Jasper Palmer and Edison Chau

jpalmer4 and edichau

CSE444 Final Report

Section 1. Overall System Architecture: (2 page)

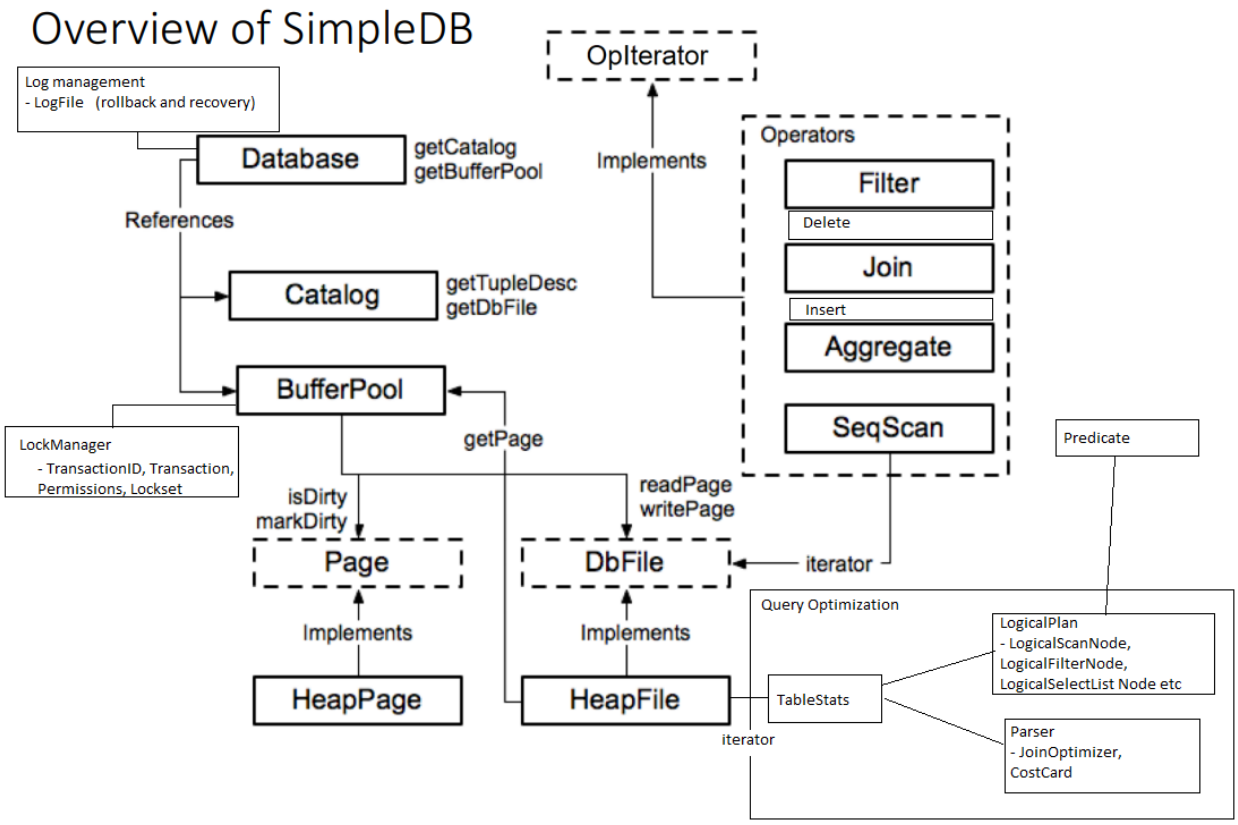
An overview description of the overall architecture of your final SimpleDB system (0.5 page)

SimpleDB is built on a set of smaller subsystems that makeup the overall database. These components include: the log management system, the lock management system, the buffer management system (i.e. the buffer pool), and the operation system. Each of these systems is built out of smaller foundational blocks that encapsulate the standard database components. Some examples of these are the Tuple, HeapPage, HeapFile, etc. While each of these subsystems play their own roles, their work is not exclusive and they often work together to complete tasks that are requested by the end-user.

In SimpleDB, the lock and buffer management systems always work together to create “transaction” behavior within the database. This is because most other elements of the database utilize the buffer pool to access the pages of the database’s files. Due to the consistency in this choice across the system, the lock manager only needs to be interacted with when the buffer pool is. When pages are requested, the buffer pool can provide them if the lock manager allows the locks to be obtained. Then they can be cleared when the transaction comes to an end. On the other hand, the implementation of the lock manager is not heavily coupled with the buffer pool. The lock manager does not need to be present, or it could be completely replaced with a different implementation as long as its external interface (that the buffer pool uses) remains consistent with the original version. In our final version the buffer pool does heavily rely on the lock manager since it uses it to assess whether or not it should be performing cycle detection while keeping track of the wait-for dependency graph.

Outside of these subsystems, there are also the Database and Catalog that are the higher levels of access to the information that is in the database. The Database holds static references to the buffer pool, log file, and Catalog. Then the Catalog allows for interactions with the tables that are within the database and the associated DBFiles. This demonstrates the chain of interaction from the user, to SimpleDB, to the Database, to the Catalog, to the DBFiles, to the DBPages, to the Tuples, and finally to their fields (where the data is stored). This is all abstracted from the user perspective and they just see the high level concepts interacting through SQL.

A diagram of the overall system architecture. You may modify our diagram from the early slides. (0.5 page)



A somewhat more detailed description of each of the following main components of your SimpleDB system: Buffer manager, operators, lock manager, and log manager (0.5 page)

One component is the `BufferPool` (our version of a buffer manager), which keeps track of the pages that have recently been accessed (i.e. it functions as a cache of pages). The `BufferPool` also has the ability to flush and evict pages from its cache. While it manages the buffer, this component cannot stand alone because it does not have its own method for reading the pages from disk. This is where the `HeapFile` comes in. It has a method for reading pages from disk into itself (since it is a representation of a file within the database). Furthermore, it can insert and delete tuples that are stored in the file. When it reads a page, it can provide it to the `BufferPool` that can cache the page for quicker retrieval later. An even lower level component within the hierarchy is the `HeapPage`. It is a random collection of `Tuple`s that are used by the `BufferPool` and `HeapFile` as an intermediary to simultaneously store them in groups and to access the `Tuple`s themselves. This backs the insert and delete functionality that the `BufferPool` and `HeapFile` provide. It also allows the system to keep track of which sets (pages) of tuples are dirty at any given time.

Operators are another key piece of the SimpleDB system. These come in many flavors such as Filter, Join, Aggregate, Insert, and Delete. Each of these operations are components of queries that allow us to create and modify a stream of tuples or the tuples on disk (in the case of insertion and deletion). Most of these operations are built predicates which allow us to specify conditions under which the operation should be performed. For example, Filter uses a predicate to achieve its goals and iterates over the child's tuples while only keeping the ones that match it. The `Predicate` itself just checks whether the condition it stores holds for the tuple that it is given.

Transactions are included as an important aspect of SimpleDB and are handled with Strict Two Phase Locking. To handle the locking we created a `LockManager`. It keeps track of which transactions are holding locks on each page, whether the locks are exclusive or shared, and which pages each `Transaction` has a lock on. This is the base functionality that the `BufferPool` utilizes to check whether a transaction should get a page when they call `getPage()`. With locking comes the possibility of deadlocks—where there are cyclical dependencies between more than one transaction. While the `LockManager` does not handle this checking directly, the SimpleDB system does detect them and aborts transactions accordingly. Transactions also support aborting and committing which are not directly part of the `LockManager`, but they release all of the locks held by a transaction and remove dependencies from the dependency graph so that cycle detection functions correctly.

For our version of a log manager, we used undo-redo logging and based it on the Aries protocol. This involves two primary aspects: rollback and recovery. For rollback, like the Aries protocol we went through and undid all of the update entries in the log by setting the impacted pages from the given transaction back to their before images and clearing the `BufferPool` so that the incorrect values are not read. For recovery, we performed analysis followed by redo and then undo phases. Unlike Aries, we did not choose to implement compensating log records since they were not necessary to the recovery algorithm.

Lab 5 answers to the question prompts in the lab 5 spec (0.5 page)

Exercise 1: Parser.java

When you launch SimpleDB, the entry point of the application is simpledb.Parser.main().

Starting from that entry point, describe the life of a query from its submission by the user to its execution. For this, list the sequence of methods that are invoked. For each method, describe its primary functions. When you describe the methods that build the physical query plan, discuss how the plan is built. Write the answers in your lab 5 writeup document.

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()

- Builds logical plan by calling simpledb.Parser.parseQueryLogicalPlan
- Finds the optimal plan by calling the logical plans physicalPlan method

Step 4: simpledb.Parser.parseQueryLogicalPlan()

- Call simpledb.LogicalPlan.addScan() to add table to the logical plan for each table in FROM clause
- Process WHERE clause by calling parseQueryLogicalPlan simpledb.Parser.processExpression to add the filter into the logical plan
- Process the GROUP BY clause by identifying the attribute and setting the value of the GROUP BY field.

- Process the SELECT clause by identifying the aggregation field using getValue(), identifying the method using getAggrgate() and using simpledb.addProjectField().
- Process ORDER BY by identifying the attribute to ORDER BY by calling getOrderBy(), call getExpression to process the result, identify the attribute to order and the ordering then Use addOrderBy() to add results to the logical plan.
- Return the logical plan

Step 5: simpledb.LogicPlan.physicalPlan()

- For each table, get the table name, SeqScan, put it into the hashmap then set the selectivity of the table
- For each filter, get the subPlan iterator, get the field, field type and TupleDesc. Add the name of the table and filter to hashmap, then get the tableStats to get selectivity, put current selectivity multiplied by new selectivity into hashmap.
- For each join, get the first and second plans, use a join optimizer to produce new iterator. If join not subquery join update the name of table2 and keep map updated by set all tables with name equal to t2 to t1. Then check for remaining subPlan
- For each selection, collect filed nums and types, generate aggregation nodes if there is an aggregate, generate orderby, if there is an orderby iterate over the selection list.
- Return new project.

6.1 Execute the following 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';

In your writeup, show the query plan that your optimizer selected. Explain why your optimizer selected that plan. Be careful as the plan may be different for the 1%, 0.1%, and 10% datasets (you do not need to test with all the datasets, just pick one).

6.2 Execute another SQL query of your choice over the IMDB database.

Show the query plan that your optimizer generates. Discuss why your optimizer generates that plan. Try to find an interesting SQL query with a combination of joins and selections.

Section 2. Discussion:

Discuss the overall performance of your SimpleDB engine.

Our SimpleDB engine is reasonably performant. While we did not make the extra effort to implement the extensions that would have elevated the performance of our engine, we were diligent in making sure that all of the code that we included was performant and relatively optimized. For example, we opted not to implement anything more advanced than a nested loop join for our join algorithm (as opposed to a more efficient option like a hash join).

One area that ended up being less performant than it could have been was our handling of the cycle detection within the buffer pool—for deadlock prevention. This algorithm used custom ADTs for storing pairs of `TransactionIds` and `PageIds`, and used frequent nested looping to make sure that all the waiting relationships and their cleanup were handled correctly. This was likely an overly-aggressive implementation that could have been better optimized for a more performant SimpleDB engine. This would likely have a significant impact since the detection occurs within `getPage()` which is called frequently across the SimpleDB engine.

Discuss what you would implement or what you would change in the implementation if you had more time.

Two of the additional changes that we would add to our SimpleDB engine are directly related to the discussion in the prior section about the performance.

The first is extending the `Join` operation by replacing its use of a nested loop join for a more performant alternative like a hash join. This would only speed up the queries that utilized join operations specifically, but the increase in performance would be significant.

The second extension would be a better cycle detection algorithm that uses less resources and does not occur when unnecessary. There are likely more clever indicators of when a deadlock is occurring that we could use to prevent abortion of transactions that will eventually make progress (i.e. detecting the liveliness of the transactions).

The third extension we would implement would be to improve the rollback and recover to be more similar to the Aries protocol. When writing our implementation of logging, we elected to skip Compensation Log Records (CLR) since they were not essential and served for a more interesting (challenging) implementation. However, it would have likely been more efficient, similar to Aries, and straightforward to use CLRs.