bobby · 更新于 2022-11-16

上一节 13. es集成到mx... 1. 事务和分布式... 下一节

Elasticsearch 是一个分布式可扩展的实时搜索和分析引擎,一个建立在全文搜索引擎 Apache Lucene™ 基础上的搜索引擎.当然 Elasticsearch 并不仅仅是 Lucene 那么简单，它不仅包括了全文搜索功能，还可以进行以下工作:

- 分布式实时文件存储，并将每一个字段都编入索引，使其可以被搜索。
- 实时分析的分布式搜索引擎。
- 可以扩展到上百台服务器，处理PB级别的结构化或非结构化数据。

基本概念

先说Elasticsearch的文件存储，Elasticsearch是面向文档型数据库，一条数据在这里就是一个文档，用JSON作为文档序列化的格式，比如下面这条用户数据:

代码块

```
{
    "name" :     "John",
    "sex" :      "Male",
    "age" :      25,
    "birthDate": "1990/05/01",
    "about" :    "I love to go rock climbing",
    "interests": [ "sports", "music" ]
}
```

用Mysql这样的数据库存储就会容易想到建立一张User表，有balabala的字段等，在Elasticsearch里这就是一个_文档_，当然这个文档会属于一个User的_类型_，各种各样的类型存在于一个_索引_当中。这里有一份简易的将Elasticsearch和关系型数据术语对照表:

代码块

```
关系数据库      ⇒ 数据库 ⇒ 表    ⇒ 行     ⇒ 列(Columns)
Elasticsearch  ⇒ 索引(Index)   ⇒ 类型(type)  ⇒ 文档(Docments)  ⇒ 字段(Fie
```

一个 Elasticsearch 集群可以包含多个索引(数据库)，也就是说其中包含了很多类型(表)。这些类型中包含了很多的文档(行)，然后每个文档中又包含了很多的字段(列)。Elasticsearch的交互，可以使用Java API，也可以直接使用HTTP的Restful API方式，比如我们打算插入一条记录，可以简单发送一个HTTP的请求:

代码块

```
PUT /megacorp/employee/1
{
    "name" :     "John",
    "sex" :      "Male",
    "age" :      25,
```

```
        "interests": [ "sports", "music" ]
    }
```

更新，查询也是类似这样的操作，具体操作手册可以参见Elasticsearch权威指南

索引

Elasticsearch最关键的就是提供强大的索引能力了，其实InfoQ的这篇时间序列数据库的秘密(2)——索引写的非常好，我这里也是围绕这篇结合自己的理解进一步梳理下，也希望可以帮助大家更好的理解这篇文章。
Elasticsearch索引的精髓：

> 一切设计都是为了提高搜索的性能

另一层意思：为了提高搜索的性能，难免会牺牲某些其他方面，比如插入/更新，否则其他数据库不用混了。前面看到往Elasticsearch里插入一条记录，其实就是直接PUT一个json的对象，这个对象有多个fields，比如上面例子中的_name, sex, age, about, interests_，那么在插入这些数据到Elasticsearch的同时，Elasticsearch还默默1的为这些字段建立索引–倒排索引，因为Elasticsearch最核心功能是搜索。

Elasticsearch是如何做到快速索引的

InfoQ那篇文章里说Elasticsearch使用的倒排索引比关系型数据库的B-Tree索引快，为什么呢?

什么是B-Tree索引?

上大学读书时老师教过我们，二叉树查找效率是logN，同时插入新的节点不必移动全部节点，所以用树型结构存储索引，能同时兼顾插入和查询的性能。因此在这个基础上，再结合磁盘的读取特性(顺序读/随机读)，传统关系型数据库采用了B-Tree/B+Tree这样的数据结构：

为了提高查询的效率，减少磁盘寻道次数，将多个值作为一个数组通过连续区间存放，一次寻道读取多个数据，同时也降低树的高度。

什么是倒排索引?

继续上面的例子，假设有这么几条数据(为了简单，去掉about, interests这两个field):

代码块

```
    | ID | Name | Age  |  Sex     |
    | -- |:-----------:| -----:| -----:|
    | 1  | Kate         | 24 | Female
    | 2  | John         | 24 | Male
    | 3  | Bill         | 29 | Male
```

ID是Elasticsearch自建的文档id，那么Elasticsearch建立的索引如下:
Name:

代码块

```
    | Term | Posting List |
    | -- |:----:|
    | Kate | 1 |
    | John | 2 |
    | Bill | 3 |
```

意见反馈 收藏教程 标记书签

代码块

```
| Term | Posting List |
| -- |:----:|
| 24 | [1,2] |
| 29 | 3 |
```

Sex:

代码块

```
| Term | Posting List |
| -- |:----:|
| Female | 1 |
| Male | [2,3] |
```

Posting List

Elasticsearch分别为每个field都建立了一个倒排索引，Kate, John, 24, Female这些叫term，而[1,2]就是**Posting List**。Posting list就是一个int的数组，存储了所有符合某个term的文档id。
看到这里，不要认为就结束了，精彩的部分才刚开始...
通过posting list这种索引方式似乎可以很快进行查找，比如要找age=24的同学，爱回答问题的小明马上就举手回答：我知道，id是1，2的同学。但是，如果这里有上千万的记录呢？如果是想通过name来查找呢？

Term Dictionary

Elasticsearch为了能快速找到某个term，将所有的term排个序，二分法查找term，logN的查找效率，就像通过字典查找一样，这就是**Term Dictionary**。现在再看起来，似乎和传统数据库通过B-Tree的方式类似啊，为什么说比B-Tree的查询快呢？

Term Index

B-Tree通过减少磁盘寻道次数来提高查询性能，Elasticsearch也是采用同样的思路，直接通过内存查找term，不读磁盘，但是如果term太多，term dictionary也会很大，放内存不现实，于是有了**Term Index**，就像字典里的索引页一样，A开头的有哪些term，分别在哪页，可以理解term index是一颗树：

这棵树不会包含所有的term，它包含的是term的一些前缀。通过term index可以快速地定位到term dictionary的某个offset，然后从这个位置再往后顺序查找。

所以term index不需要存下所有的term，而仅仅是他们的一些前缀与Term Dictionary的block之间的映射关系，再结合FST(Finite State Transducers)的压缩技术，可以使term index缓存到内存中。从term index查到对应的term dictionary的block位置之后，再去磁盘上找term，大大减少了磁盘随机读的次数。
这时候爱提问的小明又举手了:"那个FST是神马东东啊?"
一看就知道小明是一个上大学读书的时候跟我一样不认真听课的孩子，数据结构老师一定讲过什么是FST。但没办法，我也忘了，这里再补下课：

> FSTs are finite-state machines that **map** a **term (byte sequence)** to an arbitrary **output**.

假设我们现在要将mop, moth, pop, star, stop and top(term index里的term前缀)映射到序号：0，1，2，3，4，5(term dictionary的block位置)。最简单的做法就是定义个Map，大家找到自己的位置对应入座就好了，但从内存占用少的角度想想，有没有更优的办法呢？答案就是：**FST理论(可以自行百度)**

⭕表示一种状态

将单词分成单个字母通过O和–>表示出来，0权重不显示。如果O后面出现分支，就标记权重，最后整条路径上的权重加起来就是这个单词对应的序号。

> FSTs are finite-state machines that map a term (**byte sequence**) to an arbitrary output.

FST以字节的方式存储所有的term，这种压缩方式可以有效的缩减存储空间，使得term index足以放进内存，但这种方式也会导致查找时需要更多的CPU资源。
后面的更精彩，看累了的同学可以喝杯咖啡……

压缩技巧

Elasticsearch里除了上面说到用FST压缩term index外，对posting list也有压缩技巧。
小明喝完咖啡又举手了:"posting list不是已经只存储文档id了吗？还需要压缩？"
嗯，我们再看回最开始的例子，如果Elasticsearch需要对同学的性别进行索引(这时传统关系型数据库已经哭晕在厕所……)，会怎样？如果有上千万个同学，而世界上只有男/女这样两个性别，每个posting list都会有至少百万个文档id。Elasticsearch是如何有效的对这些文档id压缩的呢？

Frame Of Reference

> 增量编码压缩，将大数变小数，按字节存储

首先，Elasticsearch要求posting list是有序的(为了提高搜索的性能，再任性的要求也得满足)，这样做的一个好处是方便压缩，看下面这个图例：

如果数学不是体育老师教的话，还是比较容易看出来这种压缩技巧的。
未压缩之前，6个整数，每个整数用4个字节压缩，一共需要24个字节；通过将数据增量编码，将73，300，302…，变成73, 227, 2，没有丢掉数据信息的情况下，将数据减小。标识位8，本身需要一个字节空间，表示73,227,2每个数都用8bit = 1byte来存储，需要1byte * 3 = 3bytes内存空间来存储，标识位5，本身需要1个字节空间存储，表示30，11，29都用5bit来存储，5bit * 3 = 15bit，但是需要16bit = 2bytes来存储，那么8，73,227,2，5，30，11，29,一共需要1 + 3 + 1 + 2 = 7bytes空间即可。
原理就是通过增量，将原来的大数变成小数仅存储增量值，再精打细算按bit排好队，最后通过字节存储，而不是大大咧咧的尽管是2也是用int(4个字节)来存储。

Roaring bitmaps

说到Roaring bitmaps，就必须先从bitmap说起。Bitmap是一种数据结构，假设有某个posting list：
[1,3,4,7,10]
对应的bitmap就是：
[1,0,1,1,0,0,1,0,0,1]
非常直观，用0/1表示某个值是否存在，比如10这个值就对应第10位，对应的bit值是1，这样用一个字节就可以代表8个文档id，旧版本(5.0之前)的Lucene就是用这样的方式来压缩的，但这样的压缩方式仍然不够高效，如果有1亿个文档，那么需要12.5MB的存储空间，这仅仅是对应一个索引字段(我们往往会有很多个索引字段)。于是有人想出了Roaring bitmaps这样更高效的数据结构。
Bitmap的缺点是存储空间随着文档个数线性增长，Roaring bitmaps需要打破这个魔咒就一定要用到某些指数特性：

将posting list按照65535为界限分块，比如第一块所包含的文档id范围在065535之间，第二块的id范围是65536131071，以此类推。再用<商，余数>的组合表示每一组id，这样每组里的id范围都在0~65535内了，剩下的就好办了，既然每组id不会变得无限大，那么我们就可以通过最有效的方式对这里的id存储。

细心的小明这时候又举手了:"为什么是以65535为界限?"

表示的最大
encode as

a bit set, and otherwise as a simple array using 2 bytes per value”，如果是大块，用节省点用bitset存，小块就豪爽点，2个字节我也不计较了，用一个short[]存着方便。
那为什么用4096来区分大块还是小块呢?
个人理解：都说程序员的世界是二进制的，4096*2bytes = 8192bytes < 1KB, 磁盘一次寻道可以顺序把一个小块的内容都读出来，再大一位就超过1KB了，需要两次读。

联合索引

上面说了半天都是单field索引，如果多个field索引的联合查询，倒排索引如何满足快速查询的要求呢?

- 利用跳表(Skip list)的数据结构快速做“与”运算，或者
- 利用上面提到的bitset按位“与”

先看看跳表的数据结构:

将一个有序链表level0，挑出其中几个元素到level1及level2，每个level越往上，选出来的指针元素越少，查找时依次从高level往低查找，比如55，先找到level2的31，再找到level1的47，最后找到55，一共3次查找，查找效率和2叉树的效率相当，但也是用了一定的空间冗余来换取的。
假设有下面三个posting list需要联合索引:

如果使用跳表，对最短的posting list中的每个id，逐个在另外两个posting list中查找看是否存在，最后得到交集的结果。
如果使用bitset，就很直观了，直接按位与，得到的结果就是最后的交集。

总结和思考

Elasticsearch的索引思路:

> 将磁盘里的东西尽量搬进内存，减少磁盘随机读取次数(同时也利用磁盘顺序读特性)，结合各种奇技淫巧的压缩算法，用及其苛刻的态度使用内存。

所以，对于使用Elasticsearch进行索引时需要注意:

- 不需要索引的字段，一定要明确定义出来，因为默认是自动建索引的
- 同样的道理，对于String类型的字段，不需要analysis的也需要明确定义出来，因为默认也是会analysis的
- 选择有规律的ID很重要，随机性太大的ID(比如java的UUID)不利于查询

关于最后一点，个人认为有多个因素:
其中一个(也许不是最重要的)因素: 上面看到的压缩算法，都是对Posting list里的大量ID进行压缩的，那如果ID是顺序的，或者是有公共前缀等具有一定规律性的ID，压缩比会比较高;

全文检索

前面是针对单个字段，name,age说明的快速索引解决方案，elastic通常是全文检索，文档类型的，通常需要将文档分词。比如有这样2个文档:

1. The quick brown fox jumped over the lazy dog
2. Quick brown foxes leap over lazy dogs in summer

为了创建倒排索引，我们首先将每个文档的 content 域拆分成单独的 词（我们称它为 词条 或 tokens），创建一个包含所有不重复词条的排序列表，然后列出每个词条出现在哪个文档。结果如下所示：
Term Doc_1 Doc_2----------Quick | | XThe | X |brown | X | Xdog | X |dogs | | Xfox | X |foxes | | Xin | | Xjumped | X |lazy | X | Xleap | | Xover | X | Xquick | X |summer | | Xthe | X |

那么，对于brown这个词来说，Posting list就是[1, 2]，quick的Posting list就是[2]，当然了，对于大小写，同义词都是要考虑的，这样就变成了前面的单个词那样，同样利用倒排索引，增量编码，大数变小数，使用字节存储。

13. es集成到mxshop_srvs 上一节 下一节 1. 事务和分布式事务