Name: Eric Boris
StudentId: 1976637

Section 1: Overall System Architecture

Overview Description:
SimpleDB can be divided into 5 components.
Some components can be subdivided into subcomponents.
Each (sub)component is composed of classes.
The structure, down to the level of classes, is as follows:

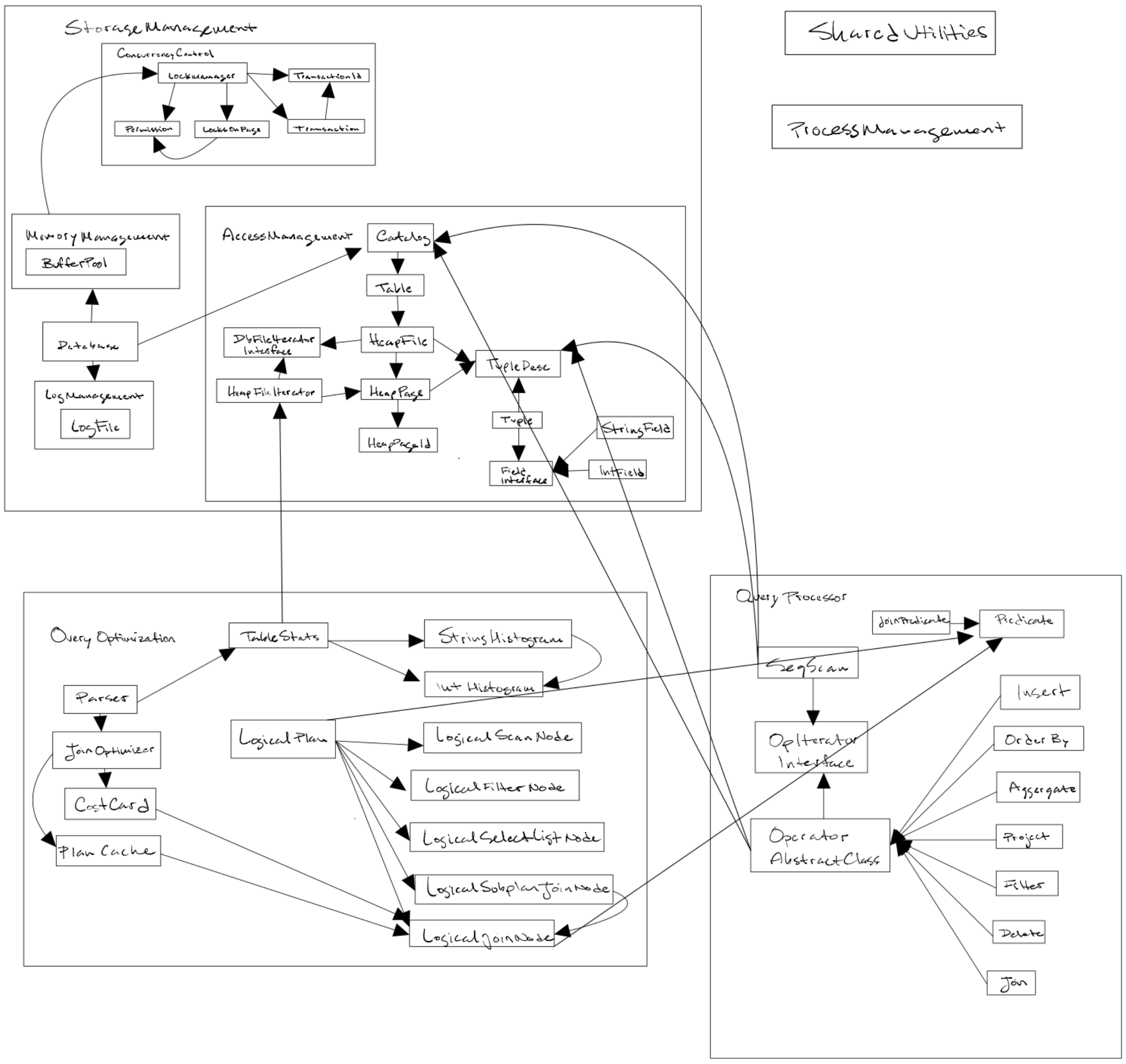
1. Storage Manager
    - 1.1. Concurrency Control
    - 1.1.a. LockManager
    - 1.1.b. Permission
    - 1.1.c. LocksOnPage
    - 1.1.d. TransactionId
    - 1.1.e. Transaction
    - 1.2. Log Manager
    - 1.2.a. LogFile
    - 1.3. Database
    - 1.4. Memory Management
    - 1.4.a. BufferPool
    - 1.5. Access Management
    - 1.5.a. Catalog
    - 1.5.b. Table
    - 1.5.c. HeapFile
    - 1.5.d. HeapPage
    - 1.5.e. HeapPageId
    - 1.5.f. DBFileIterator
    - 1.5.g. HeapFileIter
    - 1.5.h. TupleDesc
    - 1.5.i. Tuple
    - 1.5.j. Field
    - 1.5.k. StringField
    - 1.5.l. IntField
2. Query Processor
    - 2.a. SeqScan
    - 2.b. OpIterator
    - 2.c. Operator
    - 2.d. Insert
    - 2.e. OrderBy
    - 2.f. Aggregate
    - 2.g. Project
    - 2.h. Filter
    - 2.i. Delete

- 2.j. Join
- 2.k. Predicate
- 2.l. JoinPredicate

3. Query Optimizer
    - 3.a. Parser
    - 3.b. JoinOptimizer
    - 3.c. CostCard.
    - 3.d. PlanCache
    - 3.e. TableStats
    - 3.f. LogicalPlan
    - 3.g. StringHistogram
    - 3.h. IntHistogram
    - 3.i. LogicalScanNode
    - 3.j. LogicalFilterNode
    - 3.k. LogicalSelectListNode
    - 3.l. LogicalSubPlanJoinNode
    - 3.m. LogicalJoinNode
4. Shared Utilities
    - [Not covered in this course]
5. Process Management
    - [Not covered in this course]

Diagram:

Simple DB Architecture



Detailed Description:
BufferManager:
Caches pages on disk for faster random access.
Operators:
Used to distinguish the type of query being performed.
Used by the query optimizer for determining the optimal query plan.
LockManager:
Provides an interface for components to request and gain access to locks.
Enables concurrent computation.
Handles deadlocking.
LogManager:

Manages transactions.
Handles rollback, commit, checkpoint, undo, abort, and more.

Lab 5 Answers:
Exercise 1: Parser.java
Step 1: simpledb.Parser.main() and simpledb.Parser.start().
simpledb.Parser.main() is the entry point for the SimpleDB system.
Calls simpledb.Parser.start().
Performs three main actions:
1. Populates the SimpleDB catalog from the catalog text file provided by the user as an argument (Database.getCatalog().loadSchema(argv[0]);)

2. For each table defined in the system catalog:
Computes statistics over the data in the table by calling:
TableStats.computeStatistics() that for each table does:
TableStats s = new TableStats(tableid, IOCOSTPERPAGE);

3. Processes the statements submitted by the user (processNextStatement(new ByteArrayInputStream(statementBytes));)

Step 2: simpledb.Parser.processNextStatement()
Takes two key actions:
Gets a physical plan for the query by invoking handleQueryStatement((ZQuery)s);
Executes the query by calling query.execute();

Step 3: simpledb.Parser.handleQueryStatement()
1. Builds logical plan lp by calling simpledb.Parser.parseQueryLogicalPlan

2. Finds optimal plan by calling lp's physicalPlan method.

Step 4: simpedb.Parser.parseQueryLogicalPlan()
1. For each table name in FROM clause:
Call simpledb.LogicalPlan.addScan() to add table logical plan.

2. Process WHERE clause:
Call simpledb.Parser.processExpression to add corresponding filter into logical plan

3. Parse GROUP BY clause:
Identify the attribute and set the value of the GROUP BY field.

4. Process SELECT clause:
Identify the aggregation field using getValue()
Identify aggregation method (min, max, etc) using getAggregate()
Add using simpledb.Parser.addProjectField().
Or pick the column to project as specified in the SELECT clause

5. Processes ORDER BY:
Identify the attribute to order by calling getOrderBy()
Call getExpression() to process result.
Identify the attribute to order and the ordering
Use addOrderBy to add results to logical plan.

6. Return logical plan.

Step 5. simpledb.LogicPlan.physicalPlan()

1. For each table:
Get the table name
Get the corresponding SeqScan
Put into the hashmap
Set the selectivity of the table

2. For each filter:
Get the subPlan iterator
Get the field field
Get the field type fieldType
Get the field's TupleDesc td
Add the name of table and corresponding filter to hashmap
Get the tableStats to get selectivity
Get current selectivity and multiply with new selectivity
Put result into hashmap

3. For each join:
Get the first and second plans
Use join optimizer to produce new iterator
if join not subquery join:
update name of table2
Keep map updated by set all tables with names equal to t2 to t1
Check for remaining subPlan

4. For each selection:
Collect field nums and types
Generate aggregation nodes if there is an aggregate
Generate order by if there is an orderby
It iterates over the selection list. For each selection,

5. Return a new project

Exercise 6: Query Plans
The following queries are run on the 1% IMDB database.

1:

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.lname='Spicer';
```

Output:
```
The query plan is:
π(d.fname,d.lname),card:29729
                |
                ⋈(d.id=m.did),card:29729
________________________|__________________________
|                       |
|                       ⋈(m.mid=c.mid),card:29729
|               __________________________|___________________________
|               |                       |
|               |                       ⋈(a.id=c.pid),card:29729
|               |               ____________________|______________________
|               |               |                       |
|               |               σ(a.lname=Spicer),card:3260|
|               scan(Movie_Director m)  |                       |
scan(Director d)                        scan(Actor a)           scan(Casts c)

d.fname    d.lname
------------------------
Humphrey    Burton

Scott    Allen


2 rows.
Transaction 6 committed.
----------------
2.60 seconds
```

Explanation:
Optimizer chooses plan to join tables with small selectivity first.

Eliminate more invalid tuples early.
Reduce total of i/o.

2:

Query:

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

Output:

```
The query plan is:

                        π(d.fname,d.lname),card:29729
                        |
                        ⨝(d.id=m.did),card:29729
____________________________|____________________________
|                                |
σ(d.fname=John),card:320             ⨝(m.mid=c.mid),card:29729
|                            ________________|________________
scan(Director d)                 |                  |
                            scan(Movie_Director m)  scan(Casts c)

d.fname    d.lname
------------------------
John    Black

[ REMOVED 529 ROWS FOR BREVITY]

John    Timmons


531 rows.
Transaction 7 committed.
----------------
0.47 seconds
```

Explaination:
Choose steps in order of selectivity.
If step gives max selectivity, choose it first.

Section 2:

Overall, I'm pleased with the performance of my implementation. The queries I've run, run quickly. I've passed all the unit tests until lab 5, where I have one test that's not passing. I think there are some cool aspects of my implementation as well, like using both hash and nested join, that were fun. Similarly, I enjoyed the locking manager and think that that turned out well.

I'd like to have had more time to clean up some of the rough edges across the board - better documentation and cleaner implementations. I definitely wasn't able to perform as well as I'd have liked on lab 5 due to other courses. It would be nice to clean up lab 5 especially. If, for no other reason, the aforementioned and outstanding failing unit test. Same with this report, I'm not able to submit what I'd like due to time constraints.