Jonathan Ong 104135858

Quanjie Geng 204160668

CS143 Winter 2015

README --- Writeup

Design was mainly based off of the spec and the parameters given for each object.

Our design for join ordering is based off of the Selinger Optimizer (pseudocode provided in the spec). We used PlanCache to store our computed optjoin and enumerateSubsets to achieve choose(|j|, i) where i is from 1 to |j|.

We made a little change to the API in estimateJoinCardinality. While the spec says that we should fill in the estimateJoinCardinality function, in the skeleton code, this function actually makes a call to estimateTableJoinCardinality. We feel like this extra function call is unnecessary, so we sticked to the spec and put our code in estimateJoinCardinality and comment out the call to estimateTableJoinCardinality.

Our design implements all the required functions of simpledb.

For the IntHistogram, and array was chosen for the histogram. Instead of storing the values, we simply needed to convert the valid value into the proper bucket and increment the count. The width was assumed to be the same across all the buckets. The width was at least 1 and could be greater if the range was larger than the bucket number provided during construction. Selectivity was calculated by dividing the number of proper values and dividing by the total number.

For Tablestats, minimum and maximum values are first calculated for each tuple so histograms can be calculated. Once the histograms are initialized with the proper size and values, values are entered in one at time to allow the selectivity functions to properly work. Estimating selectivity relies on the IntHistogram and StringHistogram implementation.

Difficulties were had when selectivity was tested. There were problems with checking valid indexes and problems with division.

******NOTE: BECAUSE THIS PROJECT WAS TURNED IN 3 DAYS LATE (SUNDAY), 3 FLEX DAYS WERE CONSUMED. BOTH JONATHAN AND QUANJIE HAD 3 SPARE DAYS, AND THEY HAVE NONE LEFT*******

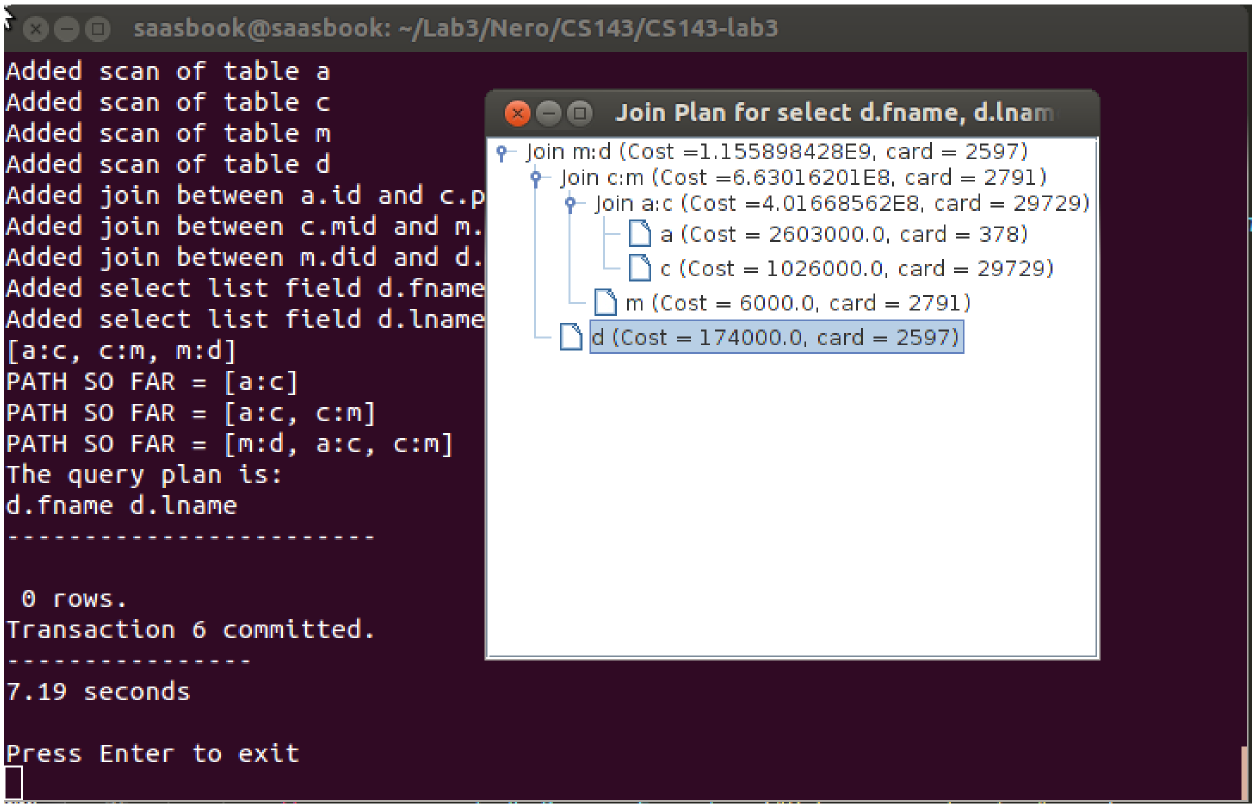
Exercise 1

Parser’s main function does some preliminary checks and then calls the start() function. The start function first loads the schema of the database, and computes relevant statistics. This includes min and max values for every field of the database. These statistics will be used to optimize the query later. There are some optional flags that can be set on the command line which prints more information for the user to see. Regardless, the parser begins parsing the SQL commands of the query until it reaches a semi-colon. It will stop at every comma and process that "phrase" if possible in the processNextStatement() function. The start function then reports the time and moves onto the next phrase.

In the processNextStatement() function, the inputstream is parsed and a transaction is generated, locking the resources. There are many different types of SQL commands this function can handle including "ZInserts, ZDeletes, and ZQueries." For a query, the query is executed and after it has finished, it reaches a commit stage, presumably to finalize its changes. If there is an error and the commit doesn't successfully follow through, and abort message is prompted.

Exercise 6

The following image shows the query plan for the 1% dataset.



```
saasbook@saasbook: ~/Lab3/Nero/CS143/CS143-lab3
Added scan of table a
Added scan of table c
Added scan of table m
Added scan of table d
Added join between a.id and c.p
Added join between c.mid and m.
Added join between m.did and d.
Added select list field d.fname
Added select list field d.lname
[a:c, c:m, m:d]
PATH SO FAR = [a:c]
PATH SO FAR = [a:c, c:m]
PATH SO FAR = [m:d, a:c, c:m]
The query plan is:
d.fname d.lname
-----------------------

 0 rows.
Transaction 6 committed.
----------------
7.19 seconds

Press Enter to exit
```

```
Join Plan for select d.fname, d.lnam
Join m:d (Cost =1.155898428E9, card = 2597)
  Join c:m (Cost =6.63016201E8, card = 2791)
    Join a:c (Cost =4.01668562E8, card = 29729)
      a (Cost = 2603000.0, card = 378)
      c (Cost = 1026000.0, card = 29729)
    m (Cost = 6000.0, card = 2791)
  d (Cost = 174000.0, card = 2597)
```

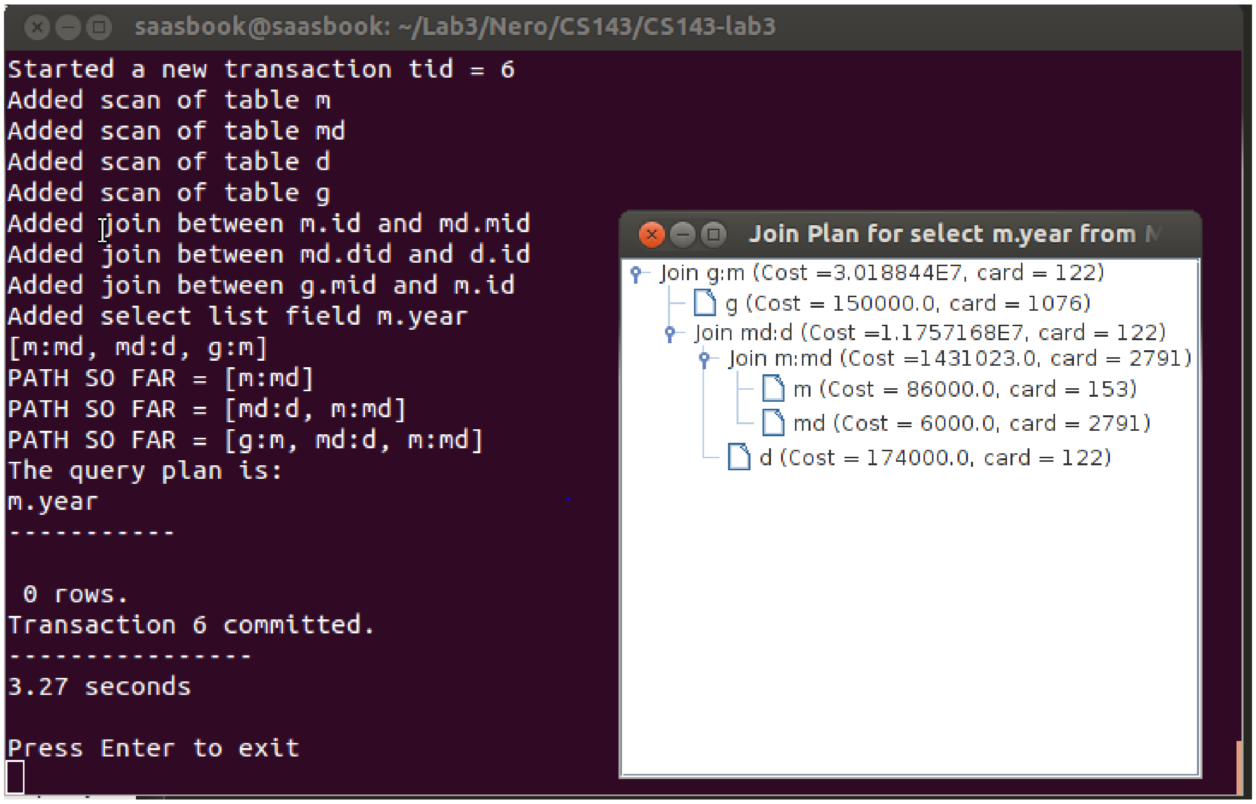
The plan is detailed in the terminal of the image: [m:d, a:c, c:m].

It first matches the specific actor name with its first and last name, since that would be too costly later on. With that subset of actors, it compares these actors with the casts under the relation "a.id=c.pid". From these matches, it joins and looks for movies associated with that cast. This can be done easily since movies are indexed with that id. From those subset of movies, the relation between movie and director is used, exploiting the index on directors.

Second query

```
select m.year
from Movie m, Movie_Director md, Director d, Genre g
where m.id=md.mid, md.did=d.id and d.fname='Gen' and g.mid=m.id
and g.genre='Shounen' and m.name='Madoka'
```

The query above was added to search for the year the movie with the name Madoka and directed by Gen was released.


saasbook@saasbook: ~/Lab3/Nero/CS143/CS143-lab3
Join Plan for select m.year from M

```
Started a new transaction tid = 6
Added scan of table m
Added scan of table md
Added scan of table d
Added scan of table g
Added join between m.id and md.mid
Added join between md.did and d.id
Added join between g.mid and m.id
Added select list field m.year
[m:md, md:d, g:m]
PATH SO FAR = [m:md]
PATH SO FAR = [md:d, m:md]
PATH SO FAR = [g:m, md:d, m:md]
The query plan is:
m.year
-----------


 0 rows.
Transaction 6 committed.
----------------
3.27 seconds

Press Enter to exit
```

```
Join g:m (Cost =3.018844E7, card = 122)
  g (Cost = 150000.0, card = 1076)
  Join md:d (Cost =1.1757168E7, card = 122)
    Join m:md (Cost =1431023.0, card = 2791)
      m (Cost = 86000.0, card = 153)
      md (Cost = 6000.0, card = 2791)
    d (Cost = 174000.0, card = 122)
```

Here is the plan.

The movie is first indexed with the movie directors and then from movie directors to directors. Once the movie is paired with the director, the optimizer has to match the genre.