



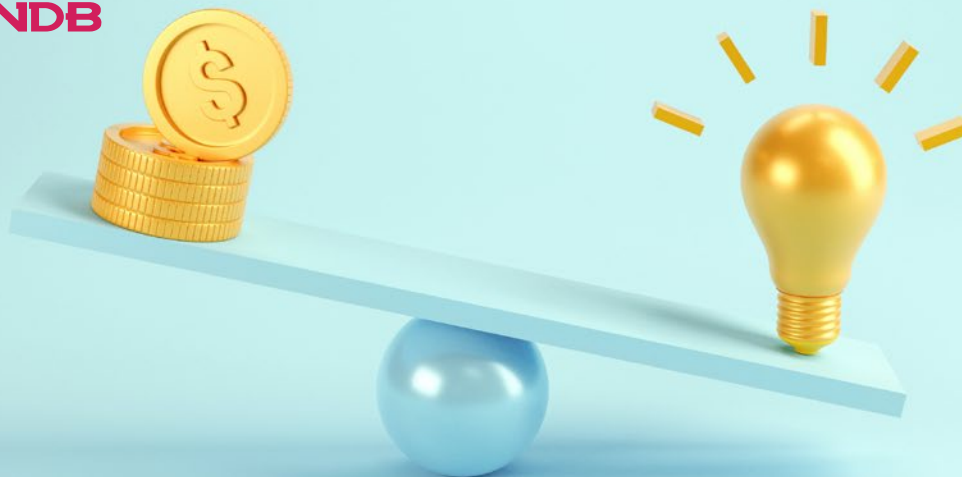
WHITEPAPER

Couchbase vs RavenDB **Performance** at Rakuten Kobo



Contents

Couchbase vs RavenDB Performance at Rakuten Kobo	2
Key Findings	3
The Dataset	3
Loading the Data	5
Disk Usage	6
The Test Environment.....	8
Environment Configuration.....	9
The Environment.....	10
The Data Model	11
The Queries	11
Indexing.....	13
Testing	13
Users and Books Highlights Queries.	13
Accessing the Data by Key	14
Discussion	16
Operational Considerations.....	20
About RavenDB.....	24



Couchbase vs RavenDB Performance at Rakuten Kobo

Rakuten Kobo Inc. is one of the world's biggest digital booksellers. Owned by Tokyo-based Rakuten and headquartered in Toronto, Rakuten Kobo enables millions of readers worldwide to read anytime, anywhere, on any device. Rakuten Kobo connects readers to stories using thoughtful and personalized curation of eBooks and audiobooks, and the best dedicated devices and apps for reading. With the singular focus of making reading the finest experience, Kobo's open platform allows people to fit reading into their busy lives.

Users purchase books, store their highlights made on a book, and track their notes. To make that possible, the backend servers must synchronize data with the e-readers on a regular basis.

With over tens of millions of devices, a normal day at Rakuten Kobo looks like a denial-of-service attack for most businesses. In determining the best storage solution for their next gen infrastructure the Kobo team, led by Trevor Hunter the CTO, wanted to see

*Under load, RavenDB matches or exceeds Couchbase performance at the **99.99 percentile** with third of the hardware resources. In the cloud this translates to **80% cost savings**.*

how RavenDB would stack against one of the databases they already had in their infrastructure.

Kobo has been using Couchbase for a while for a different purpose and wanted to see what would be the best tool for the job to hold the data for the new e-books backend. This report benchmarks RavenDB against Couchbase.

Kobo worked with us to build a clean dataset that matched their data distribution using publicly available data to have a reproducible environment for evaluation. The dataset consists of 1.35 billion documents with a database size of 985GB.

Key Findings

- Comparable performance (up to 99.99-percentile) when accessing data by key.
- RavenDB outperforms Couchbase by orders of magnitude when using queries under load.
- Couchbase proved fragile in production, with high overhead on failure conditions. RavenDB's failure model, on the other hand, proved resilient and allowed much higher operational flexibility and peace of mind.
- RavenDB stores the data at **$\frac{1}{3}$ of the storage** needed by Couchbase.
- RavenDB cloud budget configurations are **80% cheaper** at a comparable, 99.99 percentile latency.

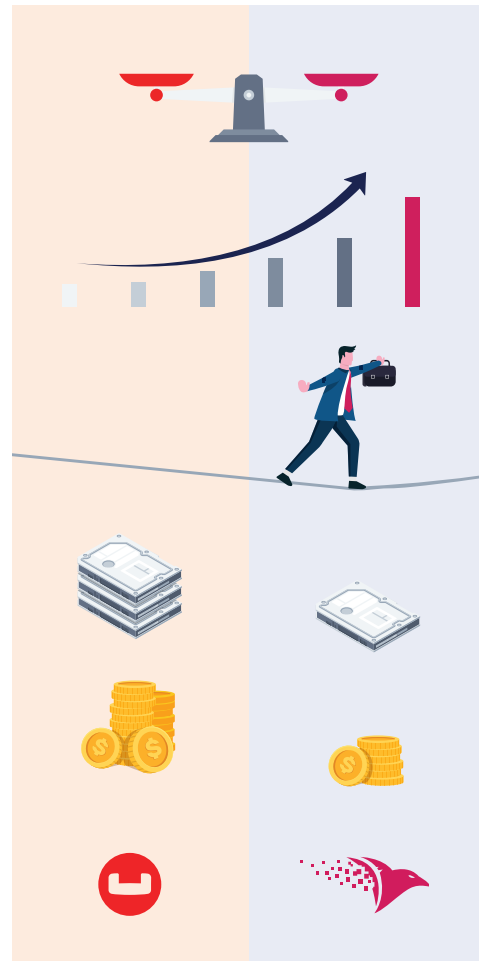
The Dataset

To let anyone study these results, our team used publicly available data. The final database export is available for anyone wanting to reproduce the findings shown in this report.

To match the typical usage scenario of the expected next gen infrastructure, we sized the deployment for a realistic expected usage scenario.

The dataset contains 1,357,692,380 (1.35 billion) documents divided into the following collections:

- **69.38 million Users.** The total number of users in the model. A user may have multiple devices



and can sync the data across all of them. We used the Reddit users' database available at: <https://bit.ly/3j4yTCe>

- **63,510 Books.** eBooks taken from *Project Gutenberg*. This includes both the book's metadata as well as their content.
- **734.22 million UsersBooks.** Associating a book to a user, usually representing a user purchasing a particular book. This was generated randomly, using long tail distribution.

- **497.03 million Highlights.** A core enhancement on the reading experience is supporting highlighting phrases and paragraphs in any book accessible by a user. The highlight data includes the position and length of the highlight as well as its actual text.
- **28.78 million Editions, 20.3 million Works and 7.89 million Authors.** These documents represent additional information about the books in the dataset.

To mimic the Rakuten Kobo data distributions found in the private data, we used a typical long tail distribution from books to users. It is expected the number of books per user would be similar. Still, in some cases, Rakuten Kobo has users that represent libraries / collections that may accumulate a lot of books. The former typically will have a small amount, and while the latter are not a normal occurrence, they would be quite taxing when accessed. As an example, the whole *Project Gutenberg* collection would include more than 60,000 books.

Given the data distribution, the relationships created is roughly three-quarters of a billion (See Table 1).

Regarding the number of highlights per user, the top 10,000 users have significantly more books than the top used in the Rakuten Kobo actual dataset. The primary objective of oversizing highlights is to test if the infrastructure can grow in functionality for both user and machine-generated content and therefore encounter queries with many potential results.

For context, Goodreads has an annual reading challenge. The average pledge for reads in 2020 was 61 books. That means that after a few years, a moder-

Number of users	Number of highlights
1	203,550
428	100,000 – 200,000
10,622	10,000 – 100,000
20,340	1,000 – 10,000
67,249	100 – 1,000
59,827	10 – 100
10,158,123	2 – 10
43,425,405	1

Table 1. Testing data distribution

ately heavy reader will accumulate a few hundred books quite easily.

In terms of highlights, Rakuten Kobo’s data has many users with no or just a few user-generated highlights. The overall average is about 10 highlights per user, with the 99-percentile coming in under 250 highlights. However, some users are really heavy users of highlights, with some approaching 20,000 highlights.

The total number of documents that we used was 1.35 billion. The database export for this dataset is 108GB of compressed data. The export file is available at (<https://bit.ly/3rhVNZ6> – note 108GB) in RavenDB dump format ready to be imported to a RavenDB database.

Loading the Data

The number of documents in the dataset represent a reasonably sized dataset, both in terms of the actual size of the data and the number of documents. Loading the dataset represents a sizable amount of time.

*For RavenDB it took less than a day.
For Couchbase it took almost four days.*

When loading the data into RavenDB, we used a machine with 8 cores and 32 GB of RAM with default configuration. The actual tests were run on several different machines.

We attempted to load the same dataset into Couchbase as well, but we ran into a few problems. By default, Couchbase stores all the metadata about documents in memory. For the workload involved, this became highly problematic. The memory used by Couchbase per key is the document key length plus 56 bytes.

With 1.35 billion documents, some of the documents have long ids (around 60 bytes). Along with the metadata overhead, **Couchbase required 162 GB of RAM just to store the document IDs** while RavenDB managed just fine to work with entire dataset using 32 GB.

Couchbase has an option that avoids holding the entire set of document keys in memory. It is called Full Ejection and allows Couchbase to reduce the amount of data that must be kept in memory.

For best performance, Couchbase *recommends* avoiding this setting. However, that requires you to significantly increase the size and capacity of your nodes. In our experiments, to get good performance without Full Ejection, we had to massively overprovision the cluster, to the point where it made no economic sense whatsoever. Performance alone isn't sufficient, part of the equation is the cost of resources needed to achieve this performance.

Even with setting the Ejection Method to Full, Couchbase nodes consumed too much memory and crashed (See Fig. 1). Without sufficient computing and memory resources, we observed the nodes consuming all available memory and processes being killed by the Operating System. We increased the size of the deployment until nodes could ingest the whole dataset. The minimum hardware required to complete the goal was a cluster of 3 nodes with 32 cores and 128 GB of RAM per node.

Even with 128GB per node, we observed Couchbase consuming more memory than was available on the machine, taking the entire machine down in the process. Using too much memory and ending up spending a lot of time in page faults rendered the node completely inaccessible, attempting to restart

name ▼	services	CPU	RAM
172.31.37.146	data query index search	4.6%	85.8%
172.31.41.235	data query index search	3.6%	96.7%
172.31.44.133	data query index search	2.5%	99.2%
Node unresponsive Not taking traffic FAILOVER to activate available replicas			

Fig. 1 Unresponsive Couchbase node after memory failure

the service or the machine was not always successful. This happened with no swap as well as with a 32GB swap in place.

We initially defined the default 90% of memory dedicated to Couchbase, but after repeated failures we reduced it to 80GB out of the 128GB during the ingest process. This reduced the outages to the point where we could complete the ingestion. During the ingestion process, we introduced breaks now and then for a few hours to let the cluster do its own book-keeping and management instead of putting it under constant load.

We also had to throttle down the ingest speed.

The ingest process for RavenDB was performed on a single node with 8 cores, 32 GB RAM and default configuration while replicating to the rest of the nodes in the cluster. For Couchbase, we set up sharding from the start. The minimum viable hardware configuration that could finish the task required 32 cores and 128 GB RAM for each of the three nodes.

That is 12 times higher than the configuration used for RavenDB for the same ingest scenario. We consider this to be the bare minimum hardware to sustain this load on Couchbase. In fact, this is explicitly *below* what we could tolerate in production. There is just a single copy of each document in Couchbase, which means a single node going down will result in data loss or data unavailability. A more realistic scenario would be a five node cluster, with the data replicated to three other nodes.

See the following section for more details on the issues we have encountered.

Disk Usage

A major difference between RavenDB and Couchbase was discovered during the ingest process. In RavenDB, the data was stored with full replication between the nodes. Every single node holds the full data set of 1.35 billion documents. The size of the data on disk was 985 GB, with another 120 GB for indexes. The system was provisioned with 2TB disks and it used just over half.

Couchbase was more challenging for the following reasons:

1. The database was sharded and we used a replication factor of 1, so each document only existed on a single node. We assumed that sharding the data would need a lot less disk space, so initially we provisioned the Couchbase nodes with 1TB disks. Instances quickly ran out of disk space once we uploaded 30% of the data, therefore we had to double the disk allocation several times to reach the ingestion and initial indexing goal.

Each Couchbase node required 6 TB of available storage to finish loading all of the data successfully.

To hold the entire dataset, we would need to double that or triple that again, per node. If each node would hold the entire 1.35 billion records as well, the amount of memory (even with Full Ejection) that would be dedicated just to metadata storage would force us to give even more memory for each node.

2. Couchbase uses an append-only + compaction model to write to the disk. Each write is written to the end of the data file. This approach allows Couchbase to simply scan from the end of the file

on startup to find the most recent valid transaction. Unfortunately, this approach comes at a cost. When documents are deleted or modified, they aren't updated in place but written (again) to the end of the file, causing the disk usage to increase substantially under write load.

Older versions of the values in the file are not modified, but they are now no longer needed. A compaction step is required to free the disk space. Couchbase performs this operation by writing a new file, with only the valid data. This was a simplified explanation of a fairly complex piece of technology, for details on the actual behavior you can consult the Couchbase documentation.

RavenDB uses a technology known as Write Ahead Log and MVCC to guarantee full ACID protection. It will hold duplicate versions of modified data as long as there are active transactions that may need to access old data. Once those transactions are complete, the space that the old values used can be immediately repurposed. There is no need for VACUUM or ongoing maintenance. The amount of space used for metadata overhead is minimal with a good balance between performance and disk utilization.

During compaction, Couchbase may require significantly more disk space. In our scenario, we were writing each document once, with no updates or deletes. We still observed compactations multiple times during the ingestion process (See Fig. 2).

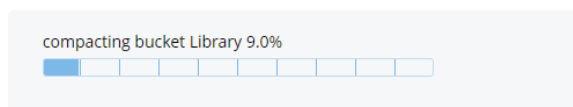


Fig. 2 Couchbase Compaction process

When compaction started, disk utilization was extremely high and the IO queue length grew very fast. During the ingest process, we observed that the compaction process could run for several hours. A late compaction process was observed taking over 12 hours of processing.

With just 1TB disks, Couchbase filled the entire disk under those conditions, then crashed when it failed to write to the disk. We compensated that by using 2TB disks. Couchbase would then consume all the disk space on those as well.

A single secondary index used 450GB of disk space (See Fig. 3). Couchbase is maintaining index snapshots to be able to roll back in the case of a node failure. You can control the number of snapshots that are kept using the Max Rollback Points configuration option, but to keep the disk usage under control for the purpose of the tests a single snapshot was used.

```
"data_size": 82607640099,
"disk_size": 484163817472,
```

Fig. 3 Couchbase metrics showing the size of a secondary index and its disk space usage. The index size is 77GB but the size on disk is 450GB

The index snapshots as well as the index itself running compactations would easily consume the entire available disk space. The final index file was 1.2TB in size, but under compaction it could balloon to 2.5TB. We were required to increase the disk size of the nodes up to 6TB in order to successfully complete the ingest and indexing processes.

3. Disk usage behavior, for data / index compaction as well as the snapshot feature shows that Couchbase requires a significantly higher number

of IOPS than the observed for RavenDB during the execution of the same task. Both peak I/O and sustained IOPS were higher than for RavenDB.

In the cloud, you pay for IOPS and you may be using burstable disks, resulting in higher financial costs.

The Test Environment

Rakuten Kobo's scenario has millions of devices that will sync periodically to the cloud with new books or annotations for each user. In many cases, the load is generated by automated sync processes and not just by users' actions.

Kobo has sold a *lot* of devices, so the work generated to the service can be expected to be a fairly constant load from a multitude of devices. Slow queries cause requests to pile up and devices may timeout. In that case, they'll retry the sync operation at a later time, deferring the load.

Infrastructure must also deal with less frequent operations. Things like users getting a new device and syncing their entire history, or pulling an old device from a drawer and starting to read.

The analysis team focused on the most common scenarios to benchmark:

- **Highlights by user.** Get the first page of all highlights for a user across all books and filter on a single property.

- **Highlights for a book and a user.** Get the first page of all highlights for a user in a specific book and filter them on multiple properties.

- **Users by ID.** When we know the document ID and can fetch it directly.

Tests for the first two queries were performed by selecting the first 100,000 users with the highest number of books and highlights in order to stress the storage solution. Then, we used a random sampling from that list to generate requests on behalf of those users causing the load to match a random-access pattern. The duration of the test and the number of requests ensure that we'll cover the entire 100,000 users.

Tests for the last query, Users by ID, were done on a list of all users, with each query requesting a different user by ID. The reason to test this use case relies on the fact that it is a core access pattern when using NoSQL databases and websites usually construct URLs that exhibit this access pattern.

These are absolutely over the top access patterns, but Rakuten Kobo will be building the next gen infrastructure and they need to understand how the system would behave when a tail risk event happens.

Rakuten Kobo needs to know what to expect on a *really really bad day*, not just when everything is smooth sailing.

Other scenarios were also tested (getting all the results for such a query, for example), but those tend to not be as performance sensitive and usually are broken apart to separate requests because of the data sizes involved.

Environment Configuration

For all tests, the operating system used was Ubuntu 20.04 with the default configuration. No modification to the operating system, kernel or base system configuration and the file system was ext4, with default configuration.

No caching was used by the web application to ensure that every request hit the storage.

RavenDB was tested on its default installation with a disk quota of 2TB.

Couchbase required changes to the default installation. Full ejection mode was necessary to avoid running out of RAM. We had to disable index snap-

shots or the 6TB disks used would not have been enough. We had to disable auto-rebalance to avoid triggering it during the ingestion process crashes.

It was our intention to run these tests on an Enterprise edition of Couchbase, but the licensing prohibited it to be used for benchmarking. Their community edition has no such restriction up to the day of publication of this technical report.

The versions tested were:

- RavenDB 5.1.12 – Released Dec 2020
- Couchbase Community 7.0 ver 3739 (beta) – Released Nov 2020

Couchbase Community was used since the Couchbase Enterprise's license forbids the publishing of benchmarks.























ROLE	Couchbase Cluster (3 servers)	RavenDB Cluster (3 servers)			Web application	Load test
						
		CONFIG 1	CONFIG 2	CONFIG 3		
MACHINE TYPE	m5a.8xlarge	m5a.8xlarge	m5a.4xlarge	m6g.large	m5a.8xlarge	m5a.2xlarge
SPECS	 32 cores  128 GB RAM  6Tb storage	 32 cores  128 GB RAM  2Tb storage	 16 cores  64 GB RAM  2Tb storage	 2 ARM cores  8 GB RAM  2Tb storage	 32 cores  128 GB RAM	 8 cores  32 GB RAM
RESERVED COST PER MONTH	\$ 5,917.79	\$ 3,767.39	\$ 2,816.93	\$ 2,080.21	\$ 632.91	\$ 158.41

Fig. 4 Testing hardware configurations

The Environment

We used cloud instances running on Amazon EC2 to make it easier to reproduce this benchmark and to provide a standardized cost calculation for the deployment.

For this report, we used several hardware configuration. All of them based on specific instance types in Amazon EC2. You can reproduce these finding by running the same instance types. (See Fig. 4)

For both databases the data was stored on io2 (provisioned IOPS) disks with 4000 IOPS.

The disk selection was done up-front to ensure fast access IO. All instances were running in the same availability zone inside a single region to reduce network latency.

For RavenDB, disk performance wasn't a major factor. We could have run with a gp3 (SSD) without any change in behavior. In a production scenario, it

is preferable that nodes in a database cluster run on separate availability zones to maximize survivability.

We wanted to compare, as much as possible, the same scenario. So we used a three nodes cluster for both databases. For RavenDB, that means a proper production deployment, each document residing in three separate nodes. For Couchbase, we were forced to set things up so each document would only reside on a single node.

On RavenDB, the entire dataset was stored on each node and we tested multiple cluster configurations with 3 nodes for high availability setup:

- m5a.8xlarge with 32 cores and 128 GB RAM
- m5a.4xlarge with 16 cores and 64 GB RAM
- m6g.large with 2 ARM cores and 8 GB RAM

For Couchbase, we tested the scenario using a cluster of three m5a.8xlarge instances with 32 cores and 128 GB RAM. The data was sharded among all three nodes with no replication (each document stored only on a

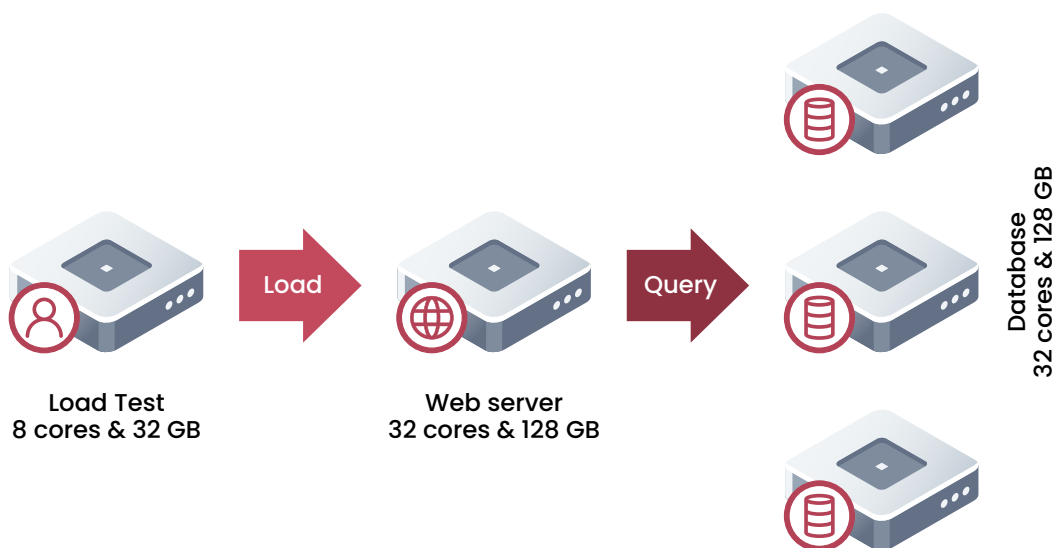


Fig. 5 Testing environment architecture

single node) to avoid increasing the size of the cluster nodes even more .

The load was generated using *wrk2* on the load test machine, targeting an ASP.NET Core application running on the web server which queries the database backend. (See Fig. 5)

The code for the web applications as well as the load generation scripts are available at: <https://github.com/ravendb/kr-benchmark-scripts/tree/mar-2021>

The Data Model

You can see a sample document in Listing 1.

The *text* field in Listing 1 contains the actual highlighted text. The other fields in the document include the book and the starting location for the highlight.

The ID of the document is composed of the following parts "highlights/{userId}-{ebookId}/{highlightId}". The reason for this document id setup is to allow us to perform the most common

queries by user and by user and book, using a simple prefix search on the document ID.

On Couchbase, all the data was sharded between each of the nodes, so each one of the servers held about one third for the data, or about 450 million documents. All the documents were held in a single bucket.

On RavenDB, replication between the nodes was used. The end result is that each node has a copy of the entire dataset (1.35 billion documents). As the full data set is available on all the nodes, requests are load balanced among nodes in the cluster.

The Queries

Both queries tested involved requesting the first page of highlights for a particular user or a particular user and book.

On RavenDB, for querying the document ID by prefix, the analysis team focused on two ways to query the data (See Listing 2 and Listing 3).

```
{
  "text": "The squabs are ready for market in four weeks...",
  "book": "ebooks/56717",
  "user": "users/5101859",
  "start": 17665,
  "at": "2011-10-16T15:49:15.1660000Z",
  "@metadata": {
    "@id": "highlights/users5101859-ebooks/56717/00002180997826-A",
  }
}
```

Listing 1. An example of a Highlight document


```
from Highlights
where user = $userid
limit 10
```

Listing 2. RavenDB – User's highlights query

```
from Highlights
where startsWith('id()', $prefix)
limit 10
```

Listing 3. RavenDB – User's highlights for a specific book using prefix query

The first allows us to perform an exact search over the Highlights collection and the second performs a prefix search on the document IDs, taking advantage of the nature of the document IDs used.

Given that the id of the document has the form: "highlights/{user}-{ebook}/" (a common pattern used in RavenDB and other NoSQL databases), we are able to get a record by user and book using just a simple prefix query.

This distinction forces RavenDB to access the data in 2 different ways. The prefix query retrieves the data directly from the storage, while the exact search, the first query, is forced to pass through the indexing engine.

While the exact search query can be engineered to avoid using the indexing mechanism, for better performance in a production system, it is defined here this way for the purpose of these tests.

For Couchbase we used matching queries (See Listing 4 and Listing 5).

```
select raw a from Library a
where a.`@metadata`.`@collection`
      = 'Highlights' and a.`user` = ?
limit 10
```

Listing 4. Couchbase – User's highlights query

```
select raw a from Library a
where a.`@metadata`.`@collection`
      = 'Highlights' and a.`user` = ?
      and a.book = ?
limit 10
```

Listing 5. Couchbase – User's highlights for a specific book query

All queries were parameterized using the client API.

We also implemented the second query using a prefix search on the ID for Couchbase (See Listing 6).

```
select raw a from Library
where META().id like
      'highlights/users/51018-ebooks/567/%'
limit 10
```

Listing 6. Couchbase – User's highlights for a specific book using prefix query

A primary index was put in place to do an efficient prefix search on document ids as *explicitly referenced in the documentation*. With the recommended method we observed very high CPU spikes (80%+), very high latencies and timeouts at just a hundred concurrent requests.

Indexing

Both RavenDB and Couchbase have asynchronous indexing processes. We let the indexing task complete and run all the queries without any contending outstanding writes.

RavenDB allows you to either define indexes explicitly or let the database engine figure out on its own what fields are of interest. RavenDB will create automatic indexes to cover those interesting fields and maintain such indexes automatically. For the purpose of this test, the *user* field on the **Highlights** collection was indexed explicitly as this is the recommended practice for large production databases.

For Couchbase, we defined two GSI indexes on the **Highlights** collection. One to cover the *user* and *book* fields, and one primary for the primary key that remained unused. In the current Rakuten Kobo's prototype implementation, views were used for this purpose but under the tests conditions their current implementation showcased higher query latencies. Therefore, we selected GSI indexing and N1QL queries to compare against.

Testing

The main driver for the next gen infrastructure is to prioritize responsiveness under load spikes than raw throughput. The key metric selected was latency of requests. Each test simulates a load spike over a period of three minutes, ensuring the cluster is stressed enough while background operations, garbage collections and maintenance/cleanup continue to be executed.

The tests were run with the wrk2 tool with 128 connections across 8 threads. The benchmark scripts as well as the web application that talk to the database can be found at: <https://github.com/ravendb/kr-benchmark-scripts/tree/mar-2021>

Service Level Agreements are usually expressed in the percentage of requests that must complete under a specific latency goal; thus, these are the most interesting numbers when you need to select your database.

At Rakuten Kobo the user experience is of paramount importance. After understanding the requirements we focused on the 95-percentile and 99-percentile which are the most likely to generate timeouts at the clients and impact user experience.

For the purposes of this study, we agreed before starting that the acceptable maximum latency was 200ms.

Users and Books Highlights Queries

The **Users' Highlights** query was tested on both databases in increments until the 200ms threshold was reached on the reference cluster (3x 32 cores with 128 GB of RAM). As shown in the latency distribution, Couchbase arrived at the 200ms threshold in the 20-percentile at 250 requests/sec - failing the test in the first run. RavenDB could handle up to 15,000 requests/sec before reaching the predefined threshold (See Fig. 6).

Of those 15,000 requests per second, 93% of the requests were served within 200 ms and over 85% were below 50ms. At 5,000 requests per second the

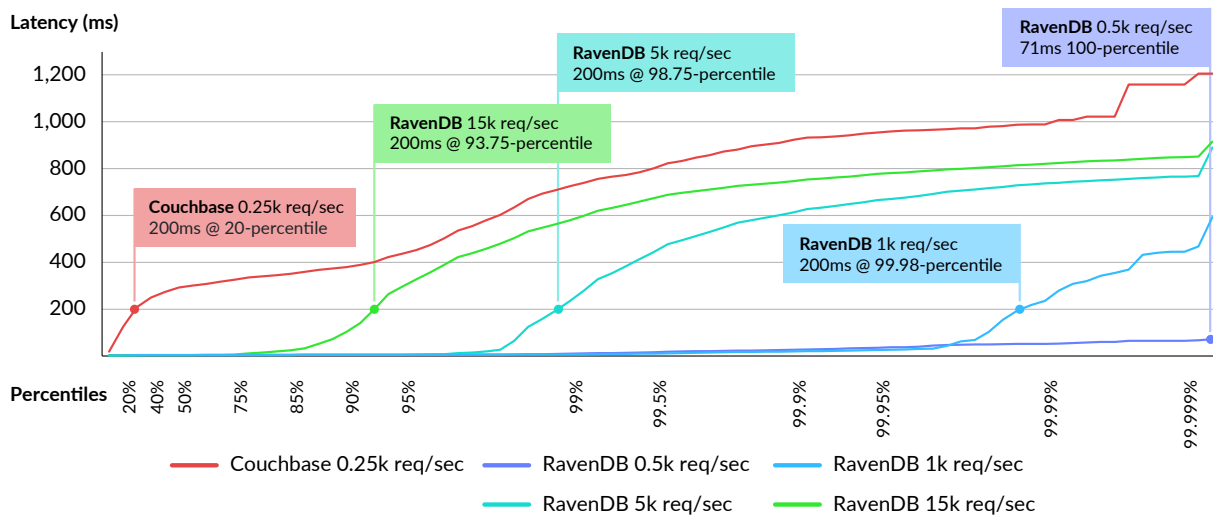


Fig. 6 Latency distribution in milliseconds for the highlights for user query (lower is better)

excess capacity enabled 97% of the requests to be served under 20ms.

In the **User's highlights** query case, we are testing the performance of using an index to find results, and then fetching them from the document store for both RavenDB and Couchbase. If you design your document id structure, you can issue some queries directly on the document store, bypassing the need for an index entirely, which is exactly what the **Books' Highlights** query is doing for RavenDB (See Fig. 7). We attempted to do the same for Couchbase, but found that the CPU cost was immense and the cluster was unable to maintain even a rate of 100 requests per second with key prefix queries.

The **Books' Highlights** query has a more stable behavior with RavenDB being able to handle effortlessly up to 99.9% of the requests below 50ms on the reference cluster setup. Being able to query the storage directly without using the indexing engine ensures the throughput is not taxing the system. At the next increment on this test (30,000 requests/sec) the client machine becomes the bottleneck

and measurements become unreliable after the 99-percentile and would require a distributed load infrastructure that was not available to the analysis team in time of the creation of this report.

These numbers were obtained with equal processing hardware: 3 nodes at 32 cores and 128 GB per node. Rakuten Kobo concluded that this scenario could not be satisfied with Couchbase as the provider.

Accessing the Data by Key

As a reference to understand the impact access to IO has, we decided to test both databases in two different scenarios. In the Cold-Start scenario, the test has to be done after rebooting the system to ensure the Operating System buffers do not contain pages belonging to the data and therefore every request requires to hit the disk.

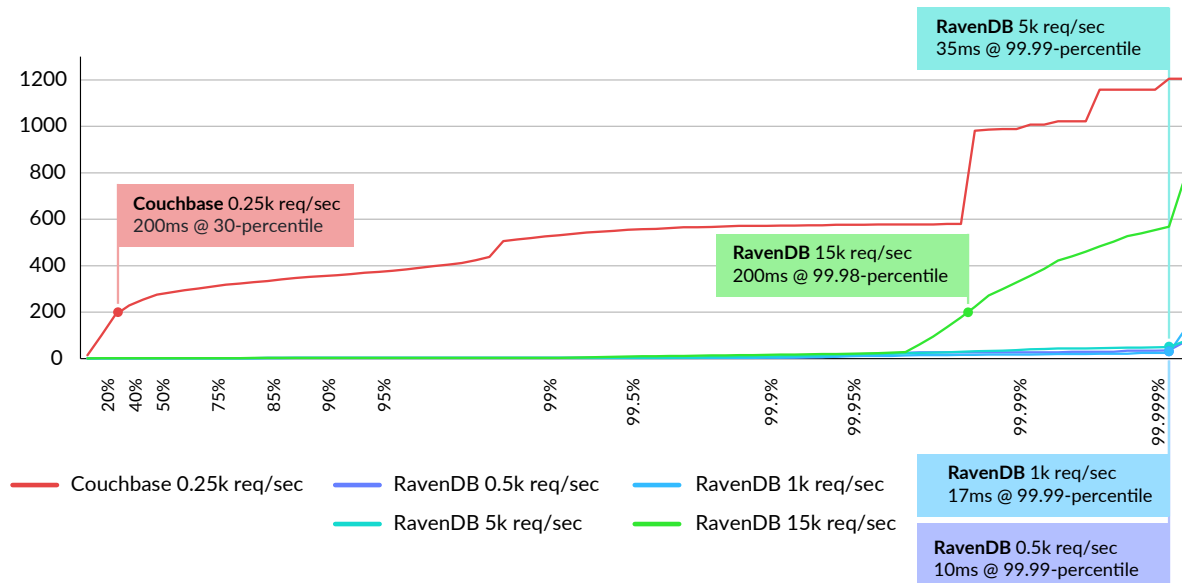


Fig. 7 Latency distribution in milliseconds for books query (lower is better)

In the second scenario, a warming run process was executed on RavenDB before the actual test to ensure a reasonable preloading of the entire database into memory to mimic Couchbase startup preloading keys and values into memory. With Couchbase, a node will not go online until it loads all the document keys from the disk. Every time a node goes up it needs to read 450 million keys and values before being able to serve requests. The forced preloading process impacts startup time and availability.

The access latency was tested until the client became the bottleneck. We concluded that this is the scenario where Couchbase shines. Having both the data and keys in-memory pays off in diminished latency. When used as a persistent caching solution the resulting latencies are very stable across the board.

When in Cold-start, Couchbase was not available to serve requests for ~13 minutes and therefore it was not included in the analysis. More on this in the operational concerns section. Conversely, RavenDB was available for servicing requests after a few seconds of starting the server at a reduced performance level.

Even for the cold start scenario, RavenDB was able to handle over 95% of the requests in the 200ms allotted time, and as it was able to move things to memory, performance improved steadily over time.

We asked specifically about the impact on the operations, Trevor gladly gave us a very detailed explanation of what this means for Rakuten Kobo, the key takeaway was:

"When we switch primary nodes for RavenDB, we notice performance drop for a while. Took us by surprise a few times, but wasn't any major concern."

Trevor, Rakuten Kobo CTO

On the other hand, when running the same scenario with Couchbase, the node is entirely unavailable while it is reading from the disk. In the case of failure, this means that RavenDB will be back and running within seconds, even if it needs some ramp up time. Couchbase will take many minutes to start fielding any requests. From the Operations team point of

view, you can imagine which situation is preferred. We'll discuss this more in the Operational Concerns section.

During the Cold-start test, there were no outstanding writes happening, so Couchbase behaves as an in-memory cache. Conversely, RavenDB doesn't have a facility to preload the entire database in memory and will do the loading on-demand in the same way it would handle databases several orders of magnitude larger than the available memory per node. (See Fig. 8)

When warmup is included, up until the 99.9-percentile with 30k request/sec, the results are almost indistinguishable. At those levels, a single request that has to hit the disk would negatively impact the results. Observations of the RavenDB CPU consumption never moved higher than 50% on any of the nodes of the clusters during the test time, and the average time for all requests was 4.52 ± 3.24 ms (4.52ms as the mean and 3.24ms as the variance). As the client starts to become the bottleneck, we couldn't increase the load higher to distinguish if that is a measurement

artifact by a few hits to the disk, or the actual behavior at that load point.

In the otherwise stable behavior at 10k requests/sec it can be seen that the latency at those levels where both systems have excess capacity is excellent. All of the requests were able to be served at less than 60ms with a staggering low latency of 6ms for the 99-percentile for both solutions. (See Fig. 9)

Discussion

Both solutions performed admirably when accessing by key for less than 30k requests/sec at almost the same price point. Couchbase has a slight advantage due to keeping all the data in memory, but this comes at a higher cost of vastly increased startup time as well as much higher disk usage. Beyond 30k requests/sec the conclusion is that a deeper investigation and an advanced distributed testing infrastructure would be needed to be done to rule out measurement artifacts.

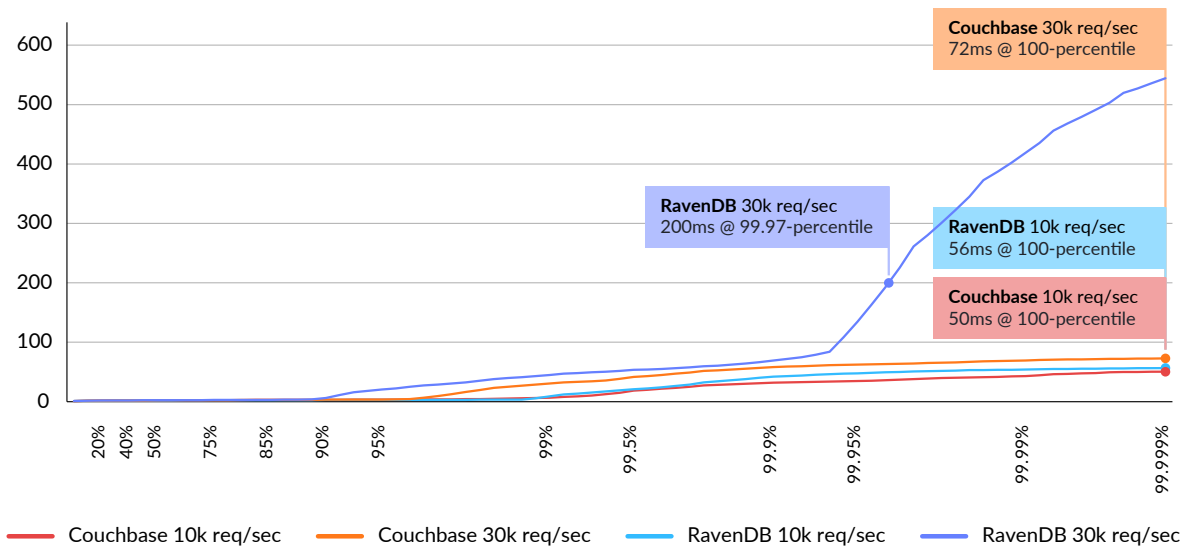


Fig. 8 Latency distribution in milliseconds for get-by-id query (lower is better)

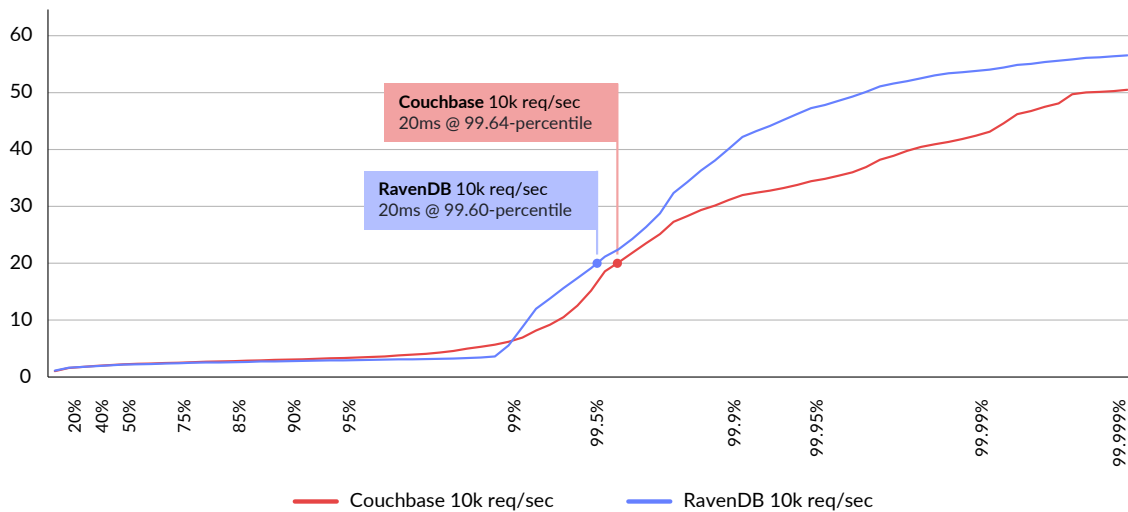


Fig. 9 Latency distribution in milliseconds for get-by-id query (lower is better)

When indexing is required, the conclusion is that an alternative solution like Elasticsearch would be required when using Couchbase.

Under the recommendation of the RavenDB Performance Team, noticing the extra capacity on the 10k request/sec scenario, a few more tests were run with downsized hardware to understand scaling costs.

Because of how Couchbase is designed, the minimum cloud hardware required to handle the benchmark on AWS is a cluster of 3 nodes of m5a.8xlarge instances (32 cores and 128 GB of RAM) with the data sharded among the nodes, without replication. That isn't a viable production configuration, of course.

Looking at the [sizing guide for Couchbase](#), and taking into account that a production environment cannot run with a single copy of the data, we estimate that a production cluster to serve this scenario would take 5 servers with 192 GB each. This assumes a working set of 20% of the data to reside in memory and three replicas for each document.

On premise, adding more RAM has a rather marginal cost, but on the cloud the relevant instance for the requirement is m5a.12xlarge, which comes at a 48% premium. A cluster of 5 such machines would have a total of 240 cores and 960 GB of RAM and a total disk usage of 30 TB.

The recommended setup for production would be bigger than the one showcased here and it would cost significantly more as well. When selecting a database solution, one does not look simply at the performance numbers, but also at what resources it takes to achieve them.

This benchmark puts a very high bar for passing. Handling thousands of queries per second with low latency is a load very few applications need to face. Therefore, we decided to see how far we could step down the hardware requirements for RavenDB and what would be the acceptable performance at each price point.

For RavenDB we selected the default setup for the cluster as reference which matches Couchbase performance to compare against: m5a.xlarge (4

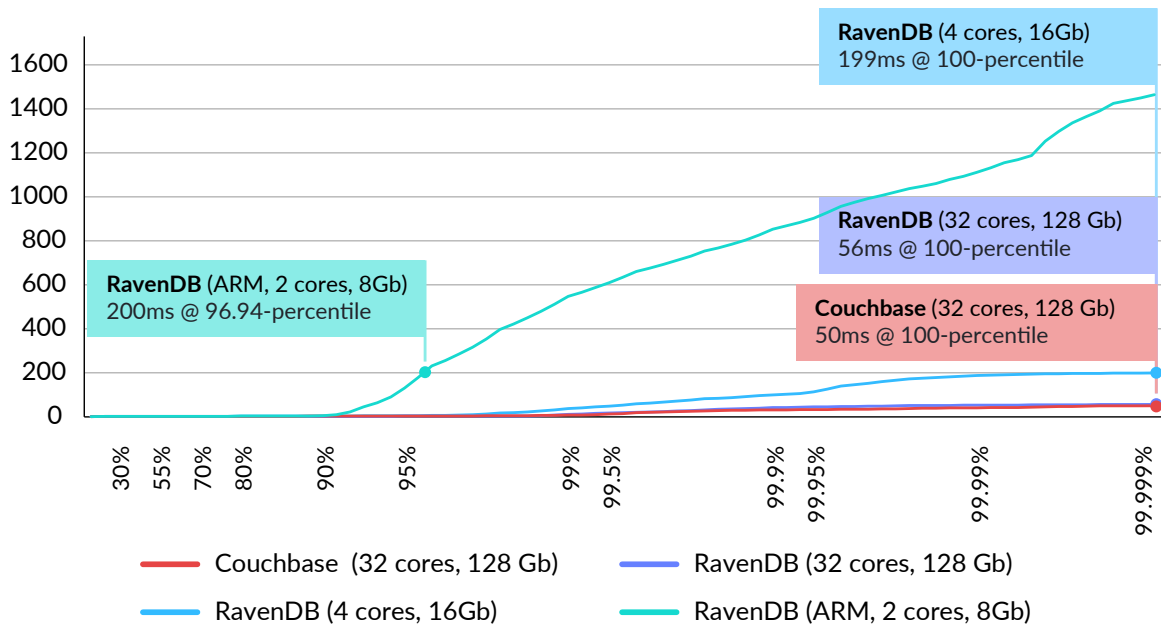


Fig. 10 Latency distribution in milliseconds at 10k requests/sec for get-by-id query (lower is better)

cores, 16GB) and a Graviton ARM m6g.large (2 cores, 8GB). (See Fig. 10)

RavenDB is able to handle gracefully 10,000 requests per second at the 99-percentile with a latency below 200ms on a 2TB database using an ARM Graviton processor (2 cores, 8 GB of RAM) at an annualized cost of \$1,185 + storage costs for the entire cluster.

For more performance conscious applications at the 99.9-percentile, a cluster composed of three m5a.xlarge nodes is able to serve 10,000 requests per second with latencies below 110ms at an annualized cost of \$2,658 + storage costs. The reference cluster would cost \$24,699 per year without including storage cost at comparable latency.

For the cases of the Books and Users highlight queries, comparisons were performed against a RavenDB reference cluster. Couchbase was not able

to complete a test run at 250 requests per second without timeouts.

If the solution requires the usage of queries, there is not enough spare capacity at the 1,000 requests per second to be able to downsize the environment up to 2 cores Graviton range (See Fig. 11). However, as shown in the mid size cluster, it is possible to sustain up to 1,000 requests per second within the threshold even using queries. The memory size and limited core count of the smallest Graviton cluster cannot sustain 1k request/sec.

The absolute downsizing limit for half the load at 500 requests per second was found to be even lower than the ARM system. The system which was used for comparison in this report was a single Raspberry PI 4 with 4GB and a USB connected SSD. At that reference load, it is able to sustain at the 99-percentile a latency well under 50ms as shown below. (See Fig. 12)

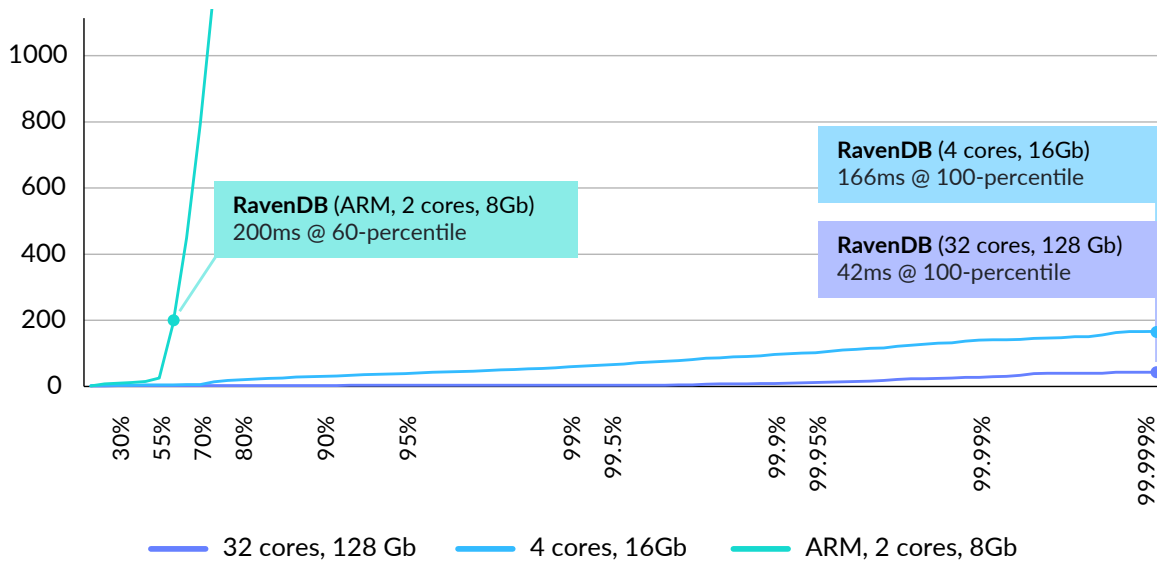


Fig. 11 RavenDB latency distribution in milliseconds at 1k requests/sec for users query (lower is better)

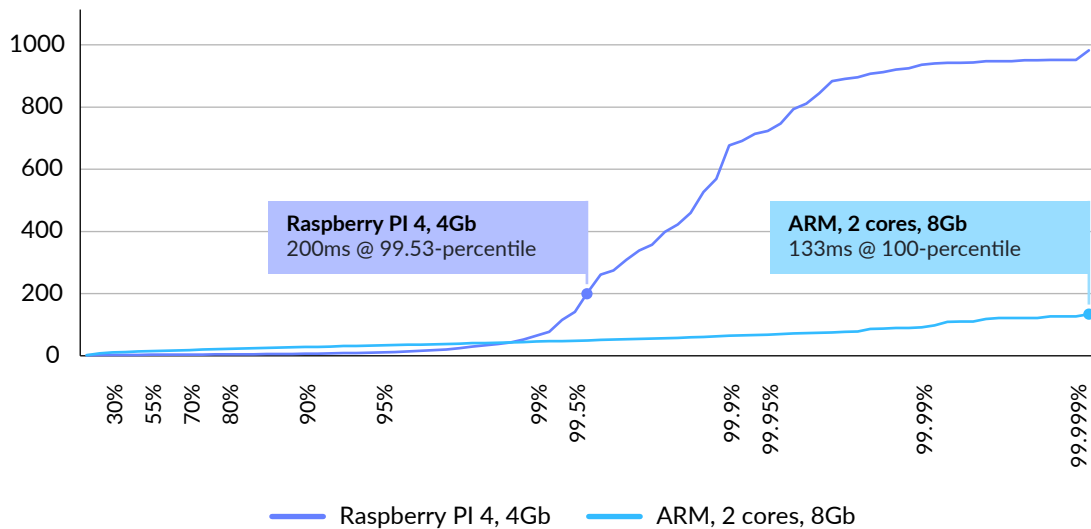


Fig. 12 RavenDB latency distribution in milliseconds at 500 requests/sec for users query (lower is better)

Couchbase cluster of three nodes with a total of 96 cores and 384 GB RAM was unable to sustain 250 queries per second. A single RavenDB node running on a Raspberry PI 4 with 4 cores and 4 GB of RAM was able to answer 500 queries / second in under 200ms in the 99 percentile.

If the system can be engineered to ensure that requests are dominated by high performance prefix queries, even at the smallest cluster composed with nodes of 2 cores with 8GB of memory, the querying system can sustain 10,000 requests per second within the 200ms threshold for the 99-percentile. (See Fig. 13)

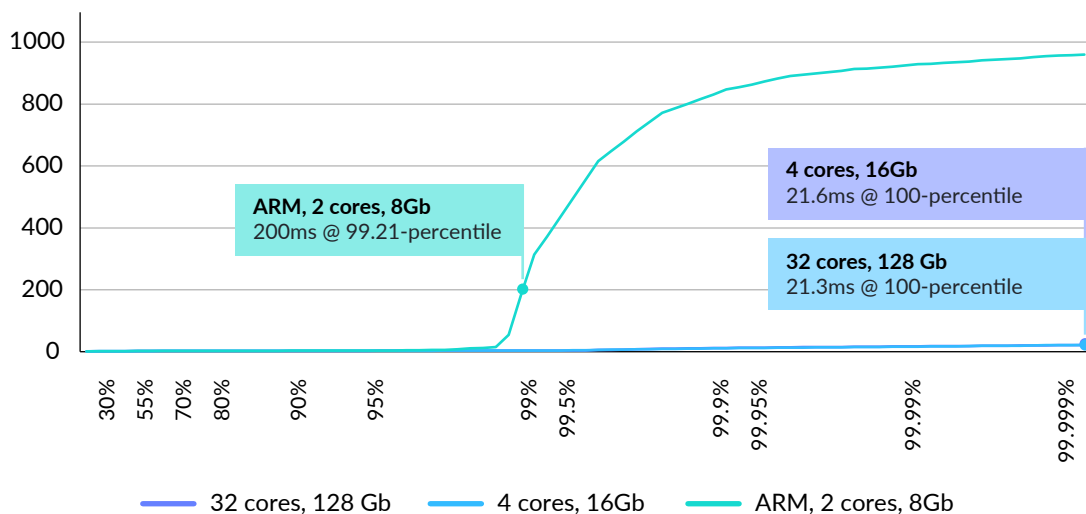


Fig. 13 RavenDB latency distribution in milliseconds at 10k requests/sec for books query (lower is better)

Operational Considerations

Reliability is key for Rakuten Kobo operations.

"Resilience and fast recovery isn't just about surviving a node crashing. They are essential for good hygiene too"

Trevor, Rakuten Kobo CTO

Therefore, we evaluated other scenarios of interest in relationship to the performance behavior under duress. Failure recovery is one of the most common situations where under a bad situation we may be adding more stress.

For high availability, the deployment of a cluster is a must. When some of those nodes fail, the cluster needs to handle that and recover. Both systems handle such failures automatically and transparently

by default, however there are important differences between their behavior.

In order to understand what to focus on, we interviewed key people at Rakuten Kobo about the key pain points on their current deployment.

A node failure in Couchbase will trigger an attempt to rebalance the data between the surviving servers. In the case of transient errors, this can lead to spikes in database loads at the moment of the fault. Faults will cause overhead on top of the reduction in the capacity caused by a fault of a node in the cluster. Therefore, it is expected that Couchbase nodes should be overprovisioned with extra idle compute capacity on standby in case a node goes down and self-healing behavior is triggered.

RavenDB systems are based on a cooperative process between the clients and the servers. A failed server does not trigger any special action on the cluster, the clients are already aware of the succession node list and will failover to the next server immediately. Transient errors will simply cause a redirection of

traffic into other nodes without the end application noticing that something happened beyond a few requests that may be slower than usual.

Behind the scenes, RavenDB will monitor the state of the node and its recovery. Experience has shown that it is rare to completely lose a node, so RavenDB defaults to ensuring the liveliness of the system and waiting for the node to return.

RavenDB Enterprise edition is able to automatically ensure the appropriate number of replicas for the data on the failed node is maintained. That is useful if you expect a node to go down and stay down for a long period of time. That requires an extended outage to trigger, in the order of minutes. In the meantime, the cluster and the clients will automatically adjust the load, without the need for expensive operations.

These rebalancing operations are only triggered after a sensible time has passed in order to avoid initiating maintenance costs after server restarts or network glitches. All maintenance operations are performed on the cluster's spare capacity, and the cluster will always prioritize users' requests over background operations.

A RavenDB node failure is not treated as a priority operation to allow DevOps to handle seamlessly rolling updates for the clusters. Updating the cluster is a routine operation where nodes are taken down one at a time in order for DevOps teams to perform maintenance operations and then rejoin the cluster.

RavenDB expects nodes to fail and handles that gracefully and seamlessly. By making the failure of a node a non-event, RavenDB provides the Operations teams with the ability to treat the nodes as hot spares. You can take one down at any time, for any reason, and nothing major will occur.

Couchbase behaves in a similar way, in theory. A node is allowed some downtime before automatic steps are taken to rebalance the cluster. A confluence of design choices may allow those small failures to have cascading impact. During startup, it will need to read the entire metadata library on the server into memory. With big databases like the one tested here, it ranged from 10 to 20 minutes on an io2 drive with 4,000 provisioned IOPS to be able to serve requests.

"We didn't upgrade the Couchbase version for years because we were fearful of taking a node down. Additionally, if we needed to increase the capacity of the cluster we had to add a node. And that would also cause a rebalance and outage."

Trevor, Rakuten Kobo CTO

Even after a Couchbase node is up, it takes even longer for indexes to become available. After a node failure, we observed that even after the node came back up and loaded the document's data, the index remained in a warmup state for a *long* time. In one scenario, a single index was still in the warmup stage after 30 minutes from the node restarting, during which time no queries could be served by those indexes.

There are many reasons why the Operations team may want to restart a node. Patching the database software or the underlying operating system is probably one of the most common reasons. However, if such operations will cause a rebalance, with its associated costs, it will be avoided at almost any cost.

Certain operations in Couchbase, like changing the hostname of the node, will require the node to be removed from the cluster and then rejoined, inducing a rebalancing operation. At the sizes tested, a rebalance operation takes 40 hours.

On the other hand, RavenDB uses a write ahead log. The only work a restarting or rejoining node needs to do on startup is to replay the unregistered transactions, which is independent of the database size. If no writes had happened between leaving and rejoining the cluster, no work needs to be done. And only the new writes will be replicated when the node joins.

After some time of Rakuten Kobo using RavenDB in production we came back to this particular question as it was of particular importance for our team.

“We're able to keep up with every minor release. Shutting down a node, upgrading the RavenDB version and restarting it is a non-event and one that takes under a minute. That's unthinkable with what happens with Couchbase and its auto-rebalancing.”

Trevor, Rakuten Kobo CTO

The Bottom Line

In every large-scale deployment, efficient software generates benefits at the bottom line. At the same performance level the current hardware budget can be reallocated to new services and other uses when the next gen infrastructure is rolled into production.

The Couchbase reference cluster is the bare minimum for the database size with no replication and no data redundancy. We tried to downscale the cluster after ingestion to 16 cores and 64 GB RAM per node, but

the cluster suffered repeated failures due to memory exhaustion.

Using the guidance from [Couchbase's documentation](#), we estimate that this workload requires (at a minimum) a 5 nodes with 192GB RAM and a replication factor of 3. We selected the m5a.12xlarge AWS instance with 48 cores and 192GB RAM as the relevant instance type for our computations.

The following chart (See Fig. 14) summarizes the annual cost of ownership at the different sustained load requirements between the competing solutions where both can successfully finish the scenario.

The Budget Cluster specification is designed around being able to trade off cost for latency at the 95-percentile level, while the High-Performance Cluster will try to match or surpass the capability at the 99.9-percentile level.

The following numbers show just how much hardware you'll need to use to meet your performance goals with Couchbase and RavenDB.

It's important to note that Couchbase according to our tests is unable to actually handle queries with any real load associated with them. In our estimation, in addition to the Couchbase cluster, you'll need to also run a separate system for queries. For example, using Elasticsearch to query the data, and then loading the data from Couchbase.

RavenDB, on the other hand, is able to fill both roles, and at a much reduced price tag.

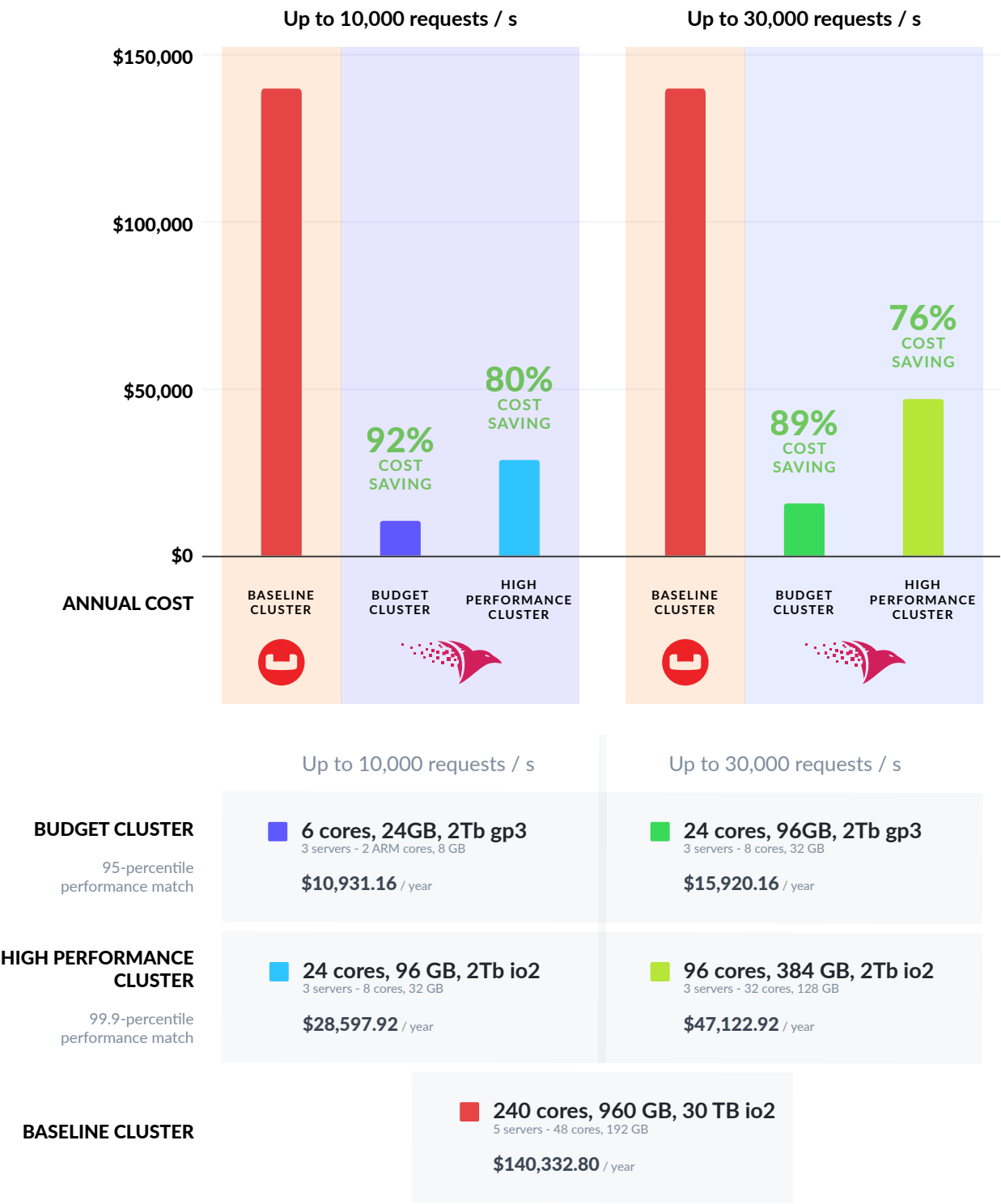


Fig. 14 Cluster costs at various workloads RavenDB and Couchbase. Achieving the same 99.99-percentile latencies



About RavenDB

RavenDB is a pioneer in NoSQL database technology with over 2 million downloads and thousands of customers from startups to Fortune 100 Large Enterprises.

Mentioned in both Gartner and Forrester research, over 1,000 businesses use RavenDB for IoT, Big Data, Microservices Architecture, fast performance, a distributed data network, and everything you need to support a modern application stack for today's user.

For more information please visit:

ravendb.net

Contact us at:

info@ravendb.net

Documentation

<https://ravendb.net/learn/docs-guide>

Use Cases

<https://ravendb.net/news/use-cases>

Free Online Training

<https://ravendb.net/learn/bootcamp>

Webinars

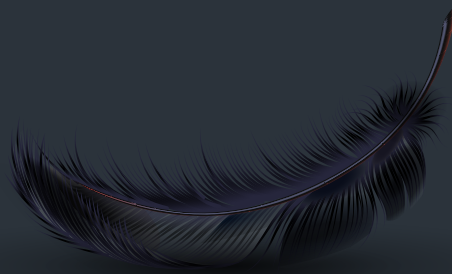
<https://ravendb.net/learn/webinars>

RavenDB Download

<https://ravendb.net/download>

RavenDB Cloud Database as a Service

<https://cloud.ravendb.net/>



Proudly developed by



Houston • Buenos Aires • Hadera • Toruń

US Number: 1-817-886-2916

info@ravendb.net

© Hibernating Rhinos, Ltd. All rights reserved.