Apache Lucene API

Matthew Morgan, 17 May 2018

About this Document

Description

This document explores, in-depth, Apache Lucene version 7.3.0, and the contents of its API. It delves into the different, main packages of Lucene, searching for the main classes that drive the full-text search engine implementation. Other features that are held aside from the core API, such as query parsers, encoders, and analyzers, are also investigated and reported on in some detail; however, features identified as being too expert for a student's understanding are omitted, but overviewed such that the existence of those features is not denied.

Document Goals

- Assist in relating Apache Lucene to the different components, and structure, of Information Retrieval systems, inclusive of, but not limited to: tokenization, indexing, similarity and models, etc
- Provide further information such that a basic understanding of the Apache Lucene packages/classes can be reached, as well as provide insight into intermediate features
- Provide abstraction of core classes in the form of charts, or provide a basis to create these charts
- Provide a basic explanation and outline of the steps required to implement Lucene, inclusive of, but not limited to: basic implementation of indexing; Lucene query syntax and query parsing; and similarity switching

Document Formatting

Formatting works as follows:

- Formatting
  - *Red text* refers to notes regarding other features of Lucene that aren't covered by this documentation, or denotes experimental APIs that may change in future releases
  - Purple text is used to provide emphasis on particular notes or sections of function tables
  - Green text is used to reference packages **aside from** Lucene's core API
  - **Bold text** provides emphasis for different class names
  - Underlined text represents a function name
- Some classes are children of others, and may, in some cases, create a large hierarchy. Effort will be made to provide thorough function tables for the hierarchy.

API Implementation

General Procedure

```
Analyzer analyzer = new StandardAnalyzer();

// Store the index in memory:
Directory directory = new RAMDirectory();
// To store an index on disk, use this instead:
// Directory directory = FSDirectory.open("/tmp/testindex");

IndexWriterConfig config = new IndexWriterConfig(analyzer);
IndexWriter iwriter = new IndexWriter(directory, config);
Document doc = new Document();
String text = "This is the text to be indexed.";
doc.add(new Field("fieldname", text, TextField.TYPE_STORED));
iwriter.addDocument(doc);
iwriter.close();

// Now search the index:
DirectoryReader ireader = DirectoryReader.open(directory);
IndexSearcher isearcher = new IndexSearcher(ireader);

// Parse a simple query that searches for "text":
QueryParser parser = new QueryParser("fieldname", analyzer);
Query query = parser.parse("text");
ScoreDoc[] hits = isearcher.search(query, null, 1000).scoreDocs;

// Iterate through the results:
for (int i = 0; i < hits.length; i++) {
  Document hitDoc = isearcher.doc(hits[i].doc);
}

ireader.close();
directory.close();
```

Implementation Outline

[1] Index Construction:

- A. Instantiate an **Analyzer** and **Directory**.
  - A1. Sample analyzers are as follows: **StandardAnalyzer**, **WhitespaceAnalyzer**, and **SimpleAnalyzer.**
  - A2. For directories, either use **RAMDirectory** (small corpa only), or **FSDirectory**.open(...).
- B. Instantiate an **IndexWriterConfig** using the Analyzer, and then instantiate an **IndexWriter**.
- C. Instantiate new **Document**s, and append **Field**s to them. After doing so, add them to the IndexWriter.
- D. Close the IndexWriter after documents have been appended. *(This commits changes to the index.)*

[2] Index Searching:

- A. Create a **DirectoryReader** by using the function **DirectoryReader**.open(...) with the Directory from before. Use this reader to then instantiate an **IndexSearcher**.
- B. Generate **Query** objects for IndexSearcher.search(...) via 1 of 2 methods:
  - B1. Generate the Query objects manually using the Query API.
  - B2. Generate Query objects via instantiating a **QueryParser**, and passing the result of parse(...).
- C. Store the result of the search(...) function - usually the scoreDocs property of type **ScoreDoc[]**.

[3] Cleanup:

- A. Close the DirectoryReader.
- B. Close the Directory.

Notes about Implementation

[1] **IndexWriter**, **IndexReader**, **DirectoryReader**, and **Directory** all throw **IOException**s – these must be handled.

[2] As discussed below, **QueryParser**, or extensions of its parent class, will throw **ParseException**s.

[3] In all cases, using the base, abstract classes for variable declaration is encouraged, such as using the type **Directory** instead of **FSDirectory** or **RAMDirectory**.

Similarity Switching

Similarity switching changes how Lucene scores documents against generated queries, with a variety of similarity implementations already available, such as TF-IDF, BM25, and other language models. For simplicity, the general procedure from above will be described as comments as to provide emphasis on the steps necessary to change similarity.

```
/* Instantiate an Analyzer (analyzer) and Directory (directory). */

// Instantiate the similarity that you want to utilize
Similarity sim = new ClassicSimilarity();

// After instantiating the IndexWriter’s configurator, set the sim
IndexWriterConfig config = new IndexWriterConfig(analyzer);
config.setSimilarity(sim);

/* Instantiate an IndexWriter (iwriter) using the configuration, and
 * then append all the documents, and their respective fields, to
 * the writer for indexing. Close the writer, then instantiate a
 * DirectoryReader (ireader). */

// After instantiating the IndexSearcher, set the similarity
IndexSearcher isearcher = new IndexSearcher(ireader);
isearcher.setSimilarity(sim);

/* Instantiate a query parser, and use it to generate queries using
 * parse(…) that are then passed to search(…) to generate TopDocs
 * results. Once done, close the DirectoryReader, and Directory. */
```

Implementation Outline

[1] Instantiate the desired **Similarity** implementation that should be used for scoring.

A. Samples of similarities are: **ClassicSimilarity**, **BM25Similarity**, and **BooleanSimilarity.**

[2] Pass the similarity instance to the setSimilarity(Similarity) function of **IndexWriterConfig.**

[3] Pass the similarity instance to the setSimilarity(Similarity) function of **IndexSearcher.**

Notes about Implementation

[1] The similarity instance must be passed to the **IndexWriterConfig** instance **BEFORE** instantiating the **IndexWriter**.

[2] If an index was previously generated, the similarity that was used to generate it must be used for searching.

Field Customization

Field customization allows incredible control over the way corpus information is stored in the index. Typically, information is indexed via the classes **IntPoint, LongPoint, TextField,** and **StringField**; however, there are some cases in which storing information, such as term vectors, would be useful. For this, the overarching **Field** class can be used with **FieldType** to define the field’s attributes.

```
/* Instantiate an Analyzer (analyzer), Directory (directory), and
 * IndexWriter (iwriter). Also generate a Document (doc). */

// Create the type. Set its properties using the various functions
FieldType fType = new FieldType();
fType.setStoreTermVectors(true);
fTtype.setStored(true);

// Add a new field to the document, specifying the type as a parameter
doc.add(new Field("fname", "txt to index", fType));

/* Perform all other actions here, such as instantiating a reader
 * and query parsing. */
```

Implementation Outline

[1] Instantiate a **FieldType** class, and change what information is stored in the index using its functions.

[2] Instantiate a **Field** with the **FieldType** as a constructor parameter.

[3] Add the customized **Field** to a **Document** using the add(…) function.

Query Parsing

The following is a simple example of how to parse a query. It's taken directly from the above.

```
/* Instantiate an Analyzer (analyzer) and perform other processes
 * prior to query parsing, such as IndexSearcher (isearch)
 * instantiation. */

QueryParser parser = new QueryParser("fieldname", analyzer);
Query query = parser.parse("text");

/* After generating the query, use isearch.search(…).scoreDocs to
 * retrieve results for the parsed query. */
```

Implementation Outline

[1] Instantiate the query parser of your selection using the analyzer of choice.
  A. Sample parsers are the **QueryParser** or **SimpleQueryParser.**
[2] Use the query parsers parse(…) function to receive a **Query** object.
[3] Pass the Query object into **IndexSearcher**'s search(…) function, from which you'll receive a **TopDocs** object.

Notes about Implementation

- Any query parser that extends the **BaseQueryParser** class may throw **ParseException**s when using parse(…).
- The syntax of a query is heavily contingent on the parser chosen for implementation.

API Exploration

Core

Codecs

The codecs package provides an abstraction of the encoding and decoding of index structure. Different implementations are available based on program necessities. *(Due to the purpose of codecs and their implementation being expert-level, no classes were explored for this package.)*

All custom codecs extend the class **Codec**, with the different formats for index information extending their respective classes. *(Different formats that don't require modification can, simply, use the default.)* The pieces of index information, and their respective classes, are in the table below.

Some example codecs are:

- **org.apache.lucene.codecs.simpletext:** A simple, plain text encoded format
- **org.apache.lucene.codecs.lucene70:** Format used by Lucene 7.0
- **org.apache.lucene.codecs.lucene50:** Format used by Lucene 5.0
- **org.apache.lucene.codecs.compressing:** StoredFieldsFormat which allows cross-document and cross-field compression of stored fields

Info Segment	Default Class	Description
Postings lists	**PostingsFormat**	*Encodes and decodes terms, postings, and proximity data*
DocValues	**DocValuesFormat**	*Encodes and decodes per-document values*
Stored fields	**StoredFieldsFormat**	*Controls the format of stored document fields*
Term vectors	**TermVectorsFormat**	*Controls the format of term vectors*
Points	**PointsFormat**	*Encodes and decodes indexed points*
FieldInfos	**FieldInfosFormat**	*Encodes and decodes **FieldInfos**, a collection of **FieldInfo** instances. (The Field Info file describes document fields and whether/not they're stored.)*
SegmentInfo	**SegmentInfoFormat**	*Expert: Controls the format of the **SegmentInfo** (segment metadata file)*
Norms	**NormsFormat**	*Encodes and decodes per-document score normalization values*
Live documents	**LiveDocsFormat**	*Format for live, or deleted, documents*

Analysis

The analysis package defines an API for **Analyzer**s, allowing conversion of text from a **Reader** to a **TokenStream** – an enumeration of token **Attributes**. The stream is composed by applying **TokenFilters** to the output of a **Tokenizer.** The filters and tokenizers are applied with an analyzer.

The package analyzers-common provides a number of Analyzer implementations, such as the following:

- **WhitespaceAnalyzer**, which utilizes a **WhitespaceTokenizer** to divide text at whitespace characters
- **SimpleAnalyzer**, which utilizes a **LetterTokenizer** and **LowerCaseFilter** to split text at numerical characters
- **StopAnalyzer**, which is similar to the SimpleAnalyzer, except it also applies a **StopFilter**

Analyzer, CharFilter, Tokenizer, and TokenFilter

*(Implementation of custom **Analyzer**, **CharFilter**, **Tokenizer**, and **TokenFilter** classes is expert-level; therefore, no function tables are available for these classes (exempt Analyzer). To understand the process of analysis, basic descriptions for the classes are provided.)*

Analyzer: Supplies a **TokenStream** which can be consumed by indexing and searching processes. Most applications can use the **StandardAnalyzer** or other analyzers supplied with Lucene. *(Analyzers use CharFilters, Tokenizers, and TokenFilters to process the text.)*

Function Name	Description
`void close()`	*Frees persistent resources used by this analyzer*
`int getOffsetGap(String fieldname)`	*Just like the below function, except for Token offsets*
`int getPositionIncrementGap(String fieldname)`	*Invoked before indexing a IndexableFIeld instance, if terms have already been added to that field*
`BytesRef normalize(String fieldname, String text)`	*Normalize string down to the representation it'd have in-index*
`TokenStream tokenStream(String fieldname, Reader reader)`	*Returns a TokenStream suitable for the field, tokenizing the contents of the reader*
`TokenStream tokenStream(String fieldname, String text)`	*Returns a TokenStream suitable for the field, tokenizing the contents of the text*

CharFilter: Extends a **Reader** to transform text before being tokenized. **Tokenizer**.setReader(Reader) accepts these.

Tokenizer: *(A Tokenizer is a **TokenStream**.)* It breaks up incoming text into tokens. *(An **Analyzer** will use these as the first step in the analysis process.)*

TokenFilter: *(A TokenFilter is a **TokenStream**.)* It modifies tokens created by **Tokenizers.** Not all **Analyzer**s will require these. Common modifications include deletion or synonym injection and stemming or case folding.

StandardAnalyzer

An **Analyzer** provided by Lucene for analysis. This analyzer uses three filters: **StandardFilter**, **LowerCaseFilter**, and **StopFilter** to filter the tokenizer **StandardTokenizer**. The following notes apply:

- **StandardAnalyzer** extends **StopwordAnalyzerBase**, which extends **Analyzer**.
- **StopwordAnalyzerBase** is a base class for Analyzers that need to make use of stopword sets.

StandardAnalyzer	
Function Name	**Description**
`StandardAnalyzer()`	*Builds an analyzer with the default stop words*
`StandardAnalyzer(CharArraySet stopWords)`	*Creates an analyzer with the given stop words*
`StandardAnalyzer(Reader stopwords)`	*Builds an analyzer with the stop words from the given reader*
`int getMaxTokenLength()`	*Returns the current maximum token length*
`void setMaxTokenLength(int length)`	*Set the max allowed token length*

StopwordAnalyzerBase	
Function Name	**Description**
`CharArraySet getStopwordSet()`	*Returns the analyzer's stopword set, or an empty set if n/a*

TokenStream

TokenStream is an abstract class, with concrete subclasses being **Tokenizer** and **TokenFilter**. It provides access to all of the token **Attribute**s. The following notes apply:

- The consumer of the TokenStream must call reset() before using the stream.
- The consumer should call incrementToken() until it returns false, consuming attributes after each call.
- The consumer must call close() to release resources when finished with the stream.

TokenStream	
Function Name	**Description**
`void close()`	*Releases resources associated with this stream*
`void end()`	*Called by the consumer after the last token is consumed, after incrementToken() returns false*
`abstract boolean incrementToken()`	*Consumers (such as **IndexWriter**) use this method to advance the stream to the next token*
`void reset()`	*Called by the consumer before consuming tokens using incrementToken()*

AttributeSource	
Function Name	**Description**
`<T extends` **`Attribute`**`> T getAttribute(Class<T> attClass)`	*Returns the instance of the passed-in Attribute in this source*

CharArraySet

Stores Strings as char[] in a hashtable. It **cannot** remove items from the set, nor does it resize its hashtable. *(The class implements **Set** but doesn't behave like it should in all cases.*

Function Name	Description
`CharArraySet(Collection<?> c, boolean ignoreCase)`	*Creates a set from a collection of objects*
`CharArraySet(int startSize, boolean ignoreCase)`	*Creates a set with enough capacity to hold 'startSize' items*
`boolean add(char[] text)`	*Add the char[] to the set*
`boolean add(String text)`	*Add this String to the set*
`void clear()`	*Clears all entries in the set*
`boolean contains(char[] text, int off, int len)`	*True if the len characters of text, starting at off, are in the set*
`boolean contains(Object o)`	
`static CharArraySet copy(Set<?> set)`	*Returns a copy of the given set as a CharArraySet*
`int size()`	
`static CharArraySet unmodifiableSet(CharArraySet set)`	*Returns an unmodifiable copy of the set*

Document

The document package provides a simple, **Document** class, which is a set of named **Fields**, whose values may be strings or instances of the **Reader** class. *(Fields implement the interface **IndexableField**.)* The user application is responsible for creating Documents based on the files being worked with.

Document

A collection of **IndexableFields** – logical representations of user content that needs to be indexed/stored. The fields have a number of properties that tell Lucene how to treat the content (indexed, tokenized, stored, etc). *(A document may be referred to as having fields, even though they're indexable fields.)* The following applies:

- Each document should, typically, contain one or more uniquely-identifying fields.
- Fields not stored aren't available in documents retrieved from the index, e.g. with **IndexReader**.document(int).

Document	
Function Name	**Description**
`Document()`	*Constructs a new document with no fields*
`void add(IndexableField field)`	*Adds a field to a document*
`void clear()`	*Removes all fields from the document*
`String get(String name)`	*Returns the string value of the field with the given name, or null*
`IndexableField getField(String name)`	*Returns a field with the given name, or null*
`List<IndexableField> getFields()`	*Returns a List of all the fields in a document*
`IndexableField[] getFields(String name)`	*Returns an array of fields with the given name*
`String[] getValues(String name)`	*Returns an array of values of the field specified*
`void removeField(String name)`	*Removes a field with the name from the document*
`void removeFields(String name)`	*Removes all fields with the given name from the document*
`Iterator<IndexableField> iterator()`	
`String toString()`	*Prints the fields of a document for human consumption*

Field

Section of a **Document** that contains a value, name, and type. *(There are four fields for non-casual users not explored: **IntRange**, **LongRange**, **DoubleRange**, and **FloatRange**.)*

Field	
Function Name	**Description**
`Field(String name, byte[] value, IndexableFieldType type)`	*Create a field with a binary value*
`Field(String name, BytesRef bytes, IndexableFieldType type)`	*Create a field with a binary value*
`Field(String name, String value, IndexableFieldType type)`	*Create a field with a String value*
`String toString()`	*Prints a field for human consumption*
`TokenStream tokenStreamValue()`	*The TokenStream used for indexing, or null*
`void set`**`T`**`Value(`**`T`**` value)`	*(Expert) Change the value of this field. Acceptable types are as follows: byte[] or BytesRef **(Bytes)**, byte **(Byte)**, double, float, int, long, Reader, short, String*

IndexableField	
Function Name	**Description**
`BytesRef binaryValue()`	*Non-null if this field has a binary value*
`IndexableFieldType fieldType()`	*Returns the FieldType for this field*
`String name()`	*Field name*
`Number numericValue()`	*Non-null if this field has a numeric value*
`Reader readerValue()`	*Value of the field as a Reader, or null*
`String stringValue()`	*Value of the field as a String, or null*
`TokenStream tokenStream(Analyzer an, TokenStream reuse)`	*Creates the TokenStream used for indexing this field*

FieldType

FieldType describes the properties of a **Field** being appended to a document. **IndexableFieldType** is an interface that FieldType implements to allow the description of these properties. Values for **IndexOptions** below are as follows:

- *IndexOptions.NONE* *IndexOptions.DOCS* *IndexOptions.DOCS_AND_FREQS*
- *IndexOptions.DOCS_AND_FREQS_AND_POSITIONS* *IndexOptions.DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS*

FieldType	
Function Name	**Description**
`FieldType()`	*Creates a new FieldType with default properties*
`FieldType(IndexableFieldType ref)`	*Creates a new, mutable FieldType with ref's properties*
`void freeze()`	*Prevent future changes*
`void setDimensions(int dimCnt, int dimNumBytes)`	*Enables point indexing*
`void setDocValuesType(DocValuesType type)`	*Sets the DocValuesType*
`void setIndexOptions(IndexOptions value)`	*Sets the indexing options for the field*
`void setOmitNorms(boolean value)`	*Set to true to omit normalization values for the field*
`void setStored(boolean value)`	*Set to true to store this field*
`void setStoreTermVectorOffsets(boolean value)`	*Set to true to also store token character offsets into the term vector*
`void setStoreTermVectorPositions(boolean value)`	*Set to true to store token positions into the term vector*
`void setStoreTermVectors(boolean value)`	*Set to true if the field's indexed form will store into term vectors*
`void setTokenized(boolean value)`	*Set to true to tokenize this field's contents via the Analyzer*
`String toString()`	*Prints a field for human consumption*

IndexableFieldType	
Function Name	**Description**
`DocValuesType docValuesType()`	*How the field's value will be indexed into docValues*
`IndexOptions indexOptions()`	*Describes what should be recorded into the inverted index*
`boolean omitNorms()`	*True if normalization values should be omitted*
`int pointDimensionCount()`	*If this is positive, the field is indexed as a point*
`int pointNumBytes()`	*The number of bytes in each dimension's values*
`boolean stored()`	
`boolean storeTermVectorOffsets()`	
`boolean storeTermVectorPositions()`	
`boolean storeTermVectors()`	
`boolean tokenized()`	

TextField

A field that's indexed and tokenized without term vectors. Use this for, example, a document's full text in a 'body' field. The acceptable values of *Field.Store* are *Field.Store.YES* and *Field.Store.NO*.

Function Name	Description
`TextField(String name, Reader reader)`	*Creates a new un-stored TextField with Reader value*
`TextField(String name, String value, Field.Store stored)`	*Creates a new TextField with String value*
`TextField(String name, TokenStream stream)`	*Creates a new, un-stored TextField with TokenStream value*

StringField

A field that's indexed but not tokenized: the entire String value is indexed as a single token. This might be used for 'country' or 'id' fields. *(If this field requires sorting, separately add a **SortedDocValuesField** to the document.)*

Function Name	Description
`StringField(String name, BytesRef val, Field.Store stored)`	*Creates a StringField, indexing the binary as a single token*
`StringField(String name, String val, Field.Store stored)`	*Creates a StringField, indexing the String as a single token*

StoredField

A field whose value is stored such that **IndexSearcher**.doc(int) and **IndexReader**.document() will return the field/value. In the constructor, '**T**' is any of the following types: **BytesRef**, double, float, int, long, byte[], or String.

Function Name	Description
`StoredField(String name, byte[] value, int offset, int length)`	*Stored-only field with the given binary value*
`StoredField(String name, T value)`	*Stored-only field*

IntPoint, LongPoint, FloatPoint, and DoublePoint

These four classes provide the ability to index integers, longs, floats, and doubles. All of them have a single constructor function, as well as the capability to construct queries for common searches. These classes are indexed for exact/range queries. *(These classes allow N-dimensional storage of values if a single value isn't enough.)* For the purposes of writing only one function set, the keyword '**T**' will be used to refer to **int**, **long**, **float**, or **double** respectively. If the values of these fields need to be stored, a separate **StoredField** instance should be added.

Function Name	Description
`TPoint(String name, T… point)`	*Creates a new point, indexing the N-dimensional T point* (ex:, IntPoint(name, point) or FloatPoint(name, point))
`static T decodeDimension(byte[] value, int offset)`	*Decode single T dimension*
`static void encodeDimension(T val, byte[] dest, int offset)`	*Encode a single T dimension*
`static Query newExactQuery(String field, T value)`	*Create a query for matching exact T value*
`static Query newRangeQuery(String field, T[] low, T[] high)`	*Create a range query for n-dimensional T values*
`static Query newRangeQuery(String field, T low, T high)`	*Create a range query for float values*
`static Query newSetQuery(String field, Collection<T> vals)`	*Create a query matching any of the 1D values*
`static Query newSetQuery(String field, T… values)`	*Create a query matching any of the 1D values*
`static T nextDown(T val)`	*Return the greatest T that compares less than val consistently with **T**.compare(T, T)*
`static T nextUp(T val)`	*Same as nextDown(), except T must compare > val*
`Number numericValue()`	*Non-null if this field has a numeric value*
`void setTValues(T… point)`	*Change the values of this field* (ex: setFloatValues(…))
`String toString()`	*Prints a Field for human consumption*

Index

The index package provides functionality for both writing an index to files on the hard disk, and accessing index info.

IndexWriter

Makes and maintains an index based on a user's document collection. *(IndexWriter allows policies for deletion – **IndexDeletionPolicy** - and merging – **MergePolicy** and **MergeScheduler** - but these policies aren't covered here.)*

The following notes apply to the below:

- Anywhere '***Iterable***<>' is written below, it stands for '***Iterable***<*? extends* ***IndexableField***>'.
- Any of the functions that modify the index return a long sequence number, expressing the effective order in which each change was applied. *(In the case of commit, the sequence number contains which updates are included/not.)*
- close() should be called after all additions, deletions, and updates are made to the documents. Either it or commit() should be called so a reader can see updates to the index.
- Only one IndexWriter may be open at a time, as opening one creates a lock file for the directory in use.

Function Name	Description
`IndexWriter(Directory d, IndexWriterConfig conf)`	*Constructs a new IndexWriter per the settings given in conf*
`long addDocument(Iterable<> doc)`	*Adds a document to this index*
`long addDocuments(Iterable<Iterable<>> docs)`	*Adds a block of documents with sequentially-assigned IDs*
`void close()`	*Closes all open resources and releases the write lock*
`long commit()`	*Commits all pending changes to the index, and syncs files so a reader can see the changes and the updates survive OS crashes. May also return -1 if no changes are pending*
`long deleteDocuments(Term… terms)`	*Deletes document(s) containing any of the terms*
`long deleteDocuments(Query… queries)`	*Deletes document(s) matching any of the provided queries*
`long deleteAll()`	*Deletes all documents in the index*
`LiveIndexWriterConfig getConfig()`	*Returns a LiveIndexWriterConfig for the IndexWriter, which can be used to query the current settings and modify 'live' ones*
`long getMaxCompletedSequenceNumber()`	*Returns the highest sequence number across all completed ops, or 0 if no operations have finished yet*
`Analyzer getAnalyzer()`	*Returns the analyzer used by this index*
`Directory getDirectory()`	*Returns the Directory used by this index*
`Set<String> getFieldNames()`	*Returns an unmodifiable set of all field names as visible from this IndexWriter, across all segments of the index*
`boolean hasDeletions()`	*Returns true if this index has deletions (including buffered del.)*
`boolean hasUncommittedChanges()`	*Returns if there may be uncommitted changes*
`boolean isOpen()`	*Returns if the IndexWriter is still open*
`int maxDoc()`	*Returns the total number of documents in the index, including not yet flushed (still in the RAM buffer) docs, not counting deletions*
`int numDocs()`	*Returns the total number of docs in the index, including docs not yet flushed (still in RAM), including deletions*
`long updateDocument(Term term, Iterable<> doc)`	*Updates a document (first deletes doc(s) containing the term, and then re-adds the new document)*
`void rollback()`	*Closes the IndexWriter without committing changes since the last commit (or since it was opened, if no commits made)*

IndexWriterConfig and LiveIndexWriterConfig

IndexWriterConfig: Holds configuration information for usage in creating an **IndexWriter**. Once the writer has been created, changes to the config won't affect the writer. *(Such desired modifications can be done via using a **LiveIndexWriterConfig** returned from the writer's getConfig() function.)* The following notes apply: *(Functions are available for specifying when to flush modifications, but not listed.)*

- Setter methods return the object to allow chaining of setting configurations. No return type is shown for them
- The parameter for setOpenMode() accepts any of these values:
  - *OpenMode.APPEND* Opens an existing index
  - *OpenMode.CREATE* Creates a new index, or overwrites an existing one
  - *OpenMode.CREATE_OR_APPEND* Creates a new index if it doesn't exist, or opens and appends

LiveIndexWriterConfig: The LiveIndexWriterConfig object uses many of the same functions as **IndexWriterConfig**. These functions are denoted with purple text in the function table for the **IndexWriterConfig** class.

***IndexWriterConfig* (*LiveIndexWriterConfig* in Purple Text)**	
Function Name	**Description**
`IndexWriterConfig()`	*Creates a new config using a StandardAnalyzer*
`IndexWriterConfig(Analyzer analyzer)`	*Creates a new config using the provided Analyzer*
`Analyzer getAnalyzer()`	*Returns the default analyzer for indexing documents*
`Similarity getSimilarity()`	*Expert: Returns the Similarity implementation used by the writer*
`IndexWriterConfig.OpenMode getOpenMode()`	*Returns the OpenMode set by setOpenMode(OpenMode)*
`setCommitOnClose(boolean commitOnClose)`	*Sets if calls **IndexWriter**.close() should first commit before closing*
`setOpenMode(IndexWriterConfig.OpenMode openMode)`	*Specifies the OpenMode of the index*
`setSimilarity(Similarity similarity)`	*Expert: Sets the similarity implementation used by the writer*

StandardDirectoryReader

A child of **IndexReader.** The following notes apply:

- ***IndexReader** instances are completely thread safe, meaning multiple threads can call any of its methods concurrently. If the application requires external synchronization, do **not** synchronize on the instance; use non-Lucene objects instead.*
- The constructor of StandardDirectoryReader is protected, and thus not available. To get a **DirectoryReader**, you'll have to use one of the 'open' functions.

StandardDirectoryReader	
Function Name	**Description**
`boolean isCurrent()`	*Check whether any changes have occurred to the index since the reader was opened*

DirectoryReader	
Function Name	**Description**
`Directory directory()`	*Returns the directory this index resides in*
`static boolean indexExists(Directory directory)`	*Returns if there is likely an index in the directory*
`static List<IndexCommit> listCommits(Directory dir)`	*Returns all commit points that exist in the Directory*
`static DirectoryReader open(Directory directory)`	*Returns an IndexReader reading the index in the given directory*
`static DirectoryReader open(IndexWriter writer)`	*Open a near real time IndexReader from the IndexWriter*
`static DirectoryReader openIfChanged(DirectoryReader oldReader)`	*If the index has changed, opens a new reader and returns it; returns null otherwise*
`static DirectoryReader openIfChanged(DirectoryReader oldReader, IndexCommit commit)`	*If the IndexCommit differs from what the provided reader is searching, open and return a new reader; otherwise, returns null*

BaseCompositeReader	
Function Name	**Description**
`int docFreq(Term term)`	*Returns the number of documents containing the term*
`int getDocCount(String field)`	*Returns the number of documents that have as least one term for this field, or -1 if this measure isn't stored by the codec*
`long getSumDocFreq(String field)`	*Returns the sum of **TermsEnum**.docFreq() for all terms in this field, or -1 if this measure isn't stored by the codec*
`long getSumTotalTerm(String field)`	*Returns the sum of **TermsEnum**.totalTermFreq() for all terms in this field, or -1 if the measure isn't stored by the codec (or if this fields omits term freq and positions)*
`Fields getTermVectors(int docID)`	*Retrieve term vectors for this document, or null if not indexed*
`int maxDoc()`	*numDocs() + 1*
`int numDocs()`	*Returns the number of documents in this index*
`long totalTermFreq(Term term)`	*Returns the total number of occurrence of the term across all documents (the sum of the freq() for each doc that has the term)*

IndexReader	
Function Name	**Description**
`void close()`	*Closes files associated with this index*
`Document document(int docID)`	*Returns the stored fields of the n-th document of the index*
`Document document(int docID, Set<String> fields)`	*Like document(int), but loads only the specified fields*
`Terms getTermVector(int docID, String field)`	*Retrieve term vector for this document and field, or null if term vectors were not indexed*
`boolean hasDeletions()`	*Returns if any documents have been deleted*
`int numDeletedDocs()`	*Returns the number of deleted documents*

TermsEnum, and PostingsEnum

TermsEnum: Iterator to seek *(using seekCeil or seekExact)* or step through *(**BytesRefIterator**.next())* terms to obtain frequency information or **PostingsEnum** for the current term. The following applies:

- Enumerations are always ordered by **BytesRef**.compareTo(…), where each term is greater than the one before it.
- TermsEnum is unpositioned when first obtained. You must first successfully call **BytesRefIterator**.next() or one of the seek methods.
- When getting the **PostingsEnum** using the postings(…) functions, deleted documents may be in the returned iterator. These must be checked on top of the enumerator.
- *The API for TermsEnum is experimental, and may change incompatibly for the next release!*

TermsEnum	
Function Name	**Description**
`abstract int docFreq()`	*Returns the number of documents containing the term*
`abstract long ord()`	*Returns the ordinal position of the current term*
`PostingsEnum postings(PostingsEnum reuse)`	*Gets PostingsEnum for the current term (use for docs + freqs)*
`abstract PostingsEnum postings(PostingsEnum reuse, int flags)`	*Gets PostingsEnum for the current term, with control over whether freqs, positions, offsets are required. (Don't call when the enum is unpositioned)*
`abstract TermsEnum.SeekStatus seekCeil(BytesRef text)`	*Seeks to the specified term, if it exists, or the next (ceil) term*
`boolean seekExact(BytesRef text)`	*Attempts to seek to the exact term. Returns true if the term's found*
`abstract void seekExact(long ord)`	*Seeks to the specified term by ordinal (position) as previously returned by ord()*
`abstract BytesRef term()`	*Returns the current term*
`abstract long totalTermFreq()`	*Returns the ttl number of occurences of the term across all documents (the sum of freq() for each document that has the term)*
`BytesRef next()`	*Returns the next term in the enumerator, or null if the end is reached*

The following are values passable for the *flags* parameter of the postings(…) function above:

▪ *PostingsEnum.ALL*	All of the below information
▪ *PostingsEnum.FREQS*	Term frequencies
▪ *PostingsEnum.NONE*	No requirement for per-document postings
▪ *PostingsEnum.OFFSETS*	Offsets
▪ *PostingsEnum.POSITIONS*	Term positions

The following are values returned by the seekCeil function:

TermsEnum.SeekStatus.END	The term wasn't found, and the end of iteration was hit
TermsEnum.SeekStatus.FOUND	The precise term was found
TermsEnum.SeekStatus.NOT_FOUND	A different term was found after the requested term

PostingsEnum (extends **DocIdSetIterator**)**:** Iterates through postings. *(The function **DocIdSetIterator.**nextDoc() must be called before using any of the per-doc methods below.)* Term vectors must have been stored for accessing the frequency of documents to work appropriately.

PostingsEnum	
Function Name	**Description**
`abstract int endOffset()`	*Returns end offset for the current position, or -1 if not indexed*
`static boolean featureRequested(int flags, short feature)`	*Returns true if the given feature is requested in the flags*
`abstract int freq()`	*Returns term frequency in the current document, or -1 if the field was indexed with IndexOptions.DOCS*
`abstract int nextPosition()`	*Returns the next position, or -1 if not indexed*
`abstract int startOffset()`	*Returns start offset for current position, or -1 if not indexed*

DocIdSetIterator	
Function Name	**Description**
`abstract int nextDoc()`	*Advances to the next doc and returns the current, or NO_MORE_DOCS if no more documents exist*
`abstract int advance(int target)`	*Advances to the first beyond the current whose document number is greater than or equal to the target, and returns the document number itself*
`abstract int docID()`	*Returns -1 if nextDoc and advance weren't called yet, NO_MORE_DOCS if the iterator has exhausted, or the doc ID currently viewed otherwise*

Terms and FieldsProducer

*(The APIs for **Terms, FieldsProducer**, and **Fields** are all experimental!)* Access to a **Fields** instance is possible through the **BaseCompositeReader**'s function getTermVectors(…). Access to a **Terms** instance is possible through the **IndexReader**'s function getTermVector(…).

FieldsProducer: Abstract API that produces terms, doc, freq, prox, offset, and payloads postings. (**Fields:** Provides a **Terms** index for fields that have it, and lists which fields do. Fields is primarily *internal/experimental API*, although also used to expose the set of term vectors per document.)

FieldsProducer	
Function Name	**Description**
`abstract void close()`	*…*
`FieldsProducer getMergeInstance()`	*Returns an instance optimized for merging*

Fields	
Function Name	**Description**
`abstract Iterator<String> iterator()`	*Returns an iterator that'll step through all fields names*
`abstract int size()`	*Returns the number of fields, or -1 if the number of distinct field names is unknown*
`abstract Terms terms(String field)`	*Gets the Terms for this field*

Terms: Access to the terms in a specific field.

Terms	
Function Name	**Description**
`abstract int getDocCount()`	*Returns the number of documents that have at least one term for this field, or -1 if the measure isn't stored by the codec*
`BytesRef getMax()`	*Returns the largest term (in lexicographic order) in the field*
`BytesRef getMin()`	*Returns the smallest term (in lexicographic order) in the field*
`abstract long getSumDocFreq()`	*Returns the sum of TermsEnum.docFreq() for all terms in this field, or -1 if the measure isn't stored by the codec*
`abstract long getSumTotalTermFreq()`	*Returns the sum of TermsEnum.totalTermFreq() for all terms in this field, or -1 if this measure isn't stored (or if the field omits term freq and positions)*
`abstract boolean hasFreqs()`	*Returns true if documents in this field store per-doc term frequency*
`abstract boolean hasOffsets()`	
`abstract boolean hasPositions()`	*Returns true if documents in the field store positions*
`abstract TermsEnum iterator()`	*Returns an iterator that'll step through all terms*
`abstract long size()`	*Returns number of terms in the field, or -1 if the it isn't stored by the codec*

Term

Represents a word from text. This is the unit of search, composed of two elements: The text of the word, as a string, and the name of the field that the text occurs in. Terms may represent more than words from text fields (dates, e-mail, etc.).

Function Name	**Description**
`Term(String fld)`	*Constructs a Term with the given field, and empty text*
`Term(String fld, BytesRef bytes)`	*Constructs a Term with the given field and bytes*
`Term(String fld, String text)`	*Constructs a Term with the given field and text*
`BytesRef bytes()`	*Returns the bytes of this term. These shouldn't be modified*
`int compareTo(Term other)`	*Compares two terms, returning negative if this term belongs before, positive if it belongs after, or zero if they are equal*
`String field()`	*Returns the field of this term*
`String text()`	*Returns the text of this term*
`String toString()`	
`static String toString(BytesRef termText)`	*Returns human-readable form of the term text*

Store

The store package provides an abstract class for storing data – **Directory** – which is a collection of files written by **IndexOutput**, and read by **IndexInput**. *(Numerous implementations are provided, such as **FSDirectory**, which uses a file system directory to store files, and **RAMDirectory** for implementing files as memory-resident data structures.)*

Directory, RAMDirectory, and FSDirectory

Directory: Abstract base for Directory implementations. These functions are available in **FSDirectory** and **RAMDirectory**.

Directory	
Function Name	**Description**
`void close()`	*Closes the store*
`void copyFrom(Directory from, String src, String dest, IOContext context)`	*Copies the file 'src' in 'from' to this directory as the file 'dest'*
`abstract IndexOutput createOutput(String name, IOContext context)`	*Creates a new, empty file in the directory with the given name*
`abstract IndexOutput createTempOutput(String pre, String suf, IOContext context)`	*Creates a empty file for writing in the directory, with a temp filename including prefix and suffix, ending with .tmp*
`abstract void deleteFile(String name)`	*Removes an existing file in the directory*
`abstract long fileLength(String name)`	*Return the length of a file in the directory*
`String[] listAll()`	*Returns an array of strings, one for each directory entry*
`abstract IndexInput openInput(String name, IOContext context)`	*Returns a stream reading an existing file*
`abstract void rename(String source, String dest)`	*Renames the source file, where 'dest' doesn't exist, yet, in the directory*

FSDirectory: Base class for a Directory implementation that store index files. *(There are three subclasses: **SimpleFSDirectory**, **NIOFSDirectory**, and **MMapDirectory**.)* The following notes apply:

- Use the function **`FSDirectory`**`.open(Path)` for Lucene to pick an implementation for the current environment.
- Accessing one of the subclasses directly or indirectly from a thread, while it's interrupted, can close the underlying channel immediately if, at the same time, the thread is clocked on IO.
- **FSDirectory** is intended for larger indexes, where **RAMDirectory** is intended for smaller ones.

FSDirectory	
Function Name	**Description**
`boolean checkPendingDeletions()`	*Tries to delete pending deleted files. Returns true if undeleted files are left*
`void deletePendingFiles()`	*Try to delete any pending files that previously tried to delete, but failed because we're on Windows and the files were still held open*
`Path getDirectory()`	
`static String[] listAll(Path dir)`	*List all files (including subdirectories) in the directory*
`static FSDirectory open(Path path)`	*Creates an FSDirectory instance, picking the best implementation*

RAMDirectory: Memory-resident **Directory** meant for indexes less than several hundred megabytes, which also has bad concurrency on multithreaded environments.

RAMDirectory	
Function Name	**Description**
`RAMDirectory()`	*Constructs an empty Directory*
`RAMDirectory(FSDirectory dir, IOContext context)`	*Makes a new RAMDirectory from other Directory implementations*
`boolean fileNameExists(String name)`	
`long ramBytesUsed()`	*Return total size, in bytes, of all files in the directory*

IOContext

Holds additional details on merge/search context. This object can never be initialized as null as passed as a parameter to openOutput(...) or openInput(...) in the **Directory** class. Values for IOContext.Context are as follows:

- *IOContext.Context.DEFAULT; IOContext.Context.FLUSH; IOContext.Context.MERGE; IOContext.Context.READ*

Function Name	Description
`IOContext(IOContext.Context context)`	
`IOContext(IOContext ctxt, boolean readOnce)`	*Initialize IOContext instance with a new value for the readOnce variable*

IndexOutput

IndexOutput: Abstract base for output to a file in a **Directory**. This class is used for all Lucene index output operations, and may only be used from one thread due to the storage of internal state data like file position. *(There are specified subclasses – **RAMOutputStream**, for example – but aren't covered by this document.)*

DataOutput: Abstract base class for performing write operations of Lucene's low-level data types. These instances may be used from only one thread.

IndexOutput	
Function Name	**Description**
`abstract void close()`	*Closes this stream to further operations*
`abstract long getFilePointer()`	*Returns the current position in this file (where the next write will occur)*
`String getName()`	*Returns the name used to create this instance*

DataOutput	
Function Name	**Description**
`void copyBytes(DataInput input, long numBytes)`	*Copies numBytes bytes from input to ourself*
`abstract void writeByte(byte b)`	*Writes a single byte*
`void writeBytes(byte[] b, int length)`	*Writes an array of bytes*
`void writeInt(int i)`	*Writes an int as four bytes*
`void writeLong(long i)`	*Writes a long as eight bytes*
`void writeMapOfStrings(Map<String,String> map)`	*Writes a String map*
`void writeSetOfStrings(Set<String> set)`	*Writes a String set*
`void writeShort(short i)`	*Writes a short as 2 bytes*
`void writeString(String s)`	*Writes a string*
`void writeVInt(int i)`	*Writes an int in variable-length format*
`void writeVLong(long i)`	*Writes a long in a variable-length format*

IndexInput

IndexInput: Abstract base for input from a **Directory**. This class is used for all Lucene index input operations, and may only be used from one thread due to the storage of internal state data like file position. The following notes apply:

- The instance must be cloned to allow multithreaded use.
- Lucene never closes cloned instances. close() will only be called on the original instance.
  - If a cloned IndexInput is accessed after closing the original object, read methods will throw exception.

DataInput: Abstract base class for performing read operations of Lucene's low-level data types. Instances of this object may only be used from one thread.

IndexInput	
Function Name	**Description**
`IndexInput clone()`	*Returns a clone of this stream*
`abstract void close()`	*Closes the stream to further operations*
`abstract long getFilePointer()`	*Returns the current position in this file (where the next read will occur)*
`abstract long length()`	*The number of bytes in the file*
`abstract void seek(long pos)`	*Sets current position in this file, where the next read will occur*
`abstract IndexInput slice(String sliceDesc, long off, long length)`	*Creates a slice of this index input, with the given desc, offset, and length*

DataInput	
Function Name	**Description**
`DataInput clone()`	*Returns a clone of the stream*
`abstract byte readByte()`	*Reads a single byte*
`abstract void readBytes(byte[] b, int offset, int len)`	*Reads a number of specified bytes into an array at the specified offset*
`int readInt()`	*Reads four bytes, returns int*
`long readLong()`	*Reads eight bytes, returns long*
`Map<String,String> readMapOfStrings()`	*Reads a map written prior with DataOutput.writeMapOfStrings*
`Set<String> readSetOfStrings()`	*Reads a set of strings written with DataOutput.writeSetOfStrings*
`short readShort()`	*Reads two bytes, returns short*
`String readString()`	*Reads a String*
`int readVInt()`	*Reads an int stored in variable-length format*
`int readVLong()`	*Reads a long stored in variable-length format*
`void skipBytes(long numBytes)`	*Skips over a number of bytes*

Search

The search package provides structures for queries – **TermQuery**, **PhraseQuery**, and **BooleanQuery**. The class **IndexSearcher** generates a **TopDocs** instance based on those queries. *(The scoring process, done when a query is submitted to the **IndexSearcher**, encompasses passing control to the **Weight** implementation and it's **Scorer/BulkScore** instances. Further, the following query types are not explored: **SpanNearQuery**, **TermRangeQuery**, **PointRangeQuery**, and **RegexpQuery**. Further, alternate search implementations allow sorting using a **Sort** instance, but aren't included.)*

The following notes apply:

- Applications should typically call **IndexSearcher.**search(Query, int) to perform a search; this yields **TopDocs.**
- **QueryBuilder** allows for easy query building; the class is in the Util package.
- A number of **QueryParsers** are provided for generation query structures from strings, or XML in the *queryparser API*. A number of query types is also available in the *queries module*.

IndexSearcher, TopDocs, and ScoreDoc

IndexSearcher: Implements search over a single **IndexReader**. Applications will only, usually, call the function search(Query,int). The following notes apply:

- If the index is unchanging, share a single **IndexSearcher** instance across multiple searches.
- If the index is changing, and you wish to see reflected changes, use **DirectoryReader**. openIfChanged(DirectoryReader) to obtain a new reader, then create a new searcher from that.
    - For low-latency turnaround, it's best to use a near-real-time reader – **DirectoryReader**. open(IndexWriter). Once you have a new **IndexReader**, it's relatively cheap to create a new **IndexSearcher** from it.
- Instances of this class are completely thread safe, meaning multiple threads can call any of its methods concurrently. External synchronization, if necessary, should **not** synchronize to the instance.

IndexSearcher	
Function Name	**Description**
`IndexSearcher(IndexReader r)`	*Creates a searcher searching the provided index*
`int count(Query query)`	*Count how many documents match the given query*
`Document doc(int docID)`	*Same as .getIndexReader().document(docID)*
`Document doc(int docID, Set<String> fieldsToLoad)`	*Same as .getIndexReader().document(docID, fieldsToLoad)*
`Explanation explain(Query q, int doc)`	*Returns an Explanation that describes doc's score against q*
`IndexReader getIndexReader()`	*Gets the IndexReader this searches*
`TopDocs search(Query q, int n)`	*Finds the top n hits for query*
`TopDocs searchAfter(ScoreDoc after, Query q, int numHits)`	*Find the top n hits for query where all results are after 'after'*
`TermStatistics termStatistics(Term term, TermContext context)`	*Returns TermStatistics for a term*

TopDocs: The results returned by using IndexSearcher's search(...) function. It contains two fields:

▪ *ScoreDocs[] scoreDocs*	The top hits for the query
▪ *long totalHits*	The number of hits for the query

TopDocs	
Function Name	**Description**
`float getMaxScore()`	*Returns the maximum score value encountered*

ScoreDoc: Contains information about a single result in a **TopDocs.** It contains two fundamental fields:

▪ *int doc*	The hit's document ID
▪ *float score*	The score of the document for the query

Query

The abstract basis for queries.

TermQuery

Query that matches documents containing a term. These may be combined with other terms with a **BooleanQuery**.

Function Name	Description
`TermQuery(Term t)`	*Creates a query for the term t*
`Term getTerm()`	*Returns the term of this query*
`String toString()`	*Prints a user-readable version of the query*

In the following example, the query will identify all **Document**s containing the **Field** named 'name' with the word 'term' in that field: <u>TermQuery q = new TermQuery(new Term("name", "term"))</u>.

BooleanQuery and BooleanClause

Query that matches documents based on boolean combinations of: **TermQuery**s, **PhraseQuery**s, or **BooleanQuery**s. The following notes apply for these classes:

- A BooleanQuery must be built using the nested class **BooleanQuery.Builder.**
- The **Builder** class returns itself to allow ease of use when constructing boolean queries; as such, <u>no return type is listed for these functions.</u>
- The values accepted for the *BooleanClause.Occur* parameter below are as follows:

BooleanClause.Occur.MUST	The clause must appear in matching documents
BooleanClause.Occur.MUST_NOT	The clause must not appear in matching documents
BooleanClause.Occur.SHOULD	The clause *should* appear in matching documents
BooleanClause.Occur.FILTER	Similar to MUST, except the clause doesn't participate in scoring

BooleanClause	
Function Name	**Description**
`BooleanClause(Query q, BooleanClause.Occur occ)`	*Constructs a BooleanClause*
`BooleanClause.Occur getOccur()`	
`Query getQuery()`	
`boolean isProhibited()`	
`boolean isRequired()`	
`boolean isScoring()`	
`String toString()`	

BooleanQuery.Builder	
Function Name	**Description**
`Builder()`	*Sole constructor*
`add(BooleanClause clause)`	*Add a new clause to the builder*
`add(Query query, BooleanClause.Occur occur)`	*Add a new clause to the builder*
`BooleanQuery build()`	*Create a new BooleanQuery based on the parameters set*
`setMinimumNumberShouldMatch(int min)`	*Specifies a min number of optional clauses that should be satisfied*

BooleanQuery	
Function Name	**Description**
`List<BooleanClause> clauses()`	*Gets a list of clauses for the query*
`static int getMaxClauseCount()`	*Return the maximum number of clauses permitted (Default: 1024)*
`int getNumberShouldMatch()`	*Gets the min number of optional clauses that must be satisfied*
`Iterator<BooleanClause> iterator()`	*Returns an iterator on the clauses in this query*
`static void setMaxClauseCount(int maxCount)`	*Set the maximum number of clauses permitted per query*
`String toString(String field)`	*Prints a user-readable version of the query*

PhraseQuery and MultiPhraseQuery

PhraseQuery and **PhraseQuery.Builder:** Match documents containing a sequence of terms. (Built for input such as "new york.") The following notes apply:

- All terms in a **PhraseQuery** must match, even those at the same position. If you want to use synonyms, a **MultiPhraseQuery** is the best choice.
- Leading holes have no particular meaning, and will be ignored. For example, searching for two terms at the positions of 4 and 5 is similar to searching for the same terms at the positions of 0 and 1.
- A factor called *slop* determines how many positions may occur between any two terms in the phrase and still be considered a match.
- The **Builder** class returns itself to allow ease of use when constructing boolean queries; as such, <u>no return type is listed for these functions.</u>

PhraseQuery.Builder	
Function Name	**Description**
`Builder()`	*Sole constructor*
`add(Term term)`	*Adds a term to the end of the phrase query*
`add(Term term, int position)`	*Adds a term to the end of the phrase query*
`PhraseQuery build()`	*Build a phrase query based on the terms that have been added*
`setSlop(int slop)`	*Set the slop*

PhraseQuery	
Function Name	**Description**
`PhraseQuery(int slop, String fld, BytesRef… terms)`	*Create a phrase query which'll match documents containing the given list of terms at consecutive positions in field, and at a maximum edit distance of slop*
`PhraseQuery(int slop, String fld, String… terms)`	*Create a phrase query. (Same as above)*
`PhraseQuery(String field, BytesRef… terms)`	*Create a phrase query that'll match documents containing the given list of terms in consecutive positions*
`PhraseQuery(String field, String… terms)`	*Create a phrase query. (Same as above)*
`int[] positions()`	*Returns the relative positions of terms in this phrase*
`int getSlop()`	*Return the slop for the query*
`Term[] getTerms()`	*Returns the list of terms in this phrase*
`String toString(String f)`	*Prints a user-readable version of this query*

MultiPhraseQuery and **MultiPhraseQuery.Builder:** A generalized version of a **PhraseQuery** that allows disjunction (OR) for terms at the same position.

MultiPhraseQuery.Builder	
Function Name	**Description**
`Builder()`	*Default constructor*
`Builder(MultiPhraseQuery mpq)`	*Copy constructor: creates a builder with the same configuration as the provided builder*
`add(Term term)`	*Adds a single term at the next position in the phrase*
`add(Term[] terms)`	*Adds multiple terms at the next position in the phrase*
`add(Term[] terms, int position)`	*Adds multiple terms at the specified position*
`MultiPhraseQuery build()`	*Builds a MultiPhraseQuery*
`setSlop(int s)`	*Sets the phrase slop*

MultiPhraseQuery	
Function Name	**Description**
`int[] positions()`	*Returns the relative positions of terms in the phrase*
`int getSlop()`	*Gets the phrase slop for this query*
`Term[][] getTermArrays()`	*Returns the arrays of arrays of terms in the multi-phrase*
`String toString(String f)`	*Prints a user-readable version of this query*

PrefixQuery and WildcardQuery

PrefixQuery: Matches documents containing the terms with a specified prefix. This type of query is built by a QueryParser for input like 'app*'. *(**WildcardQuery** is a generalized form of prefix querying.)*

PrefixQuery	
Function Name	**Description**
`PrefixQuery(Term prefix)`	*Constructs a query for terms starting with the prefix*
`Term getPrefix()`	*Returns the prefix of this query*
`String toString(String fld)`	*Prints a user-readable version of this query*

WildcardQuery: Implements wildcard search querying. Suppoted wildcards are:

- * Matches any character sequence (including the empty one)
- ? Matches a single character
- \ Escape character

The following notes apply:

- These queries can be slow, as they need to iterate over numerous terms.
- To prevent extremely-slow queries, a wildcard term shouldn't start with '*'.
  - Some **QueryParser**s will not allow a wildcard term to start with '*', but contain the function setAllowLeadingWildard() to remove such protection.

WildcardQuery	
Function Name	**Description**
`WildcardQuery(Term term)`	*Constructs a query for terms matching the term*
`WildcardQuery(Term term, int maxDetStates)`	*Constructs a query for terms matching the term*
`Term getTerm()`	*Returns the pattern term*
`String toString(String fld)`	*Prints a user-readable version of this query*

FuzzyQuery

Fuzzy queries match documents that contain terms similar to the specified term. Similarity is determined using Levenshtein distance. This query can be useful when accounting for spelling variations. The following notes apply:

- Similarity measurements are based on the Damerau-Levenshtein algorithm, though classic Levenshtein is available by passing *false* to the *transpositions* parameter.
- At most, a fuzzy query will match up to 2 edits. Higher distances are generally un-useful. *(Using higher distances is possible with an n-gram indexing technique (such as **SpellChecker** in the suggest module, but isn't covered here.))*
- Terms of length 1 or 2 sometimes won't match because of how scaled distance between two terms is computed. For a term to match, the edit distance between the terms must be less than the minimum length term. For example, *'abcd'* with maxEdits=2 will not match *'ab'* as an indexed term.

Function Name	**Description**
`FuzzyQuery(Term term)`	*Same as FuzzyQuery(term, defaultMaxEdits)*
`FuzzyQuery(Term term, int maxEdits)`	*Same as FuzzyQuery(term, maxEdits, defaultPrefixLength)*
`FuzzyQuery(Term term, int maxEdits, int prefixLength)`	*Same as FuzzyQuery(term, maxEdits, prefixLength, defaultMaxExpansions, defaultTranspositions)*
`FuzzyQuery(Term t, int maxEdit, int preLen, int maxExp, boolean transpositions)`	*Create a new FuzzyQuery that will match terms with an edit distance of, at most, maxEdits to term*
`int getMaxEdits()`	
`int getPrefixLength()`	*Returns the non-fuzzy prefix length*
`Term getTerm()`	*Returns the pattern term*
`boolean getTranspositions()`	*Returns if transpositions are treated as primitive edit operation*
`String toString(String field)`	*Prints a query to a string, with 'field' assumed to be the default field and omitted*

TermRangeQuery

Matches documents within a range of terms.

Function Name	Description
`TermRangeQuery(String fld, BytesRef lower, BytesRef upper, boolean incLower, boolean incUpper)`	*Constructs a query, selecting all terms greater/equal than lower and less/equal than upper*
`BytesRef getLowerTerm()`	*Returns the lower value of this range query*
`BytesRef getUpperTerm()`	*Returns the upper value of this range query*
`boolean includesLower()`	*Returns true if the lower endpoint is inclusive*
`boolean includesUpper()`	*Returns true if the upper endpoint is inclusive*
`static TermRangeQuery newStringRange(String fld, String lower, String upper, boolean incLower, boolean incUpper)`	*Creates a new TermRangeQuery using strings for term text*
`String toString(String fld)`	*Prints a user-readable version of this query*

BoostQuery

Allows the boosting of other queries by wrapping them. Boost values less than one give less importance to the query, while values that're greater than one give more importance to the scores returned by the query. *(More complex boosts can be applied using **FunctionScoreQuery** in the lucene-queries module, but are not discussed here.)*

Function Name	Description
`BoostQuery(Query query, float boost)`	*Wrap 'query' in such a way that the produced scores are boosted*
`float getBoost()`	*Gets the applied boost*
`Query getQuery()`	*Gets the wrapped query*
`Sring toString(String field)`	*Prings a query to a string, with 'field' assumed to be the default field and omitted*

Explanation

Explanation is a simple object that describes the score computation for a document against a query. These instances can be received from the **IndexSearcher**.

Function Name	Description
`String getDescription()`	*A description of this explanation node*
`boolean isMatch()`	*Indicates whether or not this Explanation models a match*
`String toString()`	*Render an explanation as text*

Util

The util package provides handy data structures and utility classes for use with Lucene.

BytesRef

Represents an array of bytes – byte[] – as a slice (offset+length) into an existing byte[]. The following notes apply:

- The **bytes** variable shouldn't be null. Use *EMPTY_BYTES* if necessary
- Lucene uses this class to represent terms encoded as UTF-8. To convert them to a String, use utf8ToString().

A summary of the fields of this class are as follows:

byte[] *bytes*	The contents of the BytesRef
static byte[] *EMPTY_BYTES*	Empty byte array for convenience
int *length, offset*	The length of the used bytes, and the offset of the first, valid byte

Function Name	Description
`BytesRef()`	*Create a BytesRef with EMPTY_BYTES*
`BytesRef(byte[] bytes)`	*Directly references bytes without making a copy*
`BytesRef(byte[] bytes, int off, int length)`	*Directly references bytes without making a copy*
`BytesRef(int capacity)`	*Create a BytesRef pointing to a new array of size 'capacity'*
`boolean bytesEquals(BytesRef other)`	*Expert: Compares the bytes against another BytesRef*
`BytesRef clone()`	*Returns a shallow clone of the instance (the bytes aren't copied, and will be shared by both objects)*
`int compareTo(BytesRef other)`	*Unsigned byte order comparison*
`static BytesRef deepCopyOf(BytesRef other)`	*Creates a new BytesRef that points to a copy of the bytes of 'other'*
`boolean isValid()`	*Performs internal consistency checks*
`String toString()`	*Returns hex-encoded bytes. (Ex: "[0x63 0x75]")*
`String utf8ToString()`	*Interprets stored bytes as UTF8 bytes, returning the result string*

QueryBuilder

Creates queries from the **Analyzer** chain. This class can be used as a subclass for query parsers to make generation of queries easier. Factory methods, such as newTermQuery(...), are provided such that generated queries can be customized.

Function Name	Description
`QueryBuilder(Analyzer analyzer)`	*Creates a new builder using the given analyzer*
`Query createBooleanQuery(String field, String query)`	*Creates a boolean query from the query text*
`Query createBooleanQuery(String field, String query, BooleanClause.Occur operator)`	*Creates a boolean query*
`Query createMinShouldMatchQuery(String field, String query, float fraction)`	*Creates a minimum-should-match query*
`Query createPhraseQuery(String field, String query)`	*Creates a phrase query*
`Query createPhraseQuery(String field, String query, int slop)`	*Creates a phrase query*
`Analyzer getAnalyzer()`	
`boolean getAutoGenerateMultiTermSynonymsPhraseQuery()`	*Returns true if phrase query should be auto-generated for multi-terms synonyms*
`boolean getEnablePositionIncrements()`	*Returns true if position increments are enabled*
`void setAnalyzer(Analyzer analyzer)`	*Set the analyzer used to tokenize text*
`void setAutoGenerateMultiTermSynonymPhraseQuery(boolean enable)`	*Set to true if phrase queries should be auto-generated for multi terms synonyms*
`void setEnablePositionIncrements(boolean enable)`	*Enable position increments in result query*

Query Parsing

Query parsing is a fundamental utility, with numerous query parsers and query types made available in Lucene. The different types of queries were already studied in the API's core, but query parsing is made available in a different module. Discussion about these parsers and their language syntax (for writing queries) is discussed in the Appendix.

Classes

QueryParser

A class generated by JavaCC. The most important method is the parse(String) function. This class can throw a **ParseException** when using the parse(...) function. *The parse(...) function was inherited from the QueryParserBase, and thus any parser that extends it will throw this same exception.*

QueryParser	
Function Name	**Description**
`QueryParser(String f, Analyzer a)`	*Creates a new query parser where f is the default field to search*

QueryParserBase	
Function Name	**Description**
`Query parse(String query)`	*Parses a query string, returning a Query*

SimpleQueryParser

Parses human readable query syntax. The main idea is a person should be able to type whatever they want, and the parser will attempt to interpret what to search for. Errors are ignored, and the parser attempts to decipher what it can.

Function Name	Description
`SimpleQueryParser(Analyzer an, Map<String,Float> weights)`	*Creates a new parser, searching over multiple fields with different weights*
`SimpleQueryParser(Analyzer an, Map<String,Float> weights, int flags)`	*Create parser with flags to enable/disable features*
`SimpleQueryParser(Analyzer an, String field)`	*Creates a new parser for searching over a single field*
`BooleanClause.Occur getDefaultOperator()`	*Returns implicit operating setting (SHOULD or MUST)*
`Query parse(String query)`	*Parses the query text and returns parsed query*
`void setDefaultOperator(BooleanClause.Occur op)`	*Sets implicit operator setting (SHOULD or MUST)*

Similarities

The following classes are stored in the package ***org.apache.lucene.search.similarities***. These are the similarities offered by Lucene, with BM25 being default. *(Instructions for implementing a different similarity are discussed above.)*

- **BM25Similarity**, **BooleanSimilarity**, **PerFieldSimilarityWrapper**, and **TFIDFSimilarity** (**ClassicSimilarity**) are extensions of the base class ***Similarity***.
- **Axiomatic**, **DFISimilarity**, **DFRSimilarity**, **IBSimilarity**, and **LMSimilarity** are all extensions of ***SimilarityBase***.

Classes

SimilarityBase and Similarity

*(**Similarity** and **SimilarityBase** are both experimental APIs, and may change incompatibly for future releases; as such, many of their children classes are experimental APIs. Further, both of these classes are meant for expert usage, but explanations are derived from the API documentation as to see their inner workings in some detail.)*

SimilarityBase: Subclass of **Similarity** that provides a simplified API for its descendants.

- Subclasses are only required to implement the score(BasicStats, float, float) and toString() methods. Implementation of explain(BasicStats, float, float) is optional – **SimilarityBase** provides a basic one as default.
  - **BasicStats** is a subclass of Similarity's **SimWeight** nested class, and stores all statistics commonly used for ranking methods.
- *Note:* Multi-word queries, such as phrase queries, are scored in a different way than Lucene's default ranking alg.

Similarity: Similarity defines the components of Lucene scoring, and is a low-level API intended for expert usage. It:

- Should be extended only to be used for a custom IR model's implementation
- Contains two nested classes – **Similarity.SimScorer** and **Similarity.SimWeight** – that provide API for scoring 'sloppy' queries and store the weight of a query across the indexed collection
- Determines how Lucene weighs terms, and interacts with the class at *index-time* and *query-time*
  - Index time
    - The indexer calls the computeNorm(…) function to allow the similarity to set a per-document value for the field, later accessible via **LeafReader**.getNormValues(String). Lucene makes no assumption about what's in the norm, but its useful for encoding length normalization info.
    - Many formulae require avg document length, computable via a combination of statistics available in **CollectionStatistics**, depending on whether the avg should reflect field sparsity.
    - Additional scoring factors can be stored in named **NumericDocValuesField**s, accessible at query time with **LeafReader**.getNumericDocValues(String).
  - Query time
    - Queries interact with the similarity by calling the computeWeight(…) method, allowing the implementation to compute statistics (IDF, avg doc length, etc) across the entire collection. *(Raw statistics are stored in **CollectionStatistics** and **TermStatistics**.)*
    - For each index segment, the Query creates a simScorer(…). The score() method is called for each matching document after.
  - Explanation: Queries consult the similarity's DocScorer for an explanation of how it computed its score when **IndexSearcher**.explain(Query, int) is called.

BM25Similarity

BM25 similarity, introduced in the following reference:

- Stephen E. Robertson, Steve Walker, Susan Jones, Micheline Hancock-Beaulieu, and Mike Gatford. Okapi at TREC-3. In Proceedings of the Third Text Retrieval Conference (TREC 1994). Gaithersburg, USA, November 1994.

This class contains a single field – *discountOverlaps* – that is true if overlap tokens – tokens with a position increment of 0 – should be discounted from the document's length or not.

Function Name	Description
`BM25Similarity()`	*BM25 with k1 = 1.2, b = 0.75*
`BM25Similarity(float k1, float b)`	*BM25 with the supplied parameters*
`float getB()`	
`boolean getDiscountOverlaps()`	*Returns true if overlap tokens are discounted from doc length*
`float getK1()`	
`Explanation idfExplain(CollectionStatistics colStats, TermStatistics termStat)`	*Computes a score factor for a simple term, and returns an explanation for that score factor*
`Explanation idfExplain(CollectionStatistics colStats, TermStatistics[] termStat)`	*Computes a score factor for a phrase*
`void setDiscountOverlaps(boolean v)`	*Sets whether overlap tokens are ignored when computing norm*

ClassicSimilarity

Historical scoring implementation that utilizes TF-IDF weighting for scoring. Lucene implements a VSM, but makes modifications to the concept that can be read about in the API. For example:

- Vectors aren't normalized to unit vectors – instead, a different document length normalization is used, normalizing to a vector equal to or larger than the unit vector.
- Documents can be given boosts at indexing, boosting their importance.
- Lucene is field-based, and each query term applies to a single field. The same field can be added to a document during indexing several times, thus raising the boost of that field to be the multiplication of the boosts of the separate parts of the field.
- Users can specify boosts to each query, sub-query, and query term.
- A document may match a multi term query without containing all the terms of that query.

ClassicSimilarity	
Function Name	**Description**
`ClassicSimilarity()`	
`float idf(long docFreq, long docCount)`	*log((docCount+1)/(docFreq+1))+1*
`Explanation idfExplain(CollectionStatistics colStat, TermStatistics termStat)`	*Computes a score factor for a simple term; returns explanation*
`float lengthNorm(int numTerms)`	*1/sqrt(length)*
`float sloppyFreq(int distance)`	*1/(distance+1)*
`float tf(float freq)`	*sqrt(freq)*

TFIDFSimilarity	
Function Name	**Description**
`boolean getDiscountOverlaps()`	*Returns true if overlap tokens are discounted from doc length*
`Explanation idfExplain(CollectionStatistics colStats, TermStatistics termStat)`	*Computes a score factor for a simple term, and returns an explanation for that score factor*
`Explanation idfExplain(CollectionStatistics colStats, TermStatistics[] termStat)`	*Computes a score factor for a phrase*
`void setDiscountOverlaps(boolean v)`	

BooleanSimilarity

BooleanSimilarity is a simple similarity that gives terms a score equal to their query boost. *(This similarity is typically used with disabled norms, since neither doc stats or index stats are used for scoring.)* If norms are enabled, they're computed the same way as **SimilarityBase** and **BM25Similarity**. *(No functions are discussed for this class due to its simplicity.)*

The only function to be concerned with is the constructor: **`BooleanSimilarity()`**.

MultiSimilarity

Implements the CombSUM method for combining evidence from multiple similarity values. *(Basically, this similarity combines the scores of others through summation for the resulting, final score.)*

The only function to be concerned with is the constructor: **`MultiSimilarity(Similarity[] sims)`**.

PerFieldSimilarityWrapper

This class allows a user to supply different **Similarity** implementations for different fields. Subclasses should implement the get(String) function to return an appropriate Similarity (for example, using field-specific parameter values) for fields.

Function Name	Description
`PerFieldSimilarityWrapper()`	*Sole constructor*
`abstract Similarity get(String name)`	*Returns a Similarity for scoring a field*

Axiomatic

Axiomatic is an abstract class that's used for axiomatic approaches for IR. All of the models in this family are based on BM25, Pivoted Document Length Normalization, and Language model with Dirichlet prior. Some components are modified so they follow some axiomatic constraints. *(The children classes of this abstract class are **AxiomaticF1EXP**, **AxiomaticF1LOG**, **AxiomaticF2EXP**, **AxiomaticF2LOG**, **AxiomaticF3EXP**, and **AxiomaticF3LOG**.)* These are from:

- Hui Fang and Chengxiang Zhai 2005. An Exploration of Axiomatic Approaches to Information Retrieval. In Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval (SIGIR '05). ACM, New York, NY, USA, 480-487.

DFISimilarity

This similarity implements the Divergence from Independence model based on Chi-square statistics (i.e. standardized Chi-squared distance from independence in term frequency tf). Stopword removal is **not** recommended. It's both parameter-free and non-parametric:

- *Parameter-Free:* Doesn't require any parameter tuning or training
- *Non-Parametric*: Doesn't make any assumptions about word frequency distributions on document collections

The only function to be concerned with is the constructor: **`DFISimilarity(Independence indMeasure)`**.

Variants of **Independence** are as follows:

- **IndependenceStandardized**: Described as "good at tasks that require high recall and high precision, especially against shorter queries composed of a few words, as in the case of Internet searches"
- **IndependenceSaturated**: Described as "for tasks that require high recall against long queries"
- **IndependenceChiSquared**: Described as "for tasks requiring high precision against long and shorter queries"

DFRSimilarity

This similarity implements the Divergence from Randomness framework, introduced in: Gianni Amati and Cornelis Joost Van Rijsbergen. 2002. Probabilistic models of information retrieval based on measuring the divergence from randomness. ACM Trans. Inf. Syst. 20, 4 (October 2002), 357-389. The scoring formula is composed of three separate components:

- **BasicModel**: Basic model of information content
- **AfterEffect**: First normalization of information gain
- **Normalization**: Second – length – normalization

The one function needed is a constructor: `DFRSimilarity(BasicModel bm, AfterEffect ae, Normalization norm)`.

Variants of the **BasicModel**, **AfterEffect**, and **Normalization** variants are as follows:

- **BasicModel:**

BasicModelBE	Limiting form of Bose-Einstein
BasicModelG	Geometric approximation of Bose-Einstein
BasicModelP	Poisson approximation of the Binomial
BasicModelD	Divergence approximation of the Binomial
BasicModelIn	Inverse document frequency
BasicModelIne	Inverse expected doc. frequency (mixture of Poisson/IDF)
BasicModelIF	Inverse term frequency (approximation of I(ne))

- **AfterEffect:**

AfterEffectL	Laplace's law of succession
AfterEffectB	Ratio of two Bernoulli processes
AfterEffect.NoAfterEffect	No first normalization

- **Normalization:**

NormalizationH1	Uniform distribution of TF
NormalizationH2	TF density inversely related to length
NormalizationH3	TF normalization provided by Dirichlet prior
NormalizationZ	TF normalization provided by a Zipfian relation
Normalization.NoNormalization	No second normalization

IBSimilarity

This similarity provides a framework for the family of information-based models, described in: Stephane Clinchant and Eric Gaussier. 2010. Information-based models for ad hoc IR. In Proceeding of the 33rd international ACM SIGIR conference on Research and development in information retrieval (SIGIR '10). ACM, New York, NY, USA, 234-241. A complex retrieval function is used by this similarity, where there are three factors:

- **Distribution:** Probabilistic distribution used to model term occurrence
  - **DistributionLL** (Log-logistic) or **DistributionSPL** (smoothed power-law)
- **Lambda:** Parameter of the probability distribution
  - **LambdaDF** (Average number of documents where a term occurs) or **LambdaTTF** (average number of occurrences of a term in the collection)
- **Normalization:** Term frequency normalization *(Discussed in DFR normalization)*

The one function to use is the constructor: `IBSimilarity(Distribution di, Lambda la, Normalization norm)`.

LMSimilarity

LMSimilarity is an abstract superclass for language modeling similarities. It introduces a few inner types; *however, these won't be explored in this document. Further, the following classes have more than a single constructor that allows changing of the used **LMSimilarity.CollectionModel**, but are not listed*. Two concrete implementations are as follows:

- **LMDirichletSimilarity**: Implements Bayesian smoothing using Dirichlet priors, assigning a negative score to documents that contain the term, but with fewer occurrences than predicted by the collection language model. *(Lucene returns 0 for such documents.)*
  - Constructor: `LMDirichletSimilarity()` or `LMDirichletSimilarity(float mu)`
- **LMJelinekMercerSimilarity**: Language model based on the Jelinek-Mercer smoothing method. The model takes a single parameter, lambda, with the optimal value around 0.1 for title queries, and 0.7 for long queries.
  - Constructor: `LMJelinekMercerSimilarity(float lambda)`

Appendix

Summarization

Apache Lucene's major packages provide the core functionality for using the software package, and each contains numerous classes. In the core API alone, there is approximately seven hundred, not inclusive of the query-parsers and analyzers-common packages. Whereas a full tree of classes provides a thorough insight into Lucene's structure, it does little to abstract the commonalities, making it difficult to attain a basic understanding; as such, summarizations of the different, primary components is provided here that will be used to generate abstraction diagrams.

Analysis

The analysis package provides all the functionality necessary for tokenization of a document corpa. This is done via the usage of **Analyzer**s - constructs that incorporate **CharFilter**s, **Tokenizer**s, **TokenFilter**s, and **TokenStream**s. *(**Tokenizers** and **TokenFilters** are simply **TokenStreams**.)*

The general procedure is as follows: *Raw text → CharFilter → Tokenizer → TokenFilter → TokenStream (stored in Analyzer)*

[1] Token text is passed through a **CharFilter**, transforming text prior to being tokenized.
[2] **Tokenizer**s take the text that was passed through **CharFilter**s, breaking it up into tokens.
[3] **TokenFilter**s take the tokenized text from **Tokenizer**s, and modify them through deletions, synonym injection, stemming, and case folding.
[4] The finalized series of tokens is then made available through an **Analyzer**'s **TokenStream**, which is consumed by indexing and search processes.

Examples of **Analyzer**s are the **StopAnalyzer**, **WhitespaceAnalyzer**, **SimpleAnalyzer**, and **StandardAnalyzer**. Examples of **TokenFilter**s are the **StandardFilter**, **LowerCaseFilter**, **UpperCaseFilter**, and **StopFilter**. Sample **Tokenizer**s are the **WhitespaceTokenizer**, **LetterTokenizer**, and **StandardTokenizer**. Lucene provides a few of these, and more, such as for foreign languages, in its analyzers-common package.

Document

The document package provides all necessary functionality for constructing the documents Lucene must store, index, and search utilizing queries. The **Document** class is a basis for this, for it takes a collection of **Field**s, which implement the **IndexableField** interface, with a predefined set of properties. If customization over the properties of the field is required, then the **FieldType** class is available for defining those properties – for example, storage of the field in the index, tokenization, and normalization value omission. Various types of predefined fields are made available by Lucene, inclusive of, but not limited to: **TextField**, **StringField**, **IntPoint**, **LongPoint**, and **StoredField**.

Index

The index package provides functionality for reading and writing an index. This is done via the instantiation of an **IndexWriterConfig** instance, which allows customization of appendation or overwriting of index contents, among other settings. This configuration is then used to instantiate an **IndexWriter**, which is responsible for writing the actual changes to a **Directory** *(see "Store")*. **IndexReader**s are then responsible for reading information from the index, with an example being the **StandardDirectoryReader**.

The index is stored in a segmented format, which has changed significantly throughout Lucene's development history. Control over the storage of index contents is modifiable through the "Codecs" package. Further, modifications to the index, either through document addition, deletion, or updates, is tracked via a series of "commits" – longs that identify the actions. An **IndexReader** is unable to see modifications to the index until the **IndexWriter** either updates the content, or rolls itself back from unwanted modifications.

Additional classes are provided by Lucene that expose information in the index, such as postings lists (**PostingsEnum**), and other term information (**TermsEnum**). *(Classes that assist in this effort are **Terms** and **FieldsProducer**.)* This package also defines the basic unit of search – **Term**.

Store
The store package provides code that assists in the access of index information in the form of the **Directory** class. Two main types of directories exist: **RAMDirectory**, for storing index information in RAM; and **FSDirectory**, which has three child implementations, for storing information on-disk. Whereas these classes are all that's *truly* necessary to operate Lucene, classes are made available that expose the capacity to read and write index information – **IndexInput** and **IndexOutput** for reading and writing, respectively. Manipulation of index data is done through some context, as defined in the **IOContext** class, and is usually passed to the **Directory**'s functions as a parameter when trying to fetch an **IndexOutput** or **IndexInput** instance.

Search
The search package provides functionality for querying the contents of the document corpus, after tokenization and further processing, as well as capacities to change the way processing is done on the corpus. The major, control class in doing so is **IndexSearcher**, which takes a **DirectoryReader** (child of **IndexReader**; *see "Index"*) as a constructor parameter, and contains a function – search(...) – for fetching results on queries made using the **Query** class. Results returned by this function are stored in a **TopDocs** instance, which encompasses a series of **ScoreDoc**s that contain document IDs and how those documents (associated with the IDs) scored against the query. *(If the user desires an explanation of 'how' that score was found, the package also provides an* ***Explanation*** *class that would be generated through* ***IndexSearcher****'s explain(...) function.)*

Lucene provides a variety of query types by default, such as, but not limited to: **TermQuery**, **PhraseQuery**, **BooleanQuery** (comprised of **BooleanClause**s), and **PrefixQuery.** These classes can be combined together to generate complex queries, but must be stored under the **Query** type for passing into the search(...) function. Other types of queries are made available in the *queries* module.

Further, sample similarities provided by Lucene, for usage, are **BM25Similarity**, **ClassicSimilarity**, which implements TFIDF and a modification of VSM, and **LMSimilarity** (with Dirichlet and Jelinek-Mercer smoothing variants). Lucene also provides the ability to use multiple similarities via **MultiSimilarity,** or specific ones for fields in **PerFieldSimilarityWrapper**. To incorporate these similarities, they must be used in setSimilarity(...) functions in both **IndexWriterConfig** *(see "Index")* and **IndexSearcher**, and the same similarity MUST be used for both cases.

A utility class, **QueryBuilder** *(see "Util")*, allows for building of queries on a basic level, though a series of query parsers included with Lucene, such as **QueryParser** and **SimpleQueryParser**, is available in the *query-parsers* package. Each query parser is unique, and may use a different syntax. For example, **QueryParser** uses Lucene's query syntax in its entirety, providing robust error checking on that syntax, while the **SimpleQueryParser** uses a variant of it that lacks such error-checking features.

Codecs
The codecs package provides an abstraction of the encoding and decoding functions that assist in writing and reading an index from the disk. Lucene provides various codec formats, some of which have replaced others during its development; however, the main driver of this package is the **Codec** class. Different classes exist that encompass the different portions of a segmented index, inclusive of, but not limited to: **PostingsFormat** for terms, posting, and proximity data; **TermVectorsFormat** for controlling term vector format; and **LiveDocsFormat** for controlling live, or deleted, documents. *(This package was not explored in-depth due to its nature as an expert-feature in Lucene; however, it was found that different portions of encoding/decoding can be overwritten by a custom codec, and defaults used for others.)* Various codecs are made available through the *codecs* module.

Util
The util package provides classes that may be used throughout numerous packages in the Apache Lucene API, but that may also be used by the user for various purposes. A few of the classes contained in this package are:

- **BytesRef**, which is proactively used for numerous stages of the indexing and querying process in place of Strings, meant to represent the different tokens in the document corpa;
- **QueryBuilder**, which provides a general, basic generator for **Query** instances of multiple types;
- and **AttributeSource**, which wasn't explored by this document, but which **TokenStream** and its subclasses are an extension of, allowing description of various token properties (such as **CharTermAttribute** for token text)

Query Parser Language Syntax

QueryParser

Fundamentals:

- There are two types of terms: single terms and phrases
  - Single term examples: "`test`", "`hello`"
  - Phrase examples: "`hello dolly`", "`goodnight world`"
- Multiple terms can be combined with Boolean operators to perform a complex query
  - Lucene supports the following operators: `AND ("&&"), "+", OR ("||"), NOT ("!")`, and "`-`"
    - "+" represents a required clause, and "-" represents a prohibited clause
    - The boolean operators **MUST** be written in all-caps to be successfully applied
  - OR is the default conjunction operator. If no Boolean operators are between terms, OR is implied
- Parentheses may be used to group clauses and form subqueries. Grouping may also be used in field searching
  - Grouping example: "`(Jakarta OR apache) AND website`"
  - Field Grouping example: "`title:(+return +"pink panther")`"
- Special characters may be escaped by prefixing the character with '\'
  - List of special characters: `+ - && || ! ( ) { } [ ] ^ " ~ * ? : \ /`

Term Modifiers:

- Wildcard
  - "?" performs a single character wildcard search, such as using "`te?t`" to return "test" or "text"
  - "*" Performs a multiple character wildcard search, such as using "`test*`" to return "tests" or "testing"
- Regular Expression *(Documented in **RegExp**)*
  - A pattern must be surrounded by forward slashes – '/'; escape special characters with '\'
  - Character ranges may be specified by using a hyphen. Negations may be specified with a caret
  - Examples: "`/[mb]oat/`" or "`/[^a-z]/`"
- Fuzzy Search
  - Use a tilde, ~, at the end of a single term. You may also specify the max number of edits *(Default: 2)*
  - Examples: "`roam~`" or "`roam~1`"
- Range Search
  - Match documents whose field(s) values are between a lower and upper bound.
  - They may be inclusive (surrounded by "[ ]") or exclusive (surrounded by "{ }")
  - Examples: "`title:{Aida TO Carmen}`" or "`mod_date:[20020101 TO 20030101]`"
- Term Boosting
  - The higher the boost factor you choose, the more relevant the term is considered. This manipulates, in turn, the relevance of matching documents. *(Default boost factor: 1)*
  - Boost a term by using the caret symbol with a boost factor (a number) at the end of the term searched
  - Examples: "`Jakarta^4 apache`" or "`"Jakarta apache"^4`"

SimpleQueryParser

Operator	Purpose	Example
+	AND operation	`token1+token2`
|	OR operation **(Default operator)**	`token1|token2`
-	Negates a single token	`-token0`
"	Creates phrases of terms	`"term1 term2"`
*	Specifies a prefix query *(at end of term)*	`term*`

~n	Specifies a fuzzy query *(at end of term)*
	Specifies a near query *(at the end of a phrase)*
		`term~1`
		`"term1 term2"~5`

( )	Precedence	`token1 + (token2 | token3)`

The following notes apply:

- The or operator is implied if no operators are given, such as for the query "`token1 token2`"
- Special characters can be escaped using '\'
  - The '-' character doesn't need to be escaped other than at the beginning of a term, such as "`\-token`"
  - The '*' character doesn't need to be escaped other than at the end of a term

Class Locations (Package and Binary)

The following tables are a list of locations for the different classes discussed in this document. The locations of these classes in the binary release of Lucene 7.3.0 is also listed.

- *Italic* text refers to interfaces.
- **Bold text** refers to abstract classes.
- Purple text refers to classes referenced, but not explored.

Fundamentals

lucene-core-7.3.0.jar		
org.apache.lucene.analysis		
Analyzer	CharArraySet	**CharFilter**
LowerCaseFilter	StopFilter	**StopwordAnalyzerBase**
TokenFilter	**Tokenizer**	**TokenStream**
org.apache.lucene.analysis.standard		
StandardAnalyzer	StandardFilter	StandardTokenizer
org.apache.lucene.codecs		
FieldsProducer		
org.apache.lucene.document		
Document	DoublePoint	Field
FieldType	FloatPoint	IntPoint
LongPoint	StoredField	StringField
TextField		
org.apache.lucene.index		
Fields	*IndexableField*	*IndexableFieldType*
IndexReader	IndexWriter	IndexWriterConfig
LiveIndexWriterConfig	**PostingsEnum**	StandardDirectoryReader
Term	**Terms**	**TermsEnum**
org.apache.lucene.search		
BooleanClause	BooleanQuery	BoostQuery
FuzzyQuery	IndexSearcher	MultiPhraseQuery
PhraseQuery	PrefixQuery	**Query**
ScoreDoc	TermQuery	TopDocs
WildcardQuery		
org.apache.lucene.store		
DataInput	**DataOutput**	**Directory**
FSDirectory	**IndexInput**	**IndexOutput**
IOContext	RAMDirectory	
org.apache.lucene.util		
AttributeSource	BytesRef	QueryBuilder

Analyzers

lucene-analyzers-common-7.3.0.jar		
org.apache.lucene.analysis.core		
LetterTokenizer	SimpleAnalyzer	UpperCaseFilter
WhitespaceAnalyzer	WhitespaceTokenizer	

Query Parsing

lucene-queryparser-7.3.0.jar		
org.apache.lucene.queryparser.classic		
ParseException	QueryParser	**QueryParserBase**
org.apache.lucene.queryparser.simple		
SimpleQueryParser		

Similarities

Entries in this table that have (...) have children classes that were not explored.

lucene-core-7.3.0.jar		
org.apache.lucene.search.similarities		
AfterEffect (...)	**Axiomatic (...)**	**BasicModel (...)**
BM25Similarity	BooleanSimilarity	ClassicSimilarity
DFISimilarity	DFRSimilarity	**Distribution (...)**
IBSimilarity	**Independence (...)**	**Lambda (...)**
LMSimilarity (...)	MultiSimilarity	**Normalization (...)**
PerFieldSimilarityWrapper	**Similarity**	**SimilarityBase**
TFIDFSimilarity		