Western Washington University – CSCI Department
CSCI 330 Database Systems

SURLY II Report

Martin Smith, Eric Anderson

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

Martin Smith, Eric Anderson

We divided work a little more this time. Martin got most of the JOIN logic implemented, Eric did PROJECT. We figured out the WHERE clause and temp relation logic together. We were still using Martin's original base file, so he typically created new Java classes. Eric formatted design, test cases, etc. Overall, at least half of the work was paired.

What programming language did you select and why?

SURLY1 was coded in Java, and we are both the most familiar with Java. So we continued using Java for SURLYII.

List libraries or programming language features you made use of?

Similar to before, only some of the standard packages: java.util, java.io.

Java.util package handles all the LinkedLists within the codes (of which there are many), and java.io handles the couple of I/O exceptions. We frequently called size and add methods. Occasionally we used the Integer and Character objects for type validation.

Deliverables

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

- * include at top level a file titled README.TXT that provides *Installation and Demo Instructions* containing instructions on how to install and demo your program

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

version	Feature	Covered/Comment
I	Relation	Completed last month
I	Insert	Completed last month
I	Print	Completed last month
I	Heap Storage	In effect
I	Catalog	Completed last month
I	Destroy	Completed last month
II	Delete where … AND/OR	Completed, including logical operators
II	Select where … AND/OR	Completed, including logical operators
II	Project	Complete
II	Join	Complete, including qualify attributes
other	Import/Export, GUI, …	Not required for SURLY II

How did you implement

- **Relations** – Used RelationParser.java to retrieve the name of the relation and the schema, which was then stored into the SURLY database for each Relation created.
- **Tuples –** Within Relation.java, tuples are saved within a LinkedList of Tuples object, which in Tuple.java is declared to be a LinkedList of AttributeValues, which consist the value being inputted in the relation table and the relation that it's being inserted into.
- **Attributes** – Within Relation.java, the attributes of a relation are saved within a LinkedList of Attributes objects, which consists of the name, dataType, and Length. The Attribute for each Relation is created within RelationParser.java, where the Attributes are consistently being added to the LinkedList if it fits the valid schema within the command lines.
- **Insert –** Used InsertParser.java to retrieve the name of the relation that the tuple will be sorted into, and if it fit the schema constrictions the tuple would then be inserted into that relation within the SURLY database.
- **Catalog –** Used SurlyDatabase.java, when the SurlyDatabase object would be initialized that was when we created the Relation for Catalog, and when each new relation was created within the SurlyDatabase (within the RELATION command), a Tuple containting the Relation name and the Schema size would then be inserted into Catalog. When a Relation

was Destroyed, within the DestroyRelation command within SurlyDatabase.java we invoke a method call deleteTuple upon Catalog, which deletes the specified Tuple from the Catalog.

- **Destroy** – Used DestroyParser.java to retrieve the name of the relation being destroyed, which then would be invoking the destroyRelation method that is contained within SurlyDatabase.java, which invokes a remove call on the LinkedList of Relations, which removes the Relation permanently from the list.

- **Delete where** – Checks to see if the syntax is valid. If so, then the conditions are created within DeleteParser in the object AllConditions, which will evaluate each ConditionList (separated by 'or' statements), which within each ConditionList(conditions within an 'and' statement), a Condition is evaluated. The list of Tuples are then returned, then deleted from the relation desired. If there is no 'WHERE' in delete, then it would delete all the tuples from the specified relations. Note that if there is a 'WHERE' in the statement, then it only deletes from ONE relation.
- **Select where** – Creates a new temporary relation with the new name. Adds the schema of the selected relation to the new relation. If the syntax includes a 'where', then we'll create and evaluates the conditions like how the conditions are created and evaluated within Delete where, where it returns the list of Tuples, and adds to new Relation. If there was no 'where', then the Tuples from the selected Relation will be added to the new Relation. Relation is returned, and then added to the database.
- **Project** – Similar logic as Select Where. We created ProjectParser.java, which most importantly contains addInfo() and addRelation(). AddInfo() double checks if the inputted attribute/relation info matches the requested base relation, and creates a new schema, adding it to the temp relation based on the specified attributes. AddRelation parses the tuples in the order specified and builds a new set of attributes/tuples. A check is then done for each tuple to ensure that it's not a duplicate of the others. If that check succeeds, the said tuple will be inputted into the temporary Relation. Temporary Relation is then returned, and added to the database.
- **Join –** Creates a new temporary relation with the new name. Grabs the two Relations that are desired to be joined (if they exist). After evaluating the two Relations both have the Attributes that are declared in the statement, then it will create the new Schema and Tuples for that relation, not adding the column that matched from the Relation on the right. Then it returns the temporary Relation, and adds it to the database.
- **Temp Relations** – Very simple implementation – we added a boolean isTemp variable to the Relation class that is set true if the tempBuff() function is called. Various conditionals in our LexicalAnalyzer quickly check if true or false via a getter function. Often if it is true, code is skipped (i.e. won't destroy/delete/insert into a temp relation). The case where the temporary relation overriding a relation requires the relation it's overriding, the overwritten relation (that has been saved just for this occasion) is then be passed into the specified parsers that require it.

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

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

Created a Parser class that does all the parsing for the Parser classes (RelationParser, InsertParser, DeleteParser, PrintParser, DestroyParser), so that there wouldn't be redundant code within the Parsers.

Things you are especially proud of:

The organization of all of the class files! There's a lot of them.

Recommendations

Things you would do differently if starting over now.

Probably come up with our own UML first, since a lot of the general methods require a lot of smaller methods and we felt constricted when implementing the code (hence the design and long methods).

What did you learn about databases from SURLY?

Databases require a lot of LinkedLists to work properly, at least in the scope of this course. A lot of conditionals and error exceptions are necessary to make the program more ergonomic, which can be computationally expensive if not done properly.

Any other comments?

Thanks for giving us a meaningful, progressive paired assignment. It helped us see part of what is required in the workplace.