Web Information Retrieval (67782)
Ex1: Index Structure

1 Data Set

In the exercises throughout the semester, you will build a search engine for product reviews. We will be using product review data, such as that available for "Fine Foods", "Movies" etc., in the Stanford Large Network Dataset Collection (`http://snap.stanford.edu/data/index.html`). We will also use input data of our own of the same format.

In particular, product reviews have the format:

```
product/productId: B001E4KFG0
review/userId: A3SGXH7AUHU8GW
review/profileName: delmartian
review/helpfulness: 1/1
review/score: 5.0
review/time: 1303862400
review/summary: Good Quality Dog Food
review/text: I have bought several of the Vitality canned dog food products and
have found them all to be of good quality. The product looks more like a stew
than a processed meat and it smells better. My Labrador is finicky and she
appreciates this product better than most.
```

Note that this is real data, and hence, may have some strange contents. For example, you may find new lines in the middle of a profile name, or extremely long words.

Since product reviews do not have associated identifiers, we will using a running counter to identify reviews, i.e., the first review in the input file is review 1, the second is review 2, and so on. Product reviews have a variety of data. However, we will only be interested in storing the following fields:

- product/productId
- review/helpfulness (two integers)
- review/score (integer between 1 and 5)
- review/text

When storing the text of a review, you should

1. break it into separate tokens (i.e., words) at every character that is not alphanumeric. Alphanumeric characters are lowercase letters, uppercase letters and digits. Note that non-alphanumeric characters are discarded. This may seem strange at times, e.g., the word "don't" will be two tokens: "don" and "t". However, it makes the processing simpler.
2. normalize the text by transforming all words into lower case.

To help you get started, there are two small datasets (one with 100 reviews and one with 1000 reviews) available on the course homepage. Larger datasets will also be made available later on.

2 Exercise Description

Given the raw input of a product review file, in this exercise, you will create indices which will allow efficient access to the data. Indices should be stored on disk, and used later when examining the product data. Thus, index files should remain on disk even when your program is no longer running.

The specific implementations of the indices are part of your design decisions. Implementing indices for textual data is discussed extensively during the first few weeks of the course. Your job is to build on these ideas (or propose your own), to index product review data. You must be able to analyze the expected size of the indices for different input sizes.[1] You should use some types of compression techniques to ensure that the indices are of reasonable sizes. Make sure that you focus the majority of your compression efforts on data that is likely to grow very large.

Some restrictions apply:

- First, you cannot use a database system for storing your data. Rather, you must implement the storage on your own.
- Second, you cannot use the *default* serialize and de-serialize implementations of Java objects (i.e., the `writeObject` and `readObject`) methods, as they will not allow you fine-grained tuning and analysis of the data storage. (You may, of course, override these methods).
- You can use multiple files to store the index. However, the number of files created should be constant—and not dependent on the number of reviews or size of the dictionary, etc.
- After creating the index, you should be able to discard the raw review data, i.e., all information needed for your methods should be within the index you create.

In this first exercise, you do not have to worry about making the index loading process efficient. (That will be your second exercise.) However, **your index design and implementation must be scalable enough to be able to store huge amounts of data** so that it can be used as is, once you have the capability to load large amounts of data.

2.1 Code Requirements

Given product review data, you will submit a program containing at least the following two classes: (you will most probably submit many additional classes, needed to implement these)

[1] Estimating index sizes is also discussed in this course.

1. SlowIndexWriter: Given raw review data, this class will create an on disk index that will allow access later on. All data that will be used later on must be written to disk in an index structure. Note the word "slow" in the class name. This indicates that the program constructing the index can be inefficient at this point. Note that for this exercise, *when loading the data* into the index, you can assume that the entire input fits easily in main memory. (Of course, however, you must store it efficiently on disk.) This class also allows an index to be removed from disk by deleting all index files.

2. IndexReader: After an index has been created on disk, the class IndexReader can be used to access many different types of information available in the index. You should use these operations as a guide in designing your index structure, i.e., your design should support efficient implementation of these methods. You can assume that all of these methods will be called only after an index has been created with SlowIndexWriter. These operations should be implemented in a manner that they will be efficient even when the index contains huge amounts of data.

   Note that the IndexReader can create in-memory data structures for the smaller portions of the index. However, there will not be enough memory for the entire index to be loaded to main memory.

Both of the above classes should be implemented within a package named `webdata`.

A description of the methods that must be implemented appear on the upcoming pages. (Note that after the description of the classes, this document also contains information about a written analysis that you will submit with this exercise. So, read this document until the end.)

```
package webdata;
public class SlowIndexWriter {

    /**
     * Given product review data, creates an on disk index
     * inputFile is the path to the file containing the review data
     * dir is the directory in which all index files will be created
     *     if the directory does not exist, it should be created
     */
    public void slowWrite(String inputFile, String dir) {}

   /**
     * Delete all index files by removing the given directory
     */
    public void removeIndex(String dir) {}
}

package webdata;
public class IndexReader {

    /**
     * Creates an IndexReader which will read from the given directory
```

```
 */
public IndexReader(String dir) {}

/**
 * Returns the product identifier for the given review
 * Returns null if there is no review with the given identifier
 */
public String getProductId(int reviewId) {}

/**
 * Returns the score for a given review
 * Returns -1 if there is no review with the given identifier
 */
public int getReviewScore(int reviewId) {}

/**
 * Returns the numerator for the helpfulness of a given review
 * Returns -1 if there is no review with the given identifier
 */
public int getReviewHelpfulnessNumerator(int reviewId) {}

/**
 * Returns the denominator for the helpfulness of a given review
 * Returns -1 if there is no review with the given identifier
 */
public int getReviewHelpfulnessDenominator(int reviewId) {}

/**
 * Returns the number of tokens in a given review
 * Returns -1 if there is no review with the given identifier
 */
public int getReviewLength(int reviewId) {}

/**
 * Return the number of reviews containing a given token (i.e., word)
 * Returns 0 if there are no reviews containing this token
 */
public int getTokenFrequency(String token) {}

/**
 * Return the number of times that a given token (i.e., word) appears in
 * the reviews indexed
 * Returns 0 if there are no reviews containing this token
 */
public int getTokenCollectionFrequency(String token) {}
```

```
    /**
     * Return a series of integers of the form id-1, freq-1, id-2, freq-2, ... such
     * that id-n is the n-th review containing the given token and freq-n is the
     * number of times that the token appears in review id-n
     * Only return ids of reviews that include the token
     * Note that the integers should be sorted by id
     *
     * Returns an empty Enumeration if there are no reviews containing this token
     * /
    public Enumeration<Integer> getReviewsWithToken(String token) {}

    /**
     * Return the number of product reviews available in the system
     */
    public int getNumberOfReviews() {}

    /**
     * Return the number of number of tokens in the system
     * (Tokens should be counted as many times as they appear)
     */
    public int getTokenSizeOfReviews() {}

    /**
     * Return the ids of the reviews for a given product identifier
     * Note that the integers returned should be sorted by id
     *
     * Returns an empty Enumeration if there are no reviews for this product
     */
    public Enumeration<Integer> getProductReviews(String productId) {}

}
```

2.2 Analysis Requirements

In addition to the code, you should give in an analysis of your index structure. In particular:

- Discuss the precise details behind the index structure that you have implemented. Your discussion should be very specific and should allow the reader to precisely understand the format of your index files, stored on disk. Provide a diagram that depicts the structure of the index.
- Explain which portions of the index are read into memory when an IndexReader object is created, and which portions will be read as needed.
- Theoretically analyze the expected size (in bytes) of the index. In your analysis, the size of

the index should be a function of the size of the input. Use the following variables to denote the various input size parameters:

N Number of reviews

M Total number of tokens (counting duplicates as many times as they appear)

D Number of different tokens (counting duplicates once)

L Average token length (counting each token once)

F Average token frequency, i.e., number of reviews containing a token

You can add additional variables as needed.

- Using the same variables, theoretically analyze the runtime of the functions that you implemented.

We have provided a template of your analysis in both word and tex formats. Fill in one of the templates, and save it as a pdf file, named analysis.pdf, to submit.

2.3 Testing

When testing your exercise, we will use small datasets, of approximately the size of those available on the course homepage. Your exercise should run correctly both on Linux and on Windows systems.

2.4 Exercise Submission

The code for your exercise should be submitted in a jar format via the course website. Only one submission should be made for each pair! Please make sure that the jar file uploaded is in the correct format, and contains all relevant files, including: all source files, a README (including any compiling or running issues, and the logins and teudat zehut numbers of the students submitting the project), and the file `analysis.pdf` containing your system analysis.

2.5 Frequently Asked Questions

What assumptions can we make about the data? You can make any assumptions that you wish, as long as they allow you to index all data you are given. We will not be creating "fake" data to test your system, so as long as the real data can be indexed, it is fine. When parsing the file, you should *not assume* that the only fields present in the reviews are those appearing in the small files, as there can be additional fields in the larger files. However, throughout the course, we will only be interested in storing the fields productId, helpfulness, score and text.

What if I don't know what may happen in the real world data? That is typical to a real world setting. My recommendation is to start by making assumptions that are likely to hold, and trying to work with them. If needed, you will adapt your program if you find that your assumptions do not hold.

While we could make your life easier by pre-processing all the real data, and "normalizing it" to make sure that there are no odd things, we are not doing this. Part of the exercise in this course is learning to deal with real world data. This indeed can be annoying, since it is difficult to know what will happen beforehand. You have to write a program that can degrade, or fail helpfully, in the face of unexpected inputs.

While parsing the data, can I use a data structure that may require significant memory? Yes! In this exercise, your data will easily fit into main memory. In the next exercise, you will no longer be able to assume that the data can fit into main memory.

Do we have to store the text of the reviews, i.e., be able to reconstruct them from the index? No, you do not have to reconstruct the reviews. You should store only the data needed to implement all functionality required by this exercise.

Is ... a good design idea? You are making the design decisions. Try to make sure to focus most of your effort on compressing the data that is more likely to be very large, and on efficient implementation of the functions, by avoiding scanning of large amounts of data. There is an inherent trade-off between compression and execution time (due to the need to decompress), so you will have to make hard choices.

Are the reviews are ordered by product id, meaning that reviews for the same product are consecutive? Is it guaranteed to stay that way? Yes.