CSE 444 Final Report - Lab 6
- Anupam Gupta

Overall System Architecture :

This quarter we built a simple database system - SimpleDB that has the capability of executing SQL queries successfully using several different operators that any other Database system can support. SimpleDB is a relational data base system that can run on any schema we provide it. Essentially, it consists of different components that work together to ensure query results are computed in an efficient manner. The system essentially consists of (from bottom up) Heap Pages which are basically pages that relate to physical pages in disks that store the data that's stored on the files for the table. In Simple DB, each relation is stored in a separate file called a Heap File which consists of several Heap Pages. The system keeps track of the different tables, their schemas, their primary keys as well as their underlying files using a central Catalog. There is a central cache structure named BufferPool that keeps track of dirty pages loaded into memory. The system uses a lock manager to ensure that when multiple transactions are executing simultaneously, the system doesn't go into a deadlock situation and Strict 2PL is maintained consistently through the execution of every query. In addition, there is a separate log file that keeps track of the updates/ changes made by each of the transaction which can be used in the case of a system crash to recover the system to it's state before the crash. For maximum efficiency, this system uses a Steal/No-Force policy for the database system. In addition to these features, the system is also fully parallelized and can run on different threads (depending on the number of workers) and combine results to provide the results of the query. To support the execution of these queries, multiple different operators are implemented as well including Aggregators, Joins, Filters and Sequential Scans that allow the user to execute the queries they want.

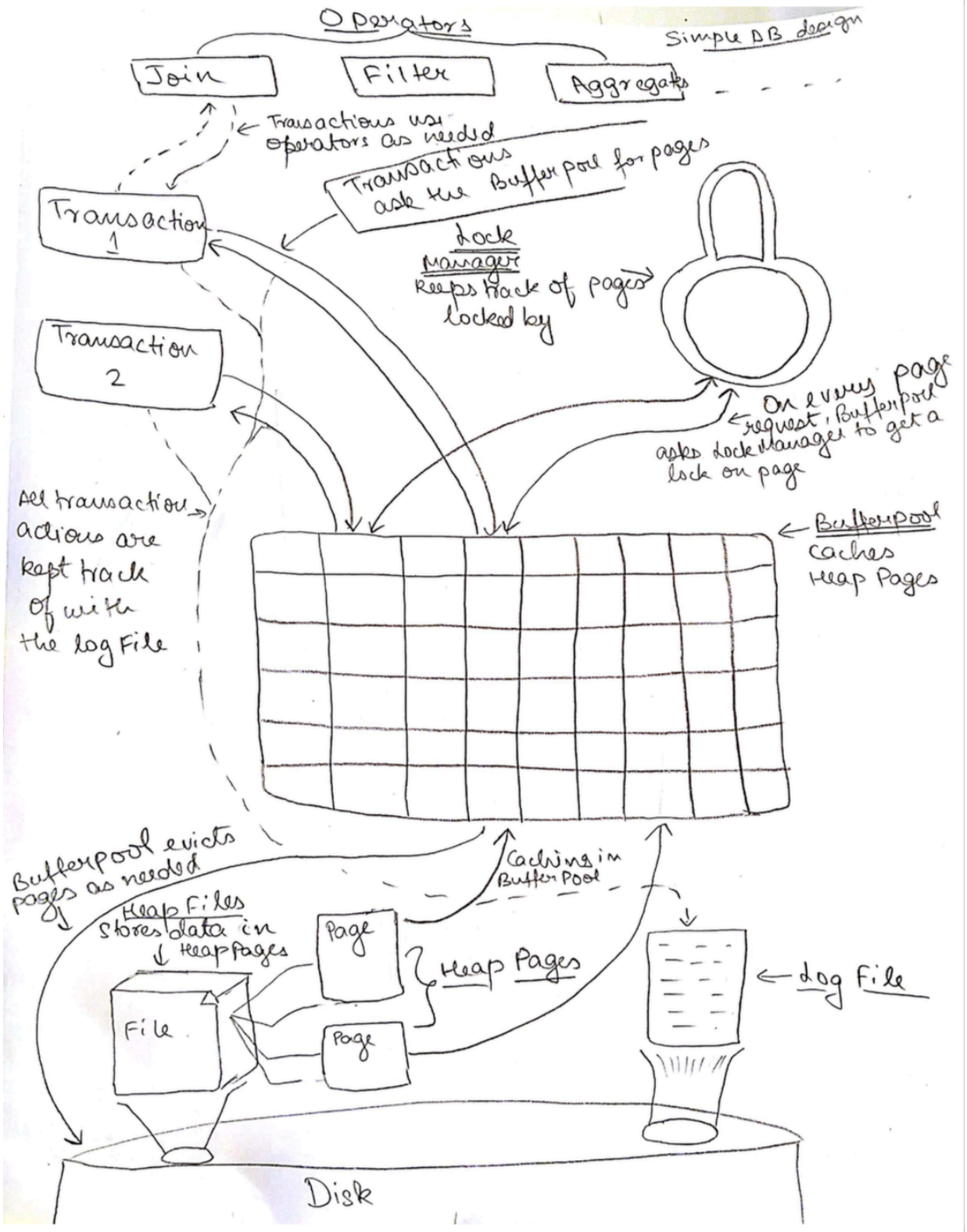


Diagram showing Simple DB's Overall System Architecture

SimpleDB is supported by various different components.Some of the integral ones are :

1. **Buffer Manager** : The Buffer Manager is an internal sort of cache for the database system that helps store pages that are currently being used by transactions. This provides fast access to any of the pages we need. The Buffer Manager has a specific limit of pages it can contain in it's cache after which to accommodate more pages, it decides to evict any of it's pages randomly back to disk. All requests for a page of data go through the Buffer Manager. The Buffer Manager can either provide the page if it has it in memory or it can read the page from memory, cache it and provide it as needed. Since the Buffer Manager is mostly responsible for providing pages to different transactions, inserting and deleting tuples, the lock Manager works very closely with the Buffer Manager to ensure locks on pages are provided at the right time and with the proper permissions.

2. **Operators :** SimpleDB supports several multiple different operators like Joins, Filter, Aggregates like MAX, MIN, AVG, SUM and COUNT, Group By, Order By, Rename and Projection operator. Each of the operators have several different options that the user can specify when they're computing the result from the operators. For example, Simple DB provides support for multiple different kinds of join operations using mathematical operators like (<, = , > , etc.). Similarly, the groupBy and aggregate operators help the user perform several different complex queries using this system as well. Each of the operators are structured in the form of Iterators that can be opened and closed and can be used to get the next tuple that the operators compute using the next() (in this case fetchNext()) method of the iterator. The operators are capable of taking the children that they will pull tuples from and compute the operations they should be computing and returning those tuples one by one.

3. **Lock Manager :** The Lock Manager is an integral component of SimpleDB that helps SimpleDB support multiple transactions executing at the same time. The Lock Manager implements locking on the granularity of Pages. As stated above, the lock Manager works in close proximity with the Buffer Pool to keep track of which transactions have locks on pages. The Lock Manager is also responsible for detecting if providing locks to pages will lead to a deadlock situation in the system. In these situations, the system decides to abort transactions as needed. The Lock Manager internally uses a dependency mapping of transactions to keep track of cyclic dependencies as well as to keep track of the kind of lock that each transaction has on a page. In essence, the lock Manager is the most important component in ensuring all the transactions obey the rules of Strict 2PL while executing queries.

4. **Log Manager :** The Log Manager is an integral component of SimpleDB that ensures that the system is recoverable in the case of a crash. The Log Manager works closes with a log file to ensure that a Steal/No-Force policy can be implemented. The Log Manager works closely with the Buffer Pool as well as it keeps track of every update by every transaction. It also keeps track of when a

transaction commits and aborts. Logs are frequently flushed to disk (after every transaction commits/aborts) or when pages are flushed from the Buffer Pool. The Log Manager also helps with rolling back updates of transactions when a transaction aborts such that the system appears as if the transaction had never executed in the first place. These redo updates are also kept track of by the Log Manager using CLR log records. In the case of a system crash, the system is capable of coming back to it's state before the system crashed by re applying all the changes stated in the log file. This helps the system maintain a comprehensive list of the pages that were changed and transactions that were in progress when the system crashed. The Log Manager hence, is vital for recovery of the system in the case of a system crash to ensure transactions/committed data isn't lost due to the crash.

Parallel Data Processing Capabilities

SimpleDB is effectively capable of completing queries in a parallel fashion. SimpleDB is capable of completing queries by parallelizing the query plan across different machines that perform the computation individually and bringing all the computed tuples together to provide the final result of the query. SimpleDB essentially spins up different threads for small computations/ different parts of the query plan that internally spin up more threads as needed to finally come together and provide all the resultant tuples. Since SimpleDB can parallelize well, the query execution times are much better in comparison to a non parallelized database system.

In SimpleDB, all data tables are partitioned horizontally into the number of machines that are working. Hence, each machine has a separate copy of all the tables stored in the catalog. SimpleDB works with the concept of workers which are essentially machines that are computing query results individually using their own set of database tables. The first step of executing the query is to come up with an efficient query plan of execution. Following this, all the workers receive the query plan and must process the query using data from the tuples that are provided to that worker. The first step to this, is to localize the query plan to make sure the query plans meta data such as operators, sequential scans, etc. are only applied using the pages/tuples that the current worker is given. After this step, the workers individually compute some parts of the query and send tuples across different workers and finally to the co-ordinating worker to return the final result of the query.

There are several different components that go into making SimpleDB working successfully in a parallel fashion. Some of the integral ones are :

1. **Worker** -> These are individual machines which hold certain portions of the database and which run different parts of the query plan. These are basically different parallel processes that are working in collaboration with other processes to compute the final results of a query. Currently, SimpleDB runs these processes

on a single machine but these could also be run on different machines such that they have access to more cores and can process faster.

2. **Shuffle Producer** -> The Shuffle Producer is an integral component behind SimpleDB being able to execute queries in parallel. The Shuffle Producer is basically an operator that is responsible for sending tuples from one worker to the next. The Shuffle Producer is responsible for distributing tuples to different workers according to a partition function that's specified. The Shuffle Producer coupled with the Shuffle Consumer is responsible for shuffling tuples between workers as needed. Shuffle Producer does this by maintaining a buffer for each of the shuffle consumer operators it needs to communicate with and sends tuple bags to these consumers when it has accumulated enough tuples. For example, a query plan might require that all tuples that match a specific characteristic reach a specific worker for the next parts of the query. This operation will be completed using two components - one of which is the Shuffle Producer which is responsible for sending tuples across workers.

3. **Shuffle Consumer** -> The Shuffle Consumer is the other component that works very closely with the Shuffle Producer to allow tuples to freely transfer from one worker to another. The Shuffle Consumer is responsible in each worker to "catch" the tuples that the Shuffle Producer sends through. Each Shuffle Consumer is responsible in a worker to work closely with other Shuffle Producers in other workers to ensure it receives the data that Shuffle Producers from different workers send across the network. Hence, a Shuffle Consumer and Shuffle Producer pair are essentially inserted between operations that require tuples to be shuffled across different workers.

4. **Collect Producer** -> The Collect Producer is a different variant of Producer. The Collect Producer is mostly used in those instances where we want to combine the tuples from different workers into a single worker. For example, while finishing up a query with aggregates, we get the tuples from different worker into a single worker that finally computes the final aggregate and outputs the result. The Collect Producer here is the sending counterpart of the sending/receiving couple that just sends the tuples to a specific Collect Consumer.

5. **Collect Consumer** -> The Collect Consumer is a different variant of the Consumer that works closely with the Collect Producer. The Collect Consumer is responsible for receiving the tuples it gets from the different Collect Producers working on different processes(workers). It forms the receiving component of the sending/ receiving couple that receives the tuples from several different Collect Producers into a single worker.

6. **Aggregates** -> In order to successfully compute queries in parallel, Aggregates also have to be changed so although different workers compute different parts of an aggregate, the query plan is shaped such that at the end, the result of the query

is accurate. To do this, aggregates in the query plan are optimized to run in parallel. For example, to compute a Count, each worker internally computes a Count while the worker at the end of the query must get all these individual counts from the workers and perform a SUM aggregation at the end to display the final result. In a similar fashion, MAX, MIN, AVG and SUM need to be changed accordingly by thinking about what operator we want the worker to compute and then what operator the final worker must compute using the tuples returned from the other workers in order to get the correct value from an aggregate.

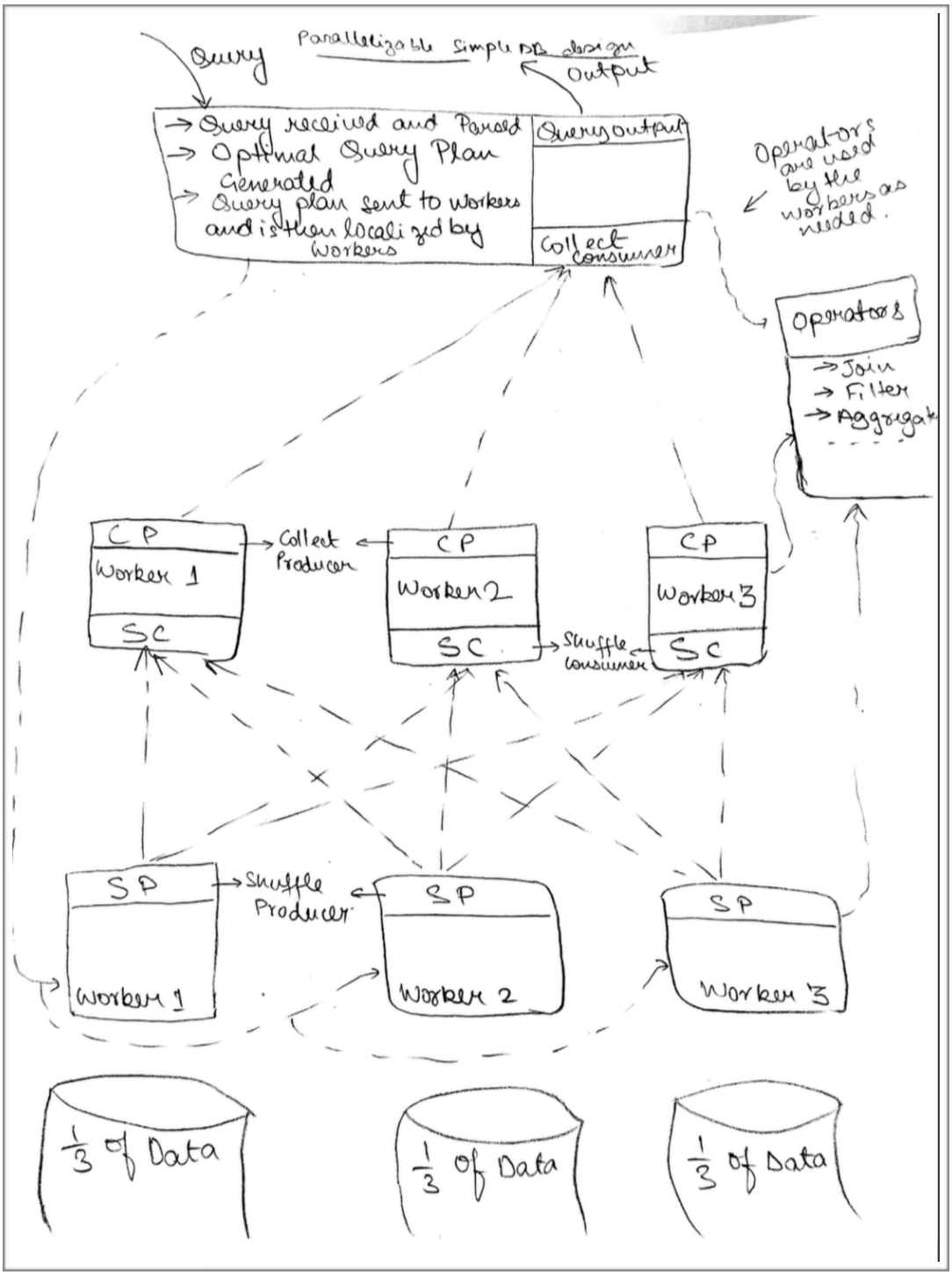


Diagram showing how SimpleDB processes Queries in a parallel fashion

Performance Analysis:

To Test the Performance of the Database, I ran a collection of 7 different queries for 1, 2 and 4 workers on both the 1 percent as well as the 10 percent database. The queries that I ran were:

Query #	Purpose of Query	Query	Expected Number of Results in 1% Dataset	Expected Number of Results in the 10% Dataset
1	Selection Test for All Elements in Table	1. select * from Actor;	26026	199195
2	Test Join with 2 Tables	1. select * from Movie_Director md, Movie m where m.id = md.mid;	2791	29762
3	Test Join with 3 Tables	1. select * from Genre g, Movie m, Movie_Director md where m.id= md.mid AND m.id=g.mid;	5087	53645
4	Test Join with 4 Tables	1. select * from Genre g, Movie m, Movie_Director md, Casts c where m.id= md.mid AND m.id=g.mid AND c.mid=m.id;	66598	725156
5	Test Aggregate with 2 Tables Joined	1. select count(id) from Actor;	1	1
6	Test Aggregate with 3 Tables Joined	1. select MAX(g.mid) from Genre g, Movie m, Movie_Director md where m.id = md.mid AND m.id=g.mid;	1	1
7	Testing Query 2 From Lab Specification	1. 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;	0	14

Query #	Purpose of Query	Query	Expected Number of Results in 1% Dataset	Expected Number of Results in the 10% Dataset
8	Testing Query 1 From Lab Specification	1. select * from Actor where id < 1000;	6	67
9	Testing Query 3 From Lab specification	1. 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;	0	5

Query Execution Results :

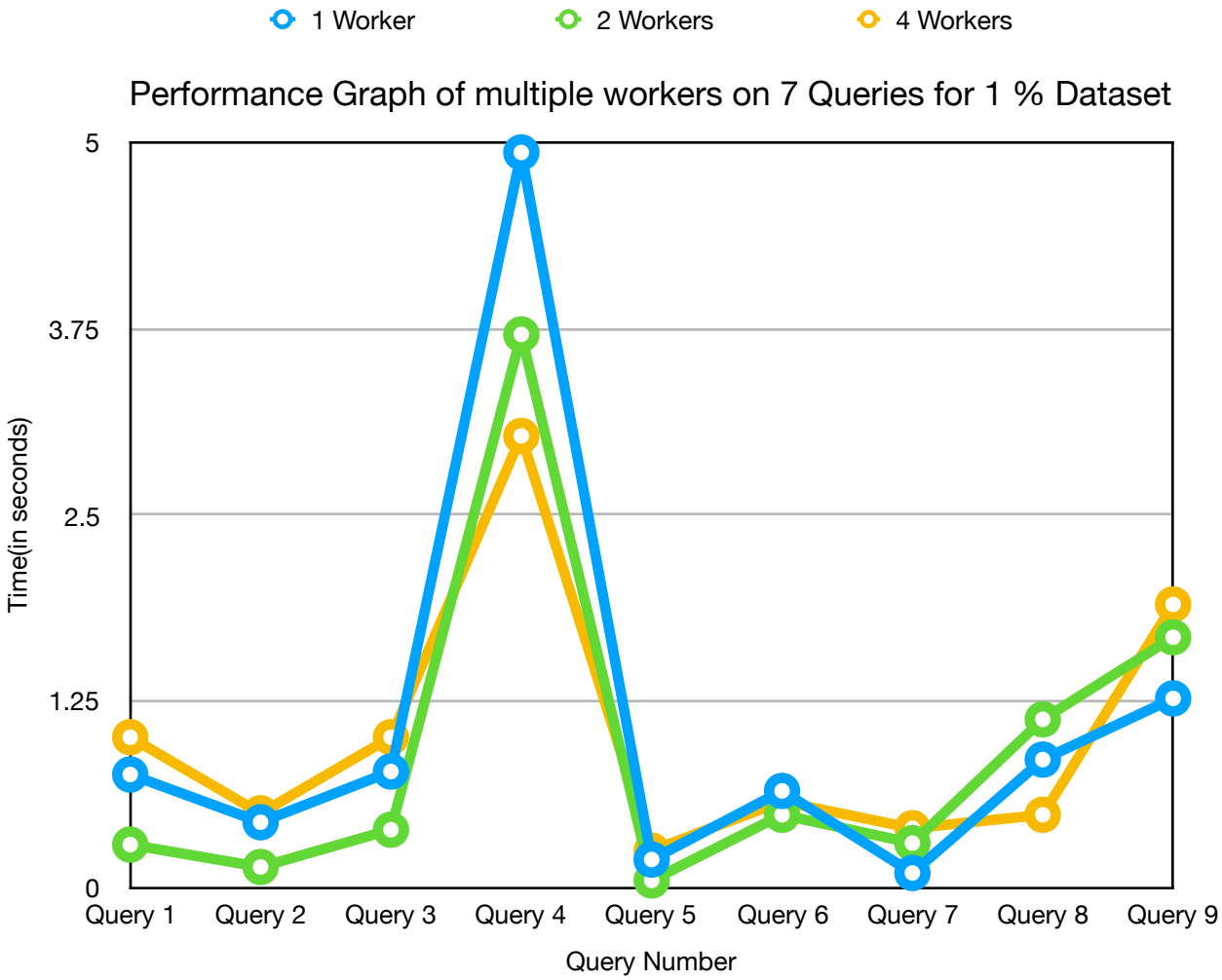
1 Percent Database(Time (in seconds) vs Number of Workers)

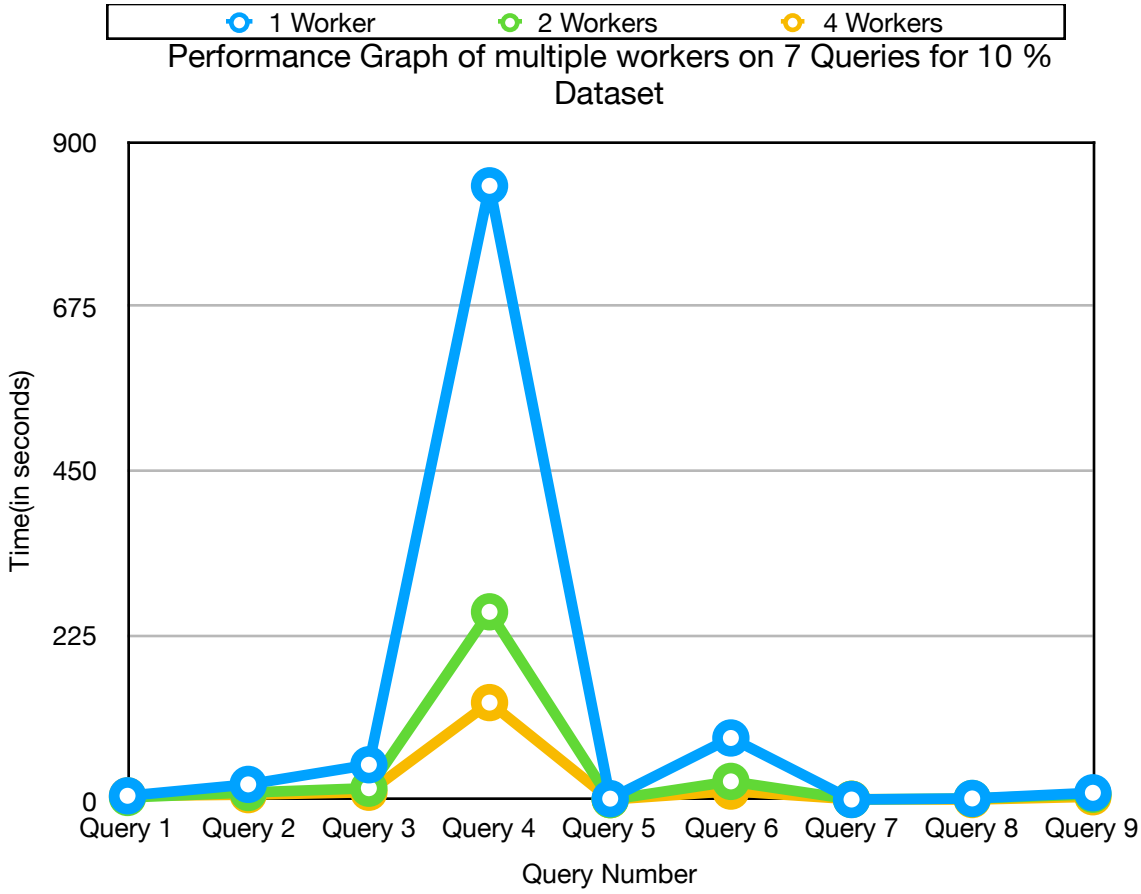
	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
1 Worker	0.76	0.44	0.78	4.93	0.19	0.65	0.10	0.86	1.27
2 Workers	0.29	0.14	0.39	3.71	0.05	0.49	0.30	1.13	1.68
4 Workers	1.01	0.5	1.01	3.03	0.25	0.58	0.40	0.49	1.90

10 Percent Database(Time (in seconds) vs Number of Workers)

	Query 1	Query 2	Query 3	Query 4	Query 5	Query 6	Query 7	Query 8	Query 9
1 Worker	5.60	21.17	48.01	839.57	1.68	84.64	0.47	1.95	9.68
2 Workers	3.28	10.09	15.95	257.12	0.23	25.27	0.66	1.81	7.65
4 Workers	5.48	7.16	10.47	133.00	0.45	12.00	0.90	0.45	4.26

We can graph these performances





As we can see from the graphs, the higher the number of workers, the lesser is the time taken for the query to complete. However, this is not always the result. In some cases, the query is completed faster with 1 or 2 queries than with 4 queries. This is because there is a higher overhead involved with computing queries in parallel. Hence, in some cases, the queries are completed faster with 1 or 2 workers as 4 workers spend more time communicating and transferring tuples than actually computing the result of the query. Overall, though, as can be seen in the case of Query 4 (which involved a join of 4 tables), the system with 4 workers completed the query in a much smaller portion than the system with 1 or 2 workers leading to an overall better system performance. In SimpleDB overall, if the queries involve lesser computation, we can spin up just a small number of workers and we'll probably get the best result while we can use higher amount of workers to get a much better performance on queries that involve larger computation in general.

Final Analysis:

Simple DB as a database system performs very well while compared to present database systems. The query times for any of the smaller queries are comparable to that of professional database systems. SimpleDB's performance with multiple workers on larger queries is also comparable with other professional database systems. Spinning up multiple workers for all queries however, might lead to a larger output time since there's an additional overhead of communicating between the workers. In addition, with the use of SimpleDB's parallelized aggregate system, the results of the queries involving aggregates are also surprisingly fast. In addition, the consistency that comes with SimpleDB's Strict 2PL policy also pays off with each run of the query getting the exact same number of rows every time when run in isolation. Overall, the performance and parallelizable structure of SimpleDB help in ensuring SimpleDB functions as a consistent database system that is both quick in it's execution as well as dependable. In addition, the easily recoverable database strategy applied in SimpleDB helps ensure the Database can be recovered even after a crash which makes the database more dependable.

If there was more time, I would definitely like to work on the query optimization part of the lab that was provided to us. I believe working on the query optimization would help me understand the underlying structure of the query plan better and make me more confident in my ability to debug the code in case of any bugs. Query Optimization is an integral component of the lab that generates the best query plan for a query and learning more about it through implementation would help me better understand what we went over in lecture (concerning optimization algorithms).