Eli Jukanovich, Thomas Jones-Moore, Megan Hong

SURLY I Report

Who was on your team and what was the division of labor?

- Eli Jukanovich
  - Built framework
  - Implemented printing
  - Join statements
  - Temporary relations
  - Qualified names
- Thomas Jones-Moore
  - Handled file parsing/lexical analyzing
  - Handled syntax-error checking
  - Delete Where
  - Nested expressions
  -
- Megan Hong
  - Implemented insert
  - Tested and handled errors
    - Errors such as:
      - Checking for datatypes when inserting
      - Checking for length of tuple before inserting
  - Select Where
  - Project

What programming language did you select and why?

We selected Java because between Java and C#, we had a firmer knowledgebase in Java.

List libraries/programming language features you made use of

We made use of the linked list class in Java to store most of the data (to take advantage of heap storage). We also used regular expressions to help parse the input file. And of course, we utilized object-oriented programming in our project. We also chose to implement java.io.Serializable.

Coverage:

- Relation - complete

- Insert - complete
- Print - complete
- Heap Storage - complete
- Catalog - complete
- Destroy – complete
- Select where – Complete
- Delete where – Complete
- Project – Complete
- Added fearure: Nested Expression

How did we implement:

Relations:

The relation is implemented via a Relation object. Each relation contains linked lists that contain information about attributes and a linked list of Tuple objects. Attributes are stored in a list named "_attributeList". The attributes are created within the Relation class, using information about the attribute that would be made. This includes an attribute's name, type, and length. This information is all parsed upon creation of the relation. Upon parsing, the values are added to their respective linked lists (such as length to a linked list named "_length". The linked lists in our relation class are parallel, meaning related information is accessed by just one index. The creation of the Attribute object accesses the information it needs via just one index.

Each relation stores its name, number of Attribute objects, and schema as properties. The schema property is used for storing information in the Catalog object. A linked list meant for the catalog, called "_metaDataList" stores the relation's name and schema. This is then made into a Tuple object (called _metaData), which is then stored specifically in the Catalog.

Tuples:

We made each tuple a list of values – these are essentially attributes, but for the sake of better understandability within the program, we opted to call attributes in tuples "values". Each relation has a list of tuples.

Each value (Value object) carries either an int or string. This is possible via overloaded constructors. Instead of primitives, we used the Integer class for storing integers. Each Value object has a _valueInt and _valueString property, which is assigned according to how the object is constructed. Values are made when during parsing (INSERT thing1 thing2 thing3 makes 3 Value objects, each for an individual "thing". These are all stored in a linked list.

Once there is an appropriate number of Value objects made during parsing (checked by comparing to number of attributes in corresponding relation), initialize a Tuple object using the linked list of Values.

Print:

For this, we chose to print in the command-line. We made print its own class and looked through a relation passed through the primary function in the Print class to convert said relation to tabular form.

The Print class is implementing using a single object. This object calls a public function, which acts as a “wrapper function” for printing the table to the command line. This is the only public function, whereas the functionality is sectioned off into several private function. One function handle finding the width of the table. The value this returns is used in almost all other functions to create each table. Then we print he outer border of a table via a function. Then another function handles displaying header information about the attributes. The next function prints the borders used in each attribute. Finally, we finish with calling the function that prints the outer border of the table again. There are functions used for printing the inner and outer borders of the table respectively.

Heap Storage:

There is a linked list that contains each We used linked list to make each relation contain a list of tuples, which contain a list of attributes. Each Relation object, as mentioned above, stores information about each of its attributes in parallel linked lists. For each attribute, the Relation object creates an Attribute object from these values and stores it within another linked list- which stores attributes of each relation.

Tuples are stored in linked lists within each relation. Tuples contain linked lists of values. All of this allows us to dynamically create our objects. This gives our database physical independence since heap storage allows the relation to not be affected by physical implementation details. The heap storage is chiefly implemented via linked lists.

Catalog

We made the catalog a relation object. To create this, we made a CatalogCreator class to set the catalog’s values before anything else happened with regards to SURLY commands. Each time a relation is created, a tuple containing metadata about that relation is put into the catalog.

The Catalog is a Relation object. The Catalog is special in that it only contains tuples containing both the name and schema of each relation. So, each tuple has only two values. The Catalog cannot be destroyed by DESTROY. Details on how the name/schema tuples are inserted into the Catalog can be found in the Relation section of this report.

Destroy

Destroy eliminates the appropriate tuple from the SURLY database. It also nulls out the appropriate object. When “DESTROY” is detected, a function known as “destroy_handler” is called. This goes through what follows DESTROY and acts on the following Relation.

There is a check to make sure DESTROY cannot act on the Catalog. If our Relation is not the Catalog, this will remove the relation from the linked list of relations we have for SURLY. This also removes information about the relation from the Catalog.

Join

Join joins two relations on some common shared attribute. The statement that specifies what attribute is shared (or what tuples in this attribute have shared values) is called a join condition. Join first has a class that reads from the file and retrieves the relations related to the operands. This is done using JoinHandler.java. Within this class is a Join class. A Join object is established with a leftOperand and a rightOperand. The class reads the join condition and cycles through tuples of the relation. For each set of tuples that satisfies the condition, there is a function that combines these tuples and adds the tuples to a "JoinedRelation". The new relation Is a field of the Join object and is called upon when the temporary relation is seeking to take the value of the new Joined relation.

Delete Where

DELETE WHERE works by first checking to see if there is an 'and' in the whole entire statement (this is of course after checking for correct syntax). If there is not an 'and' in the whole statement, then each expression separated by 'or' gets returned separately since not many comparisons have to go on. If there is an 'and' in the statement, then things get a bit more complicated. First, we assign a bunch of variables to be checked on later. This includes but is not limited to numberOfAnds, IndexOfStatement, numOfOperations, etc. We have an outer most while loop continuously looping until we hit a semi colon, that is grabbing each expression word as a String, assigning that to an ArrayList<String>, and feeding that to the deleteWhere() function. deleteWhere() cleans up the words by removing parenthesis, semicolons, and quotes.

This is then handed off to truncateStatement, which goes to the index of the 'or' or 'and' statement. It grabs the -3, -2, -1, +1, +2, +3 words, in respect to the index of the 'and' / 'or' in the ArrayList. This statement is now evaluated as needed. If we are dealing with an 'and', when statement evaluation handling is occurring, we mark the tuples index's in respect to where they are in the CATALOG to be deleted. Once all index's have been marked after all the 'and's have been parsed, we then delete the tuples with the matching index's.

Project

Stores the name of the temporary relation as a string and creates and adds the temporary relation into the catalog. Then we grab the actual relation and do a quick check to see if it exists. If it does exist we proceed to grabbing the relation's values that match the attribute values we want to project and creating and storing the desired values into a value list. From there we create tuples and add them into our temporary relation.

Select

Stores the name of the temporary relation as a String, creates a Temporary relation and adds it to the catalog. Then we get the actual relation and do a quick check to see if it exists. Next we grab the relation's tuples to preform operations (with the help of operationHandler) and check for and/or conditions using two helper functions for the two conditions respectively. Once all the conditions and operations have been preformed we set the modified tuples list into the temporary relation

Nested Expressions

For these, we used recursion. We first detected a (in our file, then passed the flow of control to NestedExpressions.java. We then used a recursive function that took in an ArrayList object and returned a Relation object. So, PRINT(SELECT (PROJECT A…))) recursively calls itself until it arrives at the innermost parenthesis. The first thing that is done is to isolate the outermost statement and begin preforming recursive calls on that outer statement. Our base case is when there is no longer a parenthesis present.

We handled the case where there is no matching parenthesis by having the loop which counts parenthesis return -1 if the index was equal to the size of the ArrayList. We chose to pass in an ArrayList because that is the structure we used while parsing our file. The nested expressions may also handle temporary values and are able to handle SELECT, PROJECT, and JOIN statements.

Things we did differently/What would we do starting over?:

We stored metadata as a property of each relation object. We also stored “attributes” in tuple objects as “values”, but this is more of a naming difference than a functional one. This implementation does not have a GUI. We are proud of how the table turned out – it prints perfectly and is a great visualization of our work.

This was also a great case in collaboration. Doing it over again, we would have met more frequently and discussed various ideas about our parts of the project with each other. We would have also worked harder to keep coding style more consistent and to more thoroughly comment our code as we push our commits.

Recommendations:

Something we would do differently is make the project object-oriented from the start. SURLY definitely met the objectives, as there is nothing missing, and the requirements are well-implemented. To improve the assignment, there should be a little more guidance on the OO aspect of it, as many students may not be up on their OO skills by the start of this assignment. To improve the course, we should do a flipped classroom. This provides a more streamlined medium of educational exchange between the professor and student. As for other comments, I personally really enjoyed this project, and I’m sure I can say the same for my teammates.

What did we learn?

We learned that a database can be implemented using an OO paradigm. We also learned how certain querying implementations may be implemented. Finally, we learned how linked lists may be used to give a database physical data independence from how the code is stored in memory.

Any other comments?

This was an interesting project.