29 SEPTEMBER 2016 HOW TO 如何

Elasticsearch as a column store
Elasticsearch 作为列存储

By 由 Adrien Grand 阿德里安·格兰德

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:

如果您不知道在开始摄取数据时要问数据哪些问题，那么列式存储可能是您的不错选择：它在两个方面有所帮助，这两个领域通常与处理大量数据的用户密切相关：

- **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.

  存储效率： 由于属于同一字段的数据存储在一起，因此可以预期它是相当同质的，这是可以利来提高压缩效率的。

- **"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.

  "快速"查询/聚合： 同一字段的数据存储在一起这一事实也有助于更好地利用文件系统缓存。 因为您不必为查询不需要的字段加载数据。此外，通过将数据拆分为块并在块的标头中添加一些有关范围或唯一值集的元数据，某些查询可能能够在查询时完全跳过某些块。还有一些技术允许在不先解压缩数据的情况下查询数据 。

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.

您可能已经注意到上一段中围绕 *Fast* 的引文。原因是，无论线性扫描的速度有多快，它仍然是线性扫描，并且根据查询的数据量以线性时间执行。Elasticsearch 采用一种不同的方法，包括默认为所有字段编制索引，并且只公开可以利用这些索引的查询，以便识别匹配的文档不需要访问所有数据。嗯，几乎**只有一个查询可能会**运行线性扫描来识别匹配的文档，即**脚本查询** 。此查询的目的是不要求您为偶尔出现的问题重新编制索引，这些问题在为数据编制索引时没有想到需要询问数据，例如查找字段 X 的值小于字段 Y 值的所有文档。但除了这个之外，所有查询都使用某种形式的索引来快速识别匹配的文档。

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.

虽然 Elasticsearch 不使用面向列的数据视图进行搜索，但它仍然需要一个用于最适合列式数据（例如排序和聚合）的工作负载。在接下来的部分中，我们将对 Lucene 和 Elasticsearch 中列式数据的历史进行快速调查。

First, there was fielddata
首先，有 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.

Lucene 最初设计为搜索库，允许用户获取与特定查询最相关的文档。但很快，用户就想做更多的事情：他们希望能够按任意字段排序，聚合有关所有匹配文档的信息，等等。为了满足这些需求，Lucene 添加了一个名为 FieldCache 的功能，它将“反相”倒排索引，以便构建内存中数据的面向列视图。2010 年发布的 Elasticsearch 初始版本使用了与 **fielddata** 相同的机制，它类似于 `FieldCache`，但具有更灵活的缓存管理。

Then doc values 然后 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.

`FieldCache` 的日益普及令人尴尬：它需要的内存量与索引数据量呈线性关系，并且由于必须在所有新 segment 上重新加载 `FieldCache` 条目，其中一些由于合并而可能非常大，因此重新打开索引的速度变慢了。因此，一方面，您有一个高效的倒排索引结构，可以找到匹配的文档，但大多数收集器不得不依赖这种低效的内存密集型数据结构来计算有关数据的有趣信息。这就是导致 Lucene 在 2012 年底发布的 Lucene 4.0 中引入**文档值**的原因。就像 `FieldCache` 一样，doc values 提供面向列的数据视图，不同之处在于它们是在索引时计算并存储在索引中。它们的内存占用非常低，并且准备好在新 segment 上使用 doc 值是打开文件的问题。

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:

doc values 是在索引时计算的这一事实也为压缩提供了更多机会。反转 fielddata 所需的时间越长，用户必须等待的时间就越长，索引的更改才会变得可见，这是不可取的。另一方面，文档值要么在合并时异步计算，要么在 flush 时对小数据集进行异步计算，因此花更多时间进行有趣的压缩是可以的。以下是 doc values 使用但 fielddata 不使用的一些压缩技术的非详尽列表：

- 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.

  由于 doc values 可以对数据执行两次传递，因此它们对数字字段进行第一次传递以计算每个值所需的位数，然后进行第二次传递以进行编码。这样，他们就可以使用每个值的细粒度位数列表进行编码。另一方面，fielddata 只使用 8、16、32 或 64，因为动态更改位数会导致昂贵的大小调整。

- The (sorted) terms dictionary for string fields is split into blocks and each block is compressed based on shared prefixes between consecutive terms.

  字符串字段的（排序）术语字典被拆分为多个块，每个块都根据连续术语之间的共享前缀进行压缩。

- 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.

  数字 doc values 计算所有值之间的最大公约数，并且仅对非公共部分进行编码。这通常适用于只有秒或天粒度的日期。

Elasticsearch integration
Elasticsearch 集成

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.

文档值在 Elasticsearch 1.0 版（2014 年 2 月）中作为可选选项提供。然而，当时的性能还不完全匹配 fielddata：Elasticsearch 将文档值隐藏在其现有的 fielddata API 后面，这带来了开销，并且 Lucene 花了一些时间才引入**用于多值数值的专用类型和用于索引输入的随机访问 API**，这对性能有很大帮助。这两个问题在 **Elasticsearch 1.4** 中都得到了解决，这是第一个具有 fielddata 和 doc 值匹配性能的版本。然后，我们在 Elasticsearch 2.0 中默认启用了文档值，Elasticsearch 的下一个主要版本 （5.0） 将不再支持 fielddata，使文档值成为具有数据列表示的唯一选项。

Specifics of Elasticsearch's column store
Elasticsearch 列存储的细节

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.

它是否使 Elasticsearch 成为列存储的良好通用替代品？不。像往常一样， **最好做一件事并把它做好** 。但 Elasticsearch 列存储有一个关键特征使其变得有趣：值按它们所属文档的文档 ID 进行索引。这是什么意思？文档 ID 是位于区段中的文档的临时标识符。Lucene 索引由几个组件组成：倒排索引、**bkd 树** 、列存储（文档值）、文档存储（存储字段）和术语向量，这些组件可以通过这些文档 ID 进行通信。例如，倒排索引能够在匹配的 doc id 上返回一个迭代器，然后由于 doc values，这些 doc id 可以用来查找字段的值：这就是聚合的工作原理。

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.

这使得 Elasticsearch 特别擅长对索引的小子集运行分析，因为您只需为与查询匹配的文档付费。这就是 Kibana 等 Elasticsearch 的用户界面如何通过递归过滤似乎具有一些有趣属性的子集并对其运行分析来轻松对数据进行切片和切块。

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

好的键，并且允许运行 eg.通过键入术语索引而不是术语字节，并使用数组而不是哈希表作为存储每个术语计数的数据结构，可以更有效地进行术语聚合。

What's next? 下一步是什么?

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.

鉴于 doc 值需要按其所属文档的 doc id 进行索引，Lucene 当前采用的方法是为每个文档保留固定数量的空间。虽然这使得文档值查找非常有效，但这有一个不希望的副作用，即对于稀疏字段来说，空间效率不高，因为没有值的文档仍然需要与有值的文档相同的存储量。这就是为什么当索引中的所有文档都有一组非常相似的字段时，Elasticsearch 效果最佳的原因。

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.

但是，有一些正在进行的开发正在探索将文档值切换到迭代器 API 而不是随机访问 API，以便可以更有效地处理这些稀疏情况，并且使用运行长度编码等技术可以更有效地进行压缩。