8排序

8.1 排序

**相关性排序**

默认情况下，结果集会按照相关性进行排序 -- 相关性越高，排名越靠前。 这一章我们会讲述相关性是什么以及它是如何计算的。 在此之前，我们先看一下sort参数的使用方法。

**排序方式**

为了使结果可以按照相关性进行排序，我们需要一个相关性的值。在ElasticSearch的查询结果中， 相关性分值会用\_score字段来给出一个浮点型的数值，所以默认情况下，结果集以\_score进行倒序排列。

有时，即便如此，你还是没有一个有意义的相关性分值。比如，以下语句返回所有tweets中 user\_id 是否 包含值 1：

GET /\_search

{

"query" : {

"filtered" : {

"filter" : {

"term" : {

"user\_id" : 1

}

}

}

}

}

过滤语句与 \_score 没有关系，但是有隐含的查询条件 match\_all 为所有的文档的 \_score 设值为 1（查询所有的文档即不指定查询条件的时候默认将所有结果集的\_score的值设置为1）。 也就相当于所有的文档相关性是相同的。

**字段值排序**

下面例子中，对结果集按照时间排序，这也是最常见的情形，将最新的文档排列靠前。 我们使用 sort 参数进行排序：

GET /\_search

{

"query" : {

"filtered" : {

"filter" : { "term" : { "user\_id" : 1 }}

}

},

"sort": { "date": { "order": "desc" }}

}

你会发现这里有两个不同点：

"hits" : {

"total" : 6,

"max\_score" : null, <1>

"hits" : [ {

"\_index" : "us",

"\_type" : "tweet",

"\_id" : "14",

"\_score" : null, <1>

"\_source" : {

"date": "2014-09-24",

...

},

"sort" : [ 1411516800000 ] <2>

},

...

}

<1> \_score 字段没有经过计算，因为它没有用作排序。

<2> date 字段被转为毫秒当作排序依据。

首先，在每个结果中增加了一个 sort 字段，它所包含的值是用来排序的。 在这个例子当中 date 字段在内部被转为毫秒，即长整型数字1411516800000等同于日期字符串 2014-09-24 00:00:00 UTC。

其次就是 \_score 和 max\_score 字段都为 null。计算 \_score 是比较消耗性能的, 而且通常主要用作排序 -- 我们不是用相关性进行排序的时候，就不需要统计其相关性（me：节省性能）。 如果你想强制计算其相关性，可以设置track\_scores为 true。

**默认排序**

作为缩写，你可以只指定要排序的字段名称：

"sort": "number\_of\_children"

字段值默认以顺序排列，而 \_score 默认以倒序排列。

**多级排序**

如果我们想要合并一个查询语句，并且展示所有匹配的结果集使用第一排序是date，第二排序是 \_score：

GET /\_search

{

"query" : {

"filtered" : {

"query": { "match": { "tweet": "manage text search" }},

"filter" : { "term" : { "user\_id" : 2 }}

}

},

"sort": [

{ "date": { "order": "desc" }},

{ "\_score": { "order": "desc" }}

]

}

排序是很重要的。结果集会先用第一排序字段来排序，当用用作第一字段排序的值相同的时候， 然后再用第二字段对第一排序值相同的文档进行排序，以此类推。

多级排序不需要包含 \_score -- 你可以使用几个不同的字段，如位置距离或者自定义数值。

**字符串参数排序**

字符查询也支持自定义排序，在查询字符串使用sort参数就可以：

GET /\_search?sort=date:desc&sort=\_score&q=search

**为多值字段排序**

在为一个字段的多个值进行排序的时候， 其实这些值本来是没有固定的排序的-- 一个拥有多值的字段就是一个集合， 你准备以哪一个作为排序依据呢？

对于数字和日期，你可以从多个值中取出一个来进行排序，你可以使用min, max, avg 或 sum这些模式。 比说你可以在 dates 字段中用最早的日期来进行排序：

"sort": {

"dates": {

"order": "asc",

"mode": "min"

}

}

8.2 字符串排序

## 多值字段字符串排序

译者注: 多值字段是指同一个字段在ES索引中可以有多个含义，即可使用多个分析器(analyser)进行分词与排序，也可以不添加分析器，保留原值。

被分析器(analyser)处理过的字符称为analyzed field(译者注：即已被分词并排序的字段，所有写入ES中的字段默认圴会被analyzed), analyzed字符串字段同时也是多值字段，在这些字段上排序往往得不到你想要的值。 比如你分析一个字符 "fine old art",它最终会得到三个值。例如我们想要按照第一个词首字母排序， 如果第一个单词相同的话，再用第二个词的首字母排序，以此类推，可惜 ElasticSearch 在进行排序时 是得不到这些信息的。

当然你可以使用 min 和 max 模式来排（默认使用的是 min 模式）但它是依据art 或者 old排序， 而不是我们所期望的那样。

为了使一个string字段可以进行排序，它必须只包含一个词（me：否则就是多值）：即完整的not\_analyzed字符串(译者注：未经分析器分词并排序的原字符串)。 当然我们需要对字段进行全文本搜索的时候还必须使用被 analyzed 标记的字段。

在 \_source 下相同的字符串上排序两次会造成不必要的资源浪费。 而我们想要的是同一个字段中同时包含这两种索引方式，我们只需要改变索引(index)的mapping即可。 方法是在所有核心字段类型上，使用通用参数 fields对mapping进行修改。 比如，我们原有mapping如下：

"tweet": {

"type": "string",

"analyzer": "english"

}

改变后的多值字段mapping如下：

"tweet": { <1>

"type": "string",

"analyzer": "english",

"fields": {

"raw": { <2>

"type": "string",

"index": "not\_analyzed"

}

}

}

<1> tweet 字段用于全文本的 analyzed 索引方式不变。

<2> 新增的 tweet.raw 子字段索引方式是 not\_analyzed。

现在，在给数据重建索引后，我们既可以使用 tweet 字段进行全文本搜索，也可以用tweet.raw字段进行排序：

GET /\_search

{

"query": {

"match": {

"tweet": "elasticsearch"

}

},

"sort": "tweet.raw"

}

**警告**： 对 analyzed 字段进行强制排序会消耗大量内存。 详情请查阅《字段类型简介》相关内容。

8.3 相关性

## 相关性简介

我们曾经讲过，默认情况下，返回结果是按相关性倒序排列的。 但是什么是相关性？ 相关性如何计算？

每个文档都有相关性评分，用一个相对的浮点数字段 \_score 来表示 -- \_score 的评分越高，相关性越高。

查询语句会为每个文档添加一个 \_score 字段。评分的计算方式取决于不同的查询类型 -- 不同的查询语句用于不同的目的：fuzzy 查询会计算与关键词的拼写相似程度，terms查询会计算 找到的内容与关键词组成部分匹配的百分比，但是一般意义上我们说的全文本搜索是指计算内容与关键词的类似程度。

ElasticSearch的相似度算法被定义为 TF/IDF，即检索词频率/反向文档频率，包括一下内容：

检索词频率::

检索词在该字段出现的频率？出现频率越高，相关性也越高。 字段中出现过5次要比只出现过1次的相关性高。

反向文档频率::

每个检索词在索引中出现的频率？频率越高，相关性越低。 检索词出现在多数文档中会比出现在少数文档中的权重更低， 即检验一个检索词在文档中的普遍重要性。

字段长度准则::

字段的长度是多少？长度越长，相关性越低。 检索词出现在一个短的 title 要比同样的词出现在一个长的 content 字段。

单个查询可以使用TF/IDF评分标准或其他方式，比如短语查询中检索词的距离或模糊查询里的检索词相似度。

相关性并不只是全文本检索的专利。也适用于yes|no的子句，匹配的子句越多，相关性评分越高。

如果多条查询子句被合并为一条复合查询语句，比如 bool 查询，则每个查询子句计算得出的评分会被合并到总的相关性评分中。

## 理解评分标准

当调试一条复杂的查询语句时，想要理解相关性评分 \_score 是比较困难的。ElasticSearch 在 每个查询语句中都有一个explain参数，将 explain 设为 true 就可以得到更详细的信息。

GET /\_search?explain <1>

{

"query" : { "match" : { "tweet" : "honeymoon" }}

}

<1> explain 参数可以让返回结果添加一个 \_score 评分的得来依据。

增加一个 explain 参数会为每个匹配到的文档产生一大堆额外内容，但是花时间去理解它是很有意义的。 如果现在看不明白也没关系 -- 等你需要的时候再来回顾这一节就行。下面我们来一点点的了解这块知识点。

首先，我们看一下普通查询返回的元数据：

{

"\_index" : "us",

"\_type" : "tweet",

"\_id" : "12",

"\_score" : 0.076713204,

"\_source" : { ... trimmed ... },

}

这里加入了该文档来自于哪个节点哪个分片上的信息，这对我们是比较有帮助的，因为词频率和 文档频率是在每个分片中计算出来的，而不是每个索引中：

"\_shard" : 1,

"\_node" : "mzIVYCsqSWCG\_M\_ZffSs9Q",

然后返回值中的 \_explanation 会包含在每一个入口，告诉你采用了哪种计算方式，并让你知道计算的结果以及其他详情：

"\_explanation": { <1>

"description": "weight(tweet:honeymoon in 0)

[PerFieldSimilarity], result of:",

"value": 0.076713204,

"details": [

{

"description": "fieldWeight in 0, product of:",

"value": 0.076713204,

"details": [

{ <2> //检索词频率

"description": "tf(freq=1.0), with freq of:",

"value": 1,

"details": [

{

"description": "termFreq=1.0",

"value": 1

}

]

},

{ <3> //反向文档频率

"description": "idf(docFreq=1, maxDocs=1)",

"value": 0.30685282

},

{ <4> //字段长度准则

"description": "fieldNorm(doc=0)",

"value": 0.25,

}

]

}

]

}

<1> honeymoon 相关性评分计算的总结

<2> 检索词频率

<3> 反向文档频率

<4> 字段长度准则

**重要**： 输出 explain 结果代价是十分昂贵的，它只能用作调试工具 --千万不要用于生产环境。

第一部分是关于计算的总结。告诉了我们 "honeymoon" 在 tweet字段中的检索词频率/反向文档频率或 TF/IDF， （这里的文档 0 是一个内部的ID，跟我们没有关系，可以忽略。）

然后解释了计算的权重是如何计算出来的：

检索词频率:

检索词 `honeymoon` 在 `tweet` 字段中的出现次数。

反向文档频率:

检索词 `honeymoon` 在 `tweet` 字段在当前文档出现次数与索引中其他文档的出现总数的比率。

字段长度准则:

文档中 `tweet` 字段内容的长度 -- 内容越长，值越小。

复杂的查询语句解释也非常复杂，但是包含的内容与上面例子大致相同。 通过这段描述我们可以了解搜索结果是如何产生的。

**提示**： JSON形式的explain描述是难以阅读的 但是转成 YAML 会好很多，只需要在参数中加上 format=yaml

## Explain Api

#### 文档是如何被匹配到的

当explain选项加到某一文档上时，它会告诉你为何这个文档会被匹配，以及一个文档为何没有被匹配。

请求路径为 /index/type/id/\_explain, 如下所示：

GET /us/tweet/12/\_explain

{

"query" : {

"filtered" : {

"filter" : { "term" : { "user\_id" : 2 }},

"query" : { "match" : { "tweet" : "honeymoon" }}

}

}

}

除了上面我们看到的完整描述外，我们还可以看到这样的描述：

"failure to match filter: cache(user\_id:[2 TO 2])"

也就是说我们的 user\_id 过滤子句使该文档不能匹配到。

8.4 数据字段

## 数据字段

本章的目的在于介绍关于ElasticSearch内部的一些运行情况。在这里我们先不介绍新的知识点， 数据字段是我们要经常查阅的内容之一，但我们使用的时候不必太在意。

当你对一个字段进行排序时，ElasticSearch 需要进入每个匹配到的文档得到相关的值。 倒排索引在用于搜索时是非常卓越的，但却不是理想的排序结构。

* 当搜索的时候，我们需要用检索词去遍历所有的文档。
* 当排序的时候，我们需要遍历文档中所有的值，我们需要做反倒序排列操作。

为了提高排序效率，ElasticSearch 会将所有字段的值加载到内存中，这就叫做"数据字段"。

**重要**： ElasticSearch将所有字段数据加载到内存中并不是匹配到的那部分数据。 而是索引下所有文档中的值，包括所有类型。

将所有字段数据加载到内存中是因为从硬盘反向倒排索引是非常缓慢的。尽管你这次请求需要的是某些文档中的部分数据， 但你下个请求却需要另外的数据，所以将所有字段数据一次性加载到内存中是十分必要的。

ElasticSearch中的字段数据常被应用到以下场景：

* 对一个字段进行排序
* 对一个字段进行聚合
* 某些过滤，比如地理位置过滤
* 某些与字段相关的脚本计算

毫无疑问，这会消耗掉很多内存，尤其是大量的字符串数据 -- string字段可能包含很多不同的值，比如邮件内容。 值得庆幸的是，内存不足是可以通过横向扩展解决的，我们可以增加更多的节点到集群。

现在，你只需要知道字段数据是什么，和什么时候内存不足就可以了。 稍后我们会讲述字段数据到底消耗了多少内存，如何限制ElasticSearch可以使用的内存，以及如何预加载字段数据以提高用户体验。