CSE444 Final Report

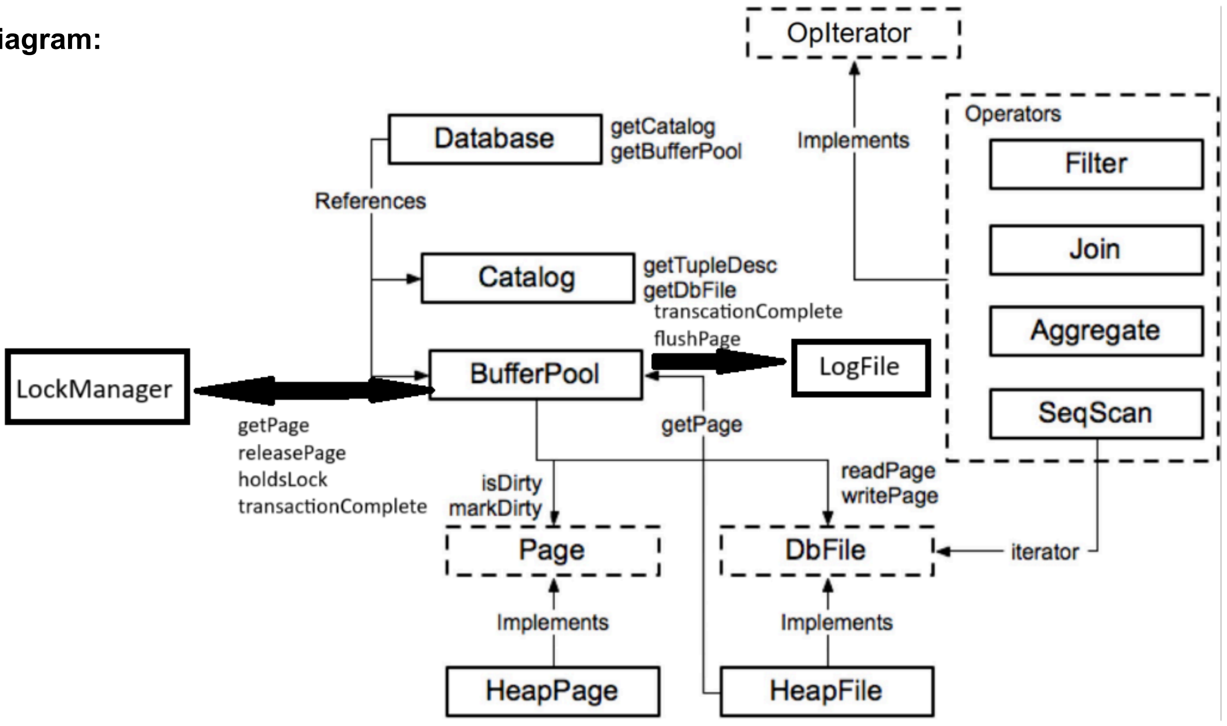
Huakun Xu
06/12/2025

Overall System Architecture

Overall Description:

The SimpleDB system is a simple database framework designed for ease of use while supporting multiple concurrency control methods. To thoroughly describe its architecture, we will first examine the process of adding tables and data, followed by executing queries. Creating a table begins with the Catalog in SimpleDB, along with the DbFile. Managing the connection between tables and files is central to all database operations. The Catalog plays a role not only in table creation but also in later stages when adding data and performing queries. Once a table is in place, data must be inserted for queries to be executed. Before requesting a page from the BufferPool, any transaction attempting to insert a tuple must first obtain a lock. The BufferPool logs this action, with lock management handled by the LockManager class, which tracks active locks on each page within the database. After securing the necessary lock, the transaction receives the requested page from BufferPool, where the appropriate function processes the tuple insertion. The BufferPool stores frequently accessed pages in memory and also manages page eviction when necessary. To modify data at the file and page level, the BufferPool interacts with the HeapFile and HeapPage classes. Using an iterator, HeapFile scans its pages for available slots to insert a new tuple. If no free space is found, it creates a new page to accommodate the tuple. With tables and data in place, queries can be executed. Query processing starts with the Parser, which retrieves the relevant table from the Catalog. The Parser then utilizes Operators and the Join Optimizer to generate and implement a query plan. Once the plan is formulated, transactions are initiated, and operators execute operations on the required tables. The final execution of the query follows, with further details later.

Diagram:



More Detailed Description:

The four main components of SimpleDB are BufferPool, the Operators, Lock Manager, and Log Manager.

The Buffer Manager in SimpleDB serves as a cache that stores pages actively used by ongoing transactions, as well as those previously utilized by transactions that have since been committed or aborted. Additionally, it is responsible for evicting pages when necessary, as there is a strict limit on the number of pages it can retain in memory. When a page is requested, the Buffer Manager first checks its cache for availability; if the page is not present, it retrieves it from disk. Furthermore, it manages the flushing of pages to disk according to the current policy in use, such as STEAL/NO FORCE or NO STEAL/FORCE. The Buffer Manager works closely with the Lock Manager, as it plays a crucial role in marking a transaction as complete. It oversees essential transaction finalization tasks, including releasing held locks. Whenever a page is read—whether from memory or disk—it ensures that appropriate locks are acquired, highlighting its tight integration with the Lock Manager. Since locks are acquired and released primarily within the Buffer Manager, it serves as one of the key points of concurrency control in SimpleDB. Finally, the Buffer Manager collaborates with the Log Manager, as outlined in the following section.

Operators in SimpleDB are fundamental for executing queries. The system supports various operators, such as JOIN, FILTER, Aggregates, GROUP BY, ORDER BY, PROJECT, and numerical operators like $<$, $=$, and $!=$. Without these, SimpleDB would simply be a collection of tables with no means for users to retrieve meaningful data. Each operator functions as an Iterator that processes incoming tuples and applies its respective operation. Additionally, operators are structured in a hierarchical fashion, where parent operators receive processed tuples from child operators within the query plan tree. These operators play a key role in transaction execution, enabling interactions with database tuples during query processing. Moreover, the JOIN operator is enhanced with a specialized JoinOptimizer class, dedicated to optimizing joins in any given query. To estimate operator cardinality, SimpleDB leverages an IntHistogram class, which determines the selectivity factor of an operator when applied to a specific table.

The Lock Manager is a critical component of SimpleDB's approach to pessimistic concurrency control. As mentioned earlier, locks are primarily acquired and released with the Buffer Manager, which delegates lock modifications to the Lock Manager. Within the Lock Manager class, two separate HashMaps are used to track shared and exclusive locks on pages, allowing the system to monitor which pages are locked and which transactions currently hold those locks. Additionally, the Lock Manager implements deadlock detection using a dependency graph. Before acquiring a lock, the system checks for existing deadlocks—regardless of their length—using BFS (Breadth-First Search). If a deadlock is detected, an error is thrown immediately, enabling the system to fail fast rather than waiting indefinitely for resolution.

The Log Manager plays an essential role in SimpleDB, particularly in optimistic concurrency control, contrasting with the pessimistic concurrency approach managed by the Lock Manager. It collaborates with the Log File to correctly implement STEAL/NO FORCE policies, as described in Lab 4. One of its primary responsibilities is maintaining a record of every action performed by currently executing transactions, working closely with the Buffer Manager to track changes. Additionally, the Log Manager oversees the rollback of aborted transactions and system recovery following a crash. By analyzing the log file, it reverses or re-applies operations as needed. Log Manager opts to automatically roll back all ABORT records during recovery, even if this approach requires additional time. The Log Manager is indispensable for restoring the system after a crash, as it is the only component capable of handling recovery. Without it, implementing STEAL/NO FORCE would be impossible, and a crash would lead to total data loss.

Lab 5 Answers:

1. Starting Point: Parser.main() and Parser.start()
   The entry point is the Parser.main() method, which invokes Parser.start():
   - It loads the database schema
   - Computes table statistics using TableStats.computeStatistics().
   - Begins a REPL loop, processing one query at a time with processNextStatement(...).
2. Query Processing: Parser.processNextStatement()
   Each user input is parsed into a Statement. If the statement is a query (ZQuery), it is handled by handleQueryStatement((ZQuery) s), which does the following:
   - Builds a LogicalPlan: LogicalPlan lp = new LogicalPlan(q, stats, explain);
   - Generates the physical plan by calling: DBIterator physicalPlan = lp.physicalPlan();
   - Executes the plan with: query.execute();
3. Logical Plan Construction: LogicalPlan.physicalPlan()
   This method generates an optimized physical plan from the logical query:
   - It constructs a graph of LogicalJoinNodes.
   - Calls JoinOptimizer.orderJoins() to compute the optimal join ordering.
   - Calls JoinOptimizer.instantiateJoinOrder(...) to build the corresponding tree of DBIterators.
4. Cost-Based Join Optimization: JoinOptimizer.orderJoins()
   This method implements dynamic programming to determine the most efficient join order:
   - It explores subsets of joins, recursively computing optimal subplans.
   - Utilizes cost and cardinality estimates with:
     - estimateJoinCost(...)
     - estimateJoinCardinality(...)
     - TableStats.estimateSelectivity(...)
5. Building the Plan: JoinOptimizer.instantiateJoinOrder()
   Once the optimal join ordering is chosen, this method constructs the actual plan:
   - Base tables are wrapped in SeqScan.
   - Joins are wrapped in Join.

   Resulting in a DBIterator that can be executed to return query results.
6. Execution: Parser.query.execute()
   Finally, the query plan is executed:
   - The DBIterator is opened and iterated.
   - Tuples are printed to the terminal.

Discussion

Overall Performance

My implementation of BufferPool uses an LRU replacement policy to manage limited memory space. This choice balances simplicity with reasonably good empirical performance across normal workloads. However, the performance could drop if there are workloads with looping access patterns or large scans that evict frequently reused pages. An improvement could be LRU-K to reduce overhead

My implementation of JoinOptimizer uses nested-loop joins which is simple to implement, but has quadratic performance in the worst case and does not scale for large relations. An improvement might be block nested-loop joins to reduce I/O by buffering inner tuples.

My implementation of IntegerAggregator keeps per-group state in a Map<Field, Integer> and iterates over all tuples to update aggregates. However, this does scale well with large groups and high cardinality. An improvement could be using an OpenHashMap instead so it works better for large groups.

My overall performance is pretty good for simpler or smaller datasets but when faced with larger datasets it falls short.

Changes

There are two 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 most of the necessary elements without using CLR records, I feel that it would make my database log records much easier to comprehend and might help with fixing the issues I had with the following tests: AbortCrash, CommitAbortCommitCrash, and OpenCommitCheckpointOpenCrash. Second, I would go back and add deadlock resolution & locking granularity at tuple level (only implemented page level). While my implementation passed all the tests, I felt like I could have done more to improve the concurrency control to work better. However, due to time limitations I wasn't able to add these features. Overall, I am satisfied with my implementation of SimpleDB in spite of the flaws it has.