CSE 444: SimpleDB Final Report

Linxing Preston Jiang Winter 2018

March 13, 2018

---

1 Overall System Architecture

In a typical database architecture, there are four main components: Process Manager, Query Executor, Share Utilities, and Storage Manger (Balazinska, Mass, Lecture 3). For our implementation of SimpleDB in the labs, we focus on the Storage Manager and Query Executor: adding access methods for data store on disk in lab1, adding both file mutability (insert/delete tuples to/from file system, eviction from full BufferPool) and query operators (SeqScan, Join, etc.) & aggregates (min, max, etc.) in lab2, adding lock manager in lab3, adding log manager in lab4, and finally, adding parallel data processing in lab6.

Figure 1 shows the parts of the architecture of SimpleDB which we implemented in the labs.

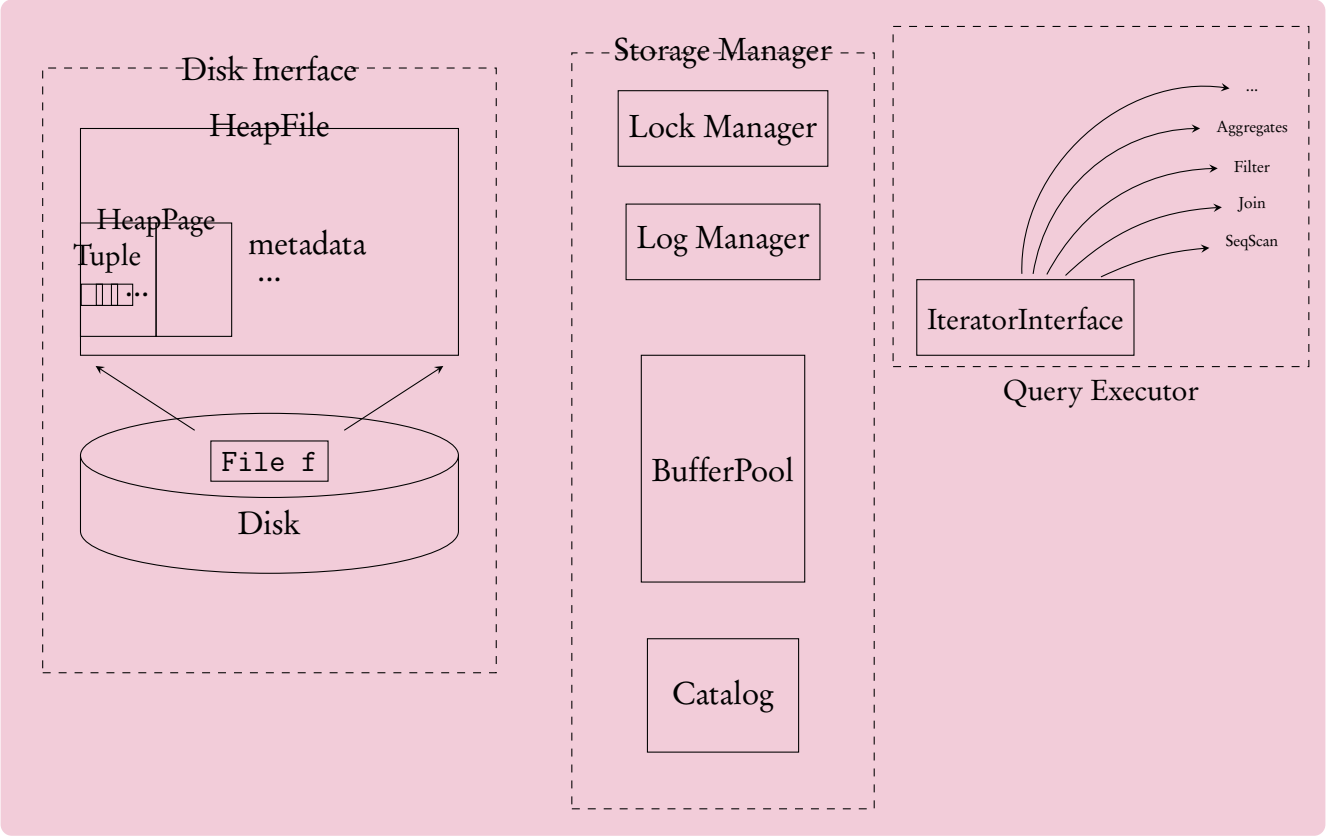

Figure 1: SimpleDB Architecture

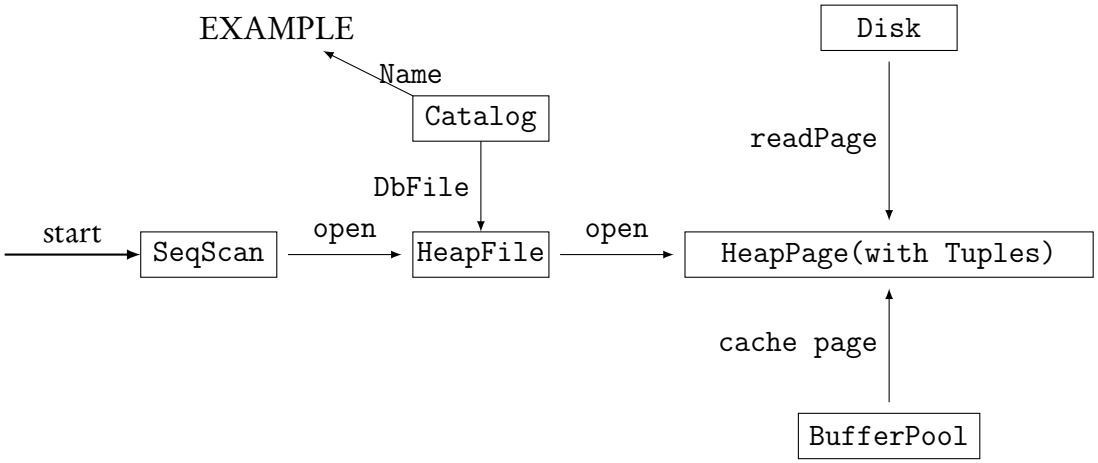

Figure 2: SimpleDB open

1.1 BufferPool and Opertors

BufferPool is responsible for both caching pages in memory that have been recently read from disk and handle concurrency and transactions. All operators read and write pages from various files on disk through the buffer pool. Operators are responsible for executing query plans. In SimpleDB, each operator implements the OpIterator interface, which supports open, hasNext, next, rewind, and close. Operators are connected together into a plan by passing lower-level operators into the constructors of higher-level operators (Lab1 ReadMe). Programs call next on the root operator and then fetch tuples recursively through the plan tree in one pass top-down and another pass bottom-up.

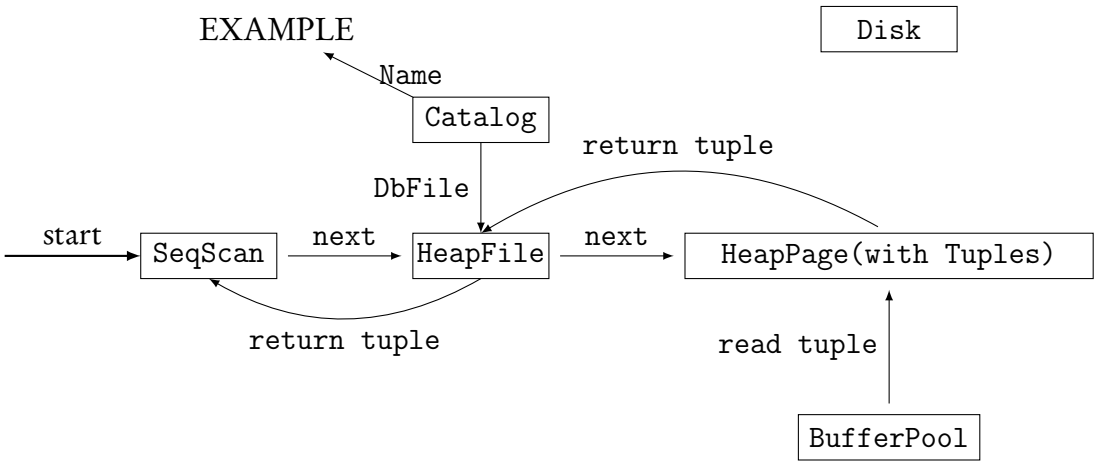

Figure 3: SimpleDB next

In Lab 1, we implemented getPage which is needed for reading Pages into memory and getting tuples, together with SeqScan operator to scan the file and return tuples. The workflow of opening opertators to read file and caching pages into BufferPool is shown by Figure 2, and the workflow of recursively getting tuples through SeqScan using next() is shown by Figure 3.

In Lab 2, We added insert, delete funcionalities in SimpleDB, which insert/delete tuples to/from the pages in BufferPool by calling the method from HeapFile to update the page, then BufferPool updates the records by re-inserting the pages into the BufferPool. When the BufferPool is full and a new page

need adding, `writePage` from `HeapFile` will be called to write the dirty page to disk and add the new page into the `BufferPool`. `BufferPool` is in charge of updating the pages because `Insert/Delete` operators directly call `insert/delete` of `BufferPool`. Besides, we also implemented `Filter`, `Join` operators and the aggregates. As an example, Figure 4 shows the workflow of using `Join` opeator and Figure 5 shows the workflow of inserting tuples.

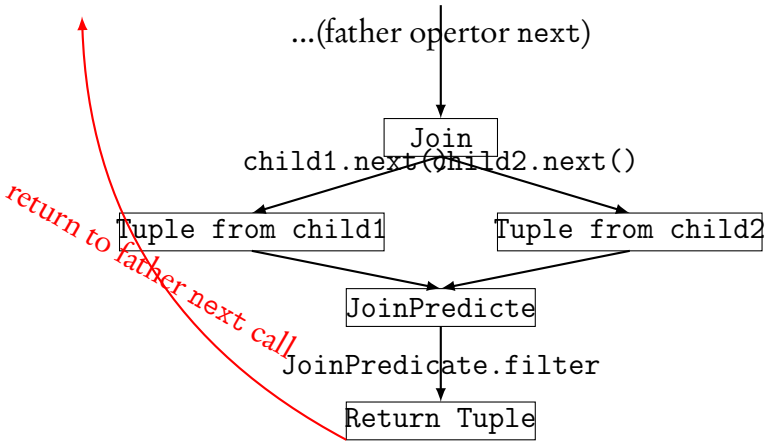

Figure 4: `SimpleDB join`

1.2 Lock Manager

Lab 3 focuses on adding Transactions functionality to simpleDB. In order to achieve this, we implemented our own `Lock` and `LockManager` which together handle acquiring/releasing both `SHARED` (read-only) and `EXCLUSIVE` (read-write) locks by different transactions on page granularity. `SimpleDB` uses time- out limits for `acquire` so that a certain period time of blocking on `acquire` will be considered as deadlock and thus abort the transaction.

- `Lock`: We need this class to represent the two types of locks simpleDB uses: `SHARED` and `EXCLUSIVE`. A shared lock is acquired by read-only transactions, and thus can be shared between many read-only transactions; an exclusive lock is acquired by read-write transactions, and thus can only be acquired by at most one read-write transaction at a time.
- `LockManager`: We need this class as the manager for all locking-related actions in simpleDB. It is created within `BufferPool` and will be called to acquire locks when `getPage` is called and release locks when `transactionComplete` is called. Note that because of the design of simpleDB, `acquire` in `LockManager` should only need calling in `getPage` when simpleDB wants to interact with a page. There are a few conditions when `acquire` will not be a blocking call. They are:
  - When the lock on that page is not locked.
  - When the lock is locked by this transaction itself (will upgrade a share lock to exclusive if this transaction is the only one which holds the lock)
  - When the lock is locked, but it is a shared lock, and the transaction wants a shared lock.
- `BufferPool.transactionComplete`: We add this method to release all locks acquired by a transaction when it commits.

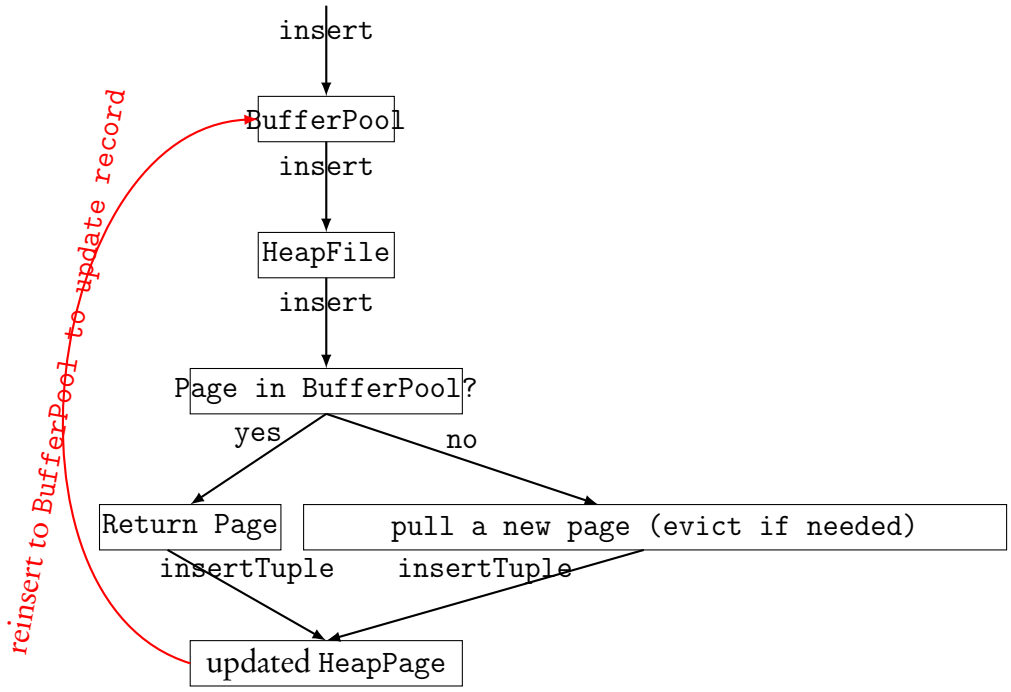

Figure 5: `SimpleDB insertTuple`

1.3 Log Manager

In Lab 4, we focus on adding rollback and recovery functionality to simpleDB upon abort and system crash. Specifically, we implemented STEAL (dirty pages may be evicted from the buffer pool even though the transaction hasn't commited yet), and NO FORCE (on transaction commit, no need to force write dirty pages to disk) for buffer pool management. In order to achieve this, we implemented log-based rollback and recovery which performs a redo-phase and an undo-phase.

`SimpleDB` supports six kinds of logs: `BEGIN`, `COMMIT`, `ABORT`, `UPDATE`, `CHECKPOINT`, and `CLR`. CLR is part of my design in order to make undo-phase easier to implement.

- `LogFile.rollback`: We need this method to roll back changes made by an aborted transaction. This methdo will read from the end of the log file and undo changes made by this transaction until its first active log record. It is also implemented that undo changes will append new CLR logs. More of this in the design part of this writeup.
- `LogFile.recover`: We need this method to recover a simpleDB system upon unexpected crashes. It will start redoing from the beginning of the log file or the last checkpoint log if any, during which a map of active transactions to their first active log line is built. Then, undo-phase will use the active transaction map and undo any changes made by these transactions bottom up.

2 Parallel Data Processing

In Lab 6, we added the ability of parallel data processing to `SimpleDB`. The basic structure of parallel `SimpleDB` contains `Worker` and `Server`. The workflow goes as follows: for every query users enter, it is first sent to `Server` for some optimization (we did not implement this part this quarter). Next step is to prepare this query to be run in parallel by inserting new operators (more on this later). The this query will be sent to all available workers. After each worker receives the query, it will localize it (more on this later) and run the query. Finally, each worker will send the results back to `Server`, then server will aggregate all the results and send the final result back to users.

Our implementation of parallel `SimpleDB` focuses on the following subparts of implementing a functional model:

- How to localize a query to run it on a local machine? (`Worker.java`)
- How to transfer data in between workers? (`ShuffleProducer.java` & `ShuffleConsumer.java`)
- How to optimize aggregate performance in a parallel setting? (`AggreagateOptimizer.java`)

The following sections explain the detailed implementations on these questions.

2.1 `Worker.java`

In `Worker.java`, we localize the query for it to run on the local machine. It does three jobs: (1) For `SeqScan` operators, we reset its `tid` and `alias`. This is because the `tid` of the *local* table may not be the same as the one of global table, and the `Catalog` is a local verlsion. We need to update the `tid` so that `SeqScan` reads the correct subtable; (2) For `Producers`, we set its worker to `this`. This is because `Worker` handles the data buffer of `Consumers` (more on this later) and the `Producer` needs to know which bufffer to send data to for the `Consumer` to get data; (3) For `Consumers`, we set its data buffer in `Worker` through its `inBuffer` map.

2.2 `ShuffleProducer.java` & `ShuffleConsumer.java`

We implement `ShuffleProducer.java` & `ShuffleConsumer.java` to enable data transfer between workers. In `ShuffleProducer.java`, in order to handle multiple connections with multiple `ShuffleConsumer` (whose addresses are stored in a `SocketInfo` wrapper), we keep three lists of `IoSession` as connections, `List` as data buffers, and `Long` as timestamps, one combination for one `Consumer`.

3 Discussion