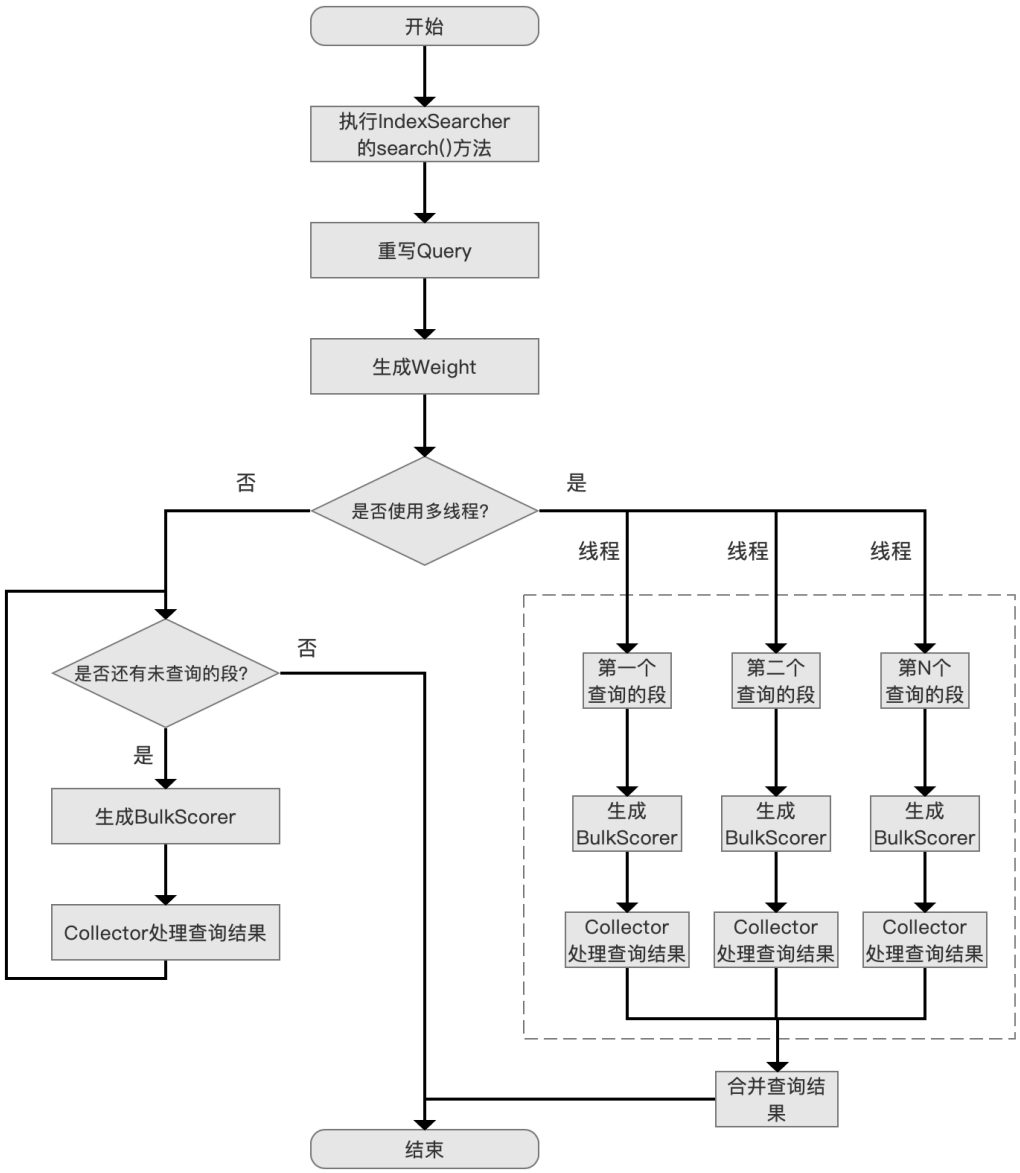
查询原理（四）

本文承接查询原理（三），继续介绍查询原理。

查询原理流程图

图1：



点击查看大图

图2、图3是BooleanQuery的查询实例，在查询原理（三）中我们根据这个例子介绍了`生成BulkScorer`的流程点，本篇文章根据这个例子，继续介绍图1中剩余的流程点。

图2：

```
Document doc ;
// 文档0
doc = new Document();
doc.add(new TextField( name: "content",  value: "h a h", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author1", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档1
doc = new Document();
doc.add(new TextField( name: "content",  value: "f f", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author2", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档2
doc = new Document();
doc.add(new TextField( name: "content",  value: "h a c", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author3", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档3
doc = new Document();
doc.add(new TextField( name: "content",  value: "a h h e", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author4", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档4
doc = new Document();
doc.add(new TextField( name: "content",  value: "a f h", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author5", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档5
doc = new Document();
doc.add(new TextField( name: "content",  value: "h", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author6", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档6
doc = new Document();
doc.add(new TextField( name: "content",  value: "c a", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author7", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档7
doc = new Document();
doc.add(new TextField( name: "content",  value: "f f f", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author8", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档8
doc = new Document();
doc.add(new TextField( name: "content",  value: "e a d f h a a ", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author9", Field.Store.YES));
indexWriter.addDocument(doc);
// 文档9
doc = new Document();
doc.add(new TextField( name: "content",  value: "a c a b c", Field.Store.YES));
doc.add(new TextField( name: "author",  value: "author10", Field.Store.YES));
indexWriter.addDocument(doc);
indexWriter.commit();
// 索引阶段结束
```

图3：

```
// 查询阶段
IndexReader reader = DirectoryReader.open(indexWriter);
IndexSearcher searcher = new IndexSearcher(reader);
BooleanQuery.Builder builder = new BooleanQuery.Builder();
// 子查询1
builder.add(new TermQuery(new Term( fld: "content",  text: "h")), BooleanClause.Occur.SHOULD);// 包含"h"的文档共6篇：0、2、3、4、5、8
// 子查询2
builder.add(new TermQuery(new Term( fld: "content",  text: "f")), BooleanClause.Occur.SHOULD);// 包含"f"的文档共4篇：1、4、7、8
// 子查询3
builder.add(new TermQuery(new Term( fld: "content",  text: "a")), BooleanClause.Occur.SHOULD);// 包含"a"的文档共7篇：0、2、3、4、6、8、9
// 子查询4
builder.add(new TermQuery(new Term( fld: "content",  text: "e")), BooleanClause.Occur.MUST_NOT);// 包含"e"的文档共2篇：3、8
builder.setMinimumNumberShouldMatch(2);
Query query = builder.build();
```

Collector处理查询结果

在查询原理（三）文章中的`生成BulkScorer`流程点，我们获得了每一个子查询对应的文档号跟词频，见图4，结合查询原理（二）文章中的`生成Weight`流程点，我们就可以在当前流程点获得满足查询条件（图3）的文档集合及对应的文档打分值。

图4：

子查询1

docDeltaBuffer	0	2	3	4	5	8		...	
数组下标	0	1	2	3	4	5	6		

freqBuffer	2	1	2	1	1	1		...	
数组下标	0	1	2	3	4	5	6		

子查询2

docDeltaBuffer	1	4	7	8				...	
数组下标	0	1	2	3	4	5	6		

freqBuffer	2	1	3	1				...	
数组下标	0	1	2	3	4	5	6		

子查询3

docDeltaBuffer	0	2	3	4	6	8	9	...	
数组下标	0	1	2	3	4	5	6		

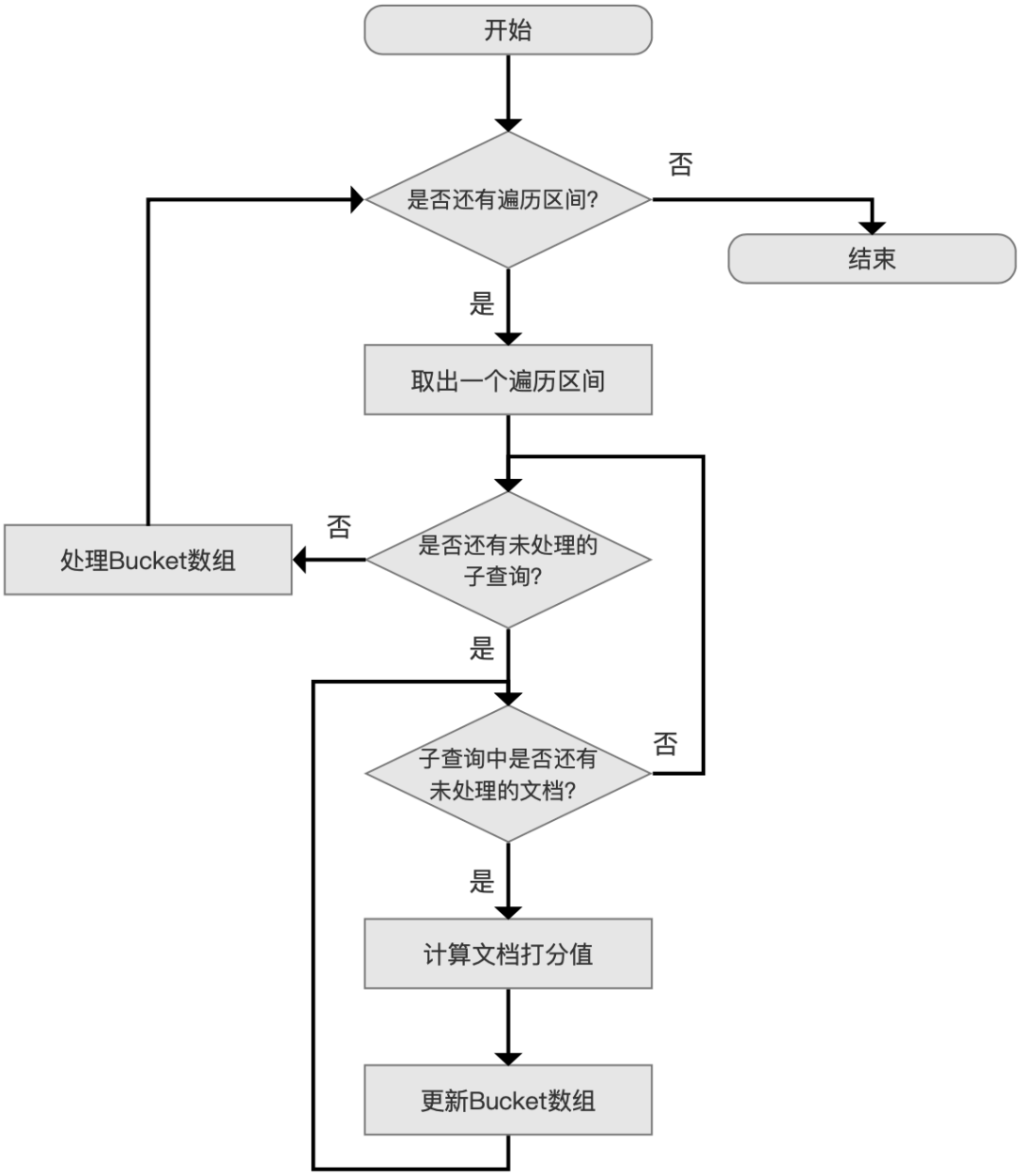
freqBuffer	1	1	1	1	1	3	2	...	
数组下标	0	1	2	3	4	5	6		

子查询4

docDeltaBuffer	3	8						...	
数组下标	0	1	2	3	4	5	6		

`Collector`处理查询结果的流程图

图5：



是否还有遍历区间?

图6：

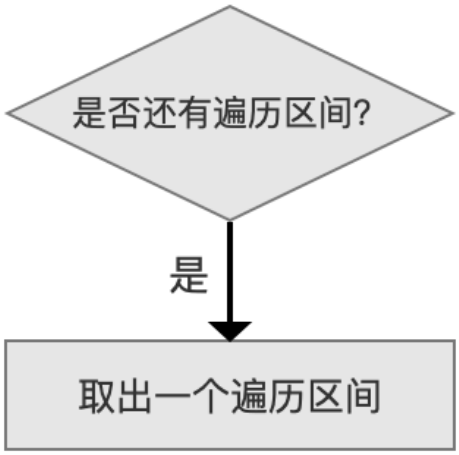


图4中的子查询4，该查询对应的文档号集合不是用户期望返回的，那么可以根据这些文档号划分出多个左闭右开的遍历区间。

满足子查询4条件的文档号集合为3、8，故可以划分出3个遍历区间：

- [0，3)
- [4，8)
- [9，2147483647)：2147483647即 Integer.MAX_VALUE

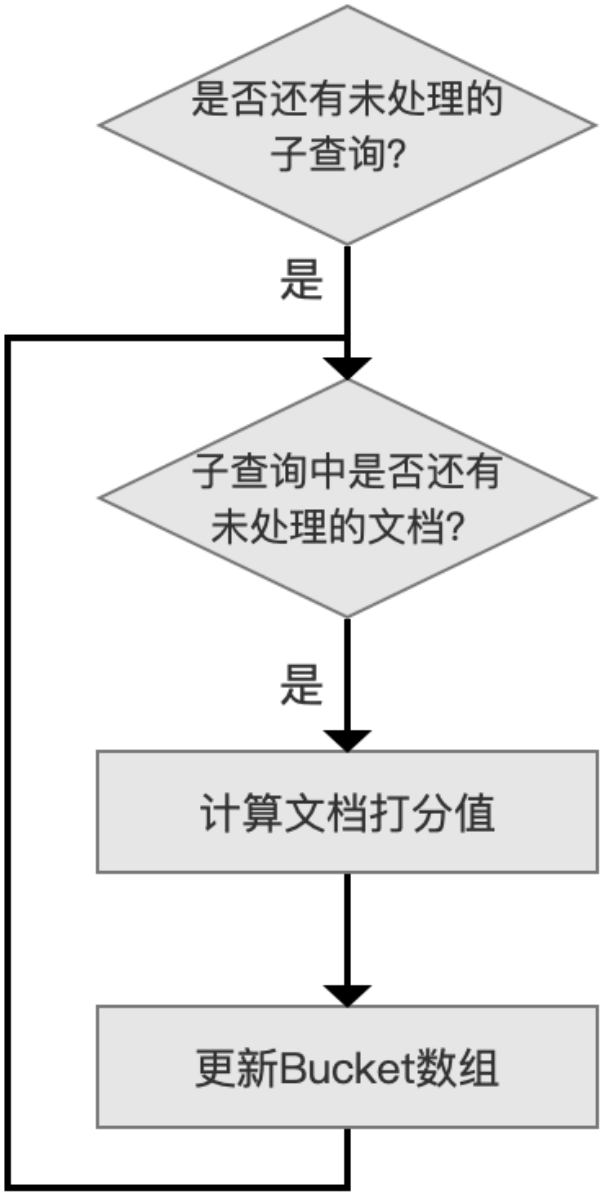
随后从每个遍历区间中找到满足子查询1、子查询2、子查询3且minShouldMatch为2的文档号，minShouldMatch通过图3中的builder.setMinimumNumberShouldMatch方法设置，描述的是用户期望的文档必须至少满足子查询1、子查询2、子查询3中的任意两个（minShouldMatch）的查询条件。

为什么要使用遍历区间：

- 降低时间复杂度：通过左闭右开实现过滤子查询4中的文档号（时间复杂度O(n)），否则当我们找出根据子查询1、子查询2、子查询3且minShouldMatch为2文档号集合后，每一个文档号都要判断是否满足子查询4的条件（时间复杂度O(n*m)），其中n跟m都为满足每一个子查询条件的文档数量，在图4中的例子中，n值为17（6 + 4 + 7，3个docDeltaBuffer数组的元素个数的和值），m值为2

处理子查询的所有文档号

图7：



该流程处理子查询的所有文档号，先看下Bucket数组，该数组的数组下标用来描述文档号，数组元素是Bucket对象，它用来记录该文档的打分值跟满足子查询条件的查询个数，Bucket类如下所示：

```
static class Bucket {
    double score;
    int freq;
}
```

- score：文档打分值
- freq：满足子查询条件的查询个数

freq

我们先说下freq，freq描述的是满足子查询条件的查询个数，例如图2中的文档8（文档号为8），因为文档号8中包含了"h"、"f"、"a"，所以它满足子查询1、子查询2、子查询3的三个查询条件，故文档号8对应的Bucket对象的freq值为3。

score

图8为BM25模型的理论打分公式：

图8：

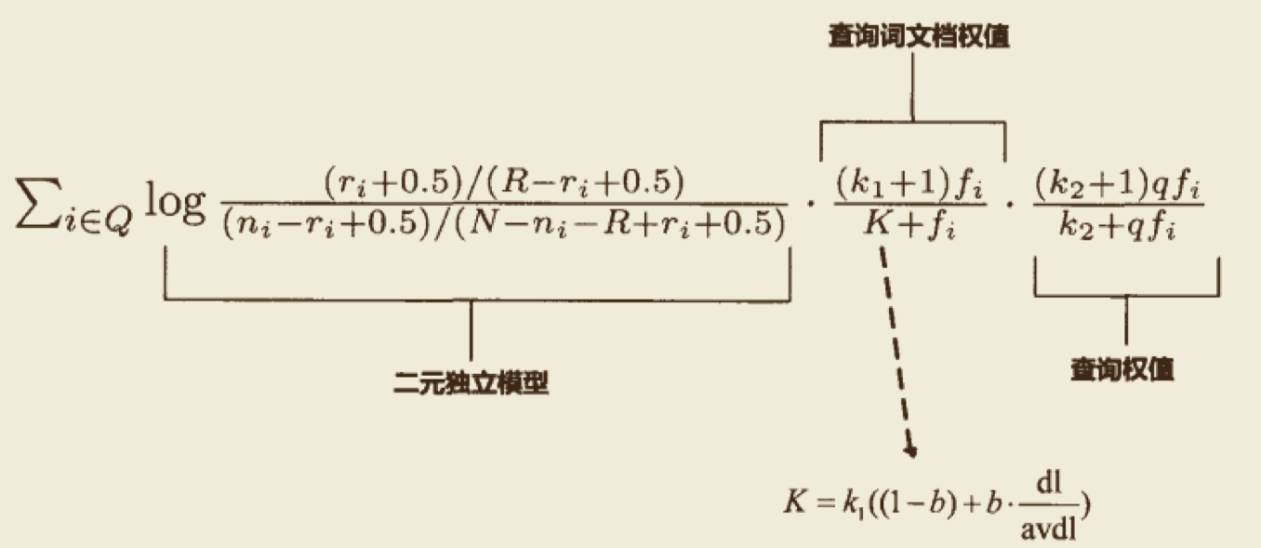


图17源自于<<这就是搜索引擎>>，作者：张俊林。

图9为在Lucene7.5.0版本中BM25模型的具体实现BM25Similarity的公式：

图9：

$$\sum_{i \in Q} \log \frac{1+(docCount_i - docFreq_i + 0.5)}{docFreq_i + 0.5} * \frac{(k_1+1)freq}{norm+freq} * boost_i$$

其中$norm = k_1 * ((1-b) + b * \frac{dl}{avgdl})$

从图9的公式可以看出，一篇文档的打分值是一个**累加值**，累加的过程即`更新Bucket数组`的流程，如果一篇文档满足多个子查询的条件，那么该文档的打分值是每个子查询对这篇文档打分的和值。

例如图2中的文档0，该文档包含了两种term，分别是 "a"，"h"，故文档0满足图3中的两个子查询的条件，分别是子查询1、子查询3，所以文档0的打分值是两个查询对这篇文档打分的和值，最后将这个和值添加到Bucket数组的数组下标为0（因为文档0的文档号是0）的数组元素Bucket对象中，该对象的freq的值同理会被赋值为2。

BM25Similarity打分公式

图10：

idf

$$\sum_{i \in Q} \log \frac{1+(docCount_i - docFreq_i + 0.5)}{docFreq_i + 0.5} * \frac{(k_1+1) freq}{norm + freq} * boost_i$$

cache[normValue]

其中$norm = k_1 * ((1-b) + b * \frac{dl}{avgdl})$

- Idf、boost、avgdl、docCount、docFreq：这些值在查询原理（二）中计算SimWeight时获得，不赘述
- freq：子查询条件中的域值在文档（正在计算打分的文档）中的词频，即图4中的freqBuffer数组的数组元素
- k_1、b：BM25模型的两个调节因子，这两个值都是经验参数，默认值为k_1 = 1.2、b = 0.75。k_1值用来控制非线性的词频标准化（non-linear term frequency normalization）对打分的影响，b值用来控制文档长度对打分的影响
- norm：该值描述的是文档长度对打分的影响，满足同一种查询的多篇文档， 会因为norm值的不同而影响打分值
    - cache数组：在查询原理（二）文章中，我们简单的提了一下cache生成的时机是在`生成Weight`的流程中，下面详细介绍该数组。
        - cache数组的数组下标normValue描述的是文档长度值，这是一个标准化后的值（下文会介绍），在Lucene中，用域值的个数来描述文档长度，例如图3中的子查询1，它查询的条件是域名为"content"，域值为"h"的文档，那么对于文档0，文档长度值为域名为"content"，term为"h"在文档0中的个数，即2；cache数组的数组元素即norm值
        - 上文说到域值的个数来描述文档长度，但是他们两个的值不总是相等，域值的个数通过标准化（normalization）后来描述文档长度，标准化的过程是将文档的长度控制在[0，255]的区间中，跟归一化的目的是类似的，为了平衡小文档相跟大文档的对打分公式的影响，标准化的计算方式不在本文中介绍，感兴趣的可以看https://github.com/LuXugang/Lucene-7.5.0/blob/master/solr-7.5.0/lucene/core/src/java/org/apache/lucene/util/SmallFloat.java中的 intToByte4(int)方法，该方法的返回值与0XFF执行与操作后就得到标准化后的文档长度值
        - 根据标准化后的文档长度值（取值范围为[0，255]）就可以计算出norm中dl的值，dl为文档长度值对应的打分值，同样两者之间的计算方法不在本文中介绍，感兴趣可以看https://github.com/LuXugang/Lucene-7.5.0/blob/master/solr-7.5.0/lucene/core/src/java/org/apache/lucene/util/SmallFloat.java中的byte4ToInt(byte)方法，图11给出了文档长度值跟dl之间的映射关系

图11：

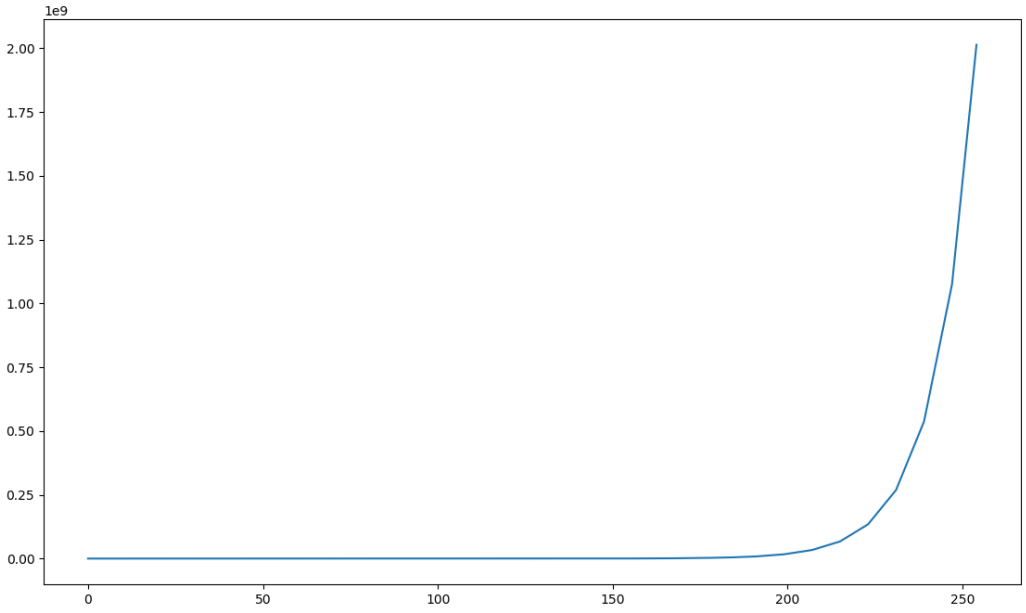


图11中，**横坐标为文档长度值，纵坐标为dl**，由于数据跨度太大，无法看出文档长度值较小的区间的趋势图，故图12给出的是文档长度值在[0，100]区间的映射图

图12：

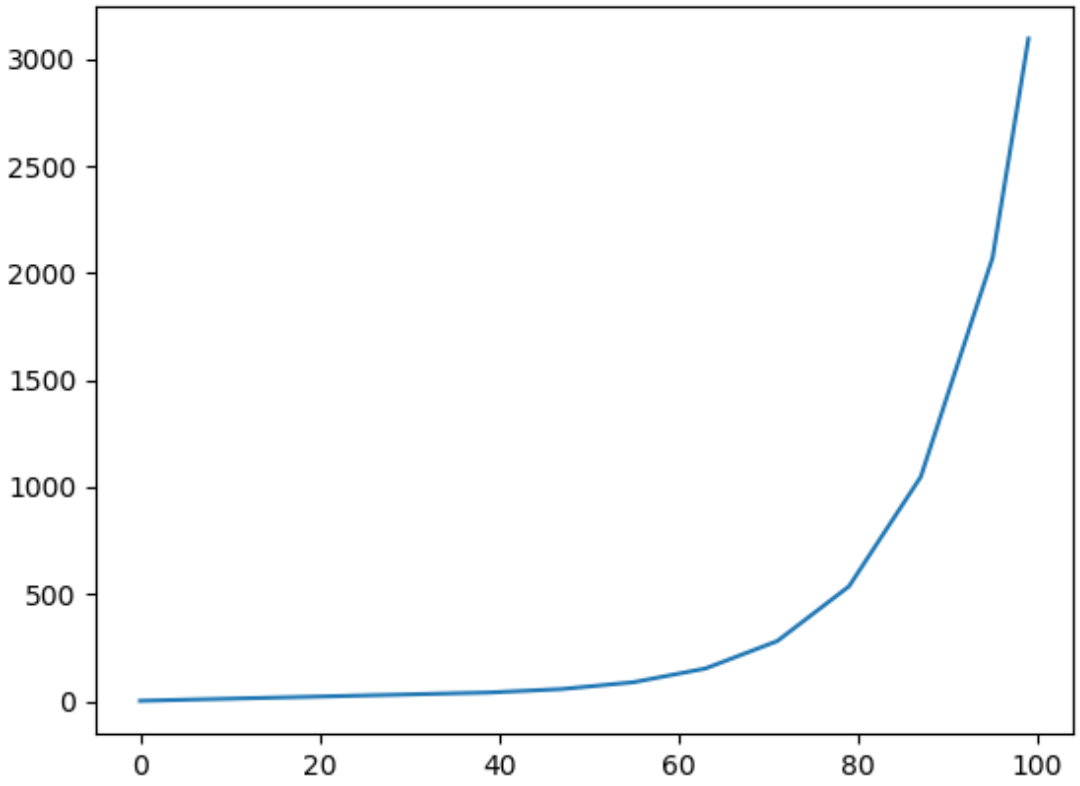


图13中，文档长度值在[0，50]区间的映射图，能进一步看出文档长度值小于等于40时，dl正比于文档长度值

图13：

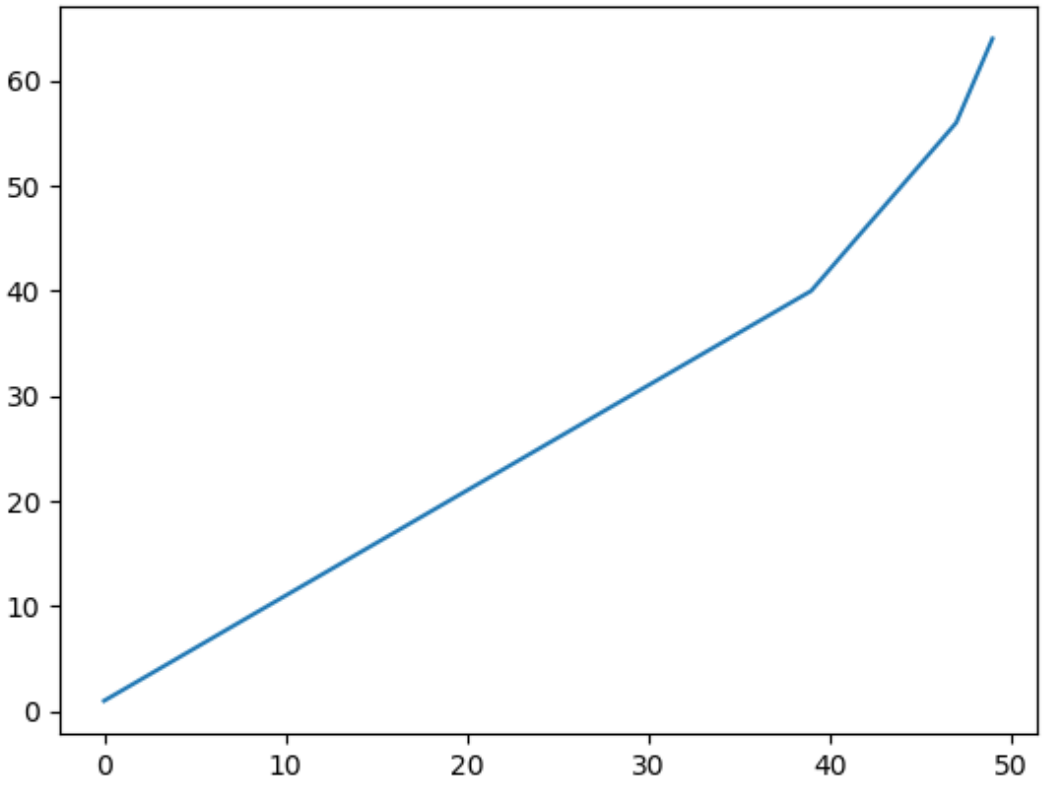
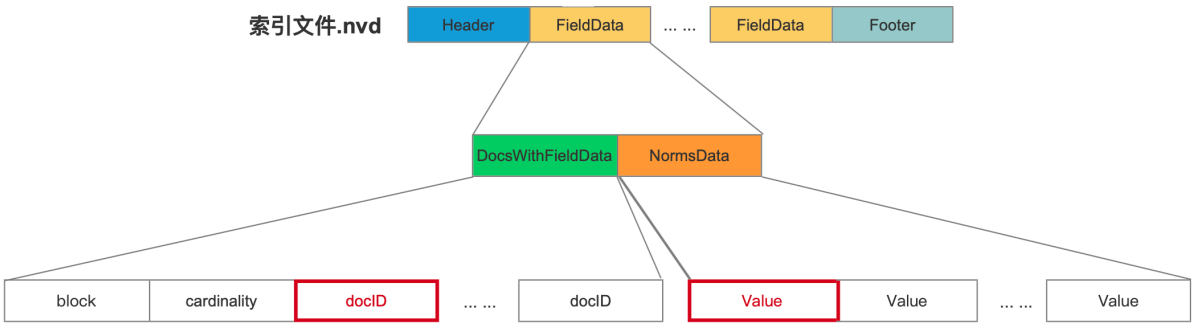


图10中的normValue根据文档号从索引文件.nvd中获得，图14中用红框标识了一篇文档的文档号及其对应的normValue。

读取索引文件的过程不展开介绍，本人不想介绍的原因是，只要了解索引文件的数据结构（见索引文件数据结构）是如何生成的，自然就明白如何读取索引文件~~

图14：



处理Bucket数组

图15：

处理Bucket数组

处理Buck数组的过程就是找出所有满足图3中minShouldMatch的文档，然后分别将文档号交给Collector收集器处理

某个遍历区间内的生成Bucket数组的过程在文档号合并（SHOULD）的文章中已经介绍，不过注意的是，在那篇文档中，没有考虑文档的打分值，故Bucket数组只介绍了freq。由于那篇中没有类似图3中的子查询4，所以遍历区间为[0，2147483647]。

对于本篇文章中图2、图3的例子，在遍历区间为[0，3)对应生成的Bucket数组如下所示，相比较文档号合并（SHOULD）中的内容，我们增加每篇文档的打分值，列出遍历区间为[0，3)的Bucket数组：

遍历区间[0，3)

图16：

在图16中，文档0跟文档2的freq 大于等于minShouldMatch（2），故这两篇文档满足图3中的查询要求。

结语

至此，我们介绍了单线程下的查询原理的所有流程点，但还有一个很重要的逻辑没有介绍，那就是在图5的`是否还有未处理的子查询`流程点，我们并没有介绍在还有未处理的子查询的情况下，如何选择哪个子查询进行处理，这个逻辑实际是个优化的过程，可能可以减少遍历区间的处理，以图2、图3为例，尽管根据子查询4，我们得出3个遍历区间，实际上我们只要处理[0，3)、[4，8)这两个逻辑区间，至于原因会在下一篇文档中展开。

图2.、图3的demo点击这里：https://github.com/LuXugang/Lucene-7.5.0/blob/master/LuceneDemo/src/main/java/lucene/query/BooleanQuerySHOULDNOTTEST.java。

另外对于多线程的情况，图1中的`合并查询结果`流程也留到下一篇文章中介绍。

点击下载附件