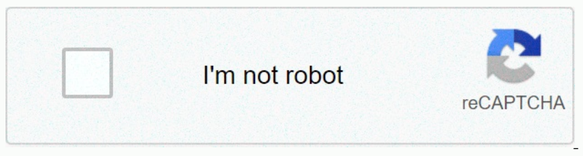

Continue

Arangodb vs mongodb performance

ArangoDB is a traditional multi-generational database competing with many single-model storage technologies. When we started the ArangoDB project, one of the key design goals was and at least was able to compete with the leading single model vendors on their home turf. Then, does the original multi-replica database make sense? To prove that we are achieving our goals and being competitive, we run and release benchmark set updates from time to time. This time we include MongoDB, PostgreSQL (Table & JSONB), OrientDB and Neo4j in this post, we will cover the following topics: Introduction to Benchmarks and acknowledgement this article is part of arangoDB's open source performance benchmark series since the previous post has a new version of competitive software to compare. There are also some major changes to the ArangoDB software, for example, in the latest version of ArangoDB, with facebook's RocksDB storage tool included, so we wait until the integration is complete before testing the new benchmark. In addition to these factors, the machine is faster, so the new benchmark makes sense. Before I enter a specific benchmark and results, I would like to send a special thank you to Hans-Peter Grahsl for his great help with mongoDB inquiries. Thank you Hans-Peter for your help! Thank you very much, as well as Max De Marzi and JakeWins, both Neo4j teams for their participation and benchmark improvement in 2018. Excellent teamwork, crew! After we published the previous benchmark, we received a lot of feedback from the community - thank you very much for their help, comments and ideas. We include that comment on this benchmark. For example, this time we have included the JSONB model for PostgreSQL test settings for comparison, we use three leading single model database systems: Neo4j. and PostgreSQL for relational databases. In addition, we compared ArangoDB with several orientDB databases, of course, our own benchmark operations can be questionable. Therefore, we publish all the scripts necessary for everyone to replicate this benchmark with minimum effort. They can be found here in Github: Setting up and importing Run-Benchmark-Script scripts, we use simple client/server settings and instances recommended by AWS for relational and non-relational databases. We use the following instance: server: i3.4xlarge. On AWS with 16 virtual cores, 122 GB RAM client: c3.xlarge on AWS with four virtual CPUs, 7.5 GB RAM and 40 GB SSD, the setting costs ~35 US dollars per day to make things all simple and easy to repeat. Get tested as it is when downloaded, so you'll need to use the same script and instance if you want to compare your numbers with ours. We use the latest version of GA (as of January 26, 2018) of all database systems and do not include rc versions, below is a list of the versions we use for each product: Neo4j 3.3.1 MongoDB 3.6.1 PostgreSQL 10.1 (Table & For this benchmark, we use NodeJS 8.9.4, the operating system for the server is Ubuntu 16.04, including the OS patch 4.4.0-1049-aws, which includes meltdown patches and Spectre V1. Description of the test, we use this set of internal benchmarks for our own assessment, our own quality control, to see how changes in ArangoDB affect performance. Our benchmark is completely open source. You can download all the scripts needed to benchmark yourself in our archives. The goal of the benchmark is to measure the performance of each database system when the query cache is not used to ensure that we disable the query cache for each proposed software. For our tests, we run the workload twenty times the average result. Each test begins with a warm-up step that allows the database system to load data in memory. For testing, we used the Pokec dataset supplied by Stanford University SNAP with 1.6 million people (vertices) connected through 30.6 million edges. It also includes graph queries to a benchmark graph database (such as the shortest route). The following test cases have been included as far as the database system can execute a query: Read once: This is a single read of a profile (such as 100,000 different documents). One-time sync: These are the same as one-time writing, but we wait for fsync in every request. Second neighbour: We searched for different neighbors directly, including a neighbor who returned an ID for 1,000 vertices, a second neighbor with information: we found different neighbors directly, including neighbors, and returned their profiles for 100 vertices, the shortest route: the shortest route, 1,000 routes found in a highly connected social graph. This answers the question of how many other people are very close to each other on social networks. Memory: This is the average of the maximum primary memory usage during the run test. Processable measurements on arangoDB testing machines - with RocksDB as storage mechanisms - determine the basis (100%) for comparison. A lower percentage indicates a higher throughput. So higher. Indicates a lower throughput. The overall results below chart show the overall results of our performance benchmarks. In the subsection after this graph, we will provide more information about each test. Learn more about ArangoDB with our white papers on what multiple versions of the database are and why they work. As you can see, many traditional versions can compete with a single-model database system. We are very pleased that our new RocksDB-based storage engine works well with the competition. I think the whole team will be proud of this combination. In basic questionnaires such as single reading, single writing, as well as one-time writing sync, we get positive results and perform better. The shortest path queries are not tested for MongoDB or PostgreSQL because those queries must be fully applied on the client side for those database systems. Note that in previous benchmarks, MongoDB shows better results in a single read/write test. The table below shows the results of the latest settings (Database+Driver on Benchmark Day). For all databases To appreciate and understand them, we need to look deeper into individual results and focus on more complex queries such as inclusion and graph functions. Age distribution in social networks - The integration test in this test, we collected more than a single collection (e.g. 1,632,803 documents), we calculated statistics on age distribution for everyone in the network by simply counting how often each age occurred. We do not use secondary indexes for this attribute on any database, so they all need to complete a full collection scan and do a counting statistic. - Is this a general ad hoc query new to multiple models and graphs? Check out our free ArangoDB chart course. The calculation of integration is effective in ArangoDB, with an average of 1.07 seconds and a basic assignment. As expected, PostgreSQL as a representative of the relational world works best with just 0.3 seconds, but only when the data is stored as a table for the same task, but with the data stored as a JSONB document, PostgreSQL requires more time compared to MongoDB, and more than twice the time compared to ArangoDB, since our previous benchmark, OrientDB does not seem to be much better and still slower by more than 20x. Additional friend networks —Finding neighbors who have tested profile information may sound like pure graph queries. But when we search within known depths, other databases can do this job to find neighbors. We tested two different doubts. Firstly, a simple search of neighbors. And two different neighbors with full detailed information OrientDB and MongoDB did a good job in this test, ArangoDB demonstrated good performance compared to neighbors in search of neighbors. A more challenging task for the database is to retrieve the profiles of those neighbors, too. ArangoDB also works effectively on this mission, but PostgreSQL is still better than 23 points (see below). The reason for ArangoDB's poor performance is the best margin index, which allows for quick search for connected edges and vertices for one node, which is assumed to be faster than a typical index search. Shortcuts between two entities — Testing the shortest path, the shortest path algorithm, is the expertise of the graph database. The algorithm finds the shortest distance between the start and end vertex. In this way, you can define the results of such queries to use, for example, on LinkedIn when displaying reciprocal connections. on someone's profile page. The task for this test is to find the shortest 1,000 routes on highly connected social networks to answer the question of how two people are so close to each other in the network. Since combining RocksDB in ArangoDB, the shortest route query has become rapid. - As fast as 416ms to find the shortest route, 1,000 ArangoDB routes are twice as fast as Neo4j and more than a hundred times faster than OrientDB. The RocksDB engine compared to ArangoDB's MMfiles engine is much better because it also has improved graphing capabilities. Using memory in the previous benchmark, primary memory use is a challenge for ArangoDB - it remains to some degree. In this benchmark, we measured memory up to 3.7 times the primary memory consumption compared to postgreSQL's best measured (tabular) results. Neo4j seems to be improving in terms of performance by increasing memory footprint. Compared to the previous benchmark, they went from the second best to the last place. Without any configuration, RocksDB can use up to two-thirds of its existing memory and do so until then, RocksDB begins to throw unverified data out of the primary memory. It is also the reason for the use of high memory ArangoDBs with RocksDB Limit. The main memory for ArangoDB with RocksDB, the great thing about RocksDB is that it is highly configurable. You can set the upper limit of allowed memory usage. We want to know what will happen if we set the limit. The memory is 10 GB and runs the complete benchmark again. The memory on RocksDB remains fast in many cases, arangoDB tests lose single writing and single reading. But achieve an acceptable overall performance. Memory usage is not yet the best on ArangoDB, however, with the RocksDB storage mechanism, you have many options so you can adapt to your use case. Also, we suspect there are additional tweaks you can make to get RocksDB's improved performance is still new for ArangoDB: we haven't touched everything available yet. In conclusion, if you are still not convinced, take a look at the Github archive, do your own tests and please share your results if you do. Note that when testing benchmarks, different hardware can produce different results. Also, please note that your performance requirements may vary and your requirements may vary. For this reason, you should use our storage space as a boiler and expand it with your own testing. In this benchmark, we can again show that ArangoDB can compete with the leading single-model database system on their home turf, and we have shown again that we can compete with a multi-format database, another OrientDB concludes that the excellent performance and superior flexibility of many indigenous models is a major advantage of ArangoDB. This nosql we use the same data and the same hardware to test each database system. If you want to better monitor or understand our results in this appendix, we will provide details about the device information and software we use. We also provide more details about the tests we conducted, as well as explain some adjustments made to accommodate the nuances of some database systems. We used a snapshot of data supplied by Stanford University SNAP, consisting of profile data from 1,632,803 people. However, each JSON document is very diverse because many branches are empty for many people. The uncompressed JSON data for vertices requires about 600 MB, and uncompressed JSON data for the desired edge is about 1.832 GB, the diameter of the graph (e.g. the shortest route) is 11, but the graph is highly connected, it is common for social networks. This makes the shortest route problem especially difficult. All hardware standards are made on i3.4xlarge virtual machines (servers) on AWS with 16 virtual cores, 122 GB RAM and 1900 GB NVMe-SSD for customers. For this, we need a language to carry out the test, so we decided to follow the following criteria: each database in comparison must have the appropriate driver. It is not one of the indigenous languages that our competitors have performed. This may provide an unfair advantage for some people. This eliminated C++ and Java languages must be reasonably popular and relevant in the market. The language should be available. All this main main platform is left JavaScript, PHP, Python, Go and Ruby. We decided to use JavaScript with .js 8.9.4. It is popular and is well known as fast, especially with network workloads. For each database, we use the latest JavaScript drivers recommended by the relevant database vendors. We use the following community versions and driver versions: ArangoDB V3.3.3. For x86_64 (driver arangojs@5.8.0) MongoDB V3.6.1 for x86_64 using wiredtiger storage mechanism (driver mongodb@3.0.1) Neo4j V3.3.1 runs on openj 1.8.0._151 (driver neo4j@1.5.3) OrientDB 2.2.29 (driver orientjs@2.2.7) PostgreSQL 10.1.1 (driver pg-promise@7.4.1) All databases are installed on the same machine. We do our best to customize the configuration parameters. For example, we close a large, transparent page and configure up to 60,000 open file tellers for each process. In addition, we have upgraded the community and vendors to configuration parameters from Michael Hunger of Neo4j and Luca Garulli of OrientDB to improve each setting. In testing, we make sure that for each trial the database has the opportunity to load all relevant data into ram. So we increase the cache size associated with and use the full collection scan as a warm-up step. We do not want to compare the query cache or similarly. - The database may require a warm-up process, but you can't compare the database based on cache size and performance. Whether the cache is useful depends on the individual usage case. For testing a single document, we use individual requests for each document, but use keep-alive and allow multiple concurrent connections. We do this because we want to test workloads rather than latency. We use a TCP/IP connection pool with up to 25 connections whenever the driver allows. All drivers seem to support this connection pool. Read one document (100,000 different documents) In this test, we stored 100,000 identifiers of people in .js nodes. The client tries to retrieve the corresponding profile from the database, each of which in a separate query. In .js everything happens in one thread, but asynchronously. To load the database connection completely, we first send all queries to the driver and then wait for all callbacks using the event loop .js. We measure the wallclock time from before we start submitting questionnaires until the final answer arrives. Obviously, this measures the processing volume of the combination of drivers and databases and does not lag. So we gave the complete wallclock time for all requests, write a single document (100,000 different documents) for this test, we carried out similarly: we loaded 100,000 different documents into the .js node. The client then measures the time the wallclock is needed to send all documents to the database using each query. We're again first. Request all to the driver, and then wait for all callbacks using the event loop .js. Above, here is a measurement of the amount that can be processed. Single document write sync (100,000 different documents) This is the same as previous tests, but we wait until the writing is synced to the disk, which is the default behavior of Neo4j, to be fair, we recommend this additional test in comparison. Integration through a single collection (1,632,803 documents) In this test, we included 1,632,803 ad hoc profile documents and counted how often each value of the AGE attribute occurred. We do not use secondary indexes for this attribute on any database, so they all need to perform a full collection scan and do a counting statistic. We only measured one request because it was enough to get the correct measurement. The amount of data scanned should be greater than any CPU cache that can be stored. Finding neighbors and neighbors (different for 1,000 vertices), this is the first test involving a case of network use. For each of the 1,000 vertices, we found all the neighbors and all neighbors of all neighbors. This must find the friends and friends of the party's friends and return a different set of friend IDs. This is a common graph matching issue, considering one or two long paths. For non-MongoDB graph databases, we use an aggregate framework to calculate results. In PostgreSQL, we use relational tables with the ID from and ID to each table supported by the index. In the Pokec dataset, we found 18,972 neighbors and 852,824 neighbors for vertices that asked our 1,000 people. To find neighbors and neighbours with profile information (different for 100 vertices), we received feedback from previous benchmarks that for real use cases, we need to return more than id, so we added a neighbour's test with a user profile that mentioned this concern and returned a complete profile. In our test case, we retrieved 84,972 profiles from the first 100 vertices we asked. A complete set of 853,000 profiles (1,000 vertices) would have been too much for nodejs to find the shortest route of 1,000 routes (in a highly connected social graph). This is a pure graph test with a search term that is specific to the graph database. We asked the database in 1000 different requests to find the shortest path between the two vertices defined in our social graph. The shortest path is not good in the more traditional database system, since the answer involves a number of previously unknown steps in the graph, which often lead to a number of previously unknown joins. The differences section above describe the tests we conducted with each database system. However, each one has some necessary differences. One cannot always, in fairness, leave all factors fixed, ArangoDB ArangoDB allows you to specify the value of the primary key attribute _key as long as there is no violation of the unique constraint. It automatically indexes the main hash in that attribute, including the margin index on the _from and _to attributes in friendship relationships (such as edge collections). No other MongoDB index is used since MongoDB treats the same edge as the documents in another collection, we helped it a little for querying the graph by creating two more indexes on _from and _to. We tested $graphlookup, but the performance was so slow that we decided not to use it and wrote the query in an old manner, as suggested by Hans-Peter Grahsl. Please note that because the statistics for MongoDB are significantly worse compared to what we measured in 2015, we set the test schedule for MongoDB with the same NodeJS version that we used in the 2015 benchmark, the results for single reading and single write improved slightly with the old NodeJS version, but did not affect the overall ranking. Neo4j In Neo4j profile document attribute values are stored as properties of vertices for fair comparison, we have created an index _key the Neo4j feature claims to use index-free pajamas. Therefore, we do not add another index on the OrientDB margin for OrientDB, we can not use version 2.2.31, which is the latest version due to an error in version 2.2.30 in the shortest_path algorithm, hindering us to complete the benchmark. We reported the error in Github and the OrientDB team immediately corrected it, but the next maintenance release was published after January 26. In the second method for comparing, we use classic relational data modeling that has all profile attributes as columns in a table. The main memory is only 128MB, we use the PostgreSQL customization configurator to provide fair conditions for everyone. All resources and code contributions used in these tests can be downloaded from our Github repository. We welcome all contributions and invite you to test other databases and other workloads. We hope you will share your results and experiences. Thanks again for all the participation the benchmark project has been so far! Excellent ☺ ☺