SimpleDB Final Report

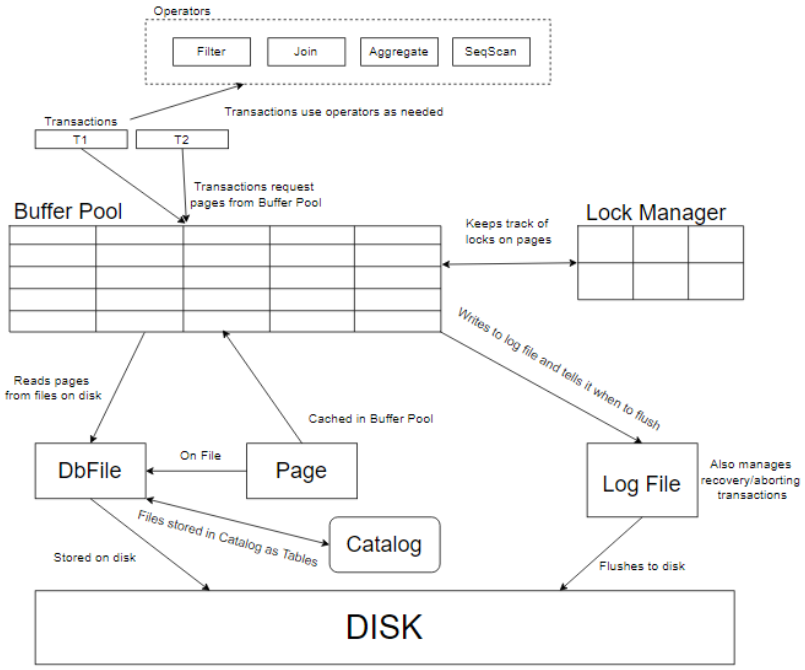
Jackson Atkins

6.9.2022

What is SimpleDB?

The SimpleDB system is a simple database system that allows for multiple methods of concurrency control designed with ease of use in mind. To accurately give a description of the architecture of this system, we will first look at adding tables and data to the system, and then the process of executing a query on this data. Adding a table begins with the Catalog in SimpleDB, as well as the DbFile. Keeping track of the connection between tables and files is the backbone of all processes in this database system. Catalog will also be seen later when we need to add data and perform queries on the database. Now that we have the table in the database, we need to add some data to perform queries on. Before requesting a page from the BufferPool, the transaction that wants to insert a tuple must acquire a lock, and the BufferPool will write this action to the log file. This is handled in the LockManager class, which tracks the locks that are held on each page contained in the database. After acquiring the proper lock, the transaction will receive the request page from BufferPool, where the correct function is called to insert a tuple. The BufferPool acts as a home for all the pages that we need to keep in memory, as well as being responsible for evicting pages once it is full. The BufferPool in turn looks to the HeapFile and HeapPage classes, which directly deal with the modification of data on a file and page level respectively. Using an iterator, the HeapFile looks for an open slot in any of its pages to insert the new tuple, eventually deciding to create an entirely new page if it cannot find an available slot. Finally, we can now execute a query on the database. The first step of a query is calling the Parser, which in turn grabs the correct table from the Catalog. The Parser then uses the Operators (described below) and the Join Optimizer to generate and perform a query plan. After generating the plan, transactions are created, and the operators are performed on the necessary tables. Finally, the query is executed, described in further detail below.

Visualization of SimpleDB



An In-Depth View of SimpleDB

To take a deeper look at SimpleDB, we will focus on the main four components of the system: the Buffer Manager, the Operators, the Lock Manager, and the Log Manager.

Buffer Manager:

The Buffer Manager in SimpleDB acts as a cache of all pages currently being used by active transactions, as well as pages that were previously used by now committed or aborted transactions. Furthermore, the Buffer Manager is also responsible for evicting pages as necessary, as it has a strict limit on the number of pages it can keep in memory. To return a page when a page is requested, the Buffer Manager will check in its cache to see if the page is available and read the page from disk if necessary. In addition, the Buffer Manager is also responsible for flushing pages to disk at the time described by the current policy being used (either STEAL/NO FORCE or NO STEAL/FORCE). It also works very closely with the Lock Manager, as it is the first step in marking a transaction as complete and performs the necessary actions related to completing a transaction, such as releasing all held locks. It also needs to make sure that locks are acquired whenever a page is read (either from disk or from memory), once again showing how closely it works with Lock Manager. This is because this is one of the few places where locks are acquired and released within SimpleDB. Lastly, it works with the Log Manager as described below.

Operators:

Operators represent the piece of SimpleDB that allows for queries to be performed. SimpleDB supports a variety of operators, including JOIN, FILTER, Aggregates, GROUP BY, ORDER BY, PROJECT, and numerical operators such as ‘<’, ‘=’, and ‘!=’. Without these operators, SimpleDB would be a collection of Tables that could not be queried by a user. Each of these operators are structured as Iterators that fetch the tuples passed to them and perform their corresponding action. In addition, these operators act as a child/parent relationship, passing completed tuples to the next operator in the query plan tree. These operators are in turn used by transactions to perform queries on tuples in the database. Furthermore, the JOIN operator has a special JoinOptimizer class build solely for the purpose of optimizing all joins in any given query. We can estimate the cardinality of these operators using an IntHistogram class that determines the selectivity factor of a given operator on any table contained in the database.

Lock Manager:

The Lock Manager in SimpleDB is incredibly important for pessimistic concurrency control. As described above, locks are mainly acquired and released through the Buffer Manager, which in turn calls the Lock Manager to modify said locks. Within the Lock Manager class, we keep track of the shared and exclusive locks on all pages by using two separate HashMaps. This allows the entire database to know what pages are locked at any given moment, and what transactions hold locks on them. Furthermore, there is also deadlock detection implemented with a dependency graph. Each time a lock is to be acquired, we check and see if there is a deadlock (of any length) within the dependency graph using BFS and throw an error accordingly. This allows the system to fail fast, rather than waiting an infinite amount of time.

Log Manager:

The Log Manager is also incredibly important but focuses on optimistic concurrency control as opposed to the pessimistic concurrency control described above. The Log Manager works with the Log File to correctly implement the STEAL/NO FORCE policies described in Lab 4. As such, the Log Manager is responsible for keeping track of every action taken by the currently running transactions, working closely with the Buffer Manager to do so. In addition, the Log Manager is also responsible for dealing with rolling back aborted transactions, as well as recovery after a system crash. It does so by looking through the log file it has and undoing and redoing actions accordingly. As described below, my implementation of a Log Manager does not use CLR records, instead opting to automatically rollback all ABORT records during the recovery process, even if it takes extra time. The Log Manager is vital for system recovery after a crash, as it is the only class that can handle such a task. Without this, it would be completely impossible to implement STEAL/NO FORCE in SimpleDB, and a system crash would result in a total loss of data.

Performance Analysis

To test the performance of my implementation of SimpleDB, I will run 5 queries and determine their runtime using the 1% IMDB database given in Lab 5 and the DBLP Dataset given in Lab 2.

Query	Runtime	Dataset
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';	850ms	IMDB 1% Dataset
SELECT d.lname, count(m1.id) FROM Director d, Movie_Director m, Movie m1 WHERE d.id=m.did AND m.mid=m1.id GROUP BY d.lname;	730ms	IMDB 1% Dataset
SELECT MIN(m.year) FROM Actor a, Casts c, Movie m WHERE a.id=c.pid AND c.mid=m.id AND a.fname='Tom' AND a.lname='Cruise';	670ms	IMDB 1% Dataset
SELECT p.title, v.name FROM papers p, authors a, paperauths pa, venues v WHERE a.name='E. F. Codd' AND pa.authorid=a.id AND pa.paperid=p.id AND p.venuid = v.id;	2.12s	DBLP Dataset
SELECT a2.name, COUNT(p.id) FROM papers p, authors a1, authors a2, paperauths pa1, paperauths pa2 WHERE a1.name= 'Michael Stonebraker' AND pa1.authorid=a1.id AND pa1.paperid=p.id AND pa2.authorid=a2.id AND pa1.paperid=pa2.paperid GROUP BY a2.name ORDER BY a2.name;	3.23s	DBLP Dataset

As we can see from the fast runtimes, my implementation of SimpleDB handles a wide variety of queries on varying datasets in an efficient and responsive manner.

What can be Added

There are three things that I would add to my implementation if I had more time. First, I would go back and add CLR records to my Log Manager implementation. While I was able to successfully implement all the necessary elements (i.e., recovery) without using CLR records, I feel that it would make my database log records much easier to comprehend and would avoid the workarounds I had to implement. In addition, I would add a better search method for detecting cycles in the precedence graph I used in my Lock Manager implementation. Once again, my current implementation works correctly, and detects cycles that are longer than length 2, but it could be improved upon. Implementing the Lock Manager deadlock detection with timers was an option I did not implement, but I would like to go back and see if they have better performance than my current precedence graph design. Lastly, I would implement a better Join Optimizer than the one I currently have, most likely by vastly improving enumerateSubsets(). As stated in the lab specification, this currently has a slow implementation that I feel could easily be improved. However, due to time constraints I was unable to go back and add these changes. Overall, I am

quite happy with my implementation of SimpleDB even without these previously described additions/changes.

Lab 5

API Changes: There was a small change made in QueryPlanVisualizer, as I altered the String representation of relational algebra terms. This was due to the fact that I received an error prior to making these changes (as described on Ed). However, this did not change any of the decisions made by the QueryOptimizer.

Exercise 1:

Steps 1 and 2 are described in the Lab 5 specification.

Step 3: Parser.handleQueryStatement()

This method takes two key actions:

- First, it parses the logical query plan by calling parseLogicalQueryPlan((ZQuery) q)
- Then it retrieves the optimized physical plan by calling LogicalPlan.physicalPlan(Map<String, TableStats> baseTableStats)

Step 4: Parser.parseLogicalQueryPlan()

- This method initially adds scans to the final logical plan by searching for all valid tables in the FROM clause
- Then it processes expressions in the WHERE clause and creates the proper Filter and Join nodes as needed
- Next, it parses the GROUP BY statements and sets the proper group by field (only one is allowed)
- Walks through the SELECT list and picks out aggregates while checking for Query validity (e.g., only one aggregate, cannot select fields that are not grouped, etc.)
- Lastly it sorts the data if the ORDER BY field is defined and returns the final logical plan

Step 5: LogicalPlan.physicalPlan()

- The method initially iterates through all tables defined in the logical plan and adds them to the initial subplan map defined in the class. It also adds the table to the statsMap and defines the table's filter selectivity as 1.0 by default (as it has simply been scanned).
- Next, it iterates through the filters. For each filter, it retrieves the logical filter node and subplan associated with it. Next, it adds the name of the logical filter and the corresponding filter to the subplanMap. It then retrieves the tableStats and uses them to estimate the selectivity of the filter, multiplying the new selectivity by the current selectivity and adding it to the filterSelectivities map.
- The most important part of generating the physical plan is iterating over the joins. For each join, it gets the left side and the right side of the join, setting their names in the process. Next, it uses the JoinOptimizer's instantiateJoin to determine the best type of

join to use depending on the stats of each join, returning an OpIterator in the process. It then adds this new join to the subplanMap. However, if this join is not a subquery join, it removes it from the map and updates the name of t2 in the equivMap (as the new node contains both tables). It then sets all tables whose name is equal to t2 to t1 to keep the map updated. Finally, it verifies that there is only one subplan left (i.e., all joins were combined).

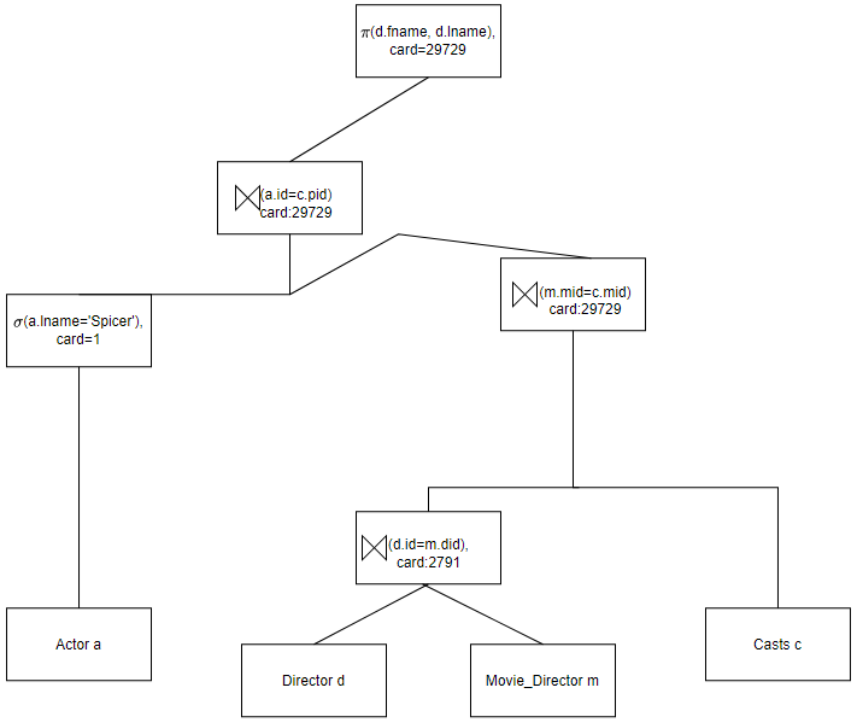
- The method then searches the select list to determine the best way to project the output fields. For each selection, it collects the number of output fields generated and their corresponding types. Furthermore, it also generates aggregation nodes as necessary, verifying that all fields are correctly accounted for. Before moving on to the final step, it creates an OrderBy node if the operation is defined in the logical query plan.
- Lastly, it returns a new Project operator given the defined list of fields and their corresponding types, as well as the combined OpIterator generated over the entire method (node).

Step 6: Query.execute()

The last step is to execute the query. After setting the Logical and Physical query plans as described above, the Parser calls Query.execute(), which uses the operators described by the physical plan to perform the query and output the final result to the user.

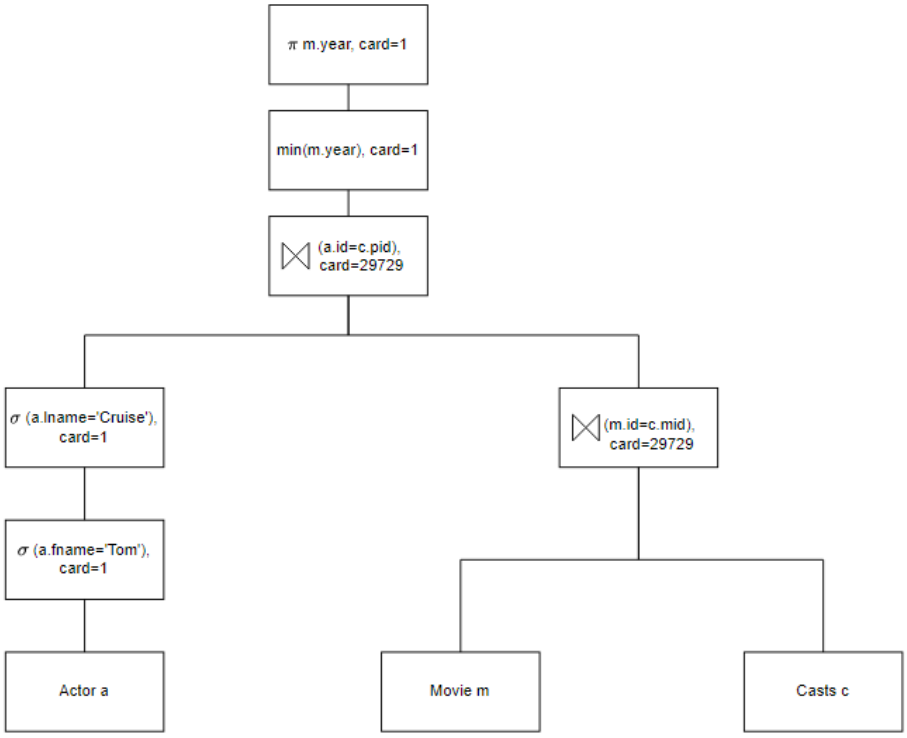
Exercise 6:

Query 1 (1st query in the table above):



The optimizer initially chose to simply scan the Actor table and determine the one tuple corresponding to the last name “Spicer”. It makes sense to do this in the beginning, as we avoid joining on the entire Actor relation, which would be quite expensive due to the large cardinality of the table. Next, we perform simple joins between the Director and Movie_Director relations, as well as between the output of the above join and the Casts relation. These decisions were made because of the overall join cost and the small output cardinality of the initial join. After performing these two joins, the optimizer joins with the Actor relation. Rather than searching through the entire Director, Movie_Director, or Casts relation to join with the one tuple on the left side of the plan, the optimizer chooses to perform this join at the second to last step in the plan. None of these joins were performed using a hash join, due to the large size of the relations leading to an expensive cost. Lastly, the optimizer simply selects the columns as described in the query.

Query 2 (3rd query in table above):



Like Query 1, the optimizer initially scans the Actor table and selects the one tuple with the name “Tom Cruise”. This is done for the same reasons as described above. The optimizer then performs a join between the Movie and Casts relations, choosing once again to join with the one tuple at the very end of the plan. After this join, the optimizer is forced to finally join with the Actor table, joining on the Actor id. Lastly, the plan selects the minimum year from the final tuple.