Conor Reisman

6/13/2013

CSE444

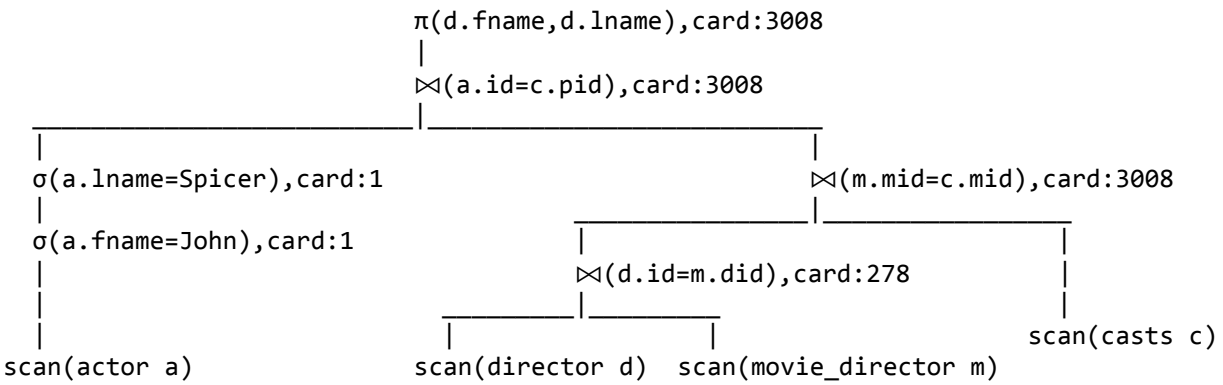
Lab 4 final report

Overall System Architecture

SimpleDB has three main modules. There's the query processor, which takes a query and generates the code to actually process the query. There's the operators, which are the building blocks of the query running (not all are shown). Finally, there's the storage manager, which handles the actual data and chooses how it is handed out to different transactions that are requesting it.

The query processor receives the raw test of a query. It parses that query in order to make sure it is actually valid and the tables/fields exist. Then it picks out all of the relevant bits of the query from among the keywords and stores it in a hierarchy that represents how to process the query. This hierarchy then goes through an optimization phase. This involves making choices like what order the joins should be in, or whether to use a nested loop join or a hash join. In more sophisticated systems it would also handle cases like query flattening with nested queries. Once the physical query plan is decided, the query processor builds the plan using the operators and begins requesting tuples from it.

Operators are the backbone of data processing in simpleDB. They form a tree that represents the physical query plan. Commands at the top of the tree propagate down until they get the data they need, then the response comes back up. For example, given this tree:



The query processor calls next() on the project operator, who calls next on the join operator, who calls (conditionally) calls next on both the select and join operators that are its children. The select operator would call next on its child select operator, who would call next on the scan operator, which loads the data from disk. Once the data is found, it travels back up, unless it doesn't satisfy the criteria of an operator. For example, if the last select called next on the scan over the actors table and it returned a tuple with fname something other than john, it wouldn't return it to the parent. It would request another tuple from the actor table until it was out of tuples or found a match. The operators can stack on top of each other in almost any combination to form a wide variety of queries, providing all the data processing functionality for simpleDB.

Finally, the storage manager handles all actual data management. The buffer pool is in charge of handing out pages to queries and limiting memory usage. It uses the lock manager to make sure that queries are not clobbering each other by providing consistency. The log manager ensures that crashes don't cause the data to be different than expected by writing logs whenever something should be written to disk. It also allows data to be written to disk before committing by keeping a record of what would need to be rolled back if aborted. This can help prevent committing from being too I/O heavy. The catalog keeps references to the file representations of the tables in the database. The Buffer Pool uses it to find the files to request pages from. Last, but not least, the DbFile, Page, Tuple, and Field classes represent the data on disk in a way that is easy to work with from a programming perspective.

Detailed Design of Basic Components

Buffer manager

The buffer manager is in charge of keeping track of and sharing all of the tables in SimpleDB. It is one of the few areas of the application that must be synchronized since it is shared among all running queries

and threads in SimpleDB. It also handles talking to the lock manager in order to ensure consistency during the execution of a query.

The interactions between the buffer manager, files, and pages are quite complicated. At first it seems relatively straightforward, albeit a little roundabout; load the file and add it to the catalog. Then, when you want a page, ask the BufferPool for page x from table y. The BufferPool then gets the file from the catalog and reads the page. Things break down a little, however, when you want to insert a tuple. Because the BufferPool needs to be in charge of any interaction with a page, insertion must go through the BufferPool. When you tell the BufferPool to insert a tuple, it tells the file to look and find an empty slot. Because everything related to pages goes through the BufferPool, the file turns around and asks the BufferPool for each page while it looks. The BufferPool then goes right back to the file and asks it to read in a page. It then proceeds to return this page right back to the file again. It gets even worse when there are no empty slots and an entirely new page has to be created. The file has to ask the BufferPool to add a page to itself, which then calls a method in the file that adds a new page and returns to the BufferPool, who proceeds to call getPage to load the page and returns it.

One of the main purposes of the BufferPool is to manage the pages in an efficient manner. This means it should make the best use of the amount of space allocated to load pages. My implementation of BufferPool uses the least-recently-used algorithm to deal with page eviction. Every time a page is requested, it removes it from the page queue and puts it back on the end. If a new page is requested and the queue, it takes the one from the front of the queue, which means it was the least recently used, and evicts it to make room for the new page. This approach isn't perfect, as I could have a pool of maximum size 50 pages and scan a 51 page file several times. This will cause the page I want to read from next to be evicted right before it is requested. However, it is considered good enough in most cases to suffice.

Operators

Filter

The filter operator is pretty simple. Each time it is asked to produce the next tuple, it requests the next tuple from its child. If it matches the filter, it is passed up. If not, the next tuple is pulled from the child.

Aggregate

The aggregate operator was much more complicated. I first had to create the appropriate aggregator, then add all of the tuples from the child into it, then loop over it for the results. There were several different types of aggregate functions that could be requested on IntFields. Several of them, like COUNT and AVG, require a count to be kept of the total number of tuples aggregated, so I just made a parent class that worked for both StringField and IntField that kept track of this data. Depending on the aggregate operation, additional information may have needed to be stored as well. Additionally, there was differing behavior for empty inputs depending on if a group by was declared or not. It was difficult to make sure all of the cases were handled correctly. Once I had loaded that, I would just return the next tuple returned from my aggregator when my parent requested it. The aggregator was cached, so it only needed to be processed once, unlike things like joins.

Join

Joins were simple to implement. I pull the first value from one child, then every value from the other child. If any pair matches the predicate, I join them together and return them to my parent. This,

however, is highly inefficient. An improvement would be to load some number of tuples I know I can hold in memory, say 50, at a time and then loop over all of them checking against all the ones from the other child. This would dramatically reduce I/O costs. Even better would be load that many tuples into a HashMap. This reduces even looping over them. This would reduce I/O costs even further, however it only works for equality joins and not for range or inequality joins.

Insert

The insert operator was quite simple. For each tuple in the child, it would insert it into the specified table through the BufferPool (see explanation of this under BufferPool). Once completed, it would return a single tuple containing the number of tuples added to the BufferPool.

Delete

This is almost identical to the Insert operator, except that instead of telling the BufferPool to insert the child tuples, it tells it to delete them.

Lock Manager

This was a very tricky class. Locks built into Java only work on threads. As a result, I had to write a new lock in addition to the manager. The manager was reasonably simple in that it just kept track of who already held a lock on different pages and allowed itself to be queried to acquire a lock between a certain transaction and a page, to release said lock, or to check if it held a lock.

The lock class is much more complicated. It had to keep track of which transactions currently had locks on it, and which type. It had a thread-based lock to prevent concurrency issues while actually trying to obtain the lock. The lock was implemented by making use of two Condition variables. If a thread wanted to read, but there was a writer, or a writer waiting, it would block on the noWriters condition. When the writer finished, it would first signal other waiting writers that they could now go, then signal readers. If there were writers, one of them would get to it first and everyone would lock up again. If there were no other writers, all of the readers would get to go. If there were readers and a writer came along, it would block on the noReaders condition. To prevent starving writers, all future readers would block until the writer finished. When the last reader finished, it would signal the waiting writers and one of them would grab the lock.

There was the additional complication of upgrading from reader to writer. I had to notice if my last reader was also one of the waiting writers and allow the writer to acquire the lock. This was done by actually signaling waiting writers when there was 0 or 1 readers remaining such that if the last reader was from the same transaction as one of the waiting writers it could get the lock.

I solved deadlock problems using timeouts. I noticed significant problems with my initial implementation, which was to all time out at the same time. It was caused by the fact that when they're all signaled and woken up, they'll all also block again (if they block again) at the same time. This means they would all time out at roughly the same time and they'd all abort, instead of just one of the ones causing the deadlock. I solved this by distributing timeouts across a range, so that if one timed out, it'd release any locks it held before the rest of them aborted, too.

Log Manager

The log manager allowed a way to serialize dirty files to disk even when they're from an uncommitted transaction and still revert them if the transaction is rolled back. Additionally, it allows files to be

recovered that were committed, but the database crashed before they were flushed to disk. It does this by writing a log record every time a page is flushed to disk.

In order to roll back, I simply read backwards through the log starting from the point where the transaction aborted until I find where the transaction began. Each time I find an update by the rolled back transaction, I'd write the before image back to disk to undo it. The last update would be the original version from before my transaction started. Additionally, I then have to make sure that the page in the BufferPool is evicted so it gets the new version.

To do the recovery, I look for the last checkpoint, since that's the last time everything was flushed to disk. I then read forward through the log. Each time I find an update by any transaction I write the after image. This ensures the latest version is on disk. If I find an aborted transaction, I immediately undo it, so I don't interfere with any writing later on in the log. If I reach the end of the log, any transactions that haven't been committed must then be aborted like in the rollback section above.

Detailed Design of the Query Optimizer

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.TableStats()

This method takes two key actions:

- It reads through the table once to collect statistics for each field (min/max).
- It reads through the table again to process each field into a histogram.

Step 3: simpledb.IntHistogram()

This class takes two key actions:

- It processes each field that's added to it to make a histogram.
- It provides the projected selectivity based on a value and an operator.

Step 4: 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 5: simpledb.Parser.handleQueryStatement()

This method has three key actions:

- It gets the logical plan for the query by calling parseQueryLogicalPlan.
- It gets the physical plan for the query by calling LogicalPlan.physicalPlan.
- It prints out the query plan.

Step 6: simpledb.Parser.parseQueryLogicalPlan()

This method has five key actions:

- It parses the query to pick out all the tables that need to be scanned and adds them to the plan.
- It parses the query to pick out all the select and join conditions and adds them to the plan.
- It parses the query to decide which fields to output after the projection. It also determines whether it needs to do any aggregation or grouping and adds them to the plan.
- It checks the query for validity.
- It parses the query to pick out any sorts that need to be done and adds them to the plan.

Step 7: simpledb.LogicalPlan.physicalPlan()

This method has six key actions:

- It looks up each necessary table and wraps it in a SeqScan operator.
- For each selection, it looks up the field type and creates the Filter operator with the appropriate Field.
- It orders the joins using JoinOptimizer.orderJoins.
- It creates a Join operator for each join based on their new ordering.
- It figures out the order to output tuples and which aggregates need to be run and creates the appropriate Project and Aggregate operators to add to the plan.
- If necessary, it adds a sort function to the plan.

Step 8: simpledb.JoinOptimizer.orderJoins()

This method has four key actions:

- For each subset of all the joins in the plan, starting with size one and growing larger, call computeCostAndCardOfSubplan.
- If the return value is not null and the cost is smaller than the current best cost, replace the best cost and add it to the cache for dynamic programming.
- As the subsets grow larger, they build off of the previous size by joining the newest join with the most optimized ordering of the subset.
- Once all subsets have been computed, get the new optimal ordering, which is the cached value for the set of all joins, optionally print it, and then return it.

Step 9: simpledb.JoinOptimizer.computerCostAndCardOfSubplan()

This method has 4 key actions:

- In the base case of a single join, it simply computes the cost and cardinality of the join and returns the ordering/cost/cardinality.

- In the case of multiple joins, it removes the one provided, then gets the optimal order for the new subset from the cache.
- From there, it figures out the optimal direction to join new node to the results of the past using estimateJoinCost and estimateTableJoinCardinality, which run off of simple heuristics based on the cost and cardinality of the two tables being joined.
- It returns this new ordering/cost/cardinality.

Detailed Design of Join Ordering

See steps 8 and 9 above.

Detailed Design of Selectivity Estimation

The selectivity estimation for IntFields worked by creating a number of buckets. The first bucket contained the minimum value and the last bucket contained the maximum value. The width of each bucket depended on the range of values and the number of buckets. As each value was added to the histogram, the count for the appropriate bucket was incremented (for example, with 10 buckets and values from 1-20, value 2 would fall into bucket 0).

In order to ascertain the selectivity of a specific value, you simply look up the number in that bucket, divide it by the total number of tuples to get the proportion in that bucket, then divide it by the width of the bucket to get the proportion of a single value in that bucket.

In order to get the selectivity of a range, it involves getting the number of values in the buckets either greater than or less than the given value and dividing it by the total number of tuples to get the proportion greater or less than. It gets a little more complicated when working with values in the same bucket. You take the fraction of the bucket that falls in the range, multiply it by the number of values in that bucket, then divide by the total number of tuples to get the proportion of values in that bucket that fall in the range.

These work reasonably well to calculate heuristics, but the values get skewed when the range is significantly larger than the number of buckets (although this saves on heuristic performance, so it is a reasonable trade off).

Detailed Design of My Extension

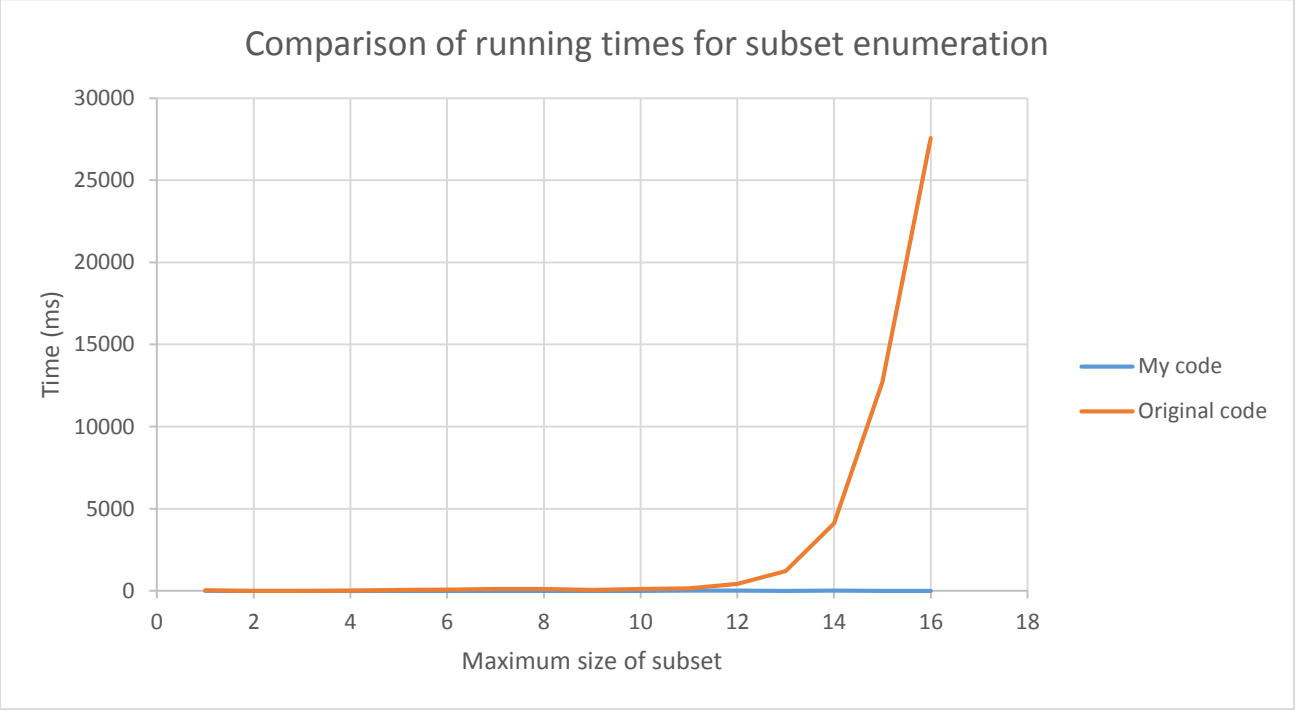
My extension was to improve the enumerateSubsets method in JoinOptimizer. My implementation is in the class file EfficientJoinOptimizer. It can be enabled/disabled by replacing any instantiation of JoinOptimizer with EfficientJoinOptimizer. PlanCache was not compatible with my change, so it was also copied and modified in EfficientPlanCache. My iterator class is called CombinationEnumerator and has an inner wrapper class over Boolean arrays called Enumeration.

The CombinationEnumerator takes a size of the set and the size of the subsets to return in its constructor. With each call to next(), it returns an Enumeration. You can loop over this Enumeration and ask for each index in the original vector whether or not it's in the subset. It is dramatically faster than constantly cloning and modifying sets. However, I didn't want to ruin this speed up by creating those Sets after generating the subsets, so I modified all methods that took a set and changed them to take an

Enumeration. For example, computeCostAndCardOfSubplan took an Enumeration (and the vector) as an argument and used those two values together instead of building a bunch of new sets.

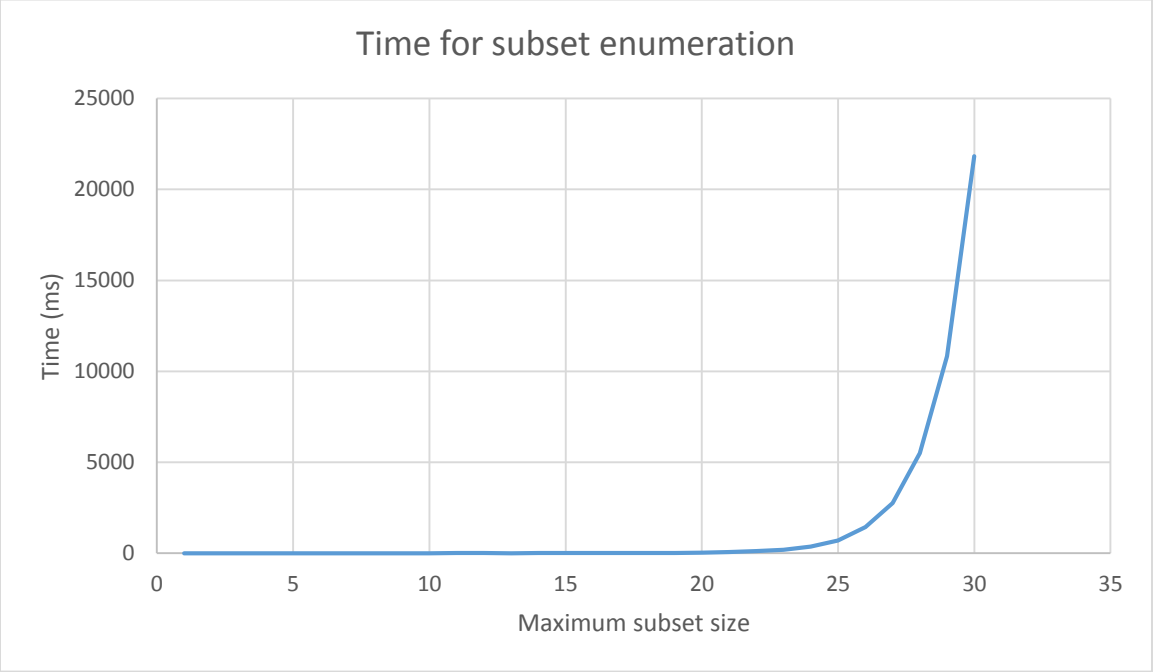
The CombinationEnumerator takes two arguments, the size of the master set, and the size of the subset to produce, we'll call these 'from' and 'choose' respectively. It works by creating a Boolean array of size 'from', and marking the first 'from' fields as true. Then, with each iteration, it moves the last true value one index forward. When it reaches the end, it removes all true values until it arrives at the first false value, recording the number of removed trues, which I'll call 'x'. It then increments the next true value forward one and writes 'x' trues after it. This continues until the last 'choose' values in the array are all true. This algorithm ensures every combination will be found.

I tested my extension by making sure it produced n choose k results and that all results are unique. This test can be found in test-extensions/simpledb/test/EnumerationTest.java. I also tested it against the old implementation. Using the values from JoinOptimizerTest.bigOrderJoinsTest, I got a speed up of over 5 times (8.422 vs 1.529) with a size of 14 joins. For a more representative demonstration of my speed up without all the overhead of the other work, I created a test that loops over increasing maximum sizes and outputs the time taken to produce all subsets of all sizes for that maximum size. The graph of those times is produced below. This used a modified version of EnumerationTest.speedTest.



As you can see from the data above, it's actually hard to compare them because mine scales so much better. In fact, at the point where it becomes infeasible to wait for the old code any longer (around 16), my code takes so little time it's not accurately measurable. Below is another graph, however, showing

that my method is still exponential, just with a much lower coefficient. In fact, I get almost twice the size for the same time, which is substantial when we're talking about exponential functions.



Executing Queries

Executing the provided query

```
The query plan is:
                            π(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    |
  |                                 ___________|___________              |
  |                                 |                       |            scan(casts c)
scan(actor a)                       scan(director d)        scan(movie_director m)

d.fname      d.lname
------------------------
Chris  Malazdrewicz

Thomas Parkinson

Alain  Zaloum
```

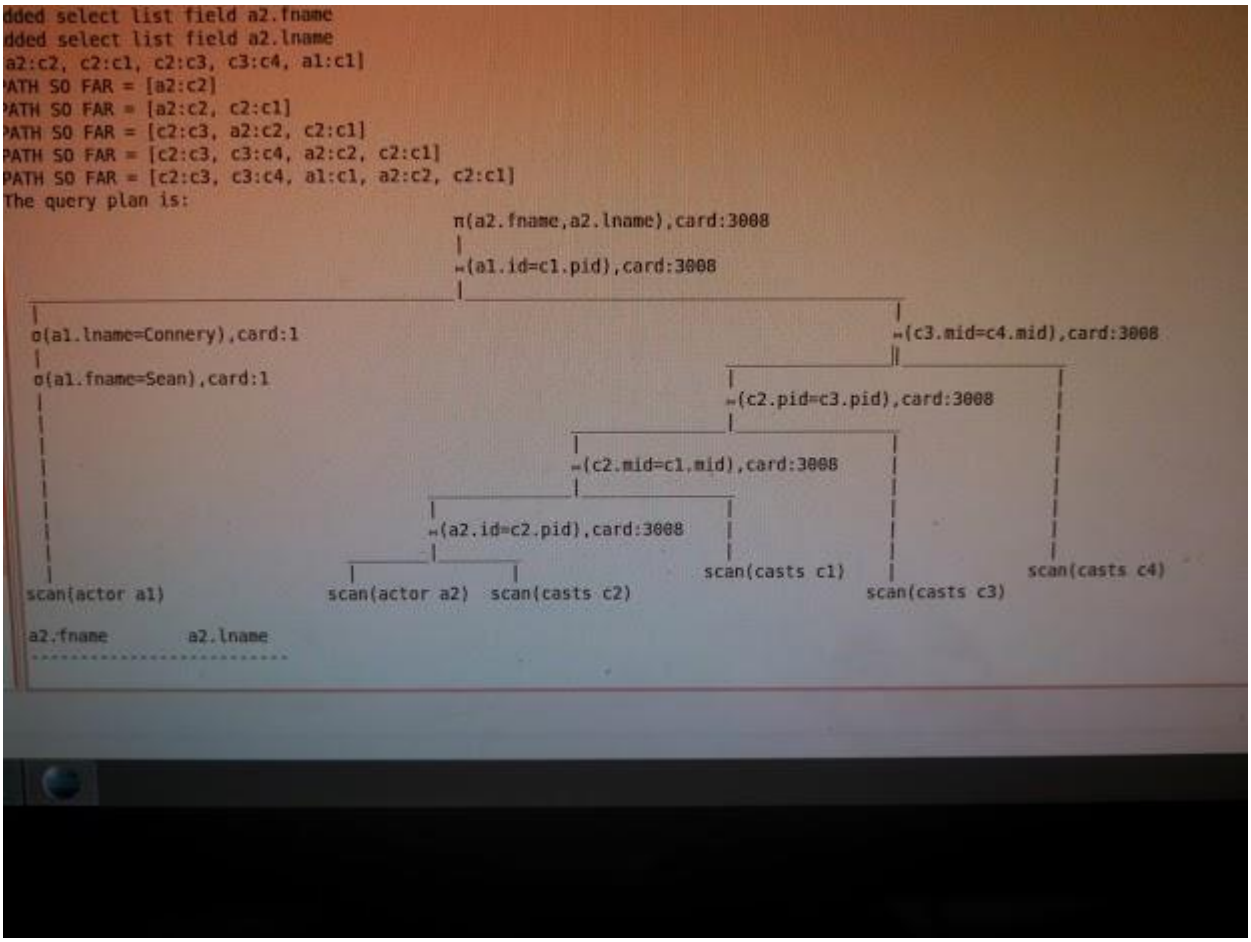
My optimizer picked this ordering by following the algorithm listed above. It tried to pick an order that would provide the lowest cost. For example, it joined the table with the smallest cardinality with the largest cardinality at the end, because the sizes of the two tables have the biggest impact on cost in the calculation (cost1 + card1 * cost2 + card1 * card2), so the larger it can pair with the smaller, the better.

The director and movie_director tables are the next smallest, so it makes sense to pair them up, so the intermediate table is also reasonably small. If movie_director joined to casts first, it would have a larger intermediate table for the second join, which would cause a higher cost, even if the output was the same.

Executing my query

My query is finding everyone with a bacon number of two for Sean Connery (since Kevin Bacon isn't in the set).
Query:

```
select a2.fname, a2.lname
from actor a1, actor a2, casts c1, casts c2, casts c3, casts c4
where a1.fname='Sean' and a1.lname='Connery'
and a1.id = c1.pid
and c1.mid = c2.mid
and c2.pid = c3.pid
and c3.mid = c4.mid
and c2.pid = a2.id;
```



The reasoning is much the same as above. It pairs the largest table with the smallest, and then tries to minimize the size of the intermediate table. Actors is the smallest, so join it with casts, then keep joining all the way up since it can't do anything else without doing a cross product.

Discussion

Performance

The performance of my simpleDB engine is pretty atrocious. While it did not seem to be a problem before this lab, I was failing QueryTest because of the timeout and could not figure out why. I even ran a profiler, and all it told me was I was spending the vast majority of my time reading in pages. This made little sense to me since I did not prematurely evict pages and the readPage code was mostly written for us. Unfortunately, it would have taken a very long time to solve this performance problem, and currently I don't have a lot of time. Thankfully, the timeout got extended so it's no longer a huge problem, just a disappointment.

One of my problems was that I started out trying to be really efficient, for example, making an index lookup in the heap page to find the index that a tuple is stored at. However, this just made things more complicated and didn't really improve performance, so I gave up on trying to do performance improvements. However, given more time I would want to implement the HashJoin so that I could actually run decent queries, since I can barely do any joining currently.

What I Learned

While I was skeptical about this project after the first lab, I do actually feel like I learned a substantial amount about how databases really work. While I assume there is some amount of magic that goes into making them run fast, I no longer feel like databases are some big black box that I'll never understand. It has substantially helped me understand some of the things I found odd about databases from past usage. It has also helped me understand the nested loop join implementation, which I never thought very deeply about before. But its slowness caused me enough problems with this assignment that it will forever be burned into my memory.

Changes

I've dumped all the notes I've taken on problems (much more heavily weighted towards earlier labs) below. A lot of problems I saw were stylistic or not following normal Java patterns. I hated anytime I had to cast, as it was usually just something missing from an interface that was easily fixable, but now I'm introducing a potential runtime error into my program. Furthermore, there was just some incorrect documentation or pieces that didn't seem to fit together well.

Lab1

```
test.java
The provided code for test.java should define a class Test for standard
naming conventions.
The provided code should use print, not println, since Tuple.toString appends
\n.

Catalog
addTable calls the other addTable with "". "" is a valid field name. You
could accidentally name a primary key.
addTable with no name calls addTable with a randomly generated String. This
could be a duplicate. It should test that
    it's unique first.
iterator instead of tableIdIterator.

DbFile
```

.writePage javadoc says PageId has a method pageno, when it's pageNumber.

DbFileIterator/DbIterator
Next should throw IllegalStateException when not open, not NoSuchElementException.

HeapFile
Hashing does NOT gaurantee uniqueness!
readPage should have throws IOException. FileNotFoundException is NOT the only IOException.

HeapPage
FIELDS SHOULD BE PRIVATE.
The iterator doesn't specify that it needs to be in the order as on disk, but the tests enforce that.
Why assume that a parse exception means that there isn't another Tuple?
Why does this take only a HeapPageId? It is LITERALLY no different from a PageId and there's no reason not to!

Page
This should be Iterable. Or at least have the .iterator method so we don't have to cast.

PageId
.pageNumber doesn't fit the standard naming convention. It should be getPageNumber.
Why is this an interface? This should be a class. There is no other information necessary and if needed, it could be
    added with a subclass. HeapPageId should just be PageId.

RecordId
.tupleno doesn't fit standard naming conventions. Should be getTupleNo.

Tuple
If a StringField is whitespace toString method could get screwed up and cause errors during parsing.
The .fields method should be .iterator to follow conventions.
Tuple implements Iterable<Field>

TupleDesc
TupleDesc implements Iterable<TDItem>
TupleDesc.getFieldName/getFieldType throws NoSuchElementException instead of the proper IndexOutOfBoundsException
TupleDesc.fieldNameToIndex doesn't match naming convention of other methods. getIndexByName is better.
TupleDesc.fieldNameToIndex should return -1 if not found, not throw an exception. They're slow and poor style.
TDItem fields should be private on public inner classes.

General
Classes don't test instanceof in .equals methods.
NoSuchElementException is an UNCHECKED exception. It does NOT belong in the method header!
The project doesn't use proper package namespacing conventions.

Lab2

Aggregate

The @link in the constructor says IntAggregator instead of IntegerAggregator.
The constructor header should specify Aggregator#NO_GROUPING, not -1.
aggregateFieldName/groupFieldName doesn't follow standard naming.

Aggregator
You don't specify the behavior if the results are empty.
The naming convention is off (and 'what' is a terrible name).
This should either take a TupleDesc or names for the columns so I don't have to implement it multiple times...

Field
Fields don't bother to check types before doing compare.

HeapFile
Going through the BufferPool means it's returned as a Page, so I have to cast to a HeapPage, which sucks.

DbFile
You don't build into the interface a way to add a new page to a file so I have to cast again.

Lab3

BufferPool
We check if a transaction holds a lock for a page... But not WHICH lock it holds!

Page.java
Any method that starts with is* should return a boolean in proper Java convention.