**Western Washington University – CSCI Department**

**CSCI 330 Database Systems**

# SURLY { II } Report

Daniel Lorigan Jason Ruffin

## Who is on your team and what's the division of labor?

In SURLY II, Jason was chosen to complete the conditionsParser.java for the SELECT/DELETE WHERE clauses. He then debugged it using (and improving) the DELETE WHERE function. Daniel implemented the DESTROY, JOIN, PROJECT, and SELECT WHERE functions. He then wrote the majority of the error handling in the late stages of the project, greatly increasing the durability of the code.

## What programming language did you select and why?

We decided to use java as we were both comfortable with the language and because it is an object oriented language and assignment. Familiarity with the syntax made it so that we could focus on the conceptual ideas behind the project instead of getting stuck on basics. It also allowed us to go above the requirements of the project, implementing functionalities such as command line inputs and saving database states.

## List libraries or programming language features you made use of?

* Java.io - Imported for user input and saving/reading the database state to/from a file
  + Database, Relation, Tuple and Attribute implement java.io.serializable for the purpose of saving the state of those objects.
* Java.util - Imported for LinkedList, HashMap and iterators
* Java.lang - Imported for errors, printing, etc.
* Java.util.regex - Used to format commands entered by the user. Checks for instances of ‘…’, where the string inside the quotes should be interpreted as one argument no matter its white spaces, replaces the white spaces with a ‘]+’ character, and then replaces those with spaces again once the string is split correctly.

## Deliverables

|  |  |
| --- | --- |
| **Checklist of deliverables** |  |
| Hardcopy of | I/II/III |
| This writeup | x |
|  |  |
| Zip file containing | I/II/III |
| This writeup | x |
| Test cases showing input/output | x |
| Source code | x |
| README.TXT \* | x |

## Coverage - Did you complete all of SURLY Part II - what is missing?

|  |  |  |
| --- | --- | --- |
| **version** | **Feature** | **Covered/Comment** |
| I | Relation | Covered |
| I | Insert | Covered |
| I | Print | Covered |
| I | Heap Storage | Covered |
| II | Destroy | Covered |
| II | Delete where … AND/OR | Covered |
| II | Select where … AND/OR | Covered |
| optional | Join, Project, Import/Export in XML, CATALOG, GUI, … | Covered Join, Project, Catalog (as ‘PRINT’ with no arguments), file import/export (binary), help functionality for each input |

## How did you implement

* **Relations** – These are objects that hold their name and a linked list of Tuple objects. Each instance also contains a flag determining if it is to be treated by Surly as a temporary or permanent relation.
* **Tuples** – These are objects that hold a linked list of Attribute objects.
* **Attributes** – These are objects that contain fields for metadata like the attribute name, type (NUM or CHAR), max size, and also a columnWidth field for the purposes of cleaner printing. These attributes are kept consistent across all tuples in a relation.
* **Insert** – Finds a specified relation and adds to it a tuple made up of the specified values, as they are ordered in the relation’s schema, by copying the base (first and null-valued) tuple of the relation, and then setting the copy’s value fields to those specified in the command arguments. It then adds this new tuple to the specified relation.
* **Destroy –** Destroy takes two arguments, a string for the relation’s name, and a Boolean for whether or not this call of destroy should work on temporary relations. We created a “findRelation(name)” function in our database class to parse through all existing relations and temp relations to find one with the given name. After finding the relation, we remove that relation from the main linked list of relations, as long as it is either a permanent relation, or an instance of destroy that is allowed to destroy temporary relations.
* **Delete where** - After the relation is found, we create a temporary list of tuples. If there are no conditions, then we add the “header” tuple to this arraylist and create a relation with just this one element in the list, if there are conditions however, we evaluate each tuple with the condionParser to check if the tuple meets the conditions. If it doesn’t, it is added to the temporary list of tuples. When all tuples have been evaluated, the relation is replaced with the temporary list. This function cannot be executed on temporary relations.
* **Select where** - After finding the specified relation, we create a temporary list of tuples. If there are no conditions, we set the list of tuples equal to the tuples in the relation and then make a temporary relation based off of these values. If there are conditions however, we send each tuple to the condition parser, and each one that meets the conditions is added to a new temporary relation with the name that the user specified.
* **Evaluation of conditions** – This functionality is vital to the delete where and select where functions when there are conditions specified to filter out tuples. It first creates two arraylists, one made up of boolean values, derived from evaluating the tuple against the conditions, and the other made up of string operators (“and” and “or”) which are parsed straight from the conditions. It then calls a recursive function to first execute all the “and” operators with their respective boolean values as the “and” operator has priority of execution. With the remaining boolean values and remaining “or” operators, if there is a single True boolean value, simply return True for that one tuple. This function can parse ‘=’, ‘!=’, ‘>’, ‘<’, ‘>=’, ‘<=’, ‘or’, and ‘and’ operators. It will fail if there are omitted spaces between operators and operands, if operators are typed incorrectly, and if there are mismatched operators or operands.
* **Join** – The two specified relations are found in the database. Then, the specified attributes in the join condition are found within their respective relation (the left value is looked for in the first relation, the right value is looked for in the second), and their respective index in the list of attributes is saved to speed up later comparisons. Each of the first relations tuples are paired with the second’s, and the join condition is checked between the two tuples. If the condition is met, then the tuples are combined as one large tuple and put in a new list of tuples. If no conditions are given, then the Cartesian product of the two relations is stored. This is then used to create into a temporary relation called the specified name.
* **Project –** The relation is located in the database, and a new list of attributes is created using the “header tuple” which only contains the attribute metadata (no values). This tuple is iterated through, and the indices of all specified attributes are stored for later use and faster searching. Each tuple in the relation is iterated through, and a new tuple is made from the values that match the previously recorded indices and stored in a new list of tuples. This list of tuples is used to create a temporary relation called the specified name.
* **Print –** The specified relations are found in the database, then printed using Relation.print(), which formats the table and calls Tuple.print(), which calls Attribute.print(), thus printing all values of the relation in neat columns under both the attribute headers and the relation’s name. If there are no relations specified, the function prints the database CATALOG, a summary of each of the relations’ schemas in the same format used when calling RELATION. This lists permanent relations first and temporary relations last.
* **Help –** When called with no parameters, a full list of all implemented commands is printed, each with a short description and a usage format. Typing ‘help’ followed by one or more command names will return the specified functions’ help messages. This function is also called when a formatting error is detected upon parsing, to help the user type commands correctly.

## Things you did differently (e.g., than the SURLY spec)

### Limitations of the current release.

This release handles most syntax errors and possibilities inside the scope of the commands. Join only implements join conditions based on two attributes, due to equivalence being the most common type of condition

### Extra features you added - e.g., going beyond the SURLY I/II spec

* Command line input: when surly is run, input is prompted until ‘exit’ is entered. While running, the user can enter commands in surly syntax, or other meta commands detailed both in the applications ‘help’ information and this document.
* Input: Entering ‘INPUT file.txt’ runs any commands specified in file.txt as if the file was run as one of the command line args.
* Save: Entering ‘SAVEAS file’ will serialize the current database object as binary data and store it in a file named ‘file.sur’.
* Load: Entering ‘LOAD file’ will load the specified file, as long as it ends in ‘.sur’.
* Catalog print (detailed above – See ‘Print’)
* Help (detailed above – See ‘Help’)

### Things you are especially proud of

We worked very hard on error handling, making sure to check for multiple possible input errors and possible user mistakes, as well as internal ones. The majority of our release of SURLY is case-insensitive. This includes input commands, relation titles and attribute titles, but not value comparisons to provide more accurate data storage. Implementing the extra functionalities mentioned above was not particularly difficult, but added greatly to the user experience, which was a priority for us. Lastly, the condition parser class was a challenge, and the implementation that we used was an interesting and original method that was optimized within our abilities, and utilizes a short circuit functionality on ‘OR’ operators.

## Recommendations

### Things you would do differently if starting over now.

We would have done more planning at the beginning of the project in order to better organize our classes functionality-wise (Database.java and SurlyParser.java became fairly complex classes), which would improve organization down the road and make changes easier to implement.

### Did SURLY meet your objectives for this course?

Creating a database management program was a very good way to get a handle on how other DBMSs might work. This project was a perfect assignment to achieve this, including new information we learned from this class and building on our previous coding experience. However, although SURLY helped us to appreciate the challenges of implementing these functions, it could have been interesting to discuss widely used DBMS implementations, and how those systems achieve such high optimization (underlying structures, strategies used to achieve relational algebra functions, etc.). The assignment was very fun to work on due to its applications, usefulness, and our freedom to add functionality.

### Suggestions on how to improve SURLY I/II assignment?

I think that touching on concepts of structuring/organizing code would have been helpful for the transition between part 1 and 2, as it seems like many groups had some issues with that aspect of the program, and it was large enough for that to make a difference. Some suggestions for improving code style about part 1 would have improved part 2’s implementation.

### Suggestions on how to improve the course?

The course was overall very interesting and useful. Highlighting professional experiences as examples, highlighting the application of concepts in real-life database scenarios, and providing lots of examples and practice was all very helpful. It felt like more information could have been covered in the time that we had, but I felt that what we did learn was certainly helpful enough to gain a solid understanding of database systems.

### Any other comments?

Not at the moment.