CSE 444: `SimpleDB` Final Report

Linxing Preston Jiang

March 14, 2018

1 Overall System Architecture

In a typical database architecture, there are four main components: Process Manager, Query Executor, Share Utilities, and Storage Manger [1]. For our implementation of `SimpleDB` in the labs, the focus is on the Storage Manager and Query Executor: adding access methods for data stored on disk in lab1, adding both file mutability (insert/delete tuples to/from file system, eviction from a full BufferPool) and query operators (SeqScan, Join, etc.) & aggregates (min, max, etc.) in lab2, adding the lock manager in lab3, adding the log manager in lab4, and finally, adding parallel data processing ability in lab6 [3].

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

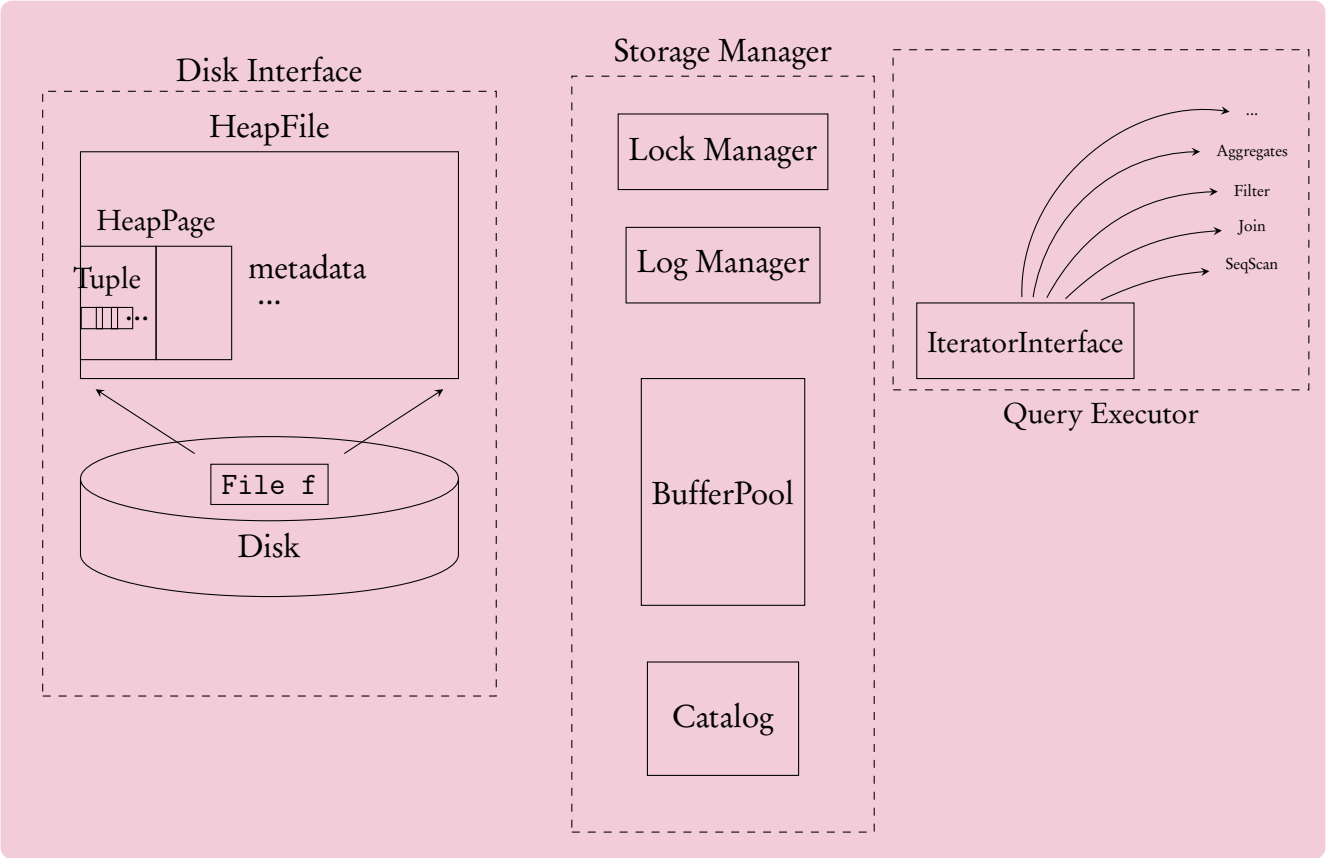

Figure 1: `SimpleDB` Architecture

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 [3]. Programs call `next` on the root operator and then fetch tuples

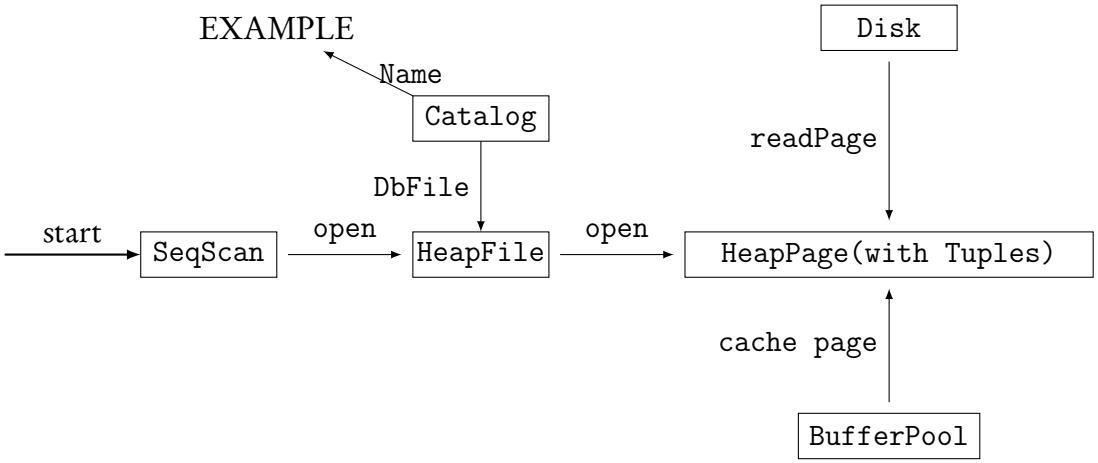

Figure 2: SimpleDB open

recursively through the plan tree in one pass top-down and another pass bottom-up.

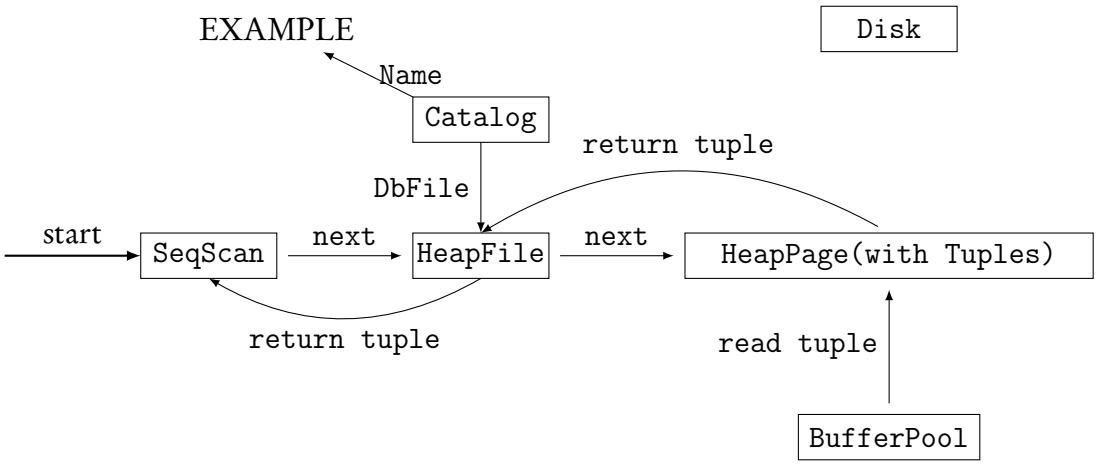

Figure 3: SimpleDB next

In Lab 1, we implemented getPage which is needed for reading pages into memory and getting tuples, together with SeqScan operator to scan files 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 pages, then BufferPool updates the records by re-inserting the pages into the BufferPool. When BufferPool is full and a new page need adding, writePage from HeapFile will be called to write a dirty page to disk and add the new page into the BufferPool. BufferPool is in charge of updating 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.

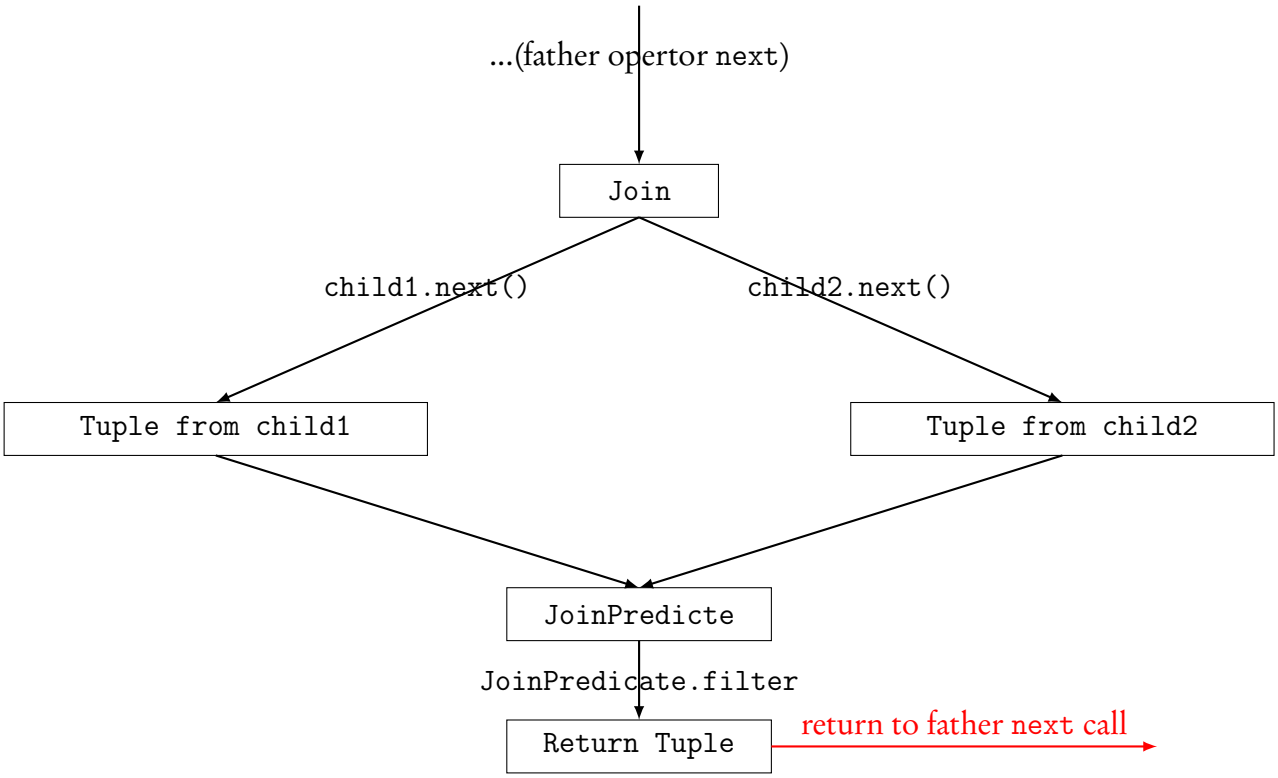

Figure 4: SimpleDB join

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 cause the system to abort the transaction.

- Lock: I 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: I 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.

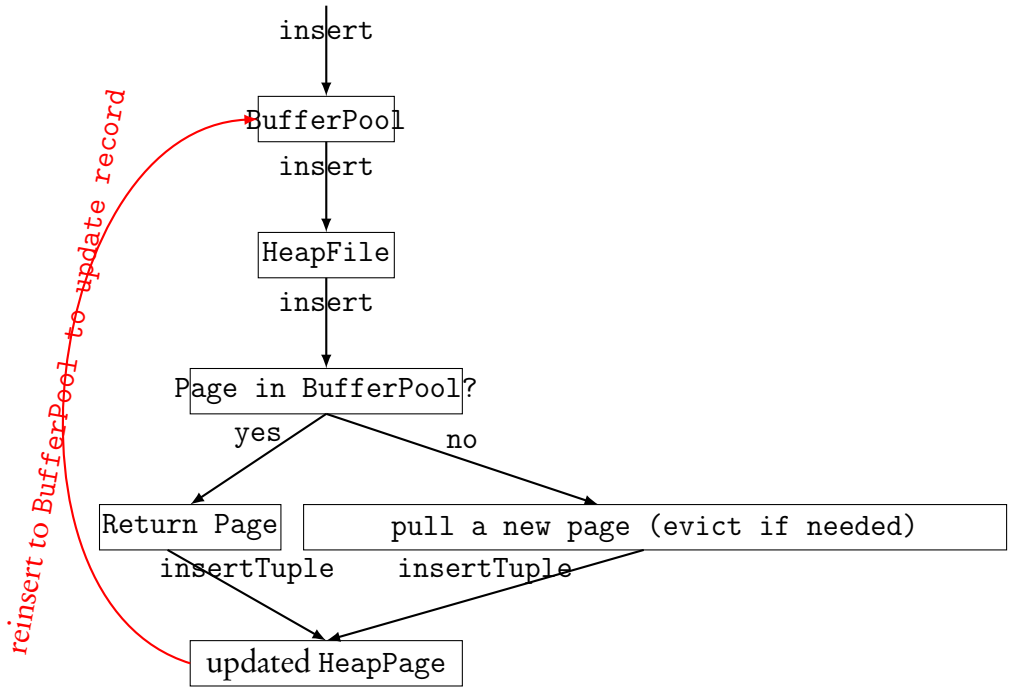

Figure 5: `SimpleDB insertTuple`

- `BufferPool.transactionComplete`: I add this method to release all locks acquired by a transaction when it commits.

1.3 Log Manager

In Lab 4, we focus on adding rollback and recovery functionality to `SimpleDB` upon abort and system crash. Specifically, Iweimplemented 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 to make undo-phase easier to implement.

- `LogFile.rollback`: we need this method to roll back changes made by an aborted transaction. This method 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.
- `LogFile.recover`: I 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 multiple Worker and one single Server. The workflow goes as follows: for every query the user enters, 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). Then 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, 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, 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 to which bufffer to send data for the Consumer; (3) For Consumers, 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, we use the same protocol in CollectProducer to send tuples to Consumer. The difference is that ShuffleProducer handles multiple connections with multiple ShuffleConsumer (whose addresses are stored in a SocketInfo wrapper), so I keep three lists of IoSession as connections, List as data buffers, and Long as timestamps, one combination for one Consumer. The working thread of ShuffleProducer takes tuples from its child operator, and uses a PartitionFunction to decide to which consumer to send this tuple.

In ShuffleConsumer, I followed the design of CollectConsumer to constantly take tuples out of the buffer (populated by ShuffleProducer) and returns an iterator of tuples in fetchNext. Together, ShuffleProducer and ShuffleConsumer enable data transfer between Workers

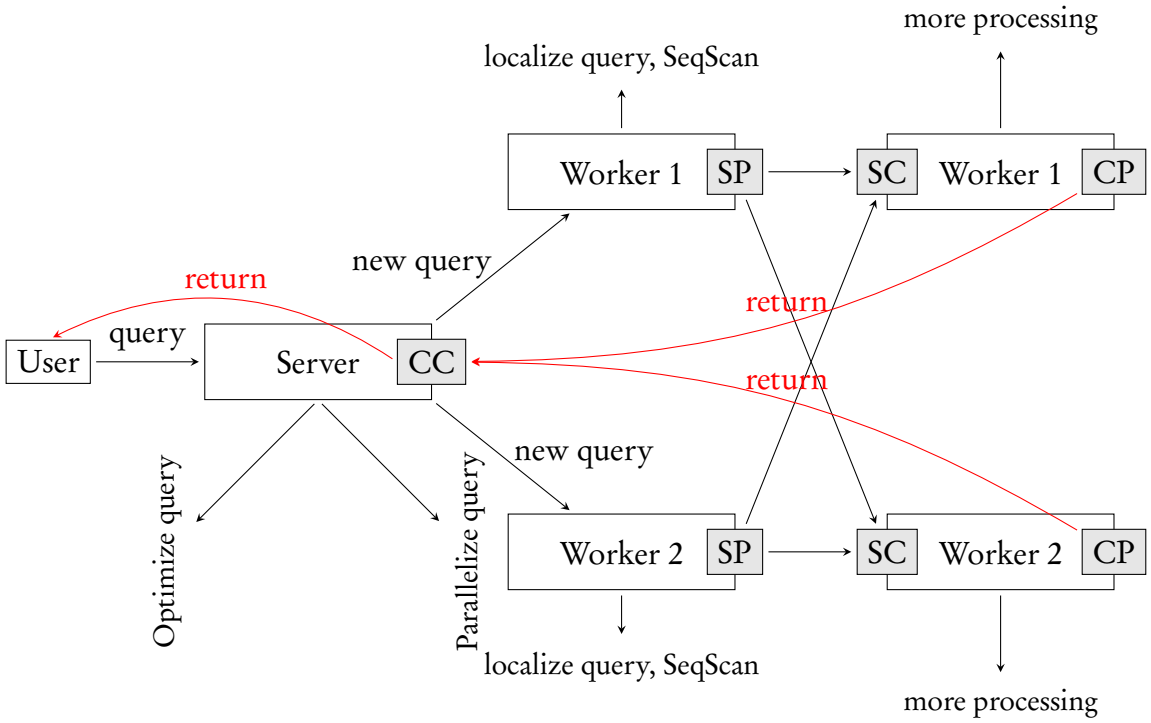

Figure 6: Parallel `SimpleDB`. SC - Shuffle Consumer, CC - Collect Consumer, SP - Shuffle Producer, SC - Shuffle Consumer

2.3 `AggregateOptimizer`

In `AggregateOptimizer.java`, we added optimization for aggregates running in parallel. The idea is that each local worker could work out an local aggregate result, then sent it to a `CollectConsumer` to return a final aggregate result. For each of the following aggregate, we optimize the performance by:

- `MIN`: calculate local `MIN` (downAgg), then take global `MIN` (upAgg).
- `MAX`: calculate local `MAX` (downAgg), then take global `MAX` (upAgg).
- `COUNT`: calculate local `COUNT` (downAgg), then take global `SUM` (upAgg).
- `SUM`: calculate local `SUM` (downAgg), then take global `SUM` (upAgg).
- `AVG`: locally calculate a `SUM` and a `COUNT` (downAgg), then take global $\frac{\sum \texttt{SUM}}{\sum \texttt{COUNT}}$ (upAgg).

As an example, Figure 6 is a diagram showing the workflow of Parallel `SimpleDB` with two workers.

2.4 Performance

In this section, I list the performance of parallel `SimpleDB` running four different queries, with 1, 2, and 4 workers available. Table 1 and table 2 shows the execution time of the following four queries on

1, 2, and 4 workers respectively, one for the cold cache (first time running the query since SimpleDB starts), the other one for already loaded cache (second time running the query since SimpleDB starts). All runtime is the average of six trials.

Query A:

```
SELECT *
    FROM Actor WHERE id < 1000;
```

Query B:

```
SELECT m.name,m.year,g.genre
    FROM Movie m, Director d, Genre g, Movie_Director md
    WHERE d.fname='Steven' AND d.lname='Spielberg'
        AND d.id=md.did AND md.mid=m.id
        AND g.mid=m.id;
```

Query C:

```
SELECT m.name, COUNT(a.id)
    FROM Movie m, Director d, Movie_Director md, Actor a, Casts c
    WHERE d.fname='Steven' AND d.lname='Spielberg'
        AND d.id=md.did AND md.mid=m.id
        AND c.mid=m.id
        AND c.pid=a.id
    GROUP BY m.name;
```

Query D:

```
SELECT m.name, AVG(a.id)
    FROM Movie m, Director d, Movie_Director md, Actor a, Casts c
    WHERE d.fname='Steven' AND d.lname='Spielberg'
        AND d.id=md.did AND md.mid=m.id
        AND c.mid=m.id
        AND c.pid=a.id
    GROUP BY m.name;
```

	Query			
# workers	A	B	C	D
1	1.44	1.83	5.65	5.49
2	1.89	1.86	5.11	6.18
4	1.48	3.11	7.49	7.22

Table 1: Performance with cold cache

	Query			
# workers	A	B	C	D
1	1.18	0.97	4.09	4.10
2	1.00	1.19	3.50	3.49
4	1.15	1.31	4.92	4.48

Table 2: Performance with loaded cache

As shown in the following bar charts (Figure 7 and Figure 8), increasing the number of workers seems to *decrease* the performance of `SimpleDB`. I suspect the reason to be that the size of the database we run is not large enough to show the advantage of data processing, i.e. the inner process communication time beats the efficiency of parallel execution.

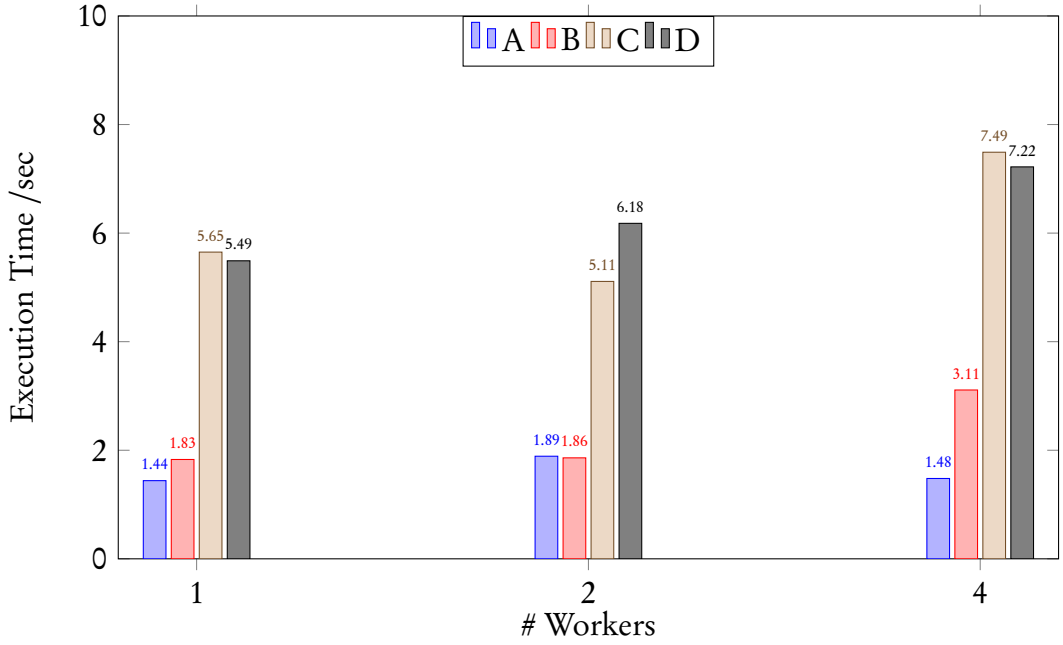

Figure 7: Performance with cold cache

3 Discussion

`SimpleDB` consists most of the modern database architecture [1]. However, there are a few aspects in my `SimpleDB` implementation (for simplicity) which affects the performance. Here are a few:

- Naive `BufferPool` eviction policy: in the current design, we choose the first page in `BufferPool` when its full. This is a naive approach because the page might be a frequently accessed page. A better approach is to use an approximation of Least-Recently-Used policy to evict dirty pages.
- Lack of indices: There are no index files in `SimpleDB` architecture, which is a drastic decrease performance wise, especially for databases which receive more look-up queries.
- Slow join: The current implementation of join is nested-loop-join, which is the least efficient. Ideally, to achieve the fastest join performance, hash-join should be used on equally join and merge-sort-join should be used on inequality join.
- Slow(and potentially inaccurate) deadlock detection: my current design for deadlock detection is using time-out, which bascially terminates a transacation if it has not been successful on grabbing a lock for a certain period of time. First, time-out is slow: if a deadlock occurs, the system will have to wait until the time-out limit is reached. Second, time-out may be inaccurate: imagine

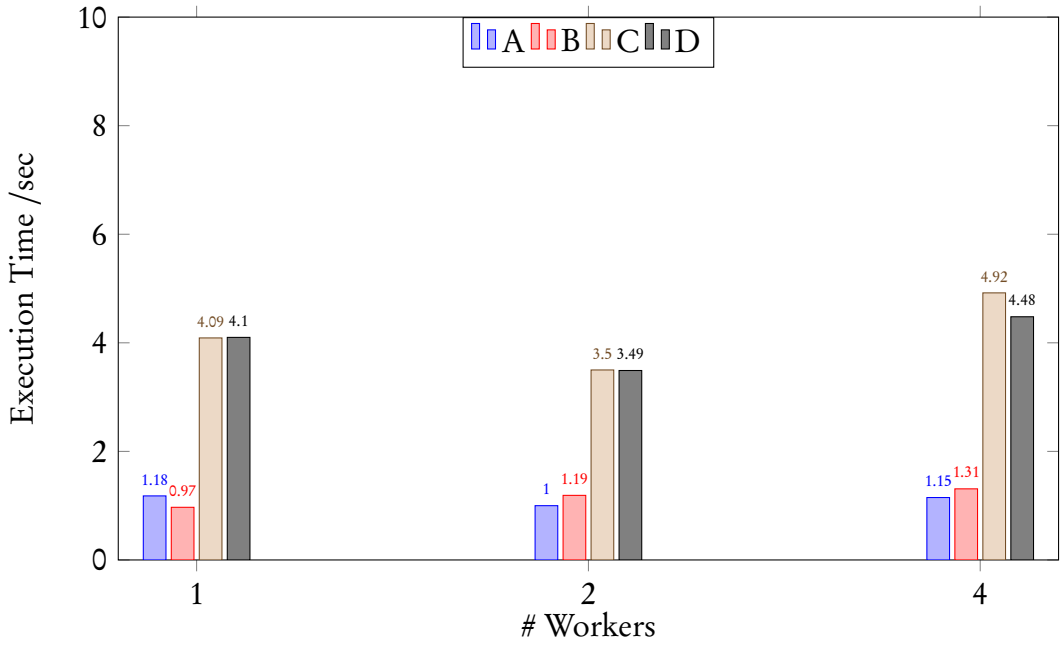

Figure 8: Performance with loaded cache

a course registration database – there are centainly times when a great number of transactions want to access the same resource, so natually the wait time for a single transaction would be longer. Time-out might terminate a transaction which is not deadlocked – it might just be slow. This is worse when the transaction has done a lot of work before this lock. Then whole work will be need to be redone. Therefore, a dependency graph would be a better choice, for many transactions

- Page-granularity lock: The lock is per-`HeapPage` for the current design. A tuple-granularity lock would bring faster performance, but more complex synchronization issue and memory need.
- Recovery Protocal: we did not implement ARIES [2], as discussed in class. Instead, current design of redoing every log is wasteful.

References

[1] Balazinska, Mass, Lecture 3
https://courses.cs.washington.edu/courses/cse444/18wi/lectures/lecture03-architecture-lar

[2] Balazinska, Mass, Lecture 18
https://courses.cs.washington.edu/courses/cse444/18wi/lectures/lecture17-19-transactions-

[3] SimpleDB Lab ReadME
https://gitlab.cs.washington.edu/cse444-18wi/simple-db