

Lab 3 Write Up

Part 1: The Life of a Query in SimpleDB

Step 1: `simplifiedb.Parser.main()` and `simplifiedb.Parser.start()`

Main is the entry point and calls start, which performs three primary actions:

- + Populates the simpleDB catalog from the catalog text file provided by the user as an argument
- + For each table defined in the system catalog, it computes statistics over the data in the table by calling `TableStats.computeStatistics()`
- + It processes new statements submitted by the user

Step 2: `simplifiedb.Parser.processNextStatement()`

- + It gets a physical plan for the query by invoking `handleQueryStatement()`
- + Is subsequently executes the query

Step 3: `simplifiedb.Parser.handleQueryStatement()`

This method facilitates the translation of our zquery object into a plan of actions to run on our database which will yield the proper answer.

- + Will construct a new query object
- + Will build a logical plan using `parseQueryLogicalPlan()`
- + Will then build a physical plan out of the logical plan using `LogicalPlan.physicalPlan()`
- + Updates operator cardinality
- + Prints out the query plan tree!

Step 4: `simplifiedb.Parser.parseQueryLogicalPlan()`

- + Will get all the table names associated in the query and put them in a list; this list comes from the "from" clause.
- + Then parses the where clause to identify filters and joins necessary in the query. Note - will not handle subqueries!
- + Will identify up to one group by field
- + Gets aggregates

Step 5: `simplifiedb.LogicalPlan.physicalPlan()`

This step basically gives us an iterator over our query result! The most important function call here.

- + Builds a map of scans
- + Iterates through all the filters and estimates the selectivity of each filter
- + Walks through the select list and determines the projection order.
- + Walks through the joins, first by ordering the joins by calling `orderJoins()` and then iteratively joining groups of data.
- + Aggregates and orders the joined, filtered tuples.
- + Then projects and returns the result in the form of an iterator!

Step 6: `simplifiedb.JoinOptimizer.OrderJoin()`

This method uses statistics and predictive modeling to order joins in some way based off the cardinalities and costs of joins

One of the methods I get to implement in this lab!

Parts 2 – 5: Coding Implementations

Part 6:

Here's the parse tree and response for the given query:

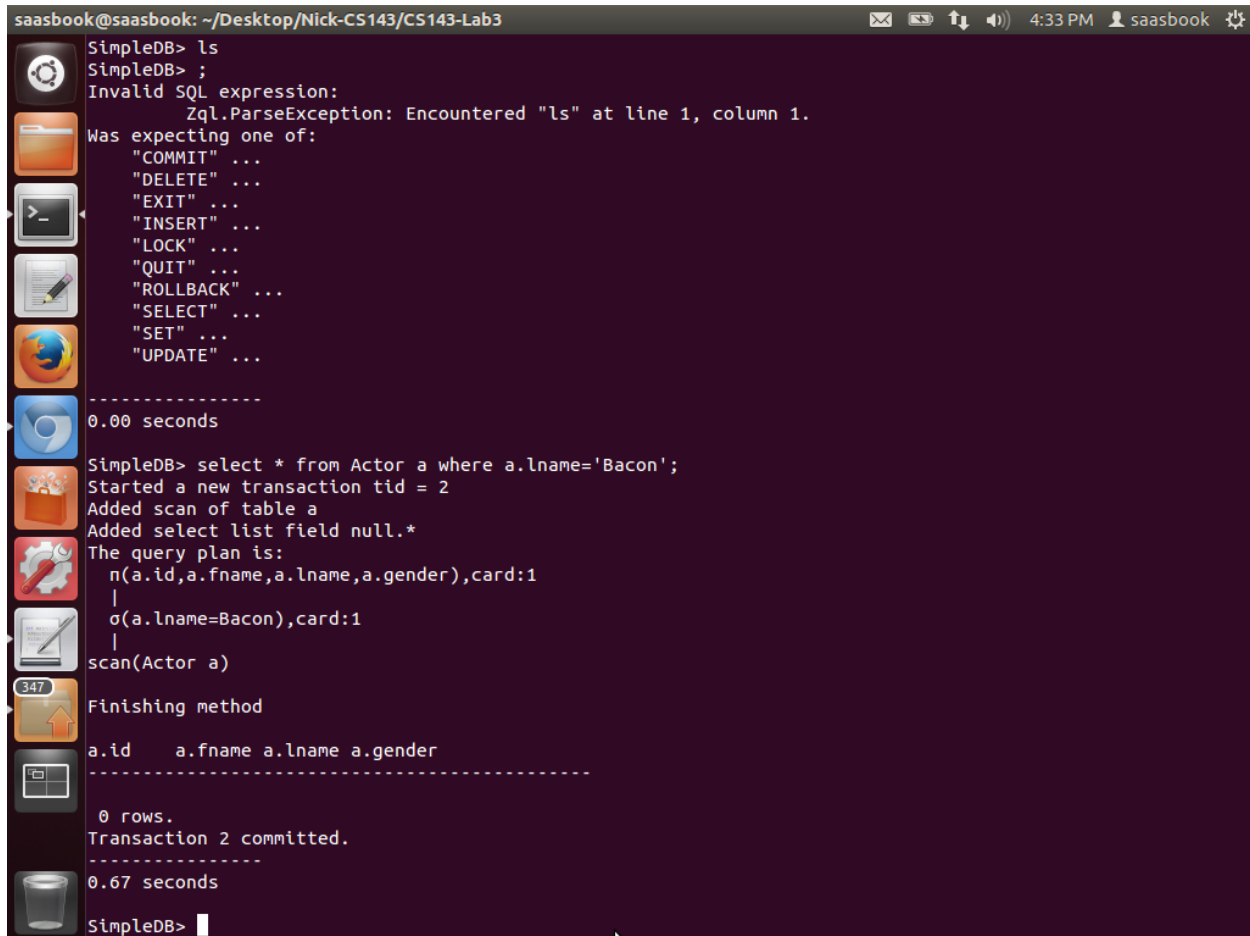
```
saasbook@saasbook: ~/Desktop/Nick-CS143/CS143-Lab3 4:31 PM saasbook
Added scan of table a
Dash home
Added scan of table c
Added scan of table m
Added scan of table d
Added join between a.id and c.pid
Added join between c.mid and m.mid
Added join between m.did and d.id
Added select list field d.fname
Added select list field d.lname
The query plan is:
      n(d.fname,d.lname),card:3008
      |
      ⋈(a.id=c.pid),card:3008
     /      \
σ(a.lname=Spicer),card:1  ⋈(m.mid=c.mid),card:3008
|                        |
σ(a.fname=John),card:1  ⋈(d.id=m.did),card:278
|                        |
c)                      |
scan(Actor a)           |
                        |
scan(Director d)  scan(Movie_Director m)
                        |
                        scan(Casts)

Finishing method
d.fname d.lname
-----
Chris   Malazdrewicz
Thomas  Parkinson
Alain   Zaloum

3 rows.
Transaction 1 committed.
-----
9.23 seconds
SimpleDB>
```

It's formed as such because of how the compute cost and compute cardinality functions are written. They give extra value to equality joins, which yield a low cardinality when one of the two keys being joined on is a primary key. It means the solution actually does all the joins first even though it's probably fastest to project the one actor's name and then join that single result onto the remaining three fields in order.

The second query I ran searched for an actor with the last name 'Bacon' and could not find him. Oh well. It's query tree is remarkably simple as there is only 1 scan and 1 operation.



```
saasbook@saasbook: ~/Desktop/Nick-CS143/CS143-Lab3
SimpleDB> ls
SimpleDB> ;
Invalid SQL expression:
    Zql.ParseException: Encountered "ls" at line 1, column 1.
Was expecting one of:
    "COMMIT" ...
    "DELETE" ...
    "EXIT" ...
    "INSERT" ...
    "LOCK" ...
    "QUIT" ...
    "ROLLBACK" ...
    "SELECT" ...
    "SET" ...
    "UPDATE" ...

-----
0.00 seconds
SimpleDB> select * from Actor a where a.lname='Bacon';
Started a new transaction tid = 2
Added scan of table a
Added select list field null.*
The query plan is:
  n(a.id,a.fname,a.lname,a.gender),card:1
  |
  σ(a.lname=Bacon),card:1
  |
  scan(Actor a)
Finishing method
a.id    a.fname a.lname a.gender
-----
0 rows.
Transaction 2 committed.
-----
0.67 seconds
SimpleDB>
```

Design Choices: For all designs, I valued simplicity and leveraged the API to build specifically what was asked of me in the spec. I used the math given to us in the spec, which did a pretty good job at building good functionality

API Changes: I built an interface called "Histogram" to help tie together the IntHistogram and StringHistogram classes, and it helped clarify my code. Simplicity over all else.

I spent about 15 hours working on this lab, mostly to a few silly bugs that I spent too much time debugging.

Note: I used the lab2 solution code as a base for this assignment. All assigned exercises contain my original solutions.