链滴

DocValues 详解

作者：llh

原文链接：https://ld246.com/article/1490785000674

来源网站：链滴

搜索引擎最简单的查询场景是

通过查询条件的 term 找到 term 对应的倒排列表
对倒排列表进行合并，得到符合查询条件的文档 id 列表并同时计算每个文档的得分
进行排序，取 topN 个文档 id
根据 topN 的文档 id 到磁盘中的正排索引中读取要返回的 field 组成返回文档
最后返回给调用者。

如果复杂点，需要对搜索结果根据字段排序（sort），聚合统计（group、facet、stat）等操作这种场景下就需要根据 docId 来获取相应的 field 值来进行计算了。但由于查询命中的结果数往往非大，不能像构造 topN 的返回文档那样到磁盘去获取字段值，这样实在太慢了。
针对这种需求，Lucene 提供了 DocValues 来快速获取 field 值的方法。
<h2 id="DocValues的功能-">DocValues 的功能。</h2>
DocVaues 的功能很简单，就是根据文档 Id，获取这个文档的字段值。下面是 NumericDocValus 类说明
<pre><code class="highlight-chroma">/**
 * A per-document numeric value.
 */
public abstract class NumericDocValues {
  /** Sole constructor. (For invocation by subclass
   * constructors, typically implicit.) */
  protected NumericDocValues() {}
  /**
   * Returns the numeric value for the specified document ID. * @param docID document ID to lookup
   * @return numeric value
   */ public abstract long get(int docID);
}
</code></pre>
可以把它看成是一个大数组，数组下标是文档 id，数组值是字段值。根据文档 id 读取字段值非快。当然这个“大数组”是经过了压缩的（它的压缩算法非常巧妙，后面会另起一篇文章详细介绍）否则非常占内存。其实，在 3.x 之前的版本（其实是 fieldcache）中，还真是一个大数组构成的。</>
<h2 id="DocValues的种类">DocValues 的种类</h2>

NumericDocValues 数值型的 DocValues，用于存放 int、long、double、float 等数值

BinaryDocValues 单值二进制 DocValues，String 类型的字段就是用这种类型来存放的。

SortedDocValues 单值、有序的二进制 DocValues，也用于存放 String 值，相当于用两个数来存放 DocValues，一个数组 dictionary[]存放去重后的值，并且是有序的，另外一个数组 order[]来记录文档的 field 值在 dictionary[]的偏移量，order[]的下标是 docId，值是 dictionary 的下标。和 BinaryDocValues 相比，如果值的重复率比较高，则由于去重了，会比 BinaryDocValues 节省很多

存，并且 dictionary[]是有序的，还可以进一步压缩。

SortedNumericDocValues 用于存放多值数值类型。

SortedSetDocValues 用于存放多值二进制类型。

<h2 id="DocValues的构建">DocValues 的构建</h2>
DocValues 需要在构建索引的时候额外生成 docId 到字段值的映射文件，这样，根据映射文件造存放于内存中的 docId 和字段值映射数据结构的时候就很快了。而且由于有 docId 到字段值的映文件，因此，docvalue 还有基于磁盘的实现（直接读取磁盘上的映射文件），虽然比内存的实现要，但也比从正排索引中读取快多了。映射文件会额外占用磁盘空间，Lucene 也允许通过不生成映射件的方式构造 DocValues，这时候就会遍历 term，从 term 找到 docId，进而构造出 docId 到字段映射的数据结构了，这种方式比生成映射文件的方式要慢很多，好处是不占用磁盘空间。获取 DocVales 实例的两种方式：

通过 DocValues 工具类获取

通过 DocValues 工具类获取，只能获取到通过映射文件生成的 DocValues
<pre><code class="highlight-chroma">public static NumericDocValues getNumeric(LeafReader reader, String field) throws IOExc ption {
  NumericDocValus dv = reader.getNumericDocValues(field);
if (dv == null) {
    checkField(read r, field, DocValuesType.NUMERIC);
return emptyNu eric();
  } else {
    return dv;
  }
}
</code></pre>
可以看到，其实是通过底层的 LeafReader 获取的，我们再看看 LeafReader 获取 DocValues 代码
<pre><code class="highlight-chroma">@Override
public final Numer cDocValues getNumericDocValues(String field) throws IOException {
  ensureOpen();
  Map,Object> vFields = docValuesLocal.get();

  Object previous dvFields.get(field);
if (previous != nul && previous instanceof NumericDocValues) {
    return (Numeri DocValues) previous;
  } else {
    FieldInfo fi = ge

</code></pre>

DVField(field, DocValuesType.NUMERIC);
 if (fi == null) {
      return null;
  }
    NumericDocValues dv = getDocValuesReader().getNumeric(fi);
  dvFields.put(field, dv);
 return dv;
  }
}
</code></pre>
可以看到最终是通过 DocValuesReader 来获取，并且这里用了一个 ThreadLocal 来缓存 DocValues，这里为什么是用 ThreadLocal 来缓存呢？如果并发线程多了，不就很占内存吗？而且，如果不用了线程池，则 ThreadLocal 就起不到缓存的作用了。其实，从 MemoryDocValuesProducer 进看一下，里面还会进行一次缓存的，不同线程引用的 DocValues 还是同一个。
<pre><code class="highlight-chroma">@Override
public synchronized NumericDocValues getNumeric(FieldInfo field) throws IOException {
  NumericDocValues instance = numericInstances.get(field.name);
 if (instance == null) {
    instance = loadNumeric(field);
 if (!merging) {
      numericInstances.put(field.name, instance);
  }
  }
  return instance;
}
</code></pre>
那 ThreadLocal 在这里是何用意呢？其实，前面也说了，DocValues 还有直接读取磁盘映射文的方式。ThreadLocal 在这里就是为了它实现的，基于磁盘读取，线程间共享是没有意义，而且基于盘的 DocValues 实例会记录读取的偏移量，这个是不能共享的，不同线程有不同的读取偏移量。所，这里用了 ThreadLocal 就很巧妙的兼容了内存和磁盘的 DocValues 实例。

通过 FieldCache 获取。

FieldCache 获取，会先看下 DocValues 映射文件是否存在，如果存在，则和 DocValues 工具读取的方式是一样的，如果不存在，则遍历 Term 构造 DocValues。
为什么不统一都从 FieldCache 来获取呢。貌似官方论坛是说 ES 默认是都生成了 DocValues 映文件的，不需要从 Term 进行构建，而 Solr 默认是没有生成 DocValues 映射文件的，因此就存在两方式了。