Introduction To Text Processing and Information Retrieval
Assignment 1

Ahmad Shayaan IMT2014004
H Vijaya Sharvani IMT2014022
Indu Ilanchezian IMT2014024

February 15, 2018

We discuss the main classes in Lucene related to the inverted index structure and the implementation of skip list datastructure to represent the postings list in Lucene.

Indexing in Lucene

The relevant and important classes which form the inverted index structure in Lucene are listed below.

1. **Document:** A document is the unit of indexing and searching in Lucene. It is composed of fields.

2. **Fields:** A field is a part of the document. Fields have a type and a name.

3. **FieldType:** FieldType class describes the properties of a field. The field type class provides high level API for the user to set whether the field type associated with a field should be storable, indexable etc. If the field type is storable, then the value of the field can be retrieved when a search hits a document. If the field is indexable, we can perform search on documents using the ”terms” in the value of this field. This class also provides API for setting whether to store term frequency vectors, term positions, term offsets etc.

4. **Terms:** A term is a word from the text. A term has two attributes: the text, which is a string and the field in which it occurred. Thus, if the same word occurs in two different ”indexable” fields, there will be two different indicies associated to each of the two terms. The term is the unit of search in Lucene. The search query phrase is compared with terms stored in the index.

5. **TermsEnum:** TermsEnum is an iterator which iterates through all terms stored in the index. During iteration, we can get the attributes of the term such as the total frequency of the term, the **docsEnum**, which is an iterator over the documents which contain the term.

6. **DocsEnum:** DocsEnum is a subclass of the DocSetIdIterator class. The freq() method of the DocsEnum also gets the frequency of the term in the current document.

The above list of classes show that Lucene uses an inverted index structure. A list of words in all the "indexable" fields form the index. This is the list of "terms". The total frequency of occurrence of each word is stored alongside the term associated with it. Each term points to a list of documents, which contain the term. The documents are iterated through in non-decreasing order of the document IDs. The frequenecy of the current term occurring in a given document is also stored if the term belongs to a field, for which the option to store term frequency vectors is set to True. The position and offset of a term in a document may also be stored.

Skip Lists in Lucene

Lucene source code has two classes which implement skip lists. These classes are MultiLevelSkipListReader and MultiLevelSkipListWriter. These classes are a part of the package org.apache.lucene.codecs. The most efficient datastructure for the PostingsEnum list is a skip list. A PostingEnum list is stored as a skip list in memory, so that it is easier to "skip to" a document with a given ID. We also know that the documents are stored in a non-decreasing order in the lucene index. A single level skip list implementation is costly for large postings list. For example, if a postings list has a million documents and the skip interval is 16, it still requires $(1000000/16) = 62500$ skips in the worst case. A multi-level skip list is more efficient and requires only a logarithmic number of skips to find a target document. Lucene implements a multi-level skip list. The skip lists are maintained in the term dictionary file. In Lucene, each skip entry in a level i for $i > 0$ has a pointer to the skip entry in the level immediately below it, i.e. level $i - 1$. This feature ensures that the target document is found in logarithmic amount of time.

Following are the key implementation details:

1. MultiLevelSkipListWriter class attributes

```
public abstract class MultiLevelSkipListWriter
{
/* Number of skip levels */
protected int numberOfSkipLevels;
/* Skip interval in the lowest level (level 0) */
private int skipInterval;
```

```
/* Skip interval for all levels other than level 0
   */
private int skipMultiplier;
/* A separate buffer is maintained for each skip
   level. Hence an array of skip buffers */
private RAMOutputStream[] skipBuffer;
}
```

2. MultiLevelSkipListWriter class constructor:

```
protected MultiLevelSkipListWriter(int skipInterval
    , int skipMultiplier, int maxSkipLevels, int df
    ) {
  this.skipInterval = skipInterval;
  this.skipMultiplier = skipMultiplier;

  if (df <= skipInterval) {
    numberOfSkipLevels = 1;
  } else {
    numberOfSkipLevels = 1+ MathUtil.log(df/
        skipInterval, skipMultiplier);
  }

  if (numberOfSkipLevels > maxSkipLevels) {
    numberOfSkipLevels = maxSkipLevels;
  }
}
```

In the above snippet, df is the document frequency for the current term. If the document frequency is less than skip interval, then there should be only one level otherwise the number of skip levels must be $\log_{skipMultiplier}(df/skipInterval)+1$. Note that the number of skip levels here, does not include the posting list level itself.

3. MultiLevelSkipListReader Atrributes: The purpose of each attribute is explained in the comments.

```
public abstract class MultiLevelSkipListReader
   implements Closeable {
/** the maximum number of skip levels possible for
    this index */
protected int maxNumberOfSkipLevels;
/** number of levels in this skip list */
private int numberOfSkipLevels;
private int numberOfLevelsToBuffer = 1;
private int docCount;
/** skipStream for each level. */
```

```
  private IndexInput[] skipStream;
  /** The start pointer of each skip level. */
  private long skipPointer[];
  /**  skipInterval of each level. */
  private int skipInterval[];
  /** Number of docs skipped per level.
  private int[] numSkipped;
  /** Doc id of current skip entry per level. */
  protected int[] skipDoc;
  /** Doc id of last read skip entry with docId &lt;=
      target. */
  private int lastDoc;
  /** Child pointer of current skip entry per level.
     */
  private long[] childPointer;
  /** childPointer of last read skip entry with docId
      &lt;=
   *  target. */
  private long lastChildPointer;
  private boolean inputIsBuffered;
  private final int skipMultiplier;
}
```

4. MultiLevelSkipListReader skipTo function:

```
public int skipTo(int target) throws IOException
{
  int level = 0;
  while (level < numberOfSkipLevels - 1 && target >
      skipDoc[level + 1]) {
    level++;
  }
  while (level >= 0) {
    if (target > skipDoc[level]) {
      if (!loadNextSkip(level)) {
        continue;
      }
    } else {
      if (level > 0 && lastChildPointer >
          skipStream[level - 1].getFilePointer()) {
        seekChild(level - 1);
      }
      level--;
    }
  }
  return numSkipped[0] - skipInterval[0] - 1;
```

```
}

private boolean loadNextSkip(int level) throws
    IOException {
  setLastSkipData(level);
  numSkipped[level] += skipInterval[level];
  if (Integer.compareUnsigned(numSkipped[level],
      docCount) > 0) {
    skipDoc[level] = Integer.MAX_VALUE;
    if (numberOfSkipLevels > level)
        numberOfSkipLevels = level;
    return false;
  }
  skipDoc[level] += readSkipData(level, skipStream[
      level]);
  if (level != 0) {
    // read the child pointer if we are not on the
        leaf level
    childPointer[level] = skipStream[level].
        readVLong() + skipPointer[level - 1];
  }
  return true;
 }

  protected void setLastSkipData(int level) {
  lastDoc = skipDoc[level];
  lastChildPointer = childPointer[level];
}
```

The skipTo function is the most important function for searching for a target document in a skip list. target is an integer denoting the document ID. First, we move up to the highest level which has is currently pointing to a document with an id lower than the target id.
We keep skipping documents in this level until the target id is less than the id of the document, which have currently skipped to. The loadNextSkip(level) first stores the last skip entry in the lastDoc attribute by calling setLastSkipData() method. It then skips a distance of skipInterval within the same level. If the skips that can be performed on the current level is exhausted, it returns false. Otherwise it skips to the next document, updates the skip entry in skipDoc[level] and the child pointer, and returns true.
If the target id value is greater than the current document id, then we move down one level and read the pointer linked by the skip list in this level. We repeat the loop till the target document id is reached. The skipTo function on completion would have updated the lastDoc attribute to the last read skip entry, which will point to the target document.