, we divided the Aerospike memory and storage totals by the memory and storage on the EC2 instances. Each I3.8xlarge instance contains 244 GB of memory and 4x1.9 TB NVMe SSDs. From the available storage, 7.6 TB was used to determine the final server counts for each of the three years. Table 7 outlines the results.

	Year 1	Year 2	Year 3
# of servers	12	15	19

Table 7: Sizing Aerospike

As you can see, the difference in required cluster size for this scenario was dramatic, helping to drive the substantial TCO savings many firms have come to realize with Aerospike.

AEROSPIKE

About Aerospike

The Aerospike enterprise-grade non-relational DBMS helps firms around the globe run mission critical operational applications that make digital transformation possible. Powered by a patented Hybrid Memory Architecture™ (HMA) and autonomic cluster management, Aerospike has proven vital to firms in financial services, telecommunications, technology, retail, and other industries that require extreme uptime, ultra-fast performance and high scalability from their operational DBMS. Indeed, Aerospike powers a wide variety of strategic applications, including fraud prevention, digital payments, recommendation engines, real-time bidding and more. Aerospike customers include Adobe, Airtel, FlipKart, Kayak, Nielsen, PayPal, Snap, and Wayfair.

© 2019 Aerospike, Inc. All rights reserved. Aerospike and the Aerospike logo are trademarks or registered trademarks of Aerospike. All other names and trademarks are for identification purposes and are the property of their respective owners.

2525 E. Charleston Road, Suite 201
Mountain View, CA 94043
Tel: 408 462 2376
www.aerospike.com