Doc Values 出现背景

> https://www.elastic.co/guide/cn/elasticsearch/guide/current/docvalues-intro.html
> https://www.elastic.co/guide/cn/elasticsearch/guide/current/docvalues.html

当你对一个字段进行排序时，Elasticsearch 需要访问每个匹配到的文档得到相关的值。倒排索引的检索性能是非常快的，但是在字段值排序时却不是理想的结构，当排序的时候，我们需要倒排索引里面某个字段值的集合。如下倒排索引：

```
Term Doc_1 Doc_2 Doc_3
-------------------------------------
brown | X | X |
dog | X | | X
dogs | | X | X
fox | X | | X
foxes | | X |
in | | X |
jumped | X | | X
 lazy | X | X |
 leap | | X |
 over | X | X | X
 quick | X | X | X
 summer | | X |
 the | X | | X
-------------------------------------
```

如果我们想要获得所有包含 `brown` 的文档的词的完整列表，我们会创建如下查询：

```
GET /my_index/_search
{
 "query" : {
 "match" : {
 "body" : "brown"
 }
 },
 "aggs" : {
 "popular_terms": {
  "terms" : {
  "field" : "body"
```

```
12   }
13   }
14   }
15 }
```

查询部分简单又高效。倒排索引是根据项来排序的，所以我们首先在词项列表中找到 `brown` ，然后扫描所有列，找到包含 `brown` 的文档。我们可以快速看到 `Doc_1` 和 `Doc_2` 包含 `brown` 这个 token。

然后，对于聚合部分，我们需要找到 `Doc_1` 和 `Doc_2` 里所有唯一的词项。 用倒排索引做这件事情代价很高： 我们会迭代索引里的每个词项并收集 `Doc_1` 和 `Doc_2` 列里面 token。这很慢而且难以扩展：随着词项和文档的数量增加，执行时间也会增加。

Doc values 通过转置两者间的关系来解决这个问题。倒排索引将词项映射到包含它们的文档，doc values 将文档映射到它们包含的词项：

```
Doc Terms
-----------------------------------------------------------------
Doc_1 | brown, dog, fox, jumped, lazy, over, quick, the
Doc_2 | brown, dogs, foxes, in, lazy, leap, over, quick, summer
Doc_3 | dog, dogs, fox, jumped, over, quick, the
-----------------------------------------------------------------
```

当数据被转置之后，想要收集到 `Doc_1` 和 `Doc_2` 的唯一 token 会非常容易。获得每个文档行，获取所有的词项，然后求两个集合的并集。

Doc Values 持久化

> https://www.elastic.co/guide/cn/elasticsearch/guide/current/_deep_dive_on_doc_values.html

Doc Values 是在索引时与 倒排索引 同时生成。也就是说 Doc Values 和 倒排索引 一样，基于 Segement 生成并且是不可变的。同时 Doc Values 和 倒排索引 一样序列化到磁盘，这样对性能和扩展性有很大帮助。

Doc Values 通过序列化把数据结构持久化到磁盘，我们可以充分利用操作系统的内存，而不是 JVM的 Heap 。 当 working set 远小于系统的可用内存，系统会自动将 Doc Values 驻留在内存中，使得其读写十分快速；不过，当其远大于可用内存时，系统会根据需要从磁盘读取 Doc Values，然后选择性放到分页缓存中。很显然，这样性能会比在内存中差很多，但是它的大小就不再局限于服务器的内存了。如果是使用 JVM 的 Heap 来实现那么只能是因为 OutOfMemory 导致程序崩溃了。

Doc Values 数据压缩

> https://www.elastic.co/guide/cn/elasticsearch/guide/current/_deep_dive_on_doc_values.html

现代 `CPU` 的处理速度要比磁盘快几个数量级（尽管即将到来的 NVMe 驱动器正在迅速缩小差距）。所以我们必须减少直接存磁盘读取数据的大小，尽管需要额外消耗 `CPU` 运算用来进行解压。要了解它如何压缩数据的，来看一组数字类型的 `Doc Values`：

```
Doc Terms
-----------------------------------------------------------------
Doc_1 | 100
Doc_2 | 1000
Doc_3 | 1500
Doc_4 | 1200
Doc_5 | 300
Doc_6 | 1900
Doc_7 | 4200
-----------------------------------------------------------------
```

按列布局意味着我们有一个连续的数据块：`[100, 1000, 1500, 1200, 300, 1900, 4200]`。因为我们已经知道他们都是数字（而不是像文档或行中看到的异构集合），所以我们可以使用统一的偏移来将他们紧紧排列。

而且，针对这样的数字有很多种压缩技巧。你会注意到这里每个数字都是 100 的倍数，`Doc Values` 会检测一个段里面的所有数值，并使用一个 *最大公约数* ，方便做进一步的数据压缩。

如果我们保存 `100` 作为此段的除数，我们可以对每个数字都除以 100，然后得到：`[1, 10, 15, 12, 3, 19, 42]` 。现在这些数字变小了，只需要很少的位就可以存储下，也减少了磁盘存放的大小。

`Doc Values` 在压缩过程中使用如下技巧。它会按依次检测以下压缩模式:

1. 如果所有的数值各不相同（或缺失），设置一个标记并记录这些值
2. 如果这些值小于 256，将使用一个 **简单的编码表**
3. 如果这些值大于 256，检测是否存在一个 **最大公约数**
4. 如果没有存在最大公约数，从最小的数值开始，统一计算 **偏移量** 进行编码

你会发现这些压缩模式不是传统的通用的压缩方式，比如 `DEFLATE` 或是 `LZ4`。因为列式存储的结构是严格且良好定义的，我们可以通过使用专门的模式来达到比通用压缩算法（如 LZ4 ）更高的压缩效果。

String 类型的压缩，通过借助 **顺序表（ordinal table**），String 类型也是类似进行编码的。String 类型是去重之后存放到顺序表的，通过分配一个 ID`，然后通过数字类型的 `ID 构建 Doc Values`。这样 `String 类型和数值类型可以达到同样的压缩效果。顺序表本身也有很多压缩技巧，比如固定长度、变长或是前缀字符编码等等。