Anton DeVore
CSE 444
Lab 6
Final Report

1. Overall System Architecture (1.5)

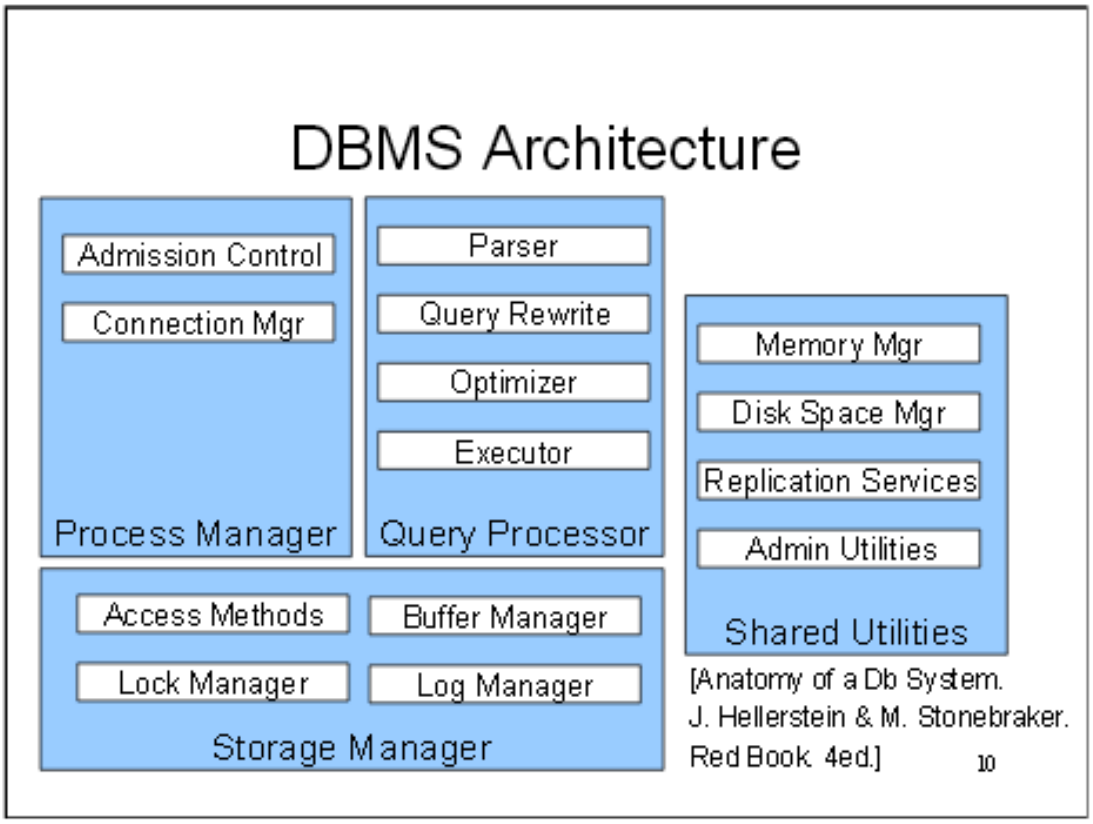
The database management system (DBMS) is designed to hold large amounts of data which can be queried over and updated by multiple users. It consists of 4 major components: process manager, query processor, storage manager and shared utilities.

The process manager handles connections to the DBMS which including multiple people accessing the DBMS concurrently and who has the proper permissions. Most of this was implemented for me. The process manager also handles multiple process connections when implemented in parallel. When executing queries in parallel, one machine acts as the coordinator which connects to a set number of workers to run in parallel to execute a query. I go into this in more detail in section 3.

The query processor handles all SQL requests made to the database system. These requests can include read and/or writing and can be made concurrently by multiple users. The DBMS starts by parsing the query. Once the DBMS verifies that the query is in proper format, the DBMS breaks the query into individual pieces or operators. The optimizer then begins forming a logical plan and physical plan which examines the operators and determines the most cost efficient way execute the query. The physical plan focuses on the cost of reading data from disks and managing that data while the logical plan focuses on the order of performing operations to reduce the size of the data needed and/or quicken the execution time of the query. The optimizer also takes into the consideration of parallel processes. The DBMS will begin executing the query once the plans are chosen. The parser was completely written for me including the optimizer. The executor was written by me, which performs filters on data, joining tables and aggregates.

The storage manager is responsible for managing the data and reducing the number of I/O requests. One way this is done is through the buffer manager also known as the buffer pool. The buffer pool keeps track of which heap pages are required for the DBMS to execute its queries and utilizes a least recently used (LRU) policy to efficiently evict and store pages. When dealing with concurrent transactions, the buffer pool manages locks on heap pages to efficiently share data between users while avoiding deadlocks. This is achieved through inclusive and exclusive locks and deadlocks are detected using a dependency graph. The DBMS achieves ACID properties by using Strict 2PL and Force/No Steal policies. Strict 2PL states that a transaction cannot release any locks until the transaction has committed. Force/No Steal allows my database to be consistent. Force meaning we write all immediate changes to the log while No Steal leaves any dirtied heap pages in the buffer pool until a transaction commits. The log manager handles writing to the log and performing undo/redo actions in the case of a system crashing or transaction aborting. I wrote the majority of the buffer manager, lock manager and log manager.

The shared utilities were written for me. This handles administrator utilities, replication services, disk space management and memory management.



Here is a simple overlook of how my DBMS's structure is laid out.

2. Detailed Design of Basic Components (2)

2.1 Buffer Manager

The buffer manager is responsible for managing heap pages. Records are represented as tuples which are stored in groups on a heap page. Together heap pages make a heap file which represents a table. There are many different eviction policies but my buffer manager utilizes a LRU policy to determine when to evict a page. The buffer manager implements a No Steal policy which means it will not evict any dirty pages until a transaction commits. This combined with Force policy allows for easy recovery but slower performance.

2.2 Lock Manager

I chose to design my lock manager to lock at the page level oppose to the tuple level. Locking at the page level was easier to implement due to the face that my buffer manager stores heap pages. Locking at a tuple level would require storing more meta-data to determine when a transaction can access a tuple. My lock manager detects dead locks by examining a dependency graph. This graph is a simple mapping of transaction waiting on other transaction. If a cycle existed in the graph then a deadlock had been created and the lock manager would choose to abort transactions until one completed and committed. The alternative to using a dependency graph is to use timeouts. Originally I chose to implement my lock manager this way but it was tough to determine an appropriate timeout limit.

2.3 Log Manager

The log manager implements Force policy which means it will write all updates to the commit log. When the system crashes the log manager reads through commit log redoing all transactions that have committed and undoing any transactions that have either aborted or haven't committed. This will leave only the updates from transactions that successfully committed before the crash occurred. No Steal along with Force policy again allows for easy recovery and slow performance.

2.4 Operators

Operators are the individual pieces of a query. Certain types of operators include filter, join, insert, delete and aggregate.

2.4.1 Filter

This operator uses a filter function to reduce the number of tuples. The filter operator applies a predicate such as <, >, =, etc… This is generally used during queries.

2.4.2 Join

This operator joins tables on a predicate. The predicates are similar to those used by the filter operators. There are three types of joins. The nested for loop join is the easiest to implement, the least memory intensive and the slowest performance. The nested for loop can be implemented in two ways: the first is to do the join tuple by tuple which is much slower in performance, the second is to do the join page by page which is better for performance. The hash-join is very fast but requires lots of memory. I implemented this version because of its quick performance. The third type of join is a merge sort join which is slower than hash-join but faster than nested for loop join. If implemented properly the merge sort join can be run with limited memory and is sometimes preferred because the tuples are sorted when it is finished.

2.4.3 Insert

This operator is used to insert a new tuples into a table. Given a list of tuples to insert it runs through necessary pages and inserts the tuple into the necessary page. The tuples are not kept in any order so the first available opening in the heap file is where the tuple is inserted. If all heap pages are full then the buffer manager puts the tuple on a new heap page which is then added to the end of the heap file. This operator will return the number of tuples properly inserted.

2.4.4 Delete

This operator is used to delete tuples from a given table. Given a list of tuples to delete it finds the tuples and deletes them from the page. If the tuple does not exist it does not do anything. This operator will return the number of tuples properly deleted.

2.4.5 Aggregate

This is operator is used to perform aggregates on tables. Aggregate types include min, max, average, sum, count and like. The 1st five can be used on integer type fields while the last two can be used for string type fields. When run in parallel, the DBMS can do local aggregates before sending those aggregates to a final worker to join the results. In the case of average, the workers do a local sum and count and send those results to a final worker to do the average mean.

3. Detailed Design of the Parallel Data Processing Capabilities (3)

3.1 There are three mandatory components: worker process, shuffle and aggregates.

3.1.1 **Worker Process**

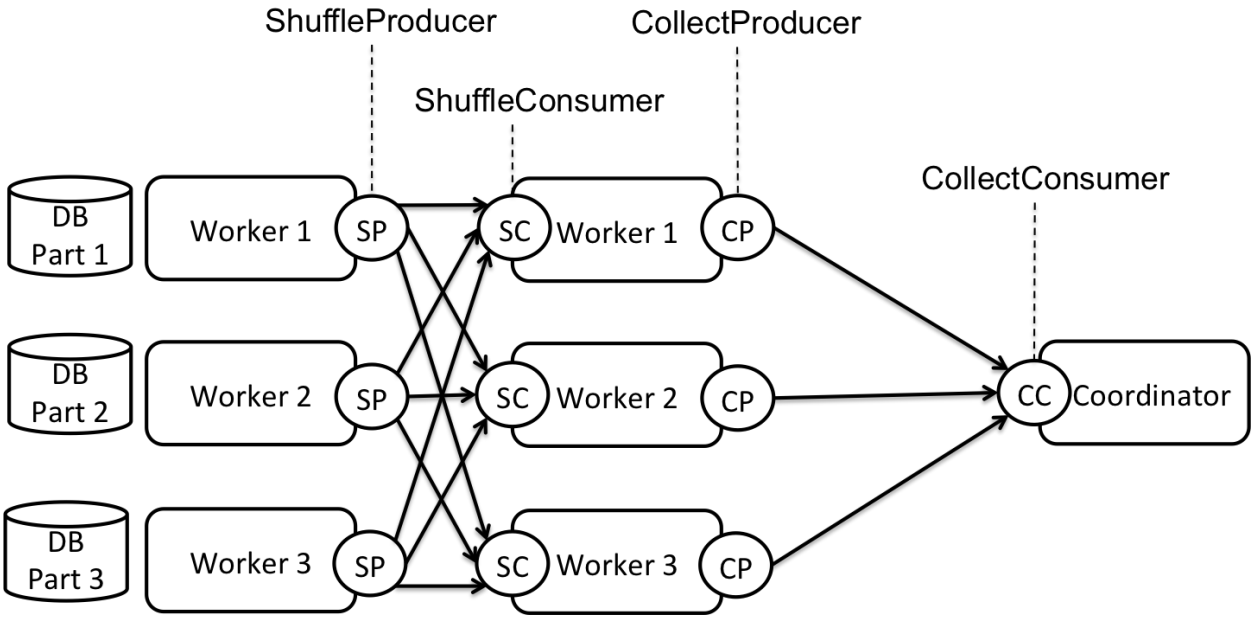
Workers are individual processes that execute part of a given query. The initial worker that receives the query is the coordinator which then spawns additional processes, either on the same machine or different machines, to do part of the query. Each worker plays two major roles:

3.1.1.1 Partition Data

Each worker gets a subset of the data. The worker applies a partition function to the data then each worker shuffles the data to all the workers for the second major function.

3.1.1.2 Localized Query

Once a worker receives the partitioned data from each worker it performs a localized query. The query is specified by the coordinator.



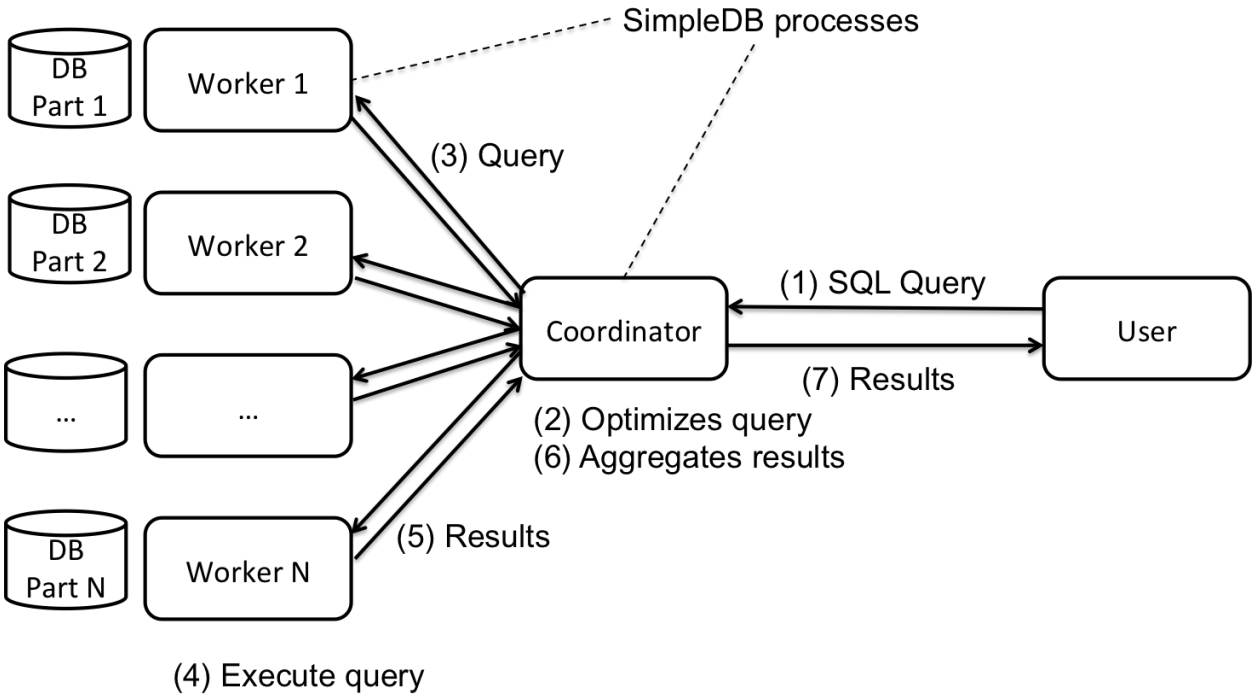
Above is an image of the process three workers would go through to complete a query. They first get a portion of the data from the database. The ShuffleProducer performs the partition function which sends the data to the appropriate workers. The ShuffleConsumer collects the partitioned data from the ShuffleProducer. The worker then performs its localized query. After that the CollectProducer sends the results of the localize query to the coordinator. The CollectConsumer organizes all the results to return to the user.

3.1.2 **Shuffle**

A worker performs the shuffle function after performing one of its major functions. After a worker partitions the data, it must send the data to the appropriate worker to apply the localized query. After a worker performs its localized query, it must send its results to the coordinate. The coordinator will organize the results and return them to the user.

3.1.3 **Aggregates**

Occasionally queries involve aggregates which can be done in parallel. The workers behave slightly different when performing aggregates. After the workers received their partitioned data, they perform a localized aggregate. They send their results to one worker. This one worker applies the aggregate again to the data received and returns the final result to the coordinator. Apply the aggregate twice is ok for min, max, sum and count but not average. Average is a special case for which each worker instead does a sum and count, then the one worker does another sum and count, calculates the average and returns the final result to the coordinator.



This diagram is an overall look at a query made by the user. The coordinator is responsible for parsing the query and optimizing the query plan. Then it spawns workers to execute localize query plans and aggregates their results to return to the user.

3.2 Extensions

I didn't have time to implement any extensions. If I had more time I would have experimented with different types of data partitioning schemes. I would expect a range partitioning would be better for range queries but might be slightly skewed. The amount of data received by each worker is dependent on the how the ranges are partitioned. A hash partition would be much quicker for point queries where there is/are selection predicates. Hash partitions are generally easier to partition equally so that each worker has an equal amount of work. I was also considering maybe implementing a composite partition which is a combination of range partitioning further subdivided by hash partitioning. If I had enough time I would have implemented all three partitions and examined the performance times on different queries.

3.3 Analysis

After implementing Worker.java I ran the first test query:

select * from actor where id < 1000;

Workers	Cold Cache (s)	Hot Cache (s)
1	4.05	3.10
2	3.06	2.26
4	2.60	1.75

The performance times for different number of workers are in the table above. It is clear that a hot cache is much quicker and as the number of workers increase the performance time improves. At some point if I continued increasing the number of workers I would expect a decrease in performance time do to the overhead of creating and communicating with many workers.

After implementing the rest of ShuffleProducer.java and ShuffleConsumer.java I was able pass the JUnit tests but wasn't able to get the query to run. I was able to run the query on my database but when I tried to run the query in parallel a NoSuchElementException() was thrown.

I also tried to implement AggregateOptimizer.java. Max was given to me but when I wrote a test for Max, the test failed. The assertion on line 84 continued to fail. This was after I changed it to be assertEquals() from assertTrue(). It originally stated assertTrue(string == string). This is an improper way to compare two strings so I changed it to assertEquals(string, string). This failed when I ran min, sum, average, count and max. Examining the line it seemed that it was comparing the group by field name to the first field. The group by field was always the second field so I don't see any way for this test to pass. I debugged it for a few hours and couldn't determine if I had an underlying bug somewhere that was causing my TupleDesc to be wrong. When removing this line min, max, sum and average passed which is odd. I would have expected min, max, count

and sum to work but not average because the parallel average is dependent on count and sum and needs to be implemented differently. I unfortunately ran out of time and could not pursue my bug.

4. Discussion

My DBMS performs queries quickly, can perform simple queries in parallel and can handle concurrent transactions. I would like to focus on four main areas that affect the performance of my DBMS: joins, synchronization, parallelism, and the optimizer.

4.1 Joins

I chose to implement a hash-join which improves the overall speed of my database queries but uses up more memory than necessary. The difference in performance between the hash-join and nested for loop join is drastic in large queries and/or large datasets. I do worry that on extremely large datasets that the memory limit might be reach which is why if I had a chance to re-implement my join, I would use a merge-sort join. While this type of join is not as quick as a hash-join, when implemented properly it can be limited to a set size of memory. This would be more reliable in a DBMS.

4.2 Synchronization

My DBMS handles multiple transactions by locking at the page level. This is much more efficient than locking at a table level and is quite fast already. I would have liked to try to implement locking at a tuple level to compare performance but overall this was great for performance.

4.3 Parallel

While I couldn't get my DBMS to work completely in parallel I was able to run a simple query using one, two and four workers. If I were able to get this working completely this would be a great improvement on performance as my DBMS could perform very large queries in parallel. If given more time I would fix this first and do an extension like a composite partitioning.

4.4 Optimizer

This lab did not focus on the optimizer at all. If I had more time I would have also improved the query optimizer to take into account costs of plans and other logistics.

I learned an immense amount of knowledge from this quarter-long project; performing queries, implementing a hash join, synchronizing my database, implementing my own locks, making my database recoverable and attempting to make it parallel. I chose the parallel lab over the optimizer because I knew it would be more challenging and more applicable later on in my future. Managing all the small pieces and slowly building up my database gave me a deeper understanding for databases than I could have ever hoped to learn through books and the opportunity to make it parallel is priceless. This is a memorable project that I might continue to work on after this class is over.