Analyzing NYC Taxi dataset with Elasticsearch and Kibana

Description	Parsing CSVs to JSON with Clojure, visualization with Kibana
Period	Spring 2017
Languages & Libs	Clojure
Tags	GitHub, JVM, Elasticsearch, Databases, Business Intelligence, Kibana
GitHub	nikonyrh/nyc-taxi-data

The NYC taxicab dataset has seen lots of love from many data scientists such as Todd W. Scheider and Mark Litwintschik. I decided to give it a go while learning Clojure, as I suspected that it might be a good language for ETL jobs. This article describes how I loaded the dataset, normalized its conventions and columns, converted from CSV to JSON and stored them to Elasticsearch.

My older desktop (let's call it "Machine A") has Intel i5 3570K (quad-core without hyper-threading), 24 GB RAM and 2 x 250 GB Samsung 840 EVOs for data. The newer ("Macine B") has Intel i7 6700K (quad-core with hyper-threading), 64 GB RAM and 2 x 500 GB Samsung 750 EVOs for data. Based on CPU benchmarks it should be about twice as fast as the i5, being roughly equivalent 40 AWS ECUs.

One main goal was to get documents indexed to Elasticsearch as fast as possible. Thus I used two SSDs RAID-0 type configuration by using ZFS on Linux. The first major obstacle was that at initially I got great write performance but after 10 minutes or so it had dropped to only 40 MB/s! Based on current and past issues at its Github repo ZoL doesn't seem production ready yet, but I managed to find a workaround for my case. When I formatted SSDs to EXT4 and created a 450 GB empty files there, I could successfully create a ZFS pool out of them it could sustain the expected IO performance. This was a quite puzzling situation, but I ended up creating a 8 GB file to EXT4-formatted Samsung 850 EVO m2 card (which is the main system disk) and using it as ZIL for my pool. For some reason this kept the IO performance at normal levels and I did not have to run ZFS on top of EXT4.

CSV files are stored in a Gzipped format, from which they are lazily read as Clojure strings by `java.util.zip.GZIPInputStream`. Initial implementation would `slurp` the whole CSV string and pass it to `read-csv`, but this at too much RAM when multiple large files are being processed in parallel. The process of converting CSV rows in to Elasticsearch documents is quite straight-forward. The code has a lookup dictionary to map CSV columns into JSON fields of specific types. In total there are 36 fields. Yellow and Green taxicab datasets had some differences on column naming but at least their meaning was easy to recognize. The code also generates some new fields to each doc, such as the time of day when the trip started (0 - 24 h), how many kilometers were travelled latitude / longitude wise and what was the average speed during the trip. It also stores that day's weather at Central Park to each document, as Elasticsearch does not have ad-hoc joins unlike SQL databases.

Originally I generated document ids in Clojure and used those to skip chunks which had already been inserted to ES, which was handy when I was iterating the code and sorting out malformed data. Later I discarded this as it slows down indexing process quite a lot. Nevertheless doc id generation code was still used it to filter out duplicates before sendin them to ES. But there are less than hundred of them so now I don't even bother removing them. It was causing quite a lot of memory and garbage collection overhead for very little gain, but I'm sure duplicate removal utility will be useful on other projects.

There are many factors which affect indexing performance, and I didn't carefully measure all of their impacts as it is quite tedious to execute and your mileage will vary anyway. Note that many articles on the topic Elasticsearch optimizations are for versions of 1.x and 2.x but aren't relevant on latest ES 5.x versions. CSV parsing is single-threaded, so to utilize multiple CPU cores many files are parsed in parallel. At first I used my own code for this, based on Clojure's pmap but with configurable number of threads. However seemed to work poorly when some files took much longer to process than others, and parallelism diminished until the larger file was finished and then new futures were dereferenced. This was resolved by using Claypoole's magnificent `upfor`, which will execute the jobs in N parallel threads and return the fastest results first. Documents were bulk-uploaded in chunks of 1000.

I had one index / year / company, and each index had only ten shards. I started with only one shard / index but it caused merge throttling in the long run as indexes grow to tens of gigabytes. For example the index `taxicab-yellow-2014` has 164.2 million docs and takes 48.5 GB of storage.

Indexing from A to B was constrained by A's CPU power to transform CSV rows to JSON, indexing them to Elasticsearch over a gigabit LAN at 100 - 150 Mbit/s. Machine B's CPU usage was fluctuating between 30 and 50 percent. In 10 minutes it could index 9.55 million documents, thus averaging at 15931 docs / second. When using only machine B to handle both workloads it would be saturated by CPU, totaling at 11.9 million docs in 10 minutes, or 19869 docs / second. When indexing from machine B to machine A it wasn't clear where the bottleneck was as both CPUs were used at 50 percent. It could be caused by buggy firmware on my 840 EVOs as I haven't updated them yet. Anyway, this setup managed to store only 4.6 million docs in 10 minutes, or averaging at 7746 docs / second. Top performance was obtained by parsing CSVs on both machines and hosting ES on machine B, totaling at 14.3 million docs, or at average of 23909 docs / second. Even at this speed indexing 1 billion documents would take about 11.6 hours.

In the end I used just machine B to index all documents in a single go, which took 910 minutes and stored 942.1 million documents. Then I "force merged" indexes to have at most 8 segments each, this took 44 minutes. Finally merging them down to 1 segment took 302 minutes, although CPU and disk IO usage was very low. It seems to have build-in throttling as well, but I couldn't find detailed documentation. The final size on disk was 309 GB.

Once the data is stored to ES there are endless possibilities for interesting analysis. Kibana makes it easy to get started with understanding the dataset and executing ad-hoc queries, but of course if you want full control over details and optimizations then you'll have to implement lots of stuff yourself. Here I'll just show patterns from a single fact: the date-time when the trip started. This is split into three parts: date, day of week and time of day. One can expect weekends having different taxi demand than rest of the days.

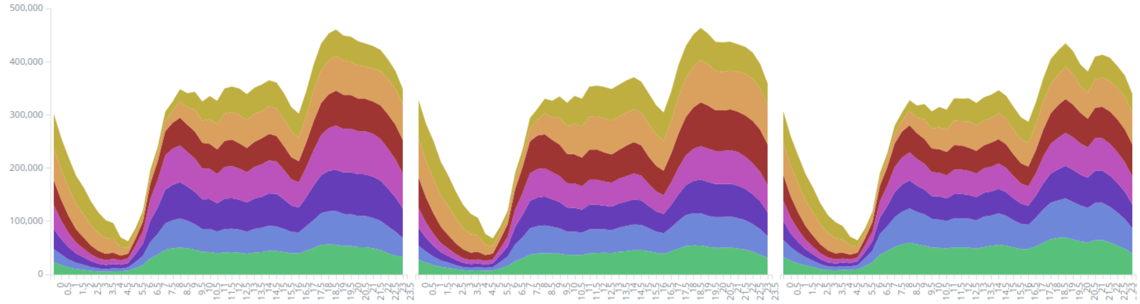

Figure 1: Monthly number of rides on different week days day and time of day.

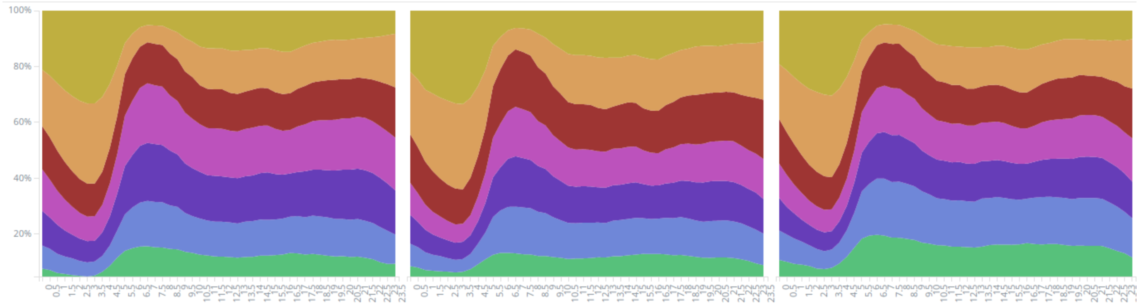

Figure 2: Monthly share of rides on different week days and time of day.

All these statistics were calculated from data from 2015-04-01 to 2015-06-30, which is a total of 43.3 million trips. The first set of figures 1 to 3 uses Kibana's Area chart, which supports many useful features such as splitting areas by sub-aggregation and showing percentages instead of totals.

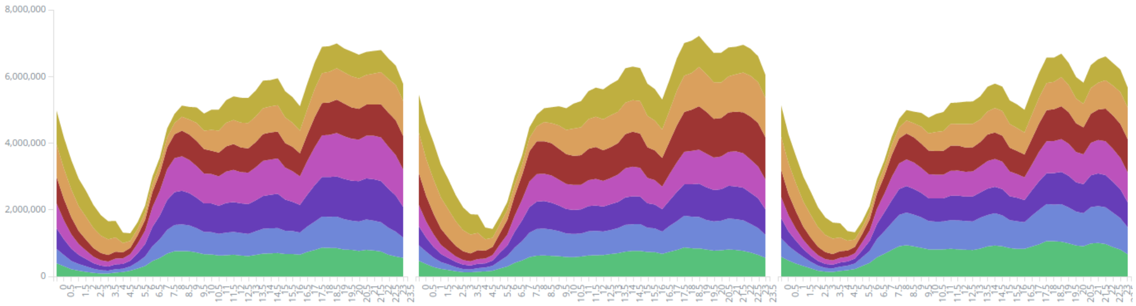

Figure 3: Monthly amount of spent dollars to taxi rides on different times of day and time of day.

An other interesting visualization is the Heatmap chart, which is seen on Figure 4. It lets you choose how documents are "bucketet" along the x and y axis and what is the visualized metric. This makes weekly cycles quite visible especially in 2 - 5 am range but I find it more difficult to extrapolate the data forward when compared to line charts. It doesn't handle outliers nicely, at least if the histogram aggregation is used. Luckily they are easy to filter out in the query phase, if the expected data range is known.

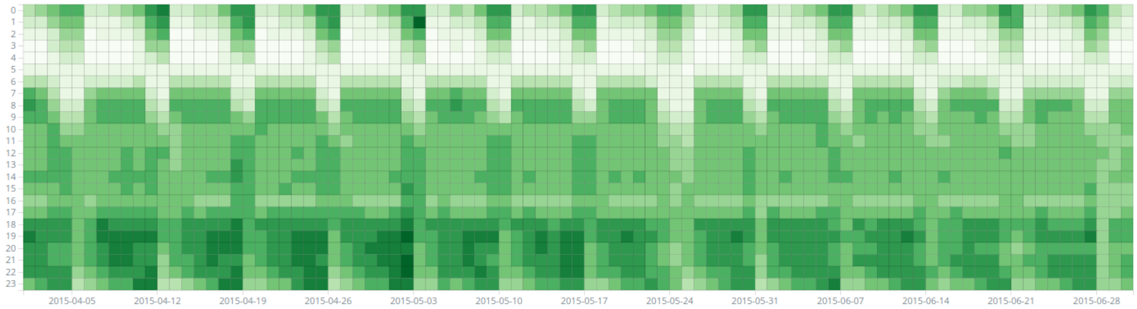

Figure 4: Heatmap on number of taxi trip at different week days and times of day.

The final example uses Line chart and percentile ranks aggregation. It calculates for each day that how many percent of that day's trips have occurred before a specific time of day, for example 6 am. Also this graph shows the major effect of weekends is on taxi trip frequency between 1 and 6 am.

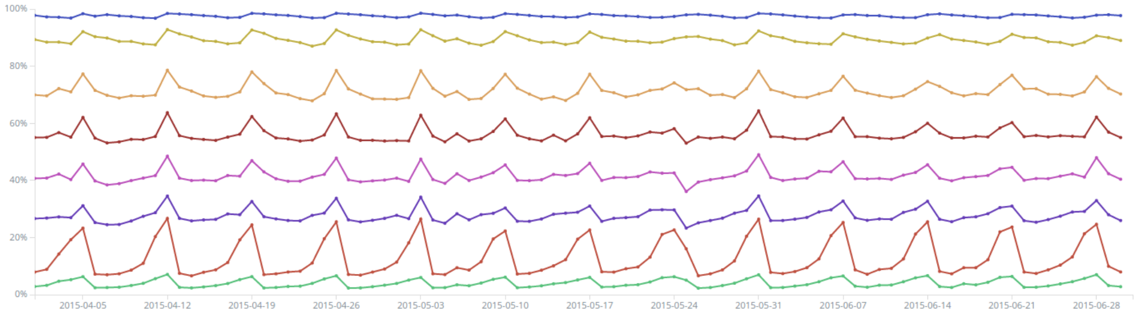

Figure 5: Percentile ranks of trips at 1, 6, 10, 13, 16, 19, 22, and 23.5 hours.