Eric Zounes, Ian Fridge, Jacques Uber
03/15/13
CS440
Assignment 4 - Part 2

Timing Results:

```
time java Main /scratch/CS440Assignment4/TY.dat
/scratch/CS440Assignment4/SY.dat /scratch/CS440Assignment4/RX.dat
/scratch/CS440Assignment4/SX.dat > results.txt

real   0m0.249s
user   0m0.469s
sys    0m0.039s
```

Please note that we did not implement the multi-pass merge sort and will have a faster skewed times because of that (DO NOT COUNT OUR SCORE FOR FASTEST TIME).

Report:

For this assignment we attempted to developed an algorithm for joining multiple database relations. We used a hash database to index the largest relation which we determine at run time by doing an initial pass over each of the files in parallel. We created a generic entry object that can take a join attribute and calculate these features. The largest relation is indexed on the join attribute X. The other relations to be joined were sorted at run time to improve the performance of the join. Our custom iterator determined whether it is operating on an indexed relation which makes our solution extremely generic.

For our execution plan we determined the total and distinct number of values for each of the relations. We found that the S relation had the greatest number of distinct values with 10000 unique values and 125000 total for both it's X and Y columns. The RY relation had 2000 total values and 1807 unique making it the relation which should be first joined with SY. The size of this join was calculated as (125000*2000)/(10000*1807). This relation was then joined with the TX relation which had 965 unique elements of the total 1000.

One of the problems we had with this assignment was that the SY and SX relationships were the same files which in turn created problems with insertion. BTREEs in Berkley DB do not support multiple entries with the same primary key and the same data entry. This is also the case with hash based databases. Not only did we have multiple entries with the same primary key, they also had the same data causing the repeated values to be ignored meaning that the S database contained 10000 records after the complete insertion (the S relation has 125000 total entries). Similarly the R table contained the unique 1807 entries since there was no data it was just they key and then the data, which is the same as the key. For the T relationship this was also the case.

The resulting table after the joins contained 956 elements, all of which were contained in TY. We tested this by writing a simple python script which took the intersection between the unique items in contained in all of the files and then returning the size of that set.

Due to these setbacks, the query execution planning was only partially implemented. When we calculated it by hand we found that the algorithm should have first joined the SX and RX relation and then joined SY and TY and finally joined those resulting intermediary relations. We attempted to debug these issues but were not successful. Despite this it did get the correct result, however not by using the optimal query execution plan.

We tried to group the files by the first value in their filename as, the relation name, and the second value as the column name. We then tried to create wrappers around these relations and find all the valid combinations between relations based on their shared columns. We then tried to compute the cost between each pair of relations. This part worked correctly but we ran into difficulties when we tried to calculate cost for joining with intermediary relations. It worked in some cases but we had bugs in our code which prevented us from finishing the algorithm.

We were significantly delayed during the development of our program as we spent a great deal of time trying to debug the fact that the duplicate values were not being inserted correctly. Once we solved this we were very much behind schedule. I think another mistake that we made was attempting to write our solution in too general a manner. We tried to make sure that the query execution planning algorithm tried to find and compute tables intelligently by allowing relations to have any number of columns, rather than hard coding some of the logic.