Project One

News Indexer – Version 1.0

CSE 435/535: Information Retrieval (Fall 2014)

Table of Contents

Version history

S.no	Version #	Date	Author	Comments
1.	1.0	21 Aug 14	Nikhil L	Initial version

1. Project Description

This project aims to build a News indexer with the following goals:

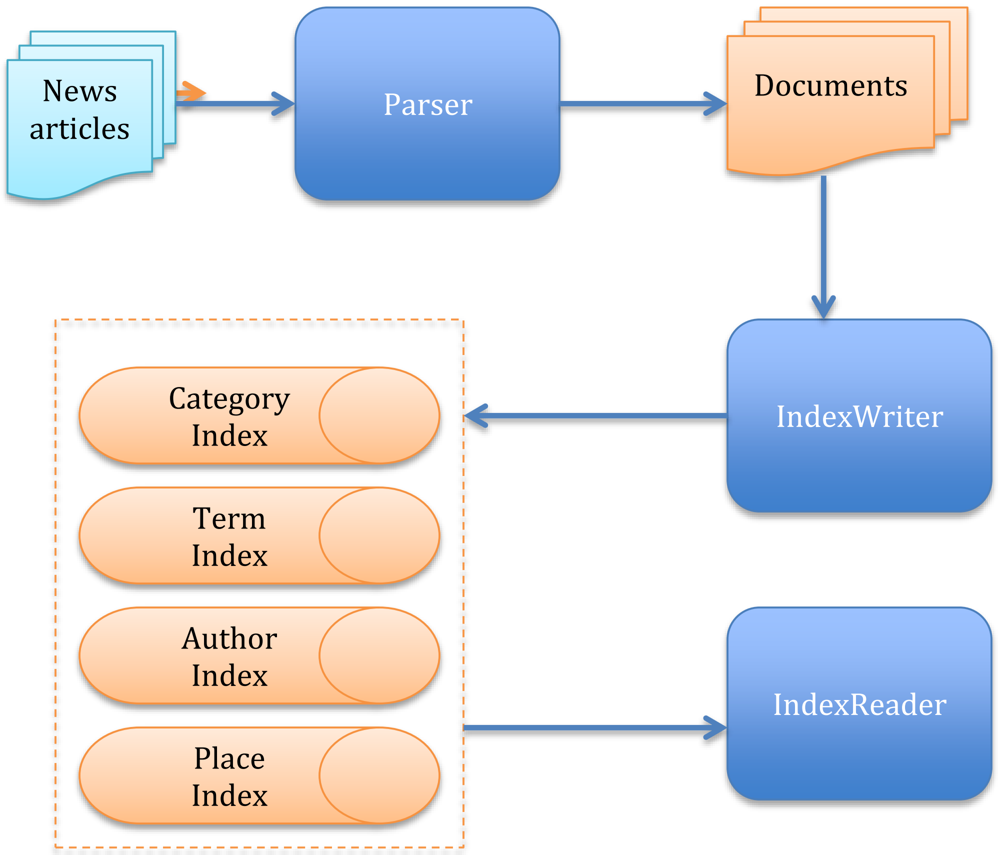
- Parse simple news articles with minimal markup
- Index a decent sized subset of the given news corpus.
- Create multiple indexes on the news articles as well as metadata.
- Provide an index introspection mechanism that can later be built upon to support queries.

You will be implementing this project purely in Java and starter code will be provided with detailed description of each component, method etc.

All deliverables are due by **28th September 2014, 23:59 EST/EDT**

2. System components

The system consists of two main components: a parser and an indexer. An overall system diagram is shown below.



The following subsections describe the functions of each component in more detail.

2.1. Parser

This component is responsible for converting a given text file into a Document representation. A Document is nothing but a collection of fields. Each field can have its own indexing strategy that would be applied by the IndexWriter. More details are provided in the following section.

Each parsed Document must have the following fields except the optional Author related fields. **The field values should be preserved as-is from the text file and no textual transformations must be applied within the parsing stage.**

S. no.	Field Name	Field Description	Example(s)
a.	FileId (FieldNames.FILEID)	The name of the file, also a unique identifier for the document	0001371
b.	Category (FieldNames.CATEGORY)	The category to which the document belongs. Each file is present within the subdirectory of its corresponding category	alum, coffee, cpi, etc.
c.	Title (FieldNames.TITLE)	The title for the document. This is usually the first line of text in all caps before other metadata like place, date or author are mentioned	U.S. COMMERCE'S ORTNER SAYS YEN UNDERVALUED
d.	Author (FieldNames.AUTHOR)	The name(s) of the author(s) of the Document. This is an optional tag and if available, would be present after the title within the <AUTHOR> tag. The format is "By/by/BY <author names separated by and>[, author org]"	Ajoy Sen, BERNICE NAPACH, etc.
e.	AuthorOrg (FieldNames.AUTHORORG)	The organization to which the author of the Document belongs. Refer to above on parsing details.	Reuters
f.	Place (FieldNames.PLACE)	The place from where the news is being reported. Usually present as the first word in the news text (after the author tag if present)	WASHINGTON, TOKYO, etc.
g.	NewsDate (FieldNames.NEWSDATE)	The date on which the given news was reported. Usually present after the place and formatted as MMMMM dd	March 5, April 15 etc.
h.	Content (FieldNames.CONTENT)	The actual news content for the Document. This is the rest of the	Commerce Dept. undersecretary

		article with all metadata removed	of economic affairs….

To reiterate, the Parser is responsible for converting a given file into a Document. Refer to the API documentation in the later sections to find out how to instantiate and use these classes. The Document class is simply a container and has been provided for you. You have to implement the Parser for the given method signatures.

2.2. IndexWriter

Once a given file has been converted into a Document, the IndexWriter is responsible for writing the fields to the corresponding indexes and dictionaries. You are expected to implement the following indexes. Representative snapshots of the indexes and dictionaries follow.

- **Term index**: An index that maps different terms to documents. This is the standard index on which you would perform retrieval.
- **Author index**: An author to document index, stores the different documents written by a given author.
- **Category index**: A category to document index, stores the different documents classified by a given category.
- **Place index**: A place to document index, stores the different documents as referenced by a given place.

Note that the snapshots are only meant for illustration purposes – you are free to implement them with any data structures you may like.

a. **Document dictionary (Optional)**: This contains a Document to document id mapping. This dictionary is optional as you may use the FileId as the document id. However, you may be able to achieve some compression benefits by assigning document ids manually. A snapshot of what this dictionary might look like is given below:

File Id	**Doc id**
…	…
0000524	12
0001210	13
…	

This dictionary if created will be referenced by **all** the indexes.

b. **Term dictionary**: Like above it contains a term to term id mapping as illustrated below

Term	**Term id**
…	…
apple	12

applicat	13
…	…

c. **Term index**: It is a simple inverted index storing the document postings list for each term as follows

1	10,22,65,….
2	17,31,88,…
3	12,16,82,112,….
……	

d. **Author dictionary & author index**: Analogous to term dictionary and index; create a dictionary and index for authors. Note that if a document has multiple authors, the mapping must be stored against each author. The author organization if available, must be stored in the author dictionary.
e. **Category dictionary & category index**: Similar to term and author indexes, create a category dictionary and index.
f. **Place dictionary & place index**: Similar to term, author and category indexes, create a place dictionary and index.

We leave it to the students to decide whether they want to create one large merged index or multiple smaller indexes. However, for each index there should not be more than 27 buckets: one for each character and one more for symbols, special characters and numbers.

You may have observed that some fields (title for example) do not have dedicated indexes. All of these fields should be added to the term index. However, you are free to use any special handling you may like to index them. One approach for example, is to boost the title field.

3. Technical details

This section provides details about the starter code, and corresponding methods that need to be implemented.

Note that you are expected to add appropriate error handling for each TODO method. Our tests will enforce this and would affect your scores.

We have provided two main classes that serve as entry points to the code: Runner and Tester. The former triggers the indexing process and the latter runs all the unit tests.

Runner:

- It simply takes two inputs: the input directory and the index directory.

- It iterates over each file within the input directory, passes it to the Parser to convert it into a Document and then calls the addDocument method on IndexWriter.
- After all files have been processed, it calls the close method

Refer to the IndexWriter section below for more information on the methods.

Tester: TBD

3.1. Parsing

This stage consists of mainly two classes:

3.1.1. Parser

This class is responsible for parsing the given file into a Document instance.

Methods to be implemented:

Document	parse	This is a static method that parses the given file into a fully populated Document instance. If any exception occurs, the method should throw a ParseException.
1.	String filename	This is the name of the news file to be parsed. It is a fully qualified filename

3.1.2. Document

This class is simply a container. The Parser must call the setField method with the correct values. The IndexWriter should call getField to read the values. We would test the Parser code by calling the different getField methods and verifying the values returned.

3.2. Tokenization

This stage is responsible for transforming the given Document class instances into Tokens (technically TokenStream instances but read on). We present the descriptions of all the important classes and the methods that must be implemented.

3.2.1. Token

This is the smallest logical element that would be indexed. At the very least, each Token would have some text associated with it. The termText field as a string and the termBuffer field as a character array represent this.

You are free to add any other information or metadata that you may like with each token. This for example can include positional information or special tokens that mark sentence boundaries for example.

You are expected to implement the following methods in this class:

String	toString	This should return the pure string representation of the token if and only if it should be indexed. The caveat has been added only in case you add special tokens. We would be using this method to verify tokenization and hence, must be implemented correctly.

void	merge	This method should merge the current token with all provided tokens. You should take care of appropriately merging all data structures that you have created. The toString() method after merging should appropriately return the merged text.
1.	Token[] tokens	The tokens to be merged.

3.2.2. TokenStream

This is an iterable stream of Tokens. Refer to the Iterator javadocs for more details about the inherited abstract methods. Apart from the inherited methods, you must implement the following methods:

void	reset	Calling this method should reset the iterator pointer back to the beginning of the stream such that hasNext() returns true and next() returns the first token.

void	append	Calling this method would append the given TokenStream to the end of the current token stream. Calling this method should not affect the stream pointer except if it has reached the end of the stream. If the argument is null or empty, no changes should occur.
1.	TokenStream other	The TokenStream to be appended

Token	getCurrent	Calling this method should return the current token as last returned by next. If remove was called on this token or no further token exists in the stream, return null.

3.2.3. Tokenizer

This class is responsible for converting strings into TokenStream objects. A Tokenizer is instantiated without any arguments or a given delimiter. The former merely implies space-delimited tokenization. Apart from the constructors, you must implement the following method:

TokenStream	consume	This method consumes the given string and converts it into a TokenStream object. The number and nature of Tokens created depends upon the delimiter the Tokenizer has been instantiated with. In case of any error, it must throw a TokenizerException
1.	String str	The string to be consumed

3.2.4. Analyzer

This is an interface that defines operations over TokenStream objects. Multiple Analyzer instances can be chained to allow successive operations on a given TokenStream. There are two levels of usage – either as a single Analyzer as a TokenFilter instance or as a chained Analyzer applicable for a given FieldName. The nuances are explained in the TokenFilterFactory and AnalyzerFactory sections that follow.

In either mode, the following methods must be implemented and as described below:

boolean	increment	Calling this method implies all operations on the current Token should be completed and the underlying stream should be moved to the next Token. The operations to be performed are determined by the constituent Analyzers. If a Token further exists in the stream, return true else return false. In case of any error, it must throw a TokenizerException

TokenStream	getStream	Calling this method would return the underlying TokenStream. This method should be called after the entire TokenStream has been iterated over

		after calling the increment method.

3.2.5. TokenFilter and TokenFilterType

A TokenFilter is a single Analyzer instance denoted by a given TokenFilterType. You must implement one class per type. The type definitions and their expected functionality are as given below:

S.no	TokenFilterType	Description
1.	SYMBOL	It should act on the following symbols with actions as described: • Any punctuation marks that possibly mark the end of a sentence (. ! ?) should be removed. Obviously if the symbol appears within a token it should be retained (a.out for example). • Any possessive apostrophes should be removed (‘s s’ or just ‘ at the end of a word). Common contractions should be replaced with expanded forms but treated as one token. (e.g. should’ve => should have). All other apostrophes should be removed. • If a hyphen occurs within a alphanumeric token it should be retained (B-52, at least one of the two constituents must have a number). If both are alphabetic, it should be replaced with a whitespace and retained as a single token (week-day => week day). Any other hyphens padded by spaces on either or both sides should be removed.
2.	ACCENTS	All accents and diacritics must be removed by folding into the corresponding English characters.
3.	SPECIALCHARS	Any character that is not a alphabet or a number and does not fit the above rules should be removed.
4.	DATES	Any date occurrence should be converted to yyyymmdd format for dates and HH:mm:ss for time stamps (yyyymmdd HH:mm:ss for both combined). Time zones can be ignored. The following defaults should be used if any field is absent: • Year should be set as 1900. • Month should be January • Date should be 1st. • Hour, minute or second should be 00.
5.	NUMBERS	Any number that is not a date should be removed.
6.	CAPITALIZATION	All tokens should be lowercased unless: • The whole word is in caps (AIDS etc.) and the

		whole sentence is not in caps • The word is camel cased and is not the first word in a sentence. • If adjacent tokens satisfy the above rule, they should be combined into a single token (San Francisco, Brad Pitt, etc.)
7.	STEMMER	A stemmer that replaces words with their stemmed versions.
8.	STOPWORDS	A stopword removal rule. It removes tokens that occur in a standard stop list.

All classes you write must use the given constructor. You should hardcode any other information that you may need (a stop word list for example). Although not a great programming practice, it does minimize testing errors on our part.

3.2.6. TokenFilterFactory

You must implement this class, i.e. the two given methods as described below:

TokenFilterFactory	getInstance	A static method to get an instance of the factory class. This method is static so as to facilitate a singleton factory instance.

TokenFilter	getFilterByType	This method must return an instance of the class that implements the given TokenFilterType filter.
1.	TokenFilterType type	The TokenFilterType instance we are interested in.
2.	TokenStream stream	The stream instance to be wrapped in

We would test your individual TokenFilter implementations by instantiating them via the above method. Ideally, you should instantiate them in the same way while building your Analyzer chains. The singleton nature of the factory would allow you to reuse the same TokenFilter instances in multiple chains if needed.

3.2.7. AnalyzerFactory

You must implement this class, i.e. the two given methods as described below:

AnalyzerFactory	getInstance	A static method to get an instance of the factory class. This method is

		static so as to facilitate a singleton factory instance.

Analyzer	getAnalyzerForField	This method must return a chained Analyzer for the given FieldName. Different fields would have different Analyzer chains and may follow different orders. You are free to implement either one single class that implements the chained behavior and return customized instances based on the provided argument or individual classes for each field name. We would be using the methods inherited from the Analyzer interface to test your classes. Thus, you are encouraged to use only these methods when calling them in your indexer code.
1.	FieldNames name	The FieldName for which the Analyzer is requested. We would call this with one of: TERM, PLACE, AUTHOR or CATEGORY.
2.	TokenStream stream	The stream instance to be wrapped in

3.3. Indexing

We provide only a single class as an entry point to the Indexing process, namely the IndexWriter. There are only two methods in this class that serve as entry points, addDocument that adds a document and close that indicates completion of all documents being added. In case of any error, you should throw an IndexerException.

You are expected to implement these methods and add classes as needed. It is expected that you use the AnalyzerFactory and TokenFilterFactory classes while implementing these methods.

Note that your program must NOT write any files outside of the indexing directory provided as an argument to the writer. Make no assumptions about the directory from where your program is invoked. You WILL lose points if your program writes any files outside the given directory even though they may be temporary.

3.4. Index reader

This is the final component. All this class does is give utility methods to read an index (which means all including referenced dictionaries). The different methods are listed below:

- Constructor: It takes two arguments: the index directory and the field on which the index was constructed.
- getTotalKeyTerms and getTotalValueTerms: This is just the size of the underlying dictionaries, the former for the key field and latter is for the value field.
- getPostings: Method to retrieve the postings list for a given term. Apart from the corresponding reverse lookups (for both keys and values), the expected result is only a map with the value field as the key of the map and the number of occurrences as the value of the map.
- getTopK: This returns the key dictionary terms that have the k largest postings list. The return type is expected to be an ordered string in the descending order of result postings, i.e., largest in the first position, and so on.
- query: This emulates an evaluation of a multiterm Boolean AND query on the index. Implement this only for the bonus. The order of the entries in the map is again defined by the cumulative sum of the number of occurrences. What that means is, when evaluating the queries, for every retained postings entry, add its local occurrences to its running count. The value with the maximum occurrences should be at the top.

4. Code testing and evaluation

The project would be evaluated by running unit tests. A majority of the tests will be shipped with the starter code. We would run some additional tests as well as run the code against a larger corpus. Refer grading guidelines for more details.

4.1. Running local tests

For every method that you are expected to implement, corresponding tests will be provided. For a Class C in package P, its corresponding test class would be named CTest and can be found in the package P.test. Each method M would have its test method named as MTest.

We encourage you to write your own tests for code you add as well as any additional tests you think pertinent. We would be evaluating your code with our own test classes, so don't be afraid to experiment.

4.2. Remote evaluation

Once your code has been submitted, we would only use your code files. We do not expect any external libraries to be required for this project. We will ignore any extraneous code submitted and you stand to lose points or worse, not get any if your code does not compile or run.

Reiterating the testing methodology and restrictions:

1. **We will overwrite all classes within the *.test directories , i.e all *Test files. We will also overwrite the Runner and Tester class. Once this is done, we will compile and run your code and tests. If your code does NOT compile, we would not evaluate any further.**
2. **You should not modify the signatures of any TODO methods. We would have corresponding test methods for most of these methods. If we're unable to compile the code (as the test cases would reference this), we will not evaluate any further.**
3. **The signature change also extends to Exceptions. Wrapper Exception classes have been provided for you at each critical stage. You are free to modify these classes as you deem fit. You should not throw any additional exceptions from TODO methods except these.**
4. **Do NOT modify any methods that are called by the Runner class (again as we will use our own Runner). Beyond this, you can add code as you please by creating classes, methods, and fields as required.**

4.3. Grading guidelines

We summarize the scoring methodology as a table below:

S.no.	Functionality	Points
1.	**Code correctness**	**60**
a)	Parser and Document	15
b)	Tokenizer and TokenStream	10
c)	TokenFilter implementations	25
	Symbol	4
	Accents	2
	SpecialChar	3
	Dates	5
	Numbers	3
	Capitalization	3
	Stemmer	3
	StopWords	2
d)	IndexWriter and IndexReader	10
2.	**Index verification**	**40**
a)	Category Index	5
b)	Author Index	10
c)	Term Index	20
d)	Place Index	5
3.	**Bonus (Optional)**	**20**
a)	IndexReader - query	5
b)	Dictionary compression*	5

c)	Postings compression*	5
d)	Fastest 10 teams#	5

We would use automated testing for the entire evaluation (except items marked *, see below). The final score would then also be automatically generated based on the test results. The curve grading would be done at the end of the semester on the total marks.

* : If you have attempted these bonuses, you should email Nikhil to set up a demo
#: Subject to them passing Index verification

5. Submission guidelines

We expect ONLY the source code to be submitted. Please zip your entire source code (the src directory, edu should be the top directory) and name it as cse535_<team name>.zip. Please convert team name to all lowercase on your submission.

To submit, from any cse unix machine (timberlake / metallica / etc.) invoke:

```
submit_cse535 <zip file name>
```

and press enter. You would receive a confirmation message. You can make multiple submissions; any new submissions would overwrite the old ones. In case of name confusions, the latest file will be used.

Late days: The following late day penalties exist –

- 1 day: Lose 10 points
- 2 days: Lose 20 points
- 3 days: Lose 30 points

These will be applied on top of your total score. If you make any submissions after the deadline, it would be considered as a late day is being used.

6. Appendix

6.1. JUnit

JUnit is a simple testing framework for Java. The power of JUnit lies in being simple but allowing extensive tests to be run. It takes very little to set it up. Let's dive right in and take a simple example: we're trying to build a function that prints the factorial of a given number. The class and method skeleton is as given below:

```
public class Factorial {

   /**
    * Method to compute the factorial
    * @param number: The number whose factorial is to be
computed
    * @return the computed value
    */
   public int compute (int number) {
        return 0;
   }
}
```

Now we want to test that this method acts as we expect and that's where JUnit comes in. Let's write the skeleton test code. If you are using Eclipse IDE, you would get a similar skeleton code:

```
public class FactorialTest {
   @Test
   public void testCompute() {
        fail("Not yet implemented");
   }
}
```

We see two interesting things here:

1. We see a "Test" annotation. This was added in JUnit 4.x. What this tells the compiler is that this is a test method. Any method in any class marked with this would be treated as a test method and executed as a test case.
2. The fail(…) method. As the name suggests, if you execute these method, the test case will fail with a message: "Not yet implemented".

So far so good but how do we write tests? Well the library provides the following overloaded functions:

- assertTrue(…): Assert that the given Boolean expression is true
- assertFalse(…): Assert that the given Boolean expression is false.
- assertEquals(…): Assert that the given two objects are equal.
- assertNotEquals(…): Assert that the given two objects are not equal.

Let's see an example and incrementally build our tests. What do we know about the factorial function? We can build a truth table as follows:

Input	Expected Output
0	1
1	1

2	2
3	6
4	24
…	…

These become our test cases and the modified FactorialTest class looks as:

```
public class FactorialTest {
   @Test
   public void testCompute() {
         assertEquals(1, Factorial.compute(0));
         assertEquals(1, Factorial.compute(1));
         assertEquals(2, Factorial.compute(2));
         assertEquals(6, Factorial.compute(3));
         assertEquals(24, Factorial.compute(4));
   }
}
```

Notice how the first input is the expected value and the second is the value returned by the method under test. You just wrote your first test cases within two minutes. It is really as simple as that.

The power of JUnit lies in its flexibility. Almost all kinds of input and output conditions for a method can be tested. JMock, a mocking library to deal with dependent objects, further extends it but you don't need to worry about that.

We have some advanced concepts used as part of the framework:

- TestSuite: Allows grouping test cases into logical groups. We might use this to create groups of test cases based on functionality.
- Paramaterized and Parameters: Allows passing parameters into the test methods without having to hardcode all tests in code. We use this to inject the Properties file into the test classes that need them.

You are encouraged to look at the code and read up on these features. Though you are not required to work with these features for this project, they could be useful to you in the future.

6.2. FAQs

This section answers some frequently asked questions about the project.

1. **What version of Java do I use?**
   We would be testing your code on JDK version 6. If you use this version, there should be no issues. Any previous version would also work and at most throw compilation warnings that we will ignore.

However, if you use a version later than 6, your code may not compile. You could still use a compiler with a later version but you would need to set the target level of compilation to 1.6. If you do not know how to do this, please look up the corresponding documentation for your IDE or seek help from the course staff.

2. **What version of JUnit do I use?**
   Please use Junit version 4. If you use version 3, the test code will not compile.

3. **Can I use an IDE? Is there a preferred IDE?**
   You are free to use any IDE you may like or choose not to use one. We are agnostic to IDE usage.

4. **What about external libraries?**
   We do not anticipate the need for any external libraries except Junit to be needed for this project.

   If you are unsure about a class that you are using, please refer to the javadocs here: http://docs.oracle.com/javase/6/docs/api/. As long as the class you are using is listed on this page or belongs to a package mentioned here, you should be fine.

   Sometimes IDEs sneak in libraries / classes accidentally thanks to their aggressive auto-complete support. When you compile your code, pay close attention to compilation warnings. You may have an unused import that could be using an unsupported library.

5. **How do I execute tests?**
   There are two ways for you to do this. Run the Tester class, it would automatically invoke all the tests within the project.

   Alternatively, you can choose a single class / method and run it as a Junit test. Most IDEs have right-click menus to run tests and provide visual feedback on the execution results.

6. **What am I allowed to change? What should I not change?**
   Remember that the methods marked TODO are our only entry points into your code. Beyond that we have no idea how your code works or what it does. What we expect is that the methods marked TODO behave as documented in the javadocs and/or the specs here.

   What this means is you are free to add classes, methods and data structures as you please. The only thing we ask is don’t change the signatures of the methods marked TODO.

When we test your code, we will overwrite the Runner, Tester and all test classes / cases. So you are free and in fact encouraged to write your own tests.

7. **What about memory considerations? Can I keep the entire index in memory?**
   We are testing your code with a small collection (the one provided to you) only because we want you to be able to code on your personal machines with relative ease and not depend on heavy computing machines. However, your implementations must not make assumptions about the test collection. You should implement your indexer as a real-life system that may be required to index large collections (possibly in gigabytes).

   We would run your code in a JVM with at most 512 MB of memory. So if you keep your entire index in memory either while indexing or reading it, you may run out of memory. In such a case, you would only be allotted marks depending on how far your code could be executed and what we could test. One of the important challenges of this project is designing efficient disk writes. This means you need to figure out when and how frequently to write data to the disk.

8. **What do I write to the disk?**
   Your entire index including the dictionaries must be written to the disk. This must be completed when the close() method on IndexWriter is called.

   Be sure that you understand the implications of what this means. Note that you may have to resort to write small chunks of the index to disk in regular intervals. However, the indexes are upper bounded by the number of chunks they have, thus, some merging may be needed to ensure the bucket constraint. All of this must be completed on the call of the close method.

9. **What is a key dictionary and what is a value dictionary?**
   Any (reverse) index is ultimately a mapping that stores a postings list against an index term. For example, the term index stores the mapping between a term to its postings list i.e. documents where it occurs. Now instead of storing the raw term and the raw document name in the index, they could be referenced by ids from the respective dictionary.

   So in this case, the term dictionary becomes the “key dictionary” and the document dictionary is the “value dictionary”. As is evident, some dictionaries would be shared between indexes. Despite this, calling getTotalKeyTerms() or getTotalValueTerms() on this dictionary should return the same number irrespective on what index it is called on.

10. **Is it mandatory to create dictionaries?**
    No.

11. **What is term boosting?**
    Term boosting refers to how much a term contributes to the relevancy score of a document for a given query. It is possible that some terms are more important than others, or their importance is determined by where they occur in the document. Hence, in theory, you could associate different weights to terms. It is common practice to store such term weights, if known or pre-determined, in the index. This allows simple query processing. However, you could always apply boosts during query processing.

    We know that some of these concepts may not be covered in class until the project is due. Hence, this is optional.

12. **Can I collaborate with other teams or students?**
    A simple way to answer this is – you can discuss with other teams in terms of concepts but not in terms of code. We use software to detect code duplication. Even though this may have been inadvertently done on your part – innocuously showing or sharing code or discuss enough details to induce similarity – we treat this as plagiarism and would mean a 0 on the project.

    It is your duty to protect your code by all means. Note that this also means you should not be "inspired" by the code submitted by previous batches that may be available to you for "reference" purposes.