

Benchmarking Elasticsearch and MS SQL on NYC Taxis



Description	Studying a few filtering and aggregation conditions
Period	Spring 2017
Languages & Libs	Clojure
Tags	GitHub, Databases, Elasticsearch, SQL
GitHub	nikonyrh/nyc-taxi-data

The NYC Taxi dataset has been used on quite many benchmarks (for example by [Mark Litwintschik](#)), perhaps because it has a quite rich set of columns but their meaning is mostly trivial to understand. I developed a Clojure project which generates Elasticsearch and SQL queries with three different templates for filters and four different templates of aggregations. This should give a decent indication of these databases performance under a typical workload, although this test did not run queries concurrently and it does not mix different query types when the benchmark is running. However benchmarks are always tricky to design and execute properly so I'm sure there is room for improvements. In this project the tested database engines were Elasticsearch 5.2.2 (with Oracle JVM 1.8.0_121) and MS SQL Server 2014.

The first filter combines a random [range query](#) on a 14-days long interval between 2013-01-01 and 2016-12-01 with a randomly positioned 2 x 2 km [bounding box](#) on the pickup position. Naturally this results in a random number of matching trips, but this is taken into account by plotting query times against the number of matching rows. 25th, 50th and 75th percentiles on the number of matching trips for this filter were about 50, 7k and 100k. This high dynamic range asks for log-log plots for visualization.

The second type of filter consists of three parts. The first one is a random 60-day long interval between 2013-01-01 and 2016-10-01. Second criterion is the amount of paid tip, having the starting value ranging from 0 to 15 dollars and the interval length was 2 dollars. The third criterion was the day of week, indicated by an integer between 1 and 7. The 25th, 50th and 75th percentiles were 9k, 38k and 232k.

The last type of filter combines a range query on the travel duration in hours starting from 0 - 3 hours and having the length of 15 minutes with a [terms query](#) on the exact number of passengers (random integer from 1 to 6). The 25th, 50th and 75th percentiles on these criteria are 1k, 21k and 738k.

An other important part of each query is that which statistics should be calculated from matching rows. On this set of benchmarks I have implemented four different aggregations. This first one is very trivial, it simply counts the number of matching rows. This sets a good baseline for performance measurements, and one would expect that calculating more complex results wouldn't be any faster than this one.

The second aggregation groups results by company (either Green or Yellow) and pickup date (1 day time-resolution), and for each bucket it calculates [stats](#) (min/max/sum) of the ad-hoc calculated USD / km metric (total paid divided by the kilometers travelled). The third aggregation groups trips by the time of day it started (2 hours time-resolution) and for each bucket calculates stats on the total amount of paid / trip. The fourth aggregation groups results by the pickup date (1 day time-resolution) and for each bucket calculates various [percentiles](#) on trips average velocity (km/h).

The dataset consists of 874.3 million rows of data and has 27 columns. On Elasticsearch the start and dropoff locations were stored as [geo-point](#) type, but on SQL Server they were stored as two separate float fields. It would have its own [Point](#) geospatial column type but its usage seemed quite verbose so I decided to ignore it for now. Data on Elasticsearch was stored into 1 index / company & year and 10 shards, pushing the total number of shards to 120 and keeping the average shard size at about 2.4 GB. Data at SQL server was kept at a single table with [clustered columnstore index](#), and data partitioned into separate files by year (see details at my [notes](#)). Total size on disk was 40.5 GB. Elasticsearch shards were [force-merged](#) down to 8 segments, and SQL column store index was [reorganized](#) to reduce fragmentation.

Elasticsearch was running on an Intel 6700K CPU, 64 GB of RAM and data was two SSDs in RAID-0 mode. 32 GB and 16 GB benchmarks were executed by having a Virtualbox instance running with 32 or 48 GB of RAM allocated to it, thus preventing it being used by Elasticsearch or OS caches. It would have been better to remove the RAM from the motherboard but I did not want to go through the extra trouble.

Unfortunately I had to use a different machine to run MS SQL Server. However it was quite comparable, having Intel 4770, 32 GB of RAM and a SSD for the data. Throughout benchmarks the SSD IO was the bottleneck, maxing out at 450 - 500 MB/s while CPU usage was 20 - 30%. One can guesstimate that with RAID-0 SSDs you could drop query times by 50% on some cases. Also it would be beneficial to have tempdb on a separate SSD from the data.

Results from different filter-aggregation combinations are shown in figures 1 - 3. Filters are chosen at random at each run of the benchmark, which means that number of matches varies a lot. Thus I chose a bit unorthodox method of visualizing the results: x-axis is percentiles (1 to 99) on the number of matching rows and y-axis is percentiles on query runtimes. I did not check the [rank correlation](#) between these values but I expect it to be reasonably high. On all graphs the green line is the SQL Server with 32 GB of RAM, blue is Elasticsearch with 16 GB, yellow is ES with 32 GB and purple is ES with 64 GB of RAM. Interestingly doubling the memory from 32 GB to 64 GB dropped query times by about 50% with Elasticsearch. SQL Server performed best with simpler aggregations, and performed relatively worst on the first filter type (presumably for not using a spatial index on lat/lon data) and the last aggregation type, in which percentiles were calculated. At worst SQL Server was 10 - 40 times slower than Elasticsearch.

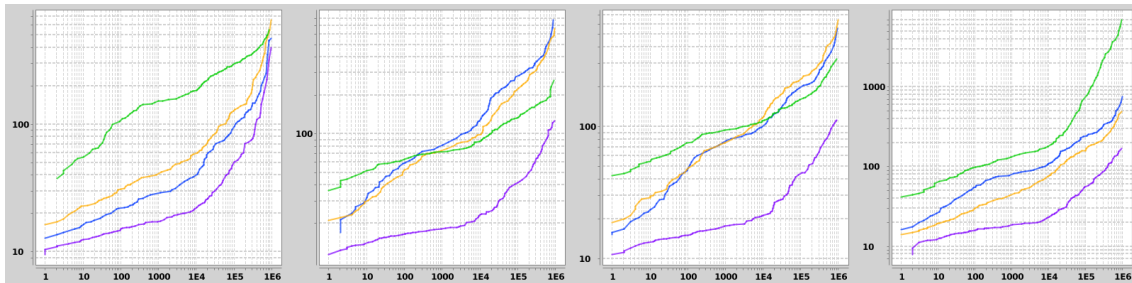


Figure 1: Results from the filter template #1.

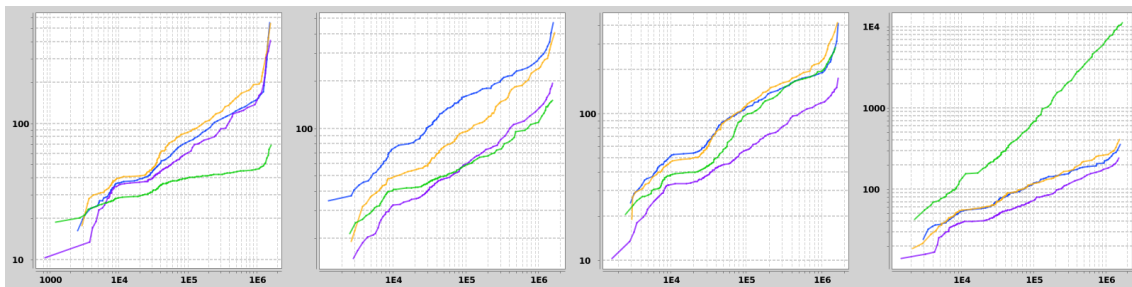


Figure 2: Results from the filter template #2.

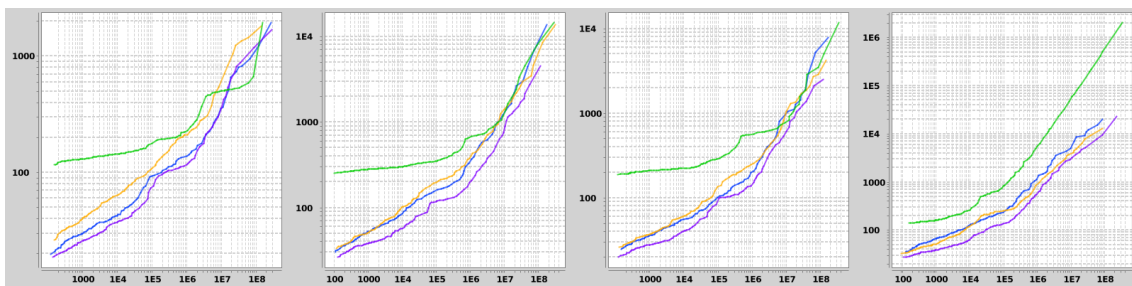


Figure 3: Results from the filter template #3.