Zemanta challenge: Real-time Business Data Analytics - Report

Rok Novosel

December 11, 2017

1 Pipeline

For my solution I decided to use the so-called ELK pipeline: Elasticsearch (6.0), Logstash (6.0) and Kibana (6.0). I fully dockerized the solution using Docker Compose to manage containers. The source code and further instruction are available on
`https://github.com/novoselrok/real-time-data-analytics-zemanta2017`.

1.1 Logstash

Logstash is an open-source data processing pipeline that can ingest, transform and output data to a multitude of available outputs. I use Logstash to parse the different JSON logs and correctly output them based on their type. Bid logs get inserted as regular documents, but click and win logs are considered as updates to their original bid logs. Final document structure in Elasticsearch looks like this:

```
{
  "bid": { ... },
  "win": { ... },
  "click": { ... }
}
```

Two important settings in Logstash are batch size and workers. Workers setting is usually set to the number of available cores. Batch size setting is the maximum number of events an individual worker thread will collect from inputs before attempting to execute its filters and outputs. Larger batch sizes are usually more efficient but come with increased memory overhead.

1.2 Elasticsearch

Elasticsearch is an open-source search engine based on Lucene. In my solution I use it to store and query the logs. In Elasticsearch documents are grouped into indices, which themselves are composed of shards (Lucene indices). I group the documents by creating an index for each day. I also define a template for each index which specifies: number of primary shards, number of replica shards, other settings, mapping of document fields and possible aliases.

Elasticsearch runs on the JVM (Java Virtual Machine). An important parameter is the size of the JVM heap. Elasticsearch recommends setting it to

half of the total memory available. It is also important to set the minimum and maximum size of the heap to the same amount to prevent expensive heap resizing during runtime.

Large number of primary shards allows us to distribute and parallelize operations across shards. But each shard comes with a small yet fixed overhead in terms of disk space, memory usage and file descriptors used. So it is important to strike a balance between disk space and query efficiency. Elasticsearch offers two tools to fix this: index rollover and shrinking the index. Rollover rolls an index over to a new one when the existing index is considered to be too old or too large (e.g. logstash-2017.12.07-000001 rolls over to logstash-2017.12.07-000002). When an index rolls over this gives us the opportunity to shrink it into a new index which uses fewer primary shards.

Another important index setting is the refresh interval. Index refresh makes all operations performed since the last refresh available for search. Since this is an expensive operation it makes sense to increase to interval significantly. I set the interval to 45 seconds, but some experimentation is needed to find the optimal value. The downside is that new documents will be available for querying after 45 seconds.

To prevent data loss, each shard has a transaction log (translog) associated with it. Any index or delete operation is written to this translog. If Elasticsearch crashes, recent transactions can be replayed from the translog when the application recovers. By default data in the translog is persisted to the disc after every request. Elasticsearch allows us to set the translog durability setting to async and persist the data after an interval (five seconds by default). Since persisting (writing to disc) is an expensive operation I increased the interval to save some indexing power. But now we run a greater risk of data loss in an event of hardware failure.

Mapping is the process of defining how a document, and the fields it contains, are stored and indexed. By default Elasticsearch tries to analyse each field before indexing the document, but that is not always optimal. That is why I map fields like device type, gender and zip code as keywords which are not analysed by Elasticsearch.

2 Benchmarks

2.1 Indexing speed

First, I tested the raw indexing speed in an ideal situation (without querying Elasticsearch during indexing). I used the **bid_request_generator** tool and the following command:

```
$ ./bin/bid_request_generator 10000 1 | nc localhost 5000
```

The command runs with 10000 logs per second and pipes it to localhost port 5000 where Logstash is listening for incoming logs. The **bid_request_generator** tool outputs the indexing speed every 10 seconds. The results are in Figure 1. The mean indexing speed was **3448.6** logs/s with standard deviation of **378.3**.

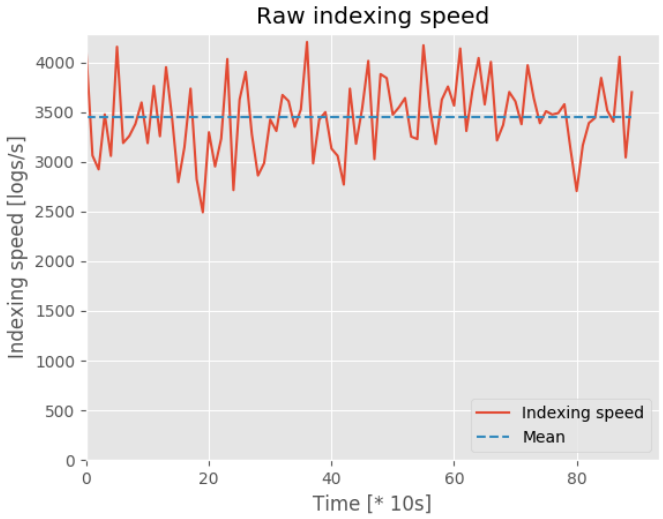

Figure 1: Raw indexing speed.

2.2 Querying speed

Next, I tested each of the five suggested queries while with and without indexing. Elasticsearch contained around 8 million documents at the time of querying. The tested queries are:

1. How a certain ad performed, what was the spend and in which locations it was spending the most.
2. Best ads in a campaign in a certain date range (clicks, impressions, spend).
3. Spend of a certain ad by day and publisher in New York.
4. Top 100 ads broken down by device and zip code.
5. Which publishers seem to be fraudulent based on a lot of impressions.

I ran all the queries one after the other multiple times with five second delay between them.

The results are in Figure 2. Here we can see the effects of aggressive caching by Elasticsearch. Only the first query takes a significant amount of time and all subsequent queries are cached and return the result almost instantly. Querying while indexing did not take significantly more time. The exceptions are spikes in response times which can be attributed to Elasticsearch refreshing the index.

Figure 3 shows the indexing speed while querying. The mean indexing speed was **3381.5** logs/s with standard deviation of **479.8**. At the start before Elasticsearch cached the queries the indexing process was slower, but after that the indexing speed reached the levels presented in Section 2.1.

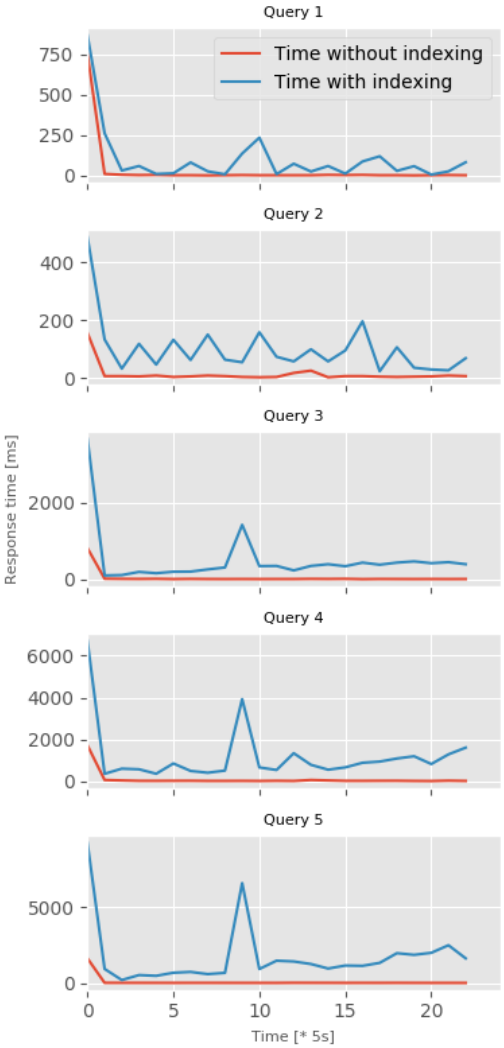

Figure 2: Response time for each of the queries.

Figure 3: Indexing speed while querying.

2.3 Stress test

For the final benchmark I decided to stress test the system. Before, I ran the queries sequentially, but now I will run all five of them in parallel every two seconds while indexing 10000 logs per second.

Figure 4 shows the indexing speed while stress testing. The mean indexing speed was **2142.4** logs/s with standard deviation of **559.7**. We can see that the stress test had a massive influence on the indexing speed. It dropped almost 40% compared to baseline from Section 2.1.

Figure 5 shows the query response times while stress testing. As expected the first query is the slowest and all subsequent queries benefit from Elasticsearch cache.

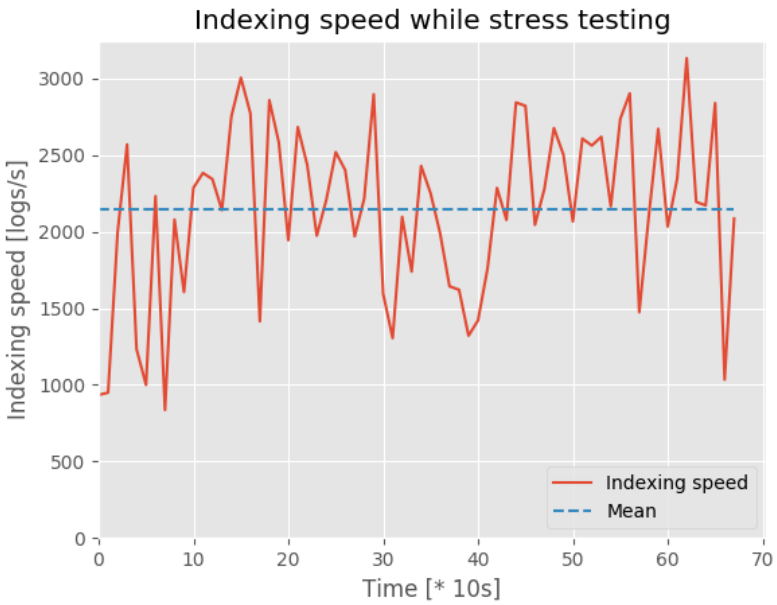

Figure 4: Indexing speed while stress testing.

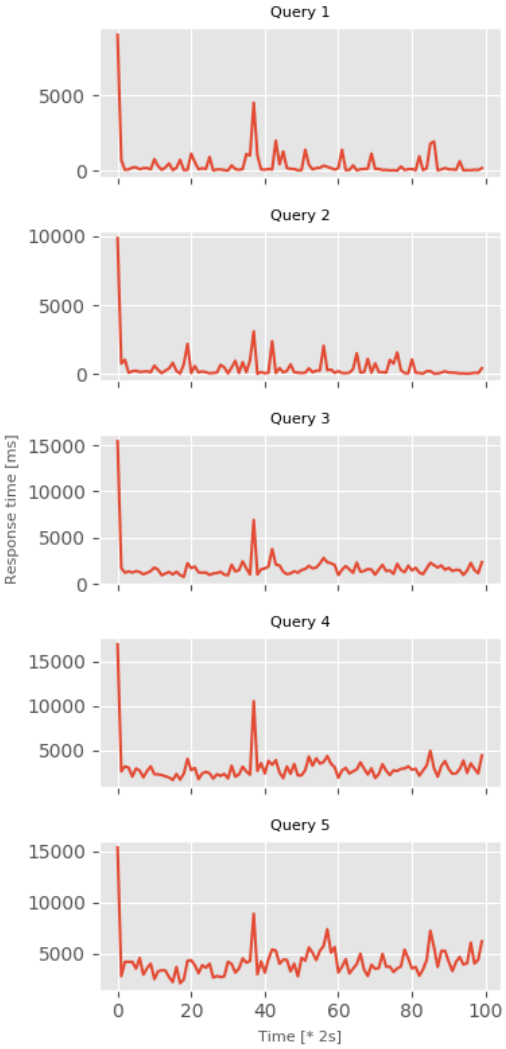

Figure 5: Query response times while stress testing.