CMPUT 291 Mini-Project 2 - Design Document:

Anson Li (1354766) and Satyen Akolkar (1353583)

General Overview

The application created for mini-project 2 is a BerkeleyDB database query. The following tasks are available for the application:

1. When provided a data text with a formatted list of products and reviews, the application can convert the text file into four text files - score.txt, reviews.txt, pterms.txt, and rterms.txt. These four text files form the basis of our database development.
2. The aforementioned text files are then cleaned and converted to .idx values for database querying.
3. The query tool can process the .idx values via BerkeleyDB, with the following functionalities:
    a. Process the aforementioned queries case insensitive and space insensitive (for relational queries).
    b. r:____ : Query the review summary and text for the selected word (must be 3 characters or more).
    c. p:____ : Query the product name for the selected word (must be 3 characters or more).
    d. _____ : Query the product name, review summary and text for the selected word.
    e. _____% : Query the product name, review summary and text for entries that begin with the input text.
    f. rscore </=/> INT : Find all reviews with a score less than / equal to / greater than the integer value.
    g. rdate </=/> DATE : Find all reviews that begin before / at / after the date selected.
    h. pprice </=/> PRICE : Find all products that have a price less than / equal to / greater than the input price (input as XX.XX).
    i. quit() : Exits the application.

User Guide

1. Parsing of text file and generation of four main text files.
Before you begin to use Berkeley DB, please enter the following commands into the terminal to instantiate the paths:

```
export CLASSPATH=$CLASSPATH:.:/usr/share/java/db.jar
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/oracle/lib
rm -f *.idx pterms.txt reviews.txt rterms.txt scores.txt
```

These ensure that the proper libaries are referenced for Berkeley DB production.
Once you have setup the correct paths, enter the /src folder. Then, simply compile every related function:

```
javac datastructs/Review.java datastructs/Product.java datastructs/GenericStack.java
datastructs/Query.java datastructs/StringEntry.java exceptions/DBMSException.java
indexer/IndexGen.java parser/DataFileGenerator.java parser/parser.java querier/DBMS.java
querier/QueryRunner.java
```

Then, run the first program with:

```
cat data.txt | java parser.parser
```

This results in the generation and the completion of the four main textfiles: reviews.txt, pterms.txt, rterms.txt, and scores.txt. Please replace 'data.txt' with the text file you are interested in using!
To compare the file to the sample on eClass, run the following where the eclass version is (review_eClass.txt, pterms_eClass.txt ...)

```
diff reviews_eClass.txt reviews.txt
```

If there is any output, the two files do not match. If there is no output then the files are identical.

2. Building indexes
Create the indexes required for part three using the aforementioned program:
```
        java indexer.IndexGen
```

Once this is run, the .idx values for all files will be generated.

3. Running queries
Once you have built the indexes, all that's left is running the query system. That can be done by:
```
        java querier.QueryRunner
```

For a list of available functions, please consult the list as described in the general overview.

Design of Software Components:
The first two components of the project were created via a combination of bash processes integrated into Java, as well as generic data text parsing methods.

In order to create the algorithms used in the query operations, we followed the following methods:

Generic queries:
For all generic queries, the key was provided with the data that was required to be searched. Then, the database and the respective cursor was generated to handle the query. Once the query was processed, if the data corresponding to the key was found, then the data was returned with the correct value. This is the most optimal method of finding generic key matches, and was used for the majority of queries, including finding score, and finding exact key matches in instances such as p: and r:.

Evaluating queries with multiple conditions:
The algorithm for processing queries was originally developed with the expectation that there were an infinite amount of conditions that a query could process, and they had different priorities. High priority conditions mean that they could operate on their own (examples are p:, r: and rscore); low priority conditions mean that they could only operate if paired with at least one high priority condition (examples are pprice, rdate). By separating these conditions, we could run the high priority conditions first, then move on to the low conditions. This ensured that we had proper condition processing flow.

Additionally, within the high priority conditions, we had to ensure that there was a correct process flow for handling the received data. Ultimately, we decided on the following logic: on the first condition, store all the correct returned values presented. On each of the following queries, if there is no match for an item in the original list, remove it. This ensures that only values that match ‘across the board’ are output.

Partial matches
When interpreting the ‘wildcard’ process, there were a number of trial algorithms developed before the final process was decided - these included iterating over every database entry via BerkeleyDB in order to pull every value that matched. However, all processes that were iterated upon were too slow for processing, and could not be used for the final product. Eventually, we decided that the query would be completed in two parts. The first part would be to grab every unique character that matched the wildcard condition that was identified - this was completed using ‘grep’ on the text file that was used to generate the respective .idx, and sorting all the unique values that existed within for processing. Once we stored all the values in the application, we ran the generic query on each unique entry that we received and combined all the unique results to get the final product.

Range searches

Range searches were generated depending on the which 'range search' was required. For rscore, range searches was structured similar to a generic query, but was completed in multiple steps. Because we knew that the limit for score was between 0 and 5, we could use these boundaries to iterate over the database to find the values we required. The final design involved an iterator that queried the database for the whole range implied in the database design, then pulling all data points that matched the requirements. For pprice and rdate, the conditional search was completed after all the products and reviews from the high priority conditions were generated. Once they were set, we used java conditional logic to parse the date and price of the respective review and product, and output the values that were correct.

Testing Strategies

Various testing functionalities were used to ensure stability and consistency with respect to BerkeleyDB queries and UI interface.

One testing strategy we implemented is creating a process flow graph. Before creating any code that would form our query language, our team decided on a process flow for the application that helped us parse the queries up to an infinite size. We ultimately designed the three-stack method as described in the design of software components section as a result of this method. Another benefit of doing so was that testing process flow became a lot easier - as there was a predetermined 'roadmap' to follow, developing the software components at each iteration required a lot less overhead.

Another strategy we used to test the application was through soft unit testing. As developers, we have a clear view of the basic I/O of the software for all functions. Using this knowledge, we can test the functionalities using predetermined inputs, and physically verify each output as correct or incorrect on the terminal. This method was helpful for testing the 10 sample queries provided on the project specifications page on multiple data sets (including the 100k tables).

One last method of testing that we used is peer code review. Our team's development strategy was to produce code for each module separately, then integrate the code (pull request) and examine whether or not it is consistent with the team's work (as well as the project requirements). This strategy is useful for two purposes: first, at a cursory glance, it can be clear in some cases if a teammate's code is not consistent with the rest of the work. As a result, clear issues in the code can be easily identified and mitigated. Second, when testing the functionalities, the teammate who did not work on the code is viewing the functionality for the first time, or with 'fresh eyes'; the teammate is compelled to use their testing overlook standard techniques in testing the methods. As a result, using peer code review was a significant benefit to our project, as we were then able to ensure that the program is consistent and largely error-free.

Group Work Strategy

Our group work strategy was initially to handle the code development individually by separating the development of each module, then combining the modules and refactor the code together. The code development, for each module:

Anson's Tasks	Time Spent (Hr)	Satyen's Tasks	Time Spent (Hr)
Part One (converting to four text files)	5	Part One (converting to four text files)	3
Part Two (parsing to .idx)	16	Part Two (parsing to .idx)	5
Part Three (generating query)	3	Part Three (generating query)	8
Refactoring part three (rewrite query)	8	Refactoring part three (rewrite query)	10
Design Documentation	3	Design Documentation	5

Refactoring	2	Refactoring	8

The method of coordination used to keep the project on track was frequent use of Google Hangouts, as well as multiple face-to-face meetings in order to confirm project schedules and soft development deadlines. No code completed was out-of-specifications.