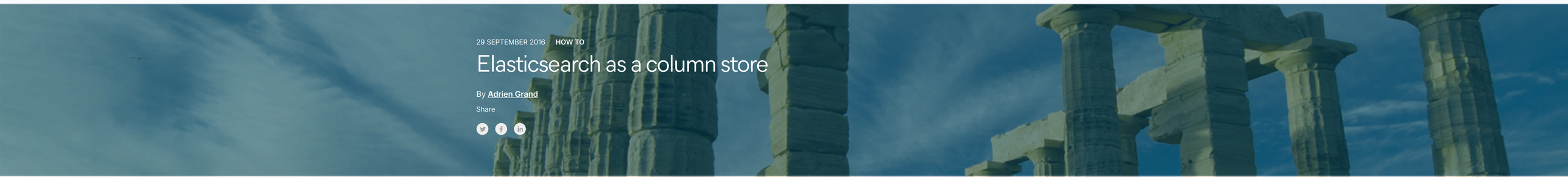

29 SEPTEMBER 2016 HOW TO

Elasticsearch as a column store

By Adrien Grand

Share

If you have no idea what questions you will want to ask your data when you start ingesting it, columnar storage is probably a good option for you: it helps in two areas that are often close to the heart of users who deal with large amounts of data:

- **Storage efficiency**: Since data that belongs to the same field is stored together, it can be expected to be quite homogeneous, which is something that can be leveraged to make compression more efficient.
- **"Fast" queries/aggregations**: The fact that data for the same field is stored together also helps make better use of the file-system cache since you will not have to load data for fields that are not needed by the query. In addition, by splitting the data into blocks and adding some meta-data about the range or set of unique values in the header of the block, some queries may be able to skip some blocks entirely at query time. There are also some techniques that allow to query the data without decompressing it first.

You might have noticed the quotes around *Fast* in the above paragraph. The reason is that however fast a linear scan is, it is still a linear scan and performs in linear time with the amount of queried data. Elasticsearch takes a different approach that consists in indexing all fields by default and only exposing queries that can leverage these indices so that identifying the matching documents does not need to visit all the data. Well, almost only, there is **one** query that might run a linear scan in order to identify matching documents, the script query. The purpose of this query is to not require you to reindex for once-in-a-while questions that you had not thought you would need to ask your data when you indexed it, such as finding all documents whose value for field X is less than the value of field Y. But other than this one, all queries make use of some form of index in order to quickly identify matching documents.

While Elasticsearch does not use a column-oriented view of the data for searching, it still needs one for workloads that work best with columnar data such as sorting and aggregations. In the next sections, we will do a quick survey of the history of columnar data in Lucene and Elasticsearch.

First, there was fielddata

Lucene was originally designed as a search library, allowing users to get the most relevant documents for a particular query. But soon users wanted to do more: they wanted to be able to sort by arbitrary fields, aggregate information about all matching documents, etc. To cover these needs, Lucene added a feature called FieldCache, which would "uninvert" the inverted index in order to build a column-oriented view of the data in memory. The initial release of Elasticsearch back in 2010 used the same mechanism with what it called **fielddata**, which was similar to `FieldCache`, but with more flexible caching management.

Then doc values

The growing use of `FieldCache` was embarassing: it required an amount of memory that was linear with the amount of indexed data, and made reopening the index slower since `FieldCache` entries had to be reloaded on all new segments, some of which being potentially very large due to merging. So on the one hand you had an efficient inverted index structure that allowed to find matching documents, but most collectors then had to rely on this inefficient memory-intensive data-structure in order to compute interesting things about the data. This is what lead Lucene to introduce doc values in Lucene 4.0, which was released in the end of 2012. Just like `FieldCache`, doc values provide a column-oriented view of the data, except that they are computed at index time and stored in the index. Their memory footprint is very low and getting doc values ready to use on a new segment is a matter of opening a file.

The fact that doc values are computed at index time also gives more opportunities for compression. The longer it takes to uninvert fielddata, the longer users have to wait before changes to the index becomes visible, which is undesirable. On the other hand doc values are either computed asynchronously at merge or on small datasets at flush time, so it is fine to spend more time doing interesting compression. Here is a non exhaustive list of some compressions techniques that doc values use that fielddata doesn't:

- Since doc values can afford to perform two passes on the data, they do a first pass on numeric fields to compute the required number of bits per value and a second one to do the encoding. This way they can use a fine-grained list of numbers of bits per value for the encoding. On the other hand fielddata only used 8, 16, 32 or 64 since changing the number of bits on the fly would incur a costly resize.
- The (sorted) terms dictionary for string fields is split into blocks and each block is compressed based on shared prefixes between consecutive terms.
- Numeric doc values compute the greatest common divisor between all values and only encode the non-common part. This is typically useful for dates that only have a second or day granularity.

Elasticsearch integration

Doc values became available in Elasticsearch in version 1.0 (February 2014) as an opt-in. However at that time, the performance did not quite match that of fielddata yet: Elasticsearch was hiding doc values behind its existing fielddata API, introducing overhead, and it took Lucene some time before introducing a dedicated type for multi-valued numerics and a random-access API for index inputs that helped performance significantly. Both these concerns got fixed in Elasticsearch 1.4, which was the first release to have matching performance of fielddata and doc values. We then enabled doc values by default in Elasticsearch 2.0 and the next major release of Elasticsearch (5.0) will not support fielddata anymore, leaving doc values as the only option for having a columnar representation of the data.

Specifics of Elasticsearch's column store

Does it make Elasticsearch a good general-purpose replacement for column-stores? No. As usual, it is better to do one thing and do it well. But the Elasticsearch column store has one key characteristic that makes it interesting: values are indexed by the doc id of the document that they belong to. What does it mean? Doc ids are transient identifiers of documents that live in a segment. A Lucene index is made of several components: an inverted index, a bkd tree, a column store (doc values), a document store (stored fields) and term vectors, and these components can communicate thanks to these doc ids. For instance, the inverted index is able to return an iterator over matching doc ids, and these doc ids can then be used to look up the value of a field thanks to doc values: this is how aggregations work.

This makes Elasticsearch particularly good at running analytics on small subsets of an index, since you will only pay the price for documents that match the query. This is how user interfaces to Elasticsearch like Kibana make it easy to slice and dice the data, by recursively filtering subsets that seem to have some interesting properties and running analytics on them.

Moreover, Lucene is geared towards making search operations fast. Thus doc values do not store the raw bytes for each document in case of a string field. Instead it writes separately a terms dictionary containing all unique values in sorted order and writes the indexes of string values in the column store. This helps since small integers make better keys than strings and allow to run eg. terms aggregations more efficiently by keying on the term index rather than the term bytes and using an array rather than a hash table as a data structure for storing per-term counts.

What's next?

Given that doc values need to be indexed by the doc id of the document they belong to, the current approach that Lucene takes is to reserve a fixed amount of space per document. While this makes doc values lookups very efficient, this has the undesired side-effect of not being space-efficient for sparse fields, since documents that do not have a value would still require the same amount of storage as documents that have values. This is why Elasticsearch works best when all documents in an index have a very similar set of fields.

However, there are ongoing developments that are exploring switching doc values to an iterator API rather than a random-access API so that these sparse cases could be handled more efficiently and that compression could be more efficient using techniques like run-length encoding.