CS 143 Lab 3 Writeup

Nathan Fischer, #804180060
Jimmy Vo, #004177090

For this lab, we will be using 1 slip day. Because we used 2 slip days for lab 1, and 1 slip day for lab 2, this is our final slip day.

Design decisions:
For selectivity estimation, we followed the guidelines in the spec. 30% was the default value used. Joins were also ordered as recommended in the spec. We did not implement any bonus exercises.

Changes to the API:
We did not make any notable changes to the API that were not already made during lab 1. We only filled out skeleton functions.

Missing or incomplete code:
We have no incomplete or missing code to the best of our knowledge. We were able to pass all tests and systemtests for this lab. The query evaluation passes as well.

Time spent on the project:
The total time spent was about 8 hours. No aspects of the lab were unreasonably challenging. The biggest challenge was locating a bug in SeqScan where table aliases were not prepended to column names

Exercise 1:

Step 1: simpledb.Parser.main() and simpledb.Parser.start() simpledb.Parser.main() is the entry point for the SimpleDB system. It calls simpledb.Parser.start(). The latter performs three main actions:
- It populates the SimpleDB catalog from the catalog text file provided by the user as argument (Database.getCatalog().loadSchema(argv[0]);).
- For each table defined in the system catalog, it computes statistics over the data in the table by calling: TableStats.computeStatistics(), which then does: TableStats s = new TableStats(tableid, IOCOSTPERPAGE);
- It processes the statements submitted by the user (processNextStatement(new ByteArrayInputStream(statementBytes));)

Step 2: simpledb.Parser.processNextStatement()
This method takes two key actions:
- First, it gets a physical plan for the query by invoking handleQueryStatement((ZQuery)s);
- Then it executes the query by calling query.execute();

Step 3: simpledb.Parser.handleQueryStatement()
- It makes a query from the tId.
- After building a logical plan, it calls “physicalPlan” to find the most optimal plan.

Step 4: simpedb.Parser.parseQueryLogicalPlan()
- First it will read the FROM clause and add tables into the logical plan.
- Then it will looks at the WHERE clause and adds filters to the plan.
- Then it looks at the GROUP BY clause to see which fields to group.
- Then it processes the SELECT clause to pick out aggregate fields.
- Then it sorts if there's an ORDER BY clause.
- The logic plan is then returned.

Step 5:simpledb.LogicalPlan.physicalPlan()

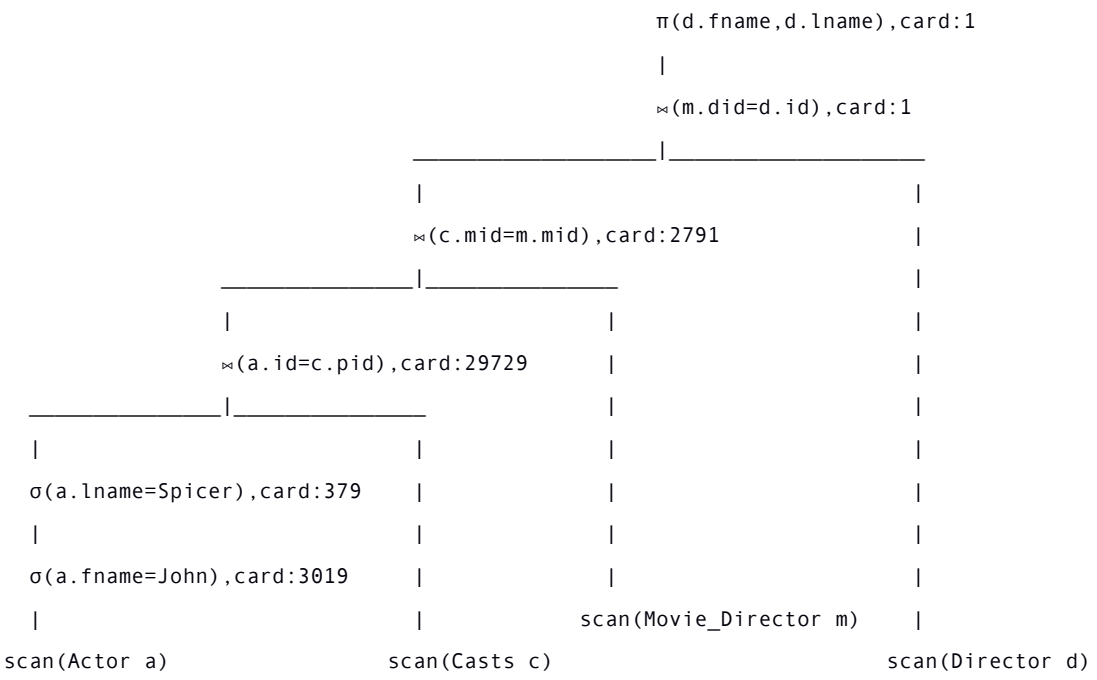
- Tries to optimize the order the tables should be joined in the plan.
- Uses the joinOptimizer with the combination of tables to find the most optimal plan.
- It returns a DbIterator object that can be used for the query.

Exercise 6:

The query plan selected for the query:

```
select d.fname, d.lname
from Actor a, Casts c, Movie_Director m, Director d
where a.id=c.pid and c.mid=m.mid and m.did=d.id
and a.fname='John' and a.lname='Spicer';
```

Looks as such:



Selecting based on name results in a large reduction factor. Joining with Casts is very efficient, because a.id is a primary key for Actor. Joining next with Movie_Director allows the final join to be with Director, which utilizes its primary key on d.id.

Our result may be different for different size data sets. If a larger data means that the different tables scale unequally (i.e. Actor grows disproportionately to Casts), then this would result in a different decision made for join order.

Our other query is as follows:

```
select a.fname, a.lname from Actor a, Casts c, Movie m where a.id=c.pid and c.mid=m.id and m.name='A
Darker Shade of Gray';
```

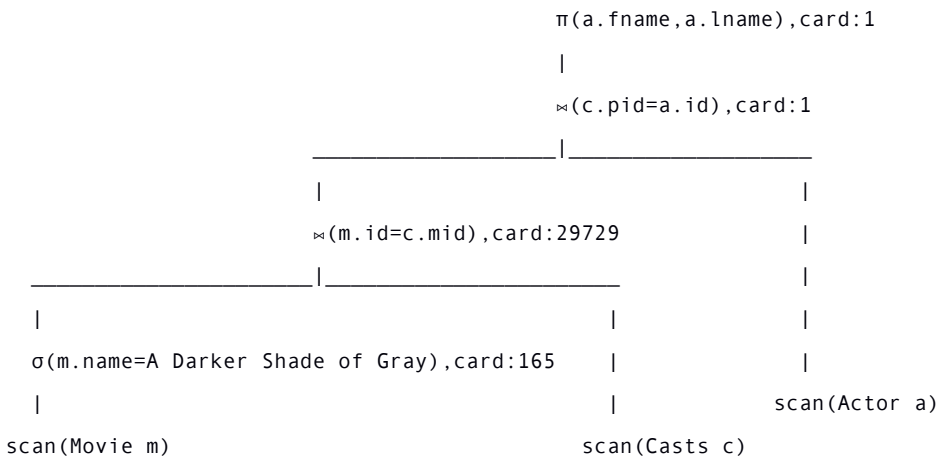
This returns 4 rows:

```
a.fname a.lname

------------------------------

Jack           Arndt

Nicodemus      Hammil
```

Kenneth Wonderley

Kristin Barker

The query plan generated looks like this:



This plan is generated because it first selects out the movie 'A Darker Shade of Gray.' This reduces the input size to the first join by a great deal. Then it joins with Casts, which will result in a small output cardinality, since there is only one cast per film (and there is probably only one film with that title). Then the final join will be very efficient because a.id is a primary key for Actor.