

[Product](#)[Resources](#)[Community](#) [Cloud](#) [Github](#)

Benchmarking Vector Search Engines

As an Open Source vector search engine, we are often compared to the competitors and asked about our performance vs the other tools. But the answer was never simple, as the world of vector databases lacked a unified open benchmark that would show the differences. So we created one, making some bold assumptions about how it should be done. Here we describe why we think that's the best way.

That is why we perform our benchmarks on exactly the same hardware, which you can rent from any cloud provider. It does not guarantee the best performance, making the whole process affordable and reproducible, so you can easily repeat it yourself. So in our benchmarks, we **focus on the relative numbers**, so it is possible to **compare** the performance of different engines given equal resources.

The list will be updated:

- Upload & Search speed on single node - [Benchmark](#)
- Filtered search benchmark - [Benchmark](#)
- Memory consumption benchmark - TBD
- Cluster mode benchmark - TBD

Some of our experiment design decisions are described at [F.A.Q Section](#).

Suggest your variants of what you want to test in our [Discord channel](#)!

Single node speed benchmark

We benchmarked several engines using various configurations of them on 3 different datasets to check how the results may vary. Those datasets may have different vector dimensionality but also vary in terms of the distance function being used. We also tried to capture the difference we can expect while using some different configuration parameters, for both the engine itself and the search operation separately. It is also quite interesting to see how the number of search threads may impact the performance of the engines, so we added that option as well.

Dataset: deep-image-96-angular ▼ Search threads: 100 ▼ Plot values: ☐ RPS | ☐ Latency | ☐ p95 latency | ☒ Index time





Engine	setup_name	Dataset	Upload Time (s)	Upload +	P95 (s)	RPS	parallel	P99 (s)	Latency (s)	Precision	engine_params
				Index Time (s)							
qdrant	qdrant-m-16-ef-128	deep-image-96-angular	629.14	1062.94	0.14	681.66	100	0.19	0.13	0.98	{"search_params":{"hnsw_ef":128}}
milvus	milvus-m-16-ef-128	deep-image-96-angular	364.46	2116.90	0.26	293.79	100	0.28	0.21	0.99	{"params":{"ef":256}}
elastic	elastic-m-16-ef-128	deep-image-96-angular	5252.68	6069.67	1.25	84.50	100	1.40	1.066	0.99	{"num_candidates":256}
weaviate	weaviate-m-32-ef-128	deep-image-96-angular	9585.44	9585.44	0.50	339.43	100	0.61	0.28	0.98	{"vectorIndexConfig":{"ef":512}}

Download raw data: [here](#)

Disclaimer

Even if we try to be objective, we are not experts in using all the existing vector databases. We develop Qdrant and try to make it stand out from the crowd. Due to that, we could have missed some important tweaks in different engines.

We tried our best, kept scrolling the docs up and down, and experimented with different configurations to get the most out of the tools. However, we believe you can do it better than us, so all **benchmarks are fully open-sourced, and contributions are welcome!**

Tested datasets

Our benchmark, inspired by github.com/erikbern/ann-benchmarks/, used the following datasets to test the performance of the engines on ANN Search tasks:

Datasets	Number of vectors	Vector dimensionality	Distance function
deep-image-96-angular	9,990,000	96	cosine



	Product	Resources	Community	 Cloud
				 Github
glove-100-angular	1,183,514	100		cosine

Hardware

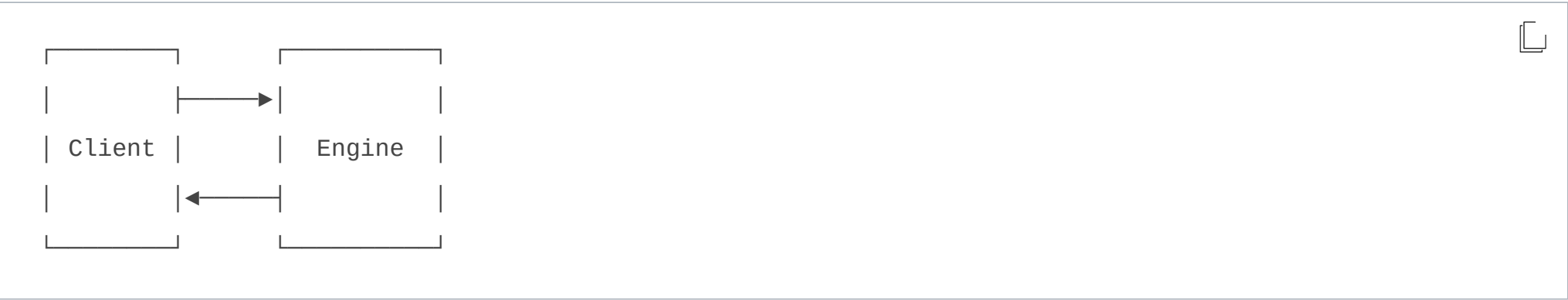
In our experiments, we are not focusing on the absolute values of the metrics but rather on a relative comparison of different engines. What is important is the fact we used the same machine for all the tests. It was just wiped off between launching different engines.

We selected an average machine, which you can easily rent from almost any cloud provider. No extra quota or custom configuration is required.

For this particular experiment, we used 8 CPUs and 32GB of RAM as a Server, with additionally limited memory to 25Gb by means of Docker, to make it exact.

And 8 CPUs + 16Gb RAM for client machine. We were trying to make the bottleneck on client side as wide as possible.

Experiment setup



The Python Client uploads data to the server, waits for all required indexes to be constructed, and then performs searches with multiple threads. We repeat this process with multiple different configurations for each engine, and then select the best one for a given precision.

Why we decided to test with the Python client

There is no consensus in the world of vector databases when it comes to the best technology to implement such a tool. You’re free to choose Go, Java or Rust-based systems. But you’re most likely to generate your embeddings using Python with PyTorch or Tensorflow, as according to stats it is the most commonly used language for Deep Learning. Thus, you’re probably going to use Python to put the created vectors in the database of your choice either way. For that reason, using Go, Java or Rust clients will rarely happen in the typical pipeline. **Python clients are also the most popular clients among all the engines, just by looking at the number of GitHub stars.**

From the user’s perspective, the crucial thing is the latency perceived while using a specific library - in most cases a Python client. Nobody can and even should redefine the whole technology stack, just because of using a specific search tool. That’s why we decided to focus primarily on official Python libraries, provided by t

An interactive chart that allows you to check the results achieved by each engine under selected circumstances. First of all, you can choose the dataset, the number of search threads and the metric you want to check. Then, you can select a precision level that would be satisfactory for you. After doing all this, the table under the chart will get automatically refreshed and will only display the best results of each of the engines, with all its configuration properties. The table is sorted by the value of the selected metric (RPS / Latency / p95 latency / Index time), and the first entry is always the winner of the category 🏆

The graph displays the best configuration / result for a given precision, so it allows us to avoid visual and measurement noise.

Please note that some of the engines might not satisfy the precision criteria, if you select a really high threshold. Some of them also failed on a specific dataset, due to memory issues. That's why the list may sometimes be incomplete and not contain all the engines.

Side notes

- **Redis** took over 8 hours to complete with indexing the **deep-image-96-angular**. That's why we interrupted the tests and didn't include those results.
- **Weaviate** was able to index the **deep-image-96-angular** only with the lightweight configuration under a given limitations (25Gb RAM). That's why there are only few datapoints with low precision for this dataset and Weaviate on the plot.

Conclusions

Some of the engines are clearly doing better than others and here are some interesting findings of us:

- **Qdrant** and **Milvus** are the fastest engines when it comes to indexing time. The time they need to build internal search structures is order of magnitude lower than for the competitors.
- **Qdrant** achives highest RPS and lowest latencies in almost all scenarios, no matter the precision threshold and the metric we choose.
- There is a noticeable difference between engines that try to do a single HNSW index and those with multiple segments. Single-segment leads to higher RPS but lowers the precision and higher indexing time. **Qdrant** allows you to configure the number of segments to achieve your desired goal.
- **Redis** does better than the others while using one thread only. When we just use a single thread, the bottleneck might be the client, not the server, where **Redis**'s custom protocol gives it an advantage. But it is architecturally limited to only a single thread execution, which makes it impossible to scale vertically.
- **Elasticsearch** is typically way slower than all the competitors, no matter the dataset and metric.

Filtered search benchmark



Benchmark Datasets - <https://github.com/qdrant/ann-filtering-benchmark-datasets>

It is similar to the ones used in the [ann-benchmarks project](#) but enriched with payload metadata and pre-generated filtering requests. It includes synthetic and real-world datasets with various filters, from keywords to geo-spatial queries.

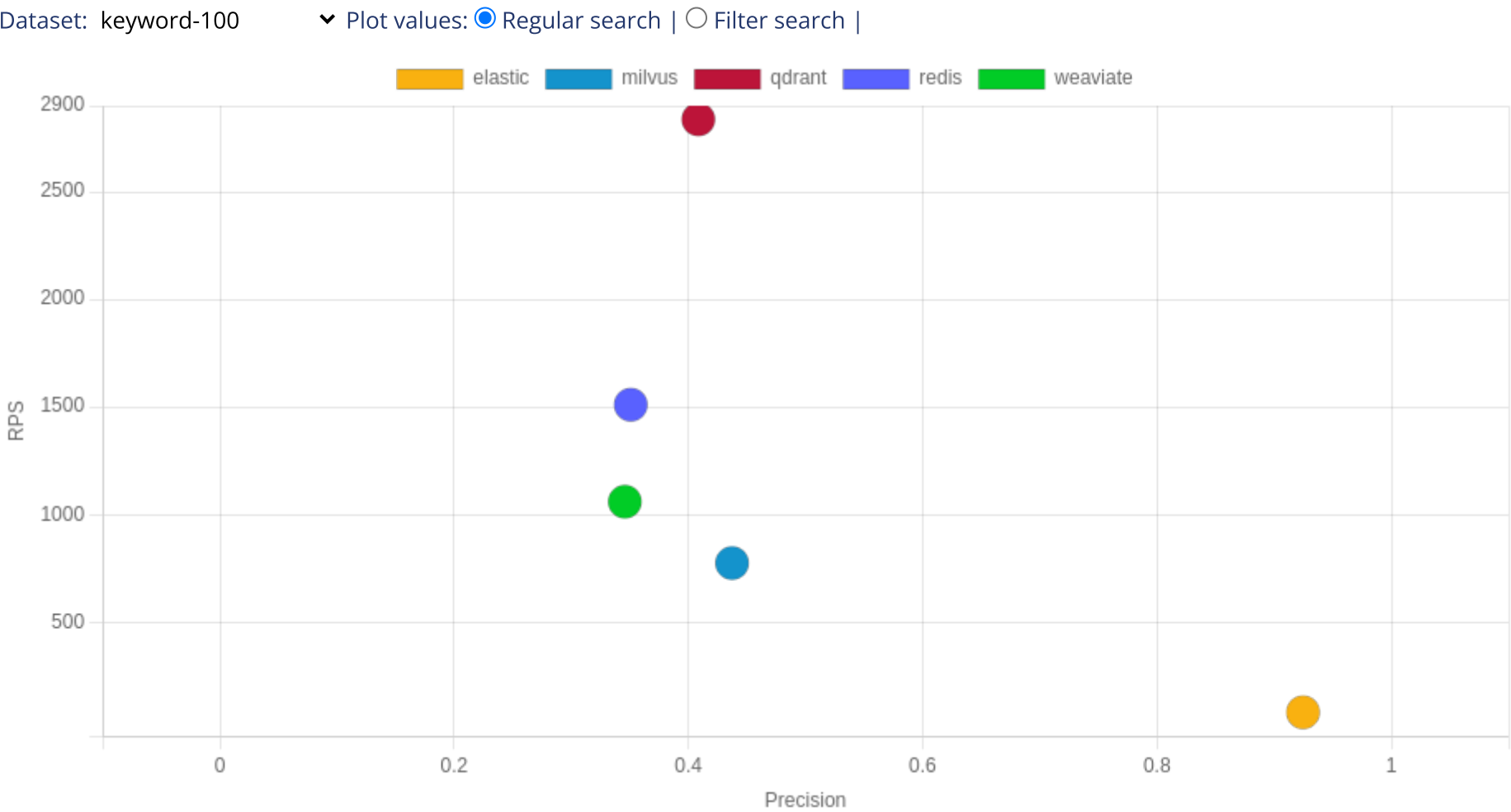
Why filtering is not trivial?

Not many ANN algorithms are compatible with filtering. HNSW is one of the few of them, but search engines approach its integration in different ways:

- Some use **post-filtering**, which applies filters after ANN search. It doesn't scale well as it either loses results or requires many candidates on the first stage.
- Others use **pre-filtering**, which requires a binary mask of the whole dataset to be passed into the ANN algorithm. It is also not scalable, as the mask size grows linearly with the dataset size.

On top of it, there is also a problem with search accuracy. It appears if too many vectors are filtered out, so the HNSW graph becomes disconnected.

Qdrant uses a different approach, not requiring pre- or post-filtering while addressing the accuracy problem. Read more about the Qdrant approach in our [Filterable HNSW](#) article.



Download raw data: [here](#)

Filtered Results

As you can see from the charts, there are three main patterns:



[Product](#)[Resources](#)[Community](#) [Cloud](#) [Github](#)

building a filtering mask for the dataset, as described above.

- **Accuracy collapse** - some engines are losing accuracy dramatically under some filters. It is related to the fact that the HNSW graph becomes disconnected, and the search becomes unreliable.

Qdrant avoids all these problems and also benefits from the speed boost, as it implements an advanced [query planning strategy](#).

i The Filtering Benchmark is all about changes in performance between filter and un-filtered queries. Please refer to the search benchmark for absolute speed comparison.

Benchmarks F.A.Q.

Are we biased?

Of course, we are! Even if we try to be objective, we are not experts in using all the existing vector databases. We develop Qdrant and try to make it stand out from the crowd. Due to that, we could have missed some important tweaks in different engines.

We tried our best, kept scrolling the docs up and down, and experimented with different configurations to get the most out of the tools. However, we believe you can do it better than us, so all **benchmarks are fully open-sourced, and contributions are welcome!**

What do we measure?

There are several factors considered while deciding on which database to use. Of course, some of them support a different subset of functionalities, and those might be a key factor to make the decision. But in general, we all care about the search precision, speed, and resources required to achieve it.

There is one important thing - **the speed of the engines has to be compared only if they achieve the same precision**. Otherwise, they could maximize the speed factors by providing inaccurate results, which everybody would rather avoid. Thus, our benchmark results are compared only at a specific search precision threshold.

We currently have planned measurements in several scenarios, from the most standard - single node deployment to a distributed cluster.



[Product](#)[Resources](#)[Community](#) [Cloud](#) [Github](#)

was just wiped out between launching different engines.

We selected an average machine, which you can easily rent from almost any cloud provider. No extra quota or custom configuration is required.

Why you are not comparing with FAISS or Annoy?

Libraries like FAISS provide a great tool to do experiments with vector search. But they are far away from real usage in production environments. If you are using FAISS in production, in the best case, you never need to update it in real-time. In the worst case, you have to create your custom wrapper around it to support CRUD, high availability, horizontal scalability, concurrent access, and so on.

Some vector search engines even use FAISS under the hood, but the search engine is much more than just an indexing algorithm.

We do, however, use the same benchmark datasets as the famous [ann-benchmarks project](#), so you can align your expectations for any practical reasons.

Why are you using Python client?

There is no consensus in the world of vector databases when it comes to the best technology to implement such a tool. You're free to choose Go, Java or Rust-based systems. But you're most likely to generate your embeddings using Python with PyTorch or Tensorflow, as according to stats it is the most commonly used language for Deep Learning. Thus, you're probably going to use Python to put the created vectors in the database of your choice either way. For that reason, using Go, Java or Rust clients will rarely happen in the typical pipeline - although, we encourage you to adopt Rust stack if you care about the performance of your application. Python clients are also the most popular clients among all the engines, just by looking at the number of GitHub stars.

What about closed-source SaaS platforms?

There are some vector databases available as SaaS only so that we couldn't test them on the same machine as the rest of the systems. That makes the comparison unfair. That's why we purely focused on testing the Open Source vector databases, so everybody may reproduce the benchmarks easily.

This is not the final list, and we'll continue benchmarking as many different engines as possible. Some applications do not support the full list of features needed for any particular benchmark, in which case we will exclude them from the list.

How to reproduce the benchmark?

The source code is available on [Github](#) and has a README file describing the process of running the benchmark for a specific engine.



the [benchmark repository](#).

Get Updates from Qdrant

We will update you on new features and news regarding Qdrant and Vector Similarity Search






Enter Your Email

Subscribe

Product

- ✔ [Use cases](#)
- ✔ [Solutions](#)
- ✔ [Benchmarks](#)
- ✔ [Demos](#)
- ✔ [Pricing](#)

Community

-  [Github](#)
-  [Discord](#)
-  [Twitter](#)
-  [Newsletter](#)
-  [Contact us](#)

Company

- ✔ [Jobs](#)
- ✔ [Privacy Policy](#)
- ✔ [Terms](#)
- ✔ [Impressum](#)
- ✔ [Credits](#)

Latest Publications

- [Scalar Quantization Is A Newly Introduced Mechanism Of Reducing The Memory Footprint And Increasing Performance](#)
- [On Unstructured Data, Vector Databases, New AI Age, And Our Seed Round.](#)
- [ChatGPT Factuality Might Be Improved With Semantic Search. Here Is How.](#)

