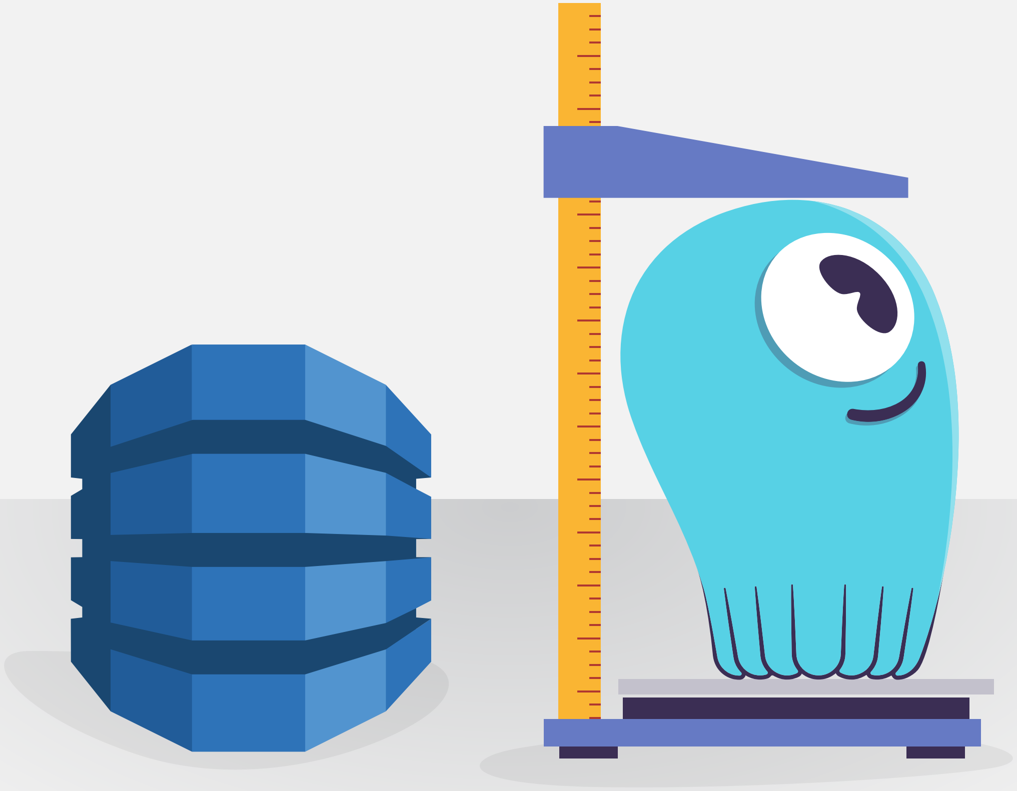
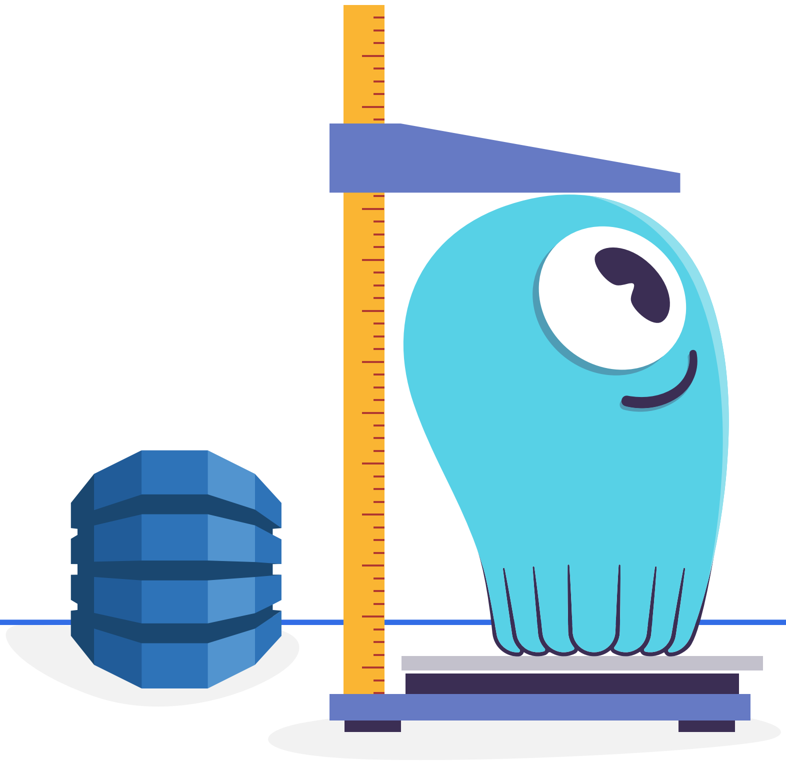


SCYLLADB COMPARISON

ScyllaDB

ScyllaDB vs. DynamoDB: Comparing Latency Across Workloads



Since early 2022, ScyllaDB has experienced a significant increase in the number of DynamoDB users moving to ScyllaDB Cloud, our database-as-a-service offering. Price performance is cited as a primary driver in virtually all of these interactions. ScyllaDB's cost is 50% of DynamoDB's cost. But the performance advantages vary. To help teams better assess whether a move makes sense, we decided to expand on our original DynamoDB benchmark with a detailed look at how latency compares across a variety of workload conditions.

METHODOLOGY

ScyllaDB and DynamoDB are architected and deployed in distinctly different ways that complicate comparison efforts.

For example, DynamoDB's provisioned model has you purchase throughput in terms of read capacity units (RCUs) and write capacity units (WCUs). If you need additional capacity, you can scale by purchasing additional RCUs and WCUs (as long as you respect the specified per-table limits). ScyllaDB, on the other hand, has no concept of provisioned throughput. You provision a cluster and get the maximum IOPS that the database can achieve with your specific hardware. If you require additional capacity beyond that, you scale out your cluster. This difference had to be accounted for in our test setup.

Additionally, the databases have fundamental differences in their architecture. DynamoDB stores data using B-trees, which makes it read-optimized. ScyllaDB, on the other hand, uses LSM trees. LSM tree-based databases are write-optimized – but ScyllaDB offers caching that is designed to reduce latency for reads. Assessing the impact of these differences required testing with a broad range of read:write ratios.

CONSIDERATIONS

In designing a comparison/ benchmarking scheme that would be as objective as possible, especially given these fundamental differences, we considered factors such as:

- Choosing the right number of loaders / threads / connections to find the sweet spot of each database's concurrency, considering their differences and limits. The Appendix provides details about an interesting scenario where we ran too few DynamoDB loaders due to Little's Law (yes, latency matters!)
- Choosing the right distribution and parameters (e.g. DynamoDB is best-suited for Uniform but Zipfian is more realistic – see the Appendix for a detailed discussion on distributions)
- Understanding loader implementation and accounting for issues (e.g. Coordinated Omission)
- Understanding database limitations and assessing their impact (e.g. DynamoDB's per partition rate limit)

SETUP

We ran benchmarks using YCSB 0.18.0+[1] on ScyllaDB Enterprise 2022.2 vs. DynamoDB. Here's what we provisioned:

ScyllaDB Cloud	DynamoDB	Loaders
Number of nodes: 3 Node instance types: i4i.2xlarge Region: us-east-1 Multi AZ (zones a, b, c)	Table class: Standard Provisioned throughput: • RCU: 200K • WCU: 250K Region: us-east-1	Number of loaders: 6 (later increased to 8) Loader nodes type: c5.2xlarge Region: us-east-1

(1) We used a slightly modified version of YCSB. See the Appendix for details.

Note that we opted to use relatively small deployments since we wanted to test throughput ranges to approximate average or smaller than average ScyllaDB deployments (e.g., 100K OPS). Previous extreme scale benchmarks demonstrated ScyllaDB's ability to achieve single-digit millisecond latency with 7.5M OPS. Since many potential users are not (yet) operating at such a massive scale, we opted to focus on the other side of the spectrum for this project.

The following loaders were used to run YCSB against both databases:

- Number of loaders: 6 to 8, depending on the test
- Loader nodes type: c5.2xlarge
- Region: us-east-1

WORKLOADS

Using YCSB, each database was preloaded with 1B rows, adding up to ~1 TB of data. The data model follows the standard YCSB "workloada" definitions: 10 fields of data, named "field0" to "field9", each of which has 100 bytes of information. This added up to a payload of approximately 1 KB per record.

We tested three workload distributions: Uniform, Zipfian, and Hotspot. DynamoDB is optimized for Uniform data distributions, so we selected the Uniform distribution to test DynamoDB within its sweet spot. Still, we also wanted to test with more realistic distributions that would be better representative of the real-world workloads handled by the majority of data-intensive applications. That's why we also tested Hotspot and Zipfian distributions. These distributions mimic real-world uneven demand for a particular set of data, as well as unforeseen spikes in demand that can occur when something "goes viral." The Hotspot distribution splits the data set into a cold and hot set, where the hot set becomes accessed more frequently than the cold set. In contrast, the Zipfian distribution has certain items exponentially more likely to receive access. In the context of a database, both distributions deviate from the regular uniform access, and introduce heavy loads on a particular piece of data. *See the Appendix for a detailed look at how and why we generated these distributions.*

For each distribution, we tested a broad range of read:write ratios, ranging from read-heavy to write-heavy.

CONSISTENCY

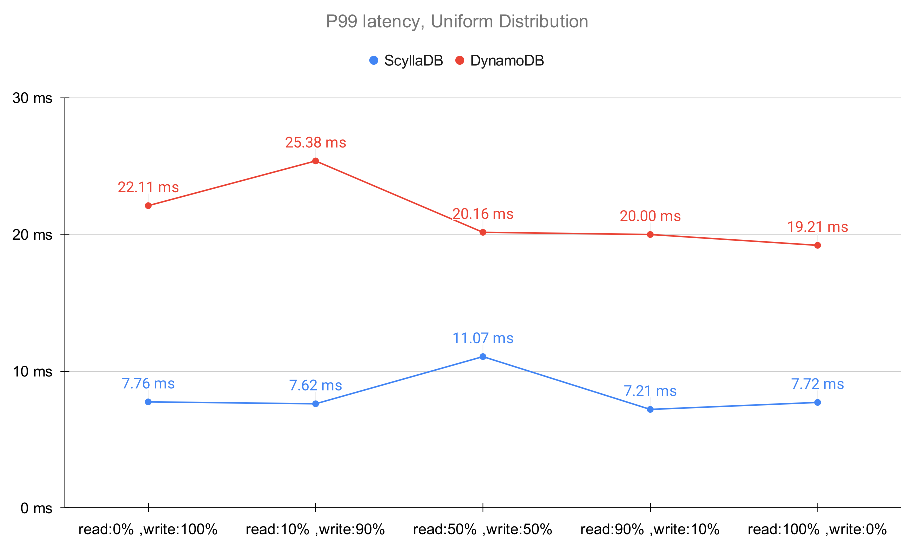
Across both databases, we wanted every successfully completed write operation to be immediately reflected in a subsequent read. For ScyllaDB, this was achieved by using QUORUM reads and writes. Achieving the same effect with DynamoDB requires using consistent reads. This is noteworthy because consistent reads are more expensive across both databases. For DynamoDB, there are direct impacts on costs. For ScyllaDB, it means less throughput and more latency than you would achieve with a consistency level of ONE, and that will ultimately impact costs because achieving the same level of throughput will require a larger cluster.

LATENCY RESULTS

For use cases that require near real-time responses (e.g., AdTech, messaging, gaming, IoT), P99 latency is critical for meeting SLAs and delivering an engaging user experience.

UNIFORM DISTRIBUTION

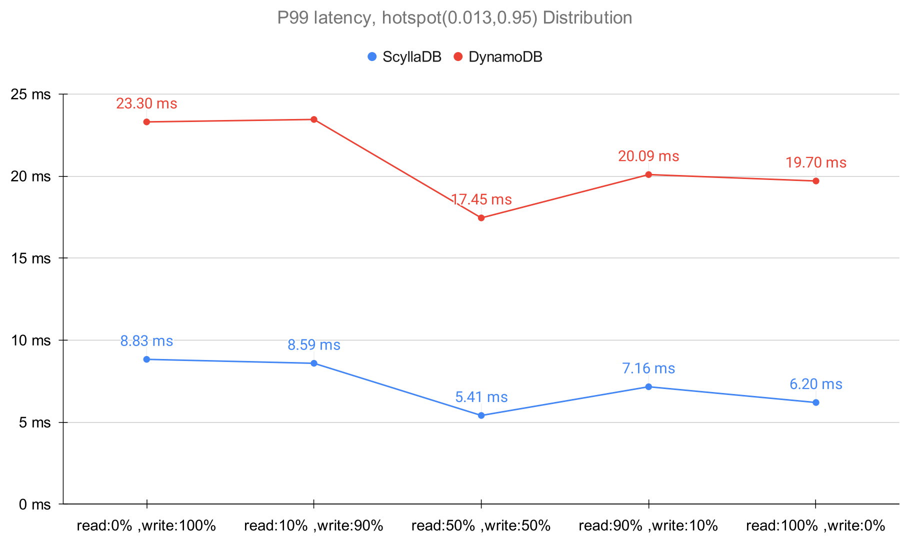
Here are the performance results for the Uniform distribution.



ScyllaDB's latency was significantly lower than that of DynamoDB. An even larger reduction in P99 latency and mean latency (under 1 ms) could be achieved by using a larger ScyllaDB cluster and reducing the utilization from 75% to the 30%-50% range. Additionally, ScyllaDB comes with several options which might further improve latencies, such as the BYPASS CACHE extension, designed for workloads that don't make effective use of caching (e.g., a workload that is much larger than the RAM with Uniform distribution). Such options have not been applied for the purpose of this performance testing.

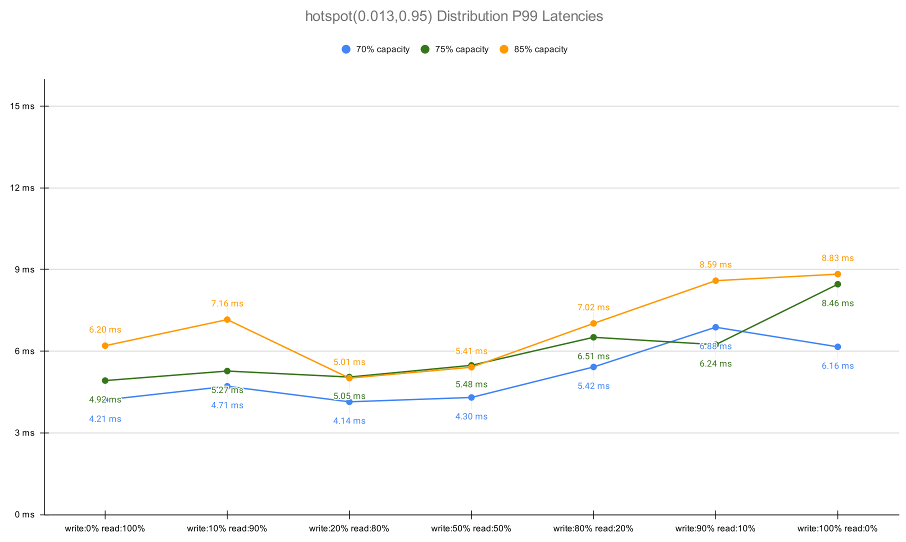
HOTSPOT DISTRIBUTION

Now, let's shift to a more realistic distribution: Hotspot. As explained in the Appendix, the Hotspot distribution has some sets of values accessed more frequently than others.



As you can see, the discrepancy between DynamoDB and ScyllaDB latencies widened for these more realistic workloads. Since a portion of the requests had a higher chance of hitting an object from the hot set, the probability of ScyllaDB's cache being touched during reads increased significantly. The results show that ScyllaDB P99 latencies decreased compared to the previous Uniform results as the read ratio increased.

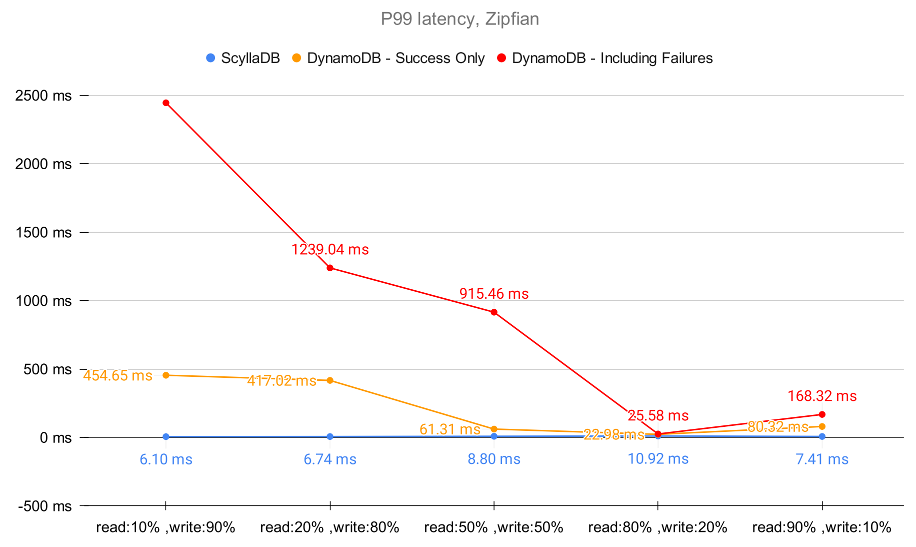
The next graph represents the P99 latency as a function of ScyllaDB's utilization. Note how the P99 decreases along with a lower cluster load:



ZIPFIAN DISTRIBUTION

Finally, let's look at the interesting case of Zipfian distribution. Zipfian distribution takes the hotspot to new levels by simulating access patterns with an exponential relation between items (a.k.a. "hotness"). As Alex Debrie notes, Zipfian is generally problematic for DynamoDB: "The most popular items are accessed orders of magnitude more than the average item. Because DynamoDB wants a more even distribution of your data, your application may get throttled as it tries to access popular items."

As expected, DynamoDB performed poorly on these workloads.



As noted by Debrie, the Zipfian distribution is likely to create hot partitions, an imbalance introduced by uneven item access from within the database. As hot partitions are a known antipattern, DynamoDB restricts the number of hits on the same partition (documented to be 3,000 RCUs and 1,000 WCUs per partition at the time of writing). Aware of this limitation and of Zipfian's imbalances, we compared both ScyllaDB and DynamoDB at scale.

Once DynamoDB limits were reached, requests started getting throttled, requiring the application to retry. As a result of the throttling and retries, the YCSB latency became severely impacted. Under some circumstances, we experienced DynamoDB throttling at up to ~2.5 seconds per request – thus preventing the application from accessing the item during that time.

At best, DynamoDB delivered 39.72% of the expected throughput (88.19 good kops/s vs. the target 222 kops/s). At worst, it delivered only 16.22% (20.28 kops/s) of what was purchased (125 kops/s) and supposedly provisioned. Given the documented DynamoDB limitations, some discrepancy is to be expected. However, the magnitude of this unfulfilled throughput was surprising.

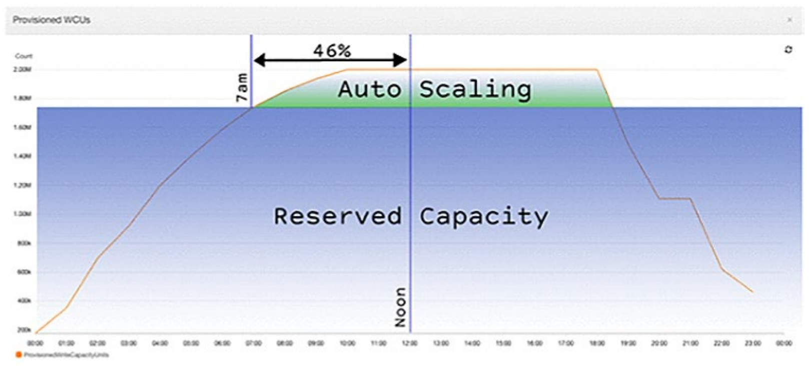
Unlike DynamoDB, ScyllaDB managed to sustain the target throughput without any throttling and still deliver single-digit millisecond latencies. In that regard, ScyllaDB does not impose any hard limits on querying hot partitions. Even though ScyllaDB implements concurrency-limiting mechanisms, frequently accessing popular items will typically benefit from its cache implementation – thus explaining why no failures have been seen during the Zipfian tests. Even then, it is worth underscoring that hot partitions are an antipattern, even for databases like ScyllaDB. Read more about ScyllaDB's advanced control mechanisms in ScyllaDB in this blog.

ADVANCED CONFIGURATIONS

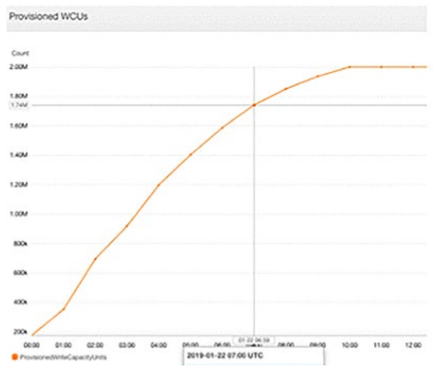
CACHING: DYNAMODB ACCELERATOR (DAX)

Given the latency results, it is interesting to note that teams using DynamoDB with latency-sensitive workloads and strict P99 requirements might require relying on DynamoDB Accelerator (DAX) to achieve their SLA targets. However, caching solutions add a significant increase to DynamoDB costs and complexity (driven by the constant need to monitor cost, reserved capacity, and scaling). ScyllaDB does not require any external caching component to sustain predictable low latencies. As mentioned earlier, ScyllaDB's BYPASS CACHE feature allows the user to give the database hints about which query results should not be cached, thus protecting against cache invalidation.

AUTO SCALING: A MIX OF PROVISIONED AND ON-DEMAND



Blended WCU reserved and auto scaling ratio



1.74 million provisioned WCUs at 7:00 AM

Once the DynamoDB cost grows, users often start to optimize it with a set of tools that AWS offers: combining on-demand and provisioned workloads, auto scaling, etc. However, this not only increases complexity; it could also incur limitations from the speed of elasticity and even throttling. Any cost savings achieved by adding this complexity are likely to decrease the gap between the two databases but wouldn't be able to close it, especially since ScyllaDB has a yearly plan and its own scale API.

Making ScyllaDB Even More Cost Effective

Unlike DynamoDB (where you provision tables), ScyllaDB is provisioned as a cluster, capable of hosting several tables – and therefore consolidating several workloads under a single deployment. Excess hardware capacity may be shared to power more use cases on that cluster. Users may rely on ScyllaDB's Workload Prioritization feature to assign different priorities to different tables (or to different use cases that share a single table).

For example, assume there are 10 use cases that require 100K OPS each. However, at any given point in time, only two of them will reach their limit. With DynamoDB, users would be forced to allocate a provisioned workload per table or to use the rather expensive on-demand mode. A standard ScyllaDB deployment is not only more cost effective. It also allows users to run all 10 workloads within a single cluster with a sustained capacity of 200k-300k OPS, and to share the idle time among all those workloads. Prioritized workloads will receive more CPU and IO resources from the scheduler, resulting in higher throughput and lower latencies.

FINAL THOUGHTS

What might begin at a seemingly reasonable cost can quickly escalate into “bill shock” with DynamoDB – especially as the throughput increases, and particularly with write-heavy workloads. This makes it a suboptimal choice for data-intensive applications anticipating steady or rapid growth. ScyllaDB’s significantly lower costs – a reflection of ScyllaDB taking full advantage of modern infrastructure for high throughput and low latency – make it a more cost-effective solution for data-intensive applications.

ScyllaDB – with its LSM-tree-based storage, unified caching, shard-per-core design, and advanced schedulers – allows you to maximize the advantages of modern hardware, from huge CPU chips to blazing-fast NVMe.

ScyllaDB is 50% of the cost of DynamiDB. And, beyond those cost savings, ScyllaDB sustains 2X peaks and provides 2X-4X better P99 latency. Additionally, it can further reduce latency when idle – or enable spare resources to be shared across multiple tables. For larger workloads spanning 500K-1M OPS and beyond, this can result in a cost saving in the millions – with better performance and fewer query limitations.

APPENDIX

A DEEPER DIVE INTO DISTRIBUTIONS

Uniform

Uniform distribution is the most basic distribution. For each request, each record has the same chance of being "touched." So for 1B records, the probability that a request chooses some record 'n' is $P_n = 10^{-9}$ for every record.

We do not believe this distribution is representative of most real-world workloads. However, we selected it because uniform distributions are reported to be DynamoDB's sweet spot. AWS states that "DynamoDB is optimized for uniform distribution of items across a table's partitions, no matter how many partitions there may be." The recommendation to use Uniform distribution for DynamoDB is also referenced in the YCSB properties file[3] and readme[4].

Hotspot

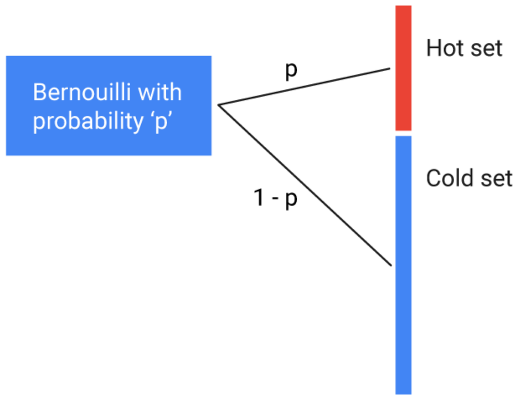
In addition to the uniformly distributed workloads, we also wanted to test with distributions that would be more representative of the real-world workloads handled by the majority of data-intensive applications. That's why we also tested Hotspot and Zipfian distributions.

A Hotspot distribution is an aggregation of two distributions, the Bernoulli distribution and one of two uniform distributions ("chosen" by the former distribution). Here, the records are partitioned into two sets: hot and cold. When generating an item from this distribution, first a "Bernoulli coin" is flipped to determine the next set to draw an item from, then an item from this set is picked with uniform distribution.

This distribution is configurable and takes two parameters:

1. The size of one of the sets (as a fraction out of the whole items set)
2. The Bernoulli probability of the defined set to be chosen

The second set size and probability can be determined from the parameters of the first one by simply subtracting them from 1.



Except for the pathological case where the Bernoulli probability is 0.5, there will always be one set that is "hotter" (hit more than the other set), hence the name, Hotspot. In this report, we note this probability as "hotspot(S, p)" where S is the size of the first set and p is the Bernoulli probability for this set.

A good schema design goal is to have the perfect, uniform distribution of your primary keys. However, in real life, some keys are accessed more than others. For example, it's common practice to use UUID for the customer or the product ID and to look them up. Some customers will be more active than others and some products will be more popular than others, so the differences in access ratios can vary significantly.

If certain items are exponentially more likely to receive hits, this resembles a Zipfian distribution.

(3) https://github.com/brianfrankcooper/YCSB/blob/ce3eb9ce51c84ee9e236998cdd2cefaeb96798a8/dynamodb/conf/dynamodb.properties#L82-L83

(4) https://github.com/brianfrankcooper/YCSB/blob/master/dynamodb/README.md#faqs

Zipfian

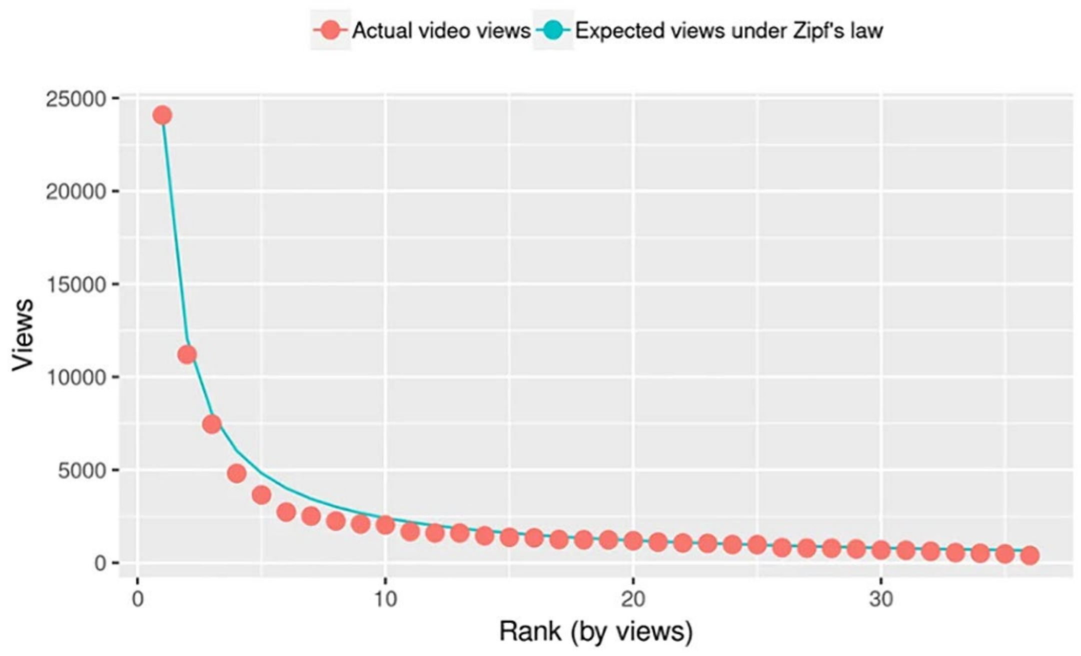
Zipfian distribution was originally used to describe word frequency in languages. However, this distribution curve, known as Zipf's Law, has also shown a correlation to many other real-life scenarios. It can often indicate a form of "rich-get-richer" self-reinforcing algorithm, such as a bandwagon or network effect, where the distribution of results is heavily skewed and disproportionately weighted. Over time, a certain search result becomes popular, and more people click on it. Thus, it has become an increasingly popular search result. Examples include the number of "likes" on social media posts or the activity distribution among Hacker News stories.

When these sorts of result skew occur in a database, it can lead to incredibly uneven access to resources and result in poor performance. For database testing, this means we'll have keys randomly accessed in a heavy-focused distribution pattern that allows us to visualize how the database in question handles hot partition scenarios.

To create the Zipfian distribution for our tests, we followed Zipf's Law to model the frequency of occurrences of items. Assuming that some items are indexed from 0 to N, it determines the rate of occurrence of each item where the smaller the index is, the more occurrences there are for it and the ratio between two consecutive items is exponential. To be able to emphasize and visualize the Zipfian distribution, we modified the class used by YCSB itself. We added the following code to YCSB: https://github.com/eliransin/YCSB/commit/359a49db79d65b5b8f103657bcec4ed91994b310

After building it, we simply ran: `‘java -cp <YCSB artifacts dir>/lib/core-0.18.0-SNAPSHOT.jar site.ycsb.generator.ScrambledZipfianGenerator 1000000000 100000’`

This generates 100K items out of 1B using the Zipfian distribution class used by YCSB. It is worth noting that such changes do not alter YCSB's behavior in any way. The code in question simply allows us to visualize what access distribution would look like for the items subject to our testing.



For 100,000 values (generated out of 1B possible values), here are the quantile stats:

Note that many of the same elements are generated repeatedly. 20% (20,000) of the generated items are actually the same 81 items. Looking at the occurrence graph below (ordered from the most common item to the least common item), it's clear how a small percentage of the items gets "hit" much more than the others:

Quantile	Max Item
0	0
0.1	6
0.2	81
0.3	932.7
0.4	6201.6
0.5	16201.5
0.6	26201.4
0.7	36201.3
0.8	46201.2
0.9	56201.1
1	66201

Little's Law and its Relevance to Uniform Distribution

Here is a small technology nugget worth sharing. We encountered some problems achieving the desired throughput in DynamoDB with Uniform distribution. When exploring the issue, we tried to apply Little's Law to see if it shed some light on how to fix it.

We tried to reach 200K operations/sec with 100% reads. We provisioned 200K RCUs and used the standard 6 loader machines. This worked nicely for ScyllaDB but couldn't meet the desired 200K ops/s with DynamoDB (we only reached 130K ops/s).

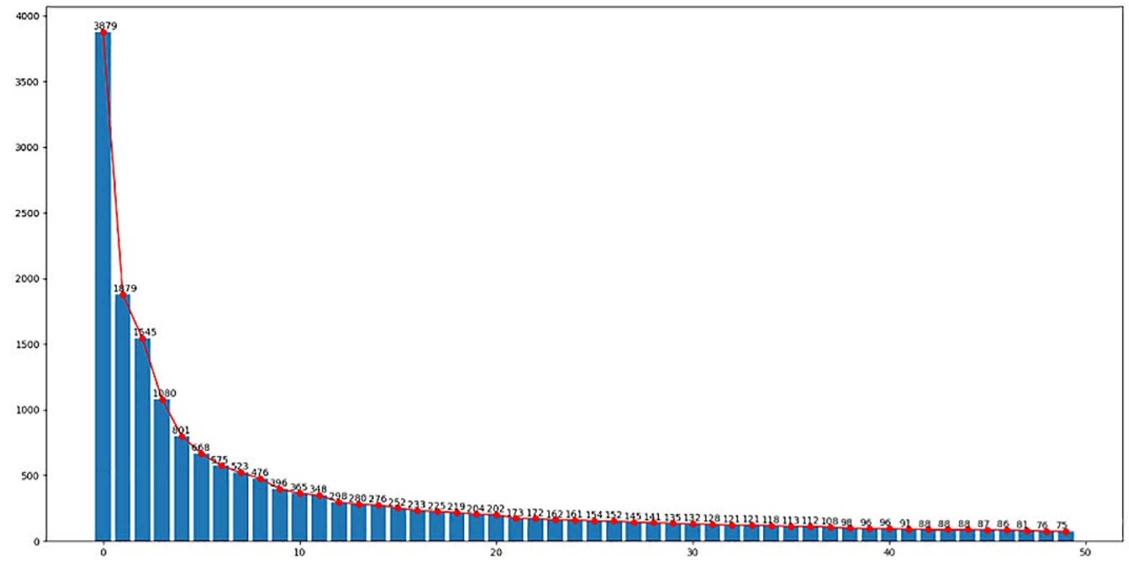
After careful analysis, we realized we hit Little's Law:

$$L = \lambda W$$

L - Average number of items within the system

λ - Average arrival rate of items into and out of the system

W - Average time an items spends in the system



In this database client/server case, this translates to:

Concurrency = throughput * mean latency

The concurrency we used was 1080 - 6 loader nodes, each running 3 YCSB processes with 60 threads each. The observed mean latency: ~6.2ms. Applying Little's Law:

Throughput * latency = 200,000 * 0.0062 = 1240

So, physics requires a concurrency of 1240 parallel threads at this latency while the client machines only had 1080 threads. It worked for ScyllaDB since our latency is dramatically lower.

Once we added more loaders for DynamoDB, the problem was resolved and the maximum throughput was achieved. It was not a hard workaround, but it is a good reminder that low latency matters.

ADDITIONAL CONFIGURATION DETAILS

Some background about using YCSB

In order to reproduce the results you'll first need to build YCSB (it is recommended to use our modified version[5] since it will not crash on heavy workload scenarios). However, for very short tests, a clean version of YCSB can be used.

- YCSB's official repo: https://github.com/brianfrankcooper/YCSB
- Our modified repo: https://github.com/eliransin/YCSB/tree/037bb5d7a3fe0b516cc6d01a0cb0fa287a70813c

Once built, upload the resulting binaries to all loader machines.

Prerequisites

You'll need access to the database you would like to evaluate: ScyllaDB, DynamoDB or both.

As a YCSB prerequisite, create the target tables for each database.

For ScyllaDB, open a cqlsh session and create the keyspace and table as follows:

```
CREATE KEYSPACE IF NOT EXISTS
ycsb WITH replication = {'class':
'NetworkTopologyStrategy', 'replication_
factor' : '3'};
```

```
CREATE TABLE IF NOT EXISTS ycsb.usertable
(y_id varchar primary key, field0 varchar,
field1 varchar, field2 varchar, field3
varchar, field4 varchar, field5 varchar,
field6 varchar, field7 varchar, field8
varchar, field9 varchar );
```

For DynamoDB, use the AWS CLI:

```
aws --region <chosen_region>
dynamodb create-table  --endpoint-
url <dynamodb_endpoint> --table-name
usertable --attribute-definitions
AttributeName=p,AttributeType=S --key-
schema AttributeName=p,KeyType=HASH
```

After the initial table creation step, the next step is to make use of the 'ycsb load' command to load the data. More information can be found in the YCSB documentation.

Properties for YCSB

YCSB takes a layered approach to configuration. The user specifies a list of files (-P) and a list of individual properties (-p). The input is then combined to a single configuration, where each property value is determined by the last value found. The command line options are traversed in order of appearance.

Unspecified properties are set to YCSB's defaults, which may be found within the YCSB GitHub repository.

List of property files used for the runs:

All property files referenced here are available at:

https://github.com/eliransin/alternator-benchmark/tree/main/workloads

The following table describes the relative file hierarchy of the workloads directory.

(5) See the following section for more details about the modifications.

File	Description
defaults/default	Some defaults to use in all runs, regardless of the database. The "threadcount" parameter may need to be overridden (as mentioned under the Little's Law and its Relevance to Uniform Distribution section). The two parameters at the top (core_workload*) are useful for the data loading stage.
defaults/measurements	Sets the measurement method for the runs. There are multiple options. However, we chose to save all of the raw data so we can process it later and be able to extract the most out of it.
defaults/scylla	Parameters for ScyllaDB binding.
defaults/dynamodb	Parameters for DynamoDB binding.
workloada	Base definitions for the YCSB workload referenced through the tests. Overrides were provided via command-line parameters. Its settings are the same as the *"workloada"* definitions shipped along with YCSB.
dynamodb_blogpost/1TB_data	Overlays some of the properties in workloada to get to the actual basic workload we used. For example, number of records were set to 1B

Running the benchmark

After completing all previous steps, remember to configure the YCSB credentials within each loader if you are testing against DynamoDB, or if you enabled authentication on your ScyllaDB cluster. Refer to the YCSB documentation for details on how to accomplish that. The next step is to start the workloads against each database you intend to test.

Building the basic command:

The following command-line arguments constitute the common set of parameters used across all runs:

```
bin/ycsb run <scylla/dynamodb> -P
workloads/defaults/default -P workloads/
defaults/measurements -P <one of:
workloads/defaults/scylla ,/workloads/
defaults/dynamodb> -P workloads/workloada
-P workloads/dynamodb_blogpost/1TB_data
-p table=usertable -p measurement.raw.
output_file=<unique file name for results>
-s -p target=<throughput target> -p
maxexecutiontime=<num seconds to run>
```

NOTE: Values enclosed between the less than and greater than signs (<>) need to be adjusted depending on the test and database you want to run against.

Read and write ratios

To control the ratio of reads vs writes, simply append the following at the end of the basic command:

```
-p updateproportion=<fraction writes> -p
readproportion=<fraction read>
```

Fractions are real numbers adding up to one. For example, for a 10% write and 90% read ratio, append the following:

```
-p updateproportion=0.1 -p
readproportion=0.9
```

Controlling the data distribution

To control the data distribution, append the following to the end of the command-line arguments:

1. For uniform: `-p requestdistribution=uniform`
2. For zipfian: `-p requestdistribution=zipfian`
3. For hotspot: `-p requestdistribution=hotspot -p hotspotdatafraction=<percentage of data in the hot set> -p hotspotopnfraction=<probability for the hot set>`

In this benchmark we used 1.3% of the data as a hot set with a probability of 95%:

```
-p requestdistribution=hotspot
-p hotspotdatafraction=0.013 -p
hotspotopnfraction=0.95
```

Database specific parameters

To specify multiple contact points for ScyllaDB, specify a comma-separated list of IP addresses along with the scylla.hosts parameter:

```
 -p scylla.hosts=<comma separated list of
ips>
```

For example, a 3-node cluster would specify:

```
 -p scylla.hos
ts=10.12.9.201,10.12.7.132,10.12.0.74
```

For DynamoDB, specify the region your table is located, the credentials file used for authentication, and – if applicable – the DynamoDB HTTP endpoint YCSB should connect to:

```
-p dynamodb.region=<region> -p dynamodb.
awsCredentialsFile=<credentials file path>
-p dynamodb.endpoint=<the DynamoDB api
endpoint>
```

Hints and tips with regards to running the benchmarks

1. In order to achieve high throughput, run multiple loaders concurrently. In our benchmarks, we used between 6 to 8 loaders, each running 3 YCSB instances. When doing so, remember to divide the expected throughput by the number of YCSB processes, where the final throughput may be found by the total_YCSB_processes* target formula, where target is the value specified by the "-p target=N" command-line argument.

2. You may use the same set of files in order to perform the initial data load against each database. To speed up the ingestion process, you may also run multiple YCSB processes concurrently, ensuring that each one loads a different range of data. Refer to YCSB documentation for more details.

3. Each process will produce raw measurements in the path as provided by the "-p measurement.raw.output_file=<unique file name for results>" arguments. Each resulting CSV file contains the operations run, the measurement start time, and the elapsed time for completing the measurements. Afterward, simply process the resulting CSV files in order to retrieve the resulting latencies for the run in question.

YCSB MODIFICATIONS

As previously mentioned, we modified YCSB to prevent it from crashing under heavy workload scenarios. The commits for this branch are listed at https://github.com/eliransin/YCSB/commits/037bb5d7a3fe0b516cc6d01a0cb0fa287a70813c

The top 5 commits cover the changes implemented by ScyllaDB. A summary of the modifications we carried out is provided below:

- We've bumped the HdrHistograms package version (45a27c9) and introduced the javax.xml.bind dependency in order to support compiling under newer Java (>8) releases (829a930).
- One commit (18534e2) introduced support on YCSB for consuming credentials data from the ~/.aws/credentials file, typically generated by the AWS CLI and other Boto3 tools. This improvement was made in order to support temporary sessions, and avoid hardcoded credentials (which could potentially expose a security breach).
- We introduced commit (359a49d) to produce and print out Zipfian's distributed elements. This allowed us to analyze the Zipfian distribution prior to running the tests. Note that this commit doesn't change the default YCSB behavior, although we believe YCSB Zipfian's generation is only a close approximation of the pure mathematical one.
- The last and most important commit (037bb5d) that we introduced addressed the crash situation described earlier. YCSB held all measurements in memory until the end of the run; this crashed the process due to the elevated number of operations executed in a single cycle. We modified YCSB to dump its measurements at every 10 seconds interval, resulting in bounded memory consumption. The alternative was to add many more loaders, but this became impractical for the purposes of our testing.

ABOUT SCYLLADB

ScyllaDB is the database for data-intensive apps that require high performance and low latency. It enables teams to harness the ever-increasing computing power of modern infrastructures – eliminating barriers to scale as data grows. Unlike any other database, ScyllaDB is built with deep architectural advancements that enable exceptional end-user experiences at radically lower costs. Over 400 game-changing companies like Disney+ Hotstar, Expedia, FireEye, Discord, Crypto.com, Zillow, Starbucks, Comcast, and Samsung use ScyllaDB for their toughest database challenges. ScyllaDB is available as free open source software, a fully-supported enterprise product, and a fully managed service on multiple cloud providers. For more information: ScyllaDB.com

SCYLLADB.COM

ScyllaDB

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

Israel Headquarters
11 Galgalei Haplada
Herzelia, Israel