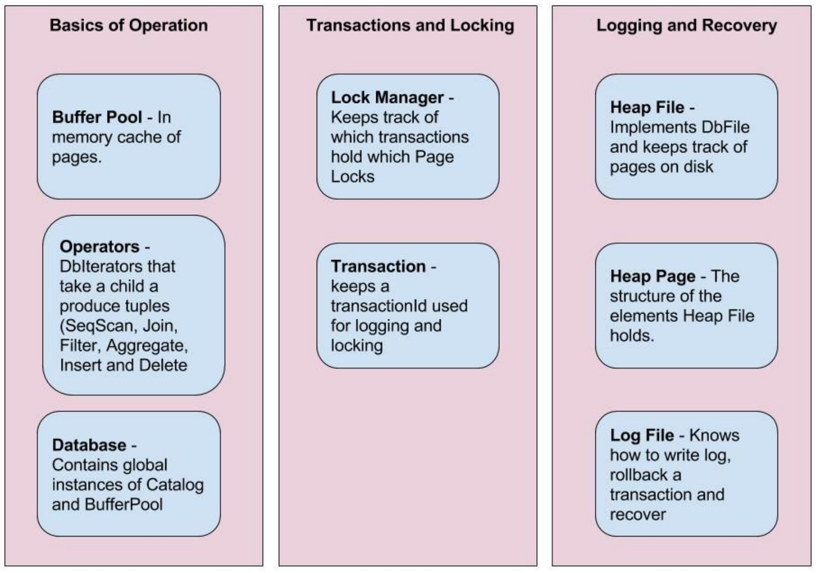
Andrew Dawson
1220796
Lab 6 Write Up

Section 1

SimpleDB is a relational database management system. That is recoverable and capable of running in parallel. The basic parts of SimpleDB are the BufferPool, the operators, locking/transaction support, logging/recovery support and support for running in parallel. The BufferPool is simply an in memory cache that holds recently used pages. The BufferPool is used by all the operators to get pages so that used pages are cached. The operators consist of Scanning, Joining, Filtering, Aggregating, Inserting and Deleting. The operators are all implemented as Iterators. The operators take some input (another operator) and produce output tuples from their child iterators. Chaining these operators together is what creates a query in SimpleDB. SimpleDB is capable of running transactions via its support for locking. Through implementing strict two phase locking simpleDB provides ACID properties for transactions. The locking is all taken care of by the LockManager class in conjunction with the BufferPool. In order for a transaction to operate on a page it must acquire the correct lock in BufferPool before it can operate on that page. When a transaction commits or aborts it releases all its locks - allowing other transactions to make progress. SimpleDB is also a recoverable system - if it crashes in the middle of execution it can recovery via an append only log that is flushed to disk. This log is used both to rollback aborted transactions as well as to restore the state of the database in the case of a crash. Lastly, SimpleDB also provides support for running in parallel via a Worker/Master relationship. The master node (Server class) receives a query and generates a non parallel query plan. Then it optimizes the query for parallel execution by replacing the operators with parallelized operators. Then the data (stored in HeapFile) is partitioned across many worker nodes. Each of these worker nodes receives a query plan and a subsection of the data. The subsection of the data is stored locally on each worker's machine. The workers execute their query and send the results back to the Server.


Basics of Operation
Buffer Pool - In memory cache of pages.
Operators - DbIterators that take a child a produce tuples (SeqScan, Join, Filter, Aggregate, Insert and Delete
Database - Contains global instances of Catalog and BufferPool
Transactions and Locking
Lock Manager - Keeps track of which transactions hold which Page Locks
Transaction - keeps a transactionId used for logging and locking
Logging and Recovery
Heap File - Implements DbFile and keeps track of pages on disk
Heap Page - The structure of the elements Heap File holds.
Log File - Knows how to write log, rollback a transaction and recover

Relationships

1. Operators access pages through BufferPool and BufferPool caches those pages
2. BufferPool holds an instance of Lock Manager and uses it to keep track of which transactions locks on which pages
3. Lock manager maintains a map between HeapPages and Transactions
4. HeapFile consists of many HeapPages and has methods for reading and writting pages to and from disk
5. Log File marks each log record with a transactionId for rollback and recovery
6. HeapPage keeps track of which Transaction dirtied it last

Buffer Manager
BufferPool is responsible for reading pages into memory. All operators access pages through BufferPool - this is important so that recently used pages are cached. While BufferPool is nothing more than an in memory cache of pages it performs three important functions: locking pages, implementing page eviction and interacting with disk. BufferPool works with LockManager to implement locking. Locking occurs at the page level. A transaction must hold an exclusive lock if it wants to write a page and it must hold a shared lock if it wants to read a page. Lock Manager is simply a map from transactions to the pages they hold locks on. In BufferPool.getPage() a transaction requests a lock (with a given permission level) on a page. This method uses the LockManager to acquire the lock before the getPage method can return. In large databases not all pages can fit in memory - therefore when the BufferPool fills up an eviction policy must be used. BufferPool uses a slight modification on least recently used eviction policy. When BufferPool.getPage() is called the BufferPool first checks if it has the requested page, if it does the page is simply returned. Otherwise the page must be read from disk. BufferPool uses an instance of DbFile to read pages from disk. BufferPool caches this read page and returns it.

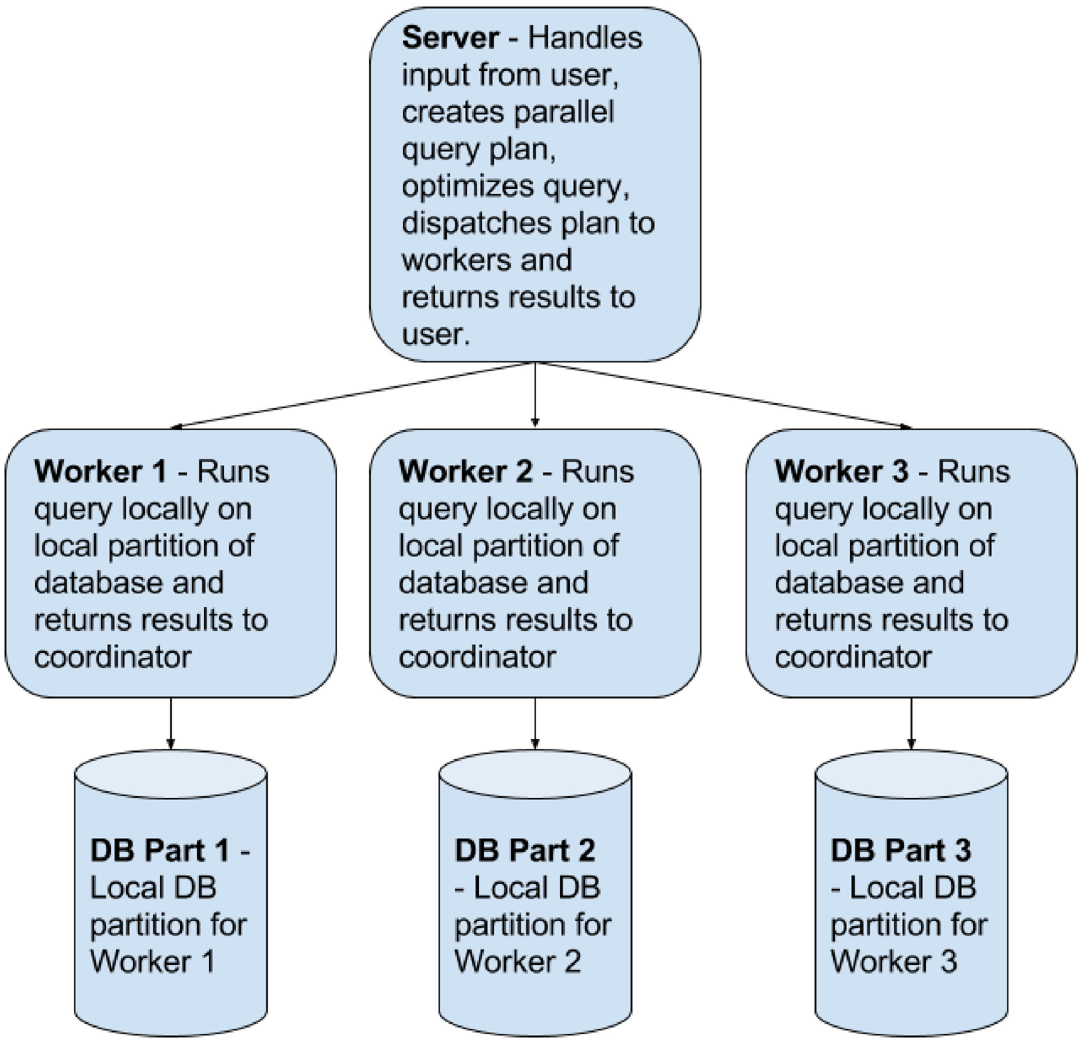
Operators
All operators are instances of DbIterator. DbIterator requires operators to implement methods such as next, hasNext, getTupleDesc, rewind and a few more. All these methods are focused around generating and returning tuples. All operators except SeqScan use the Operator abstract class to factor out common operator functionality. All operators take input tuples from some source and provide an interface to iterate through the generated tuples. SeqScan is unique in that it gets its tuples from disk (or BufferPool's cache if the page is cached). SeqScan gets its input tuples via a DbFileIterator. All other operators get their input tuples from one or two other DbIterators. Chaining these operators together is what creates a query.

Lock Manager
Lock Manager simply keeps track of which transactions holds which locks. It is used by BufferPool as explained above. Lock Manager implements strict two phase locking. It does this using a map from transactionId to a set of pages that are locked by that transaction. Lock Manager avoids deadlock using simple timeouts. If a transaction requests a lock but is unable to acquire the lock for a timeout amount of time - then the transaction is aborted.

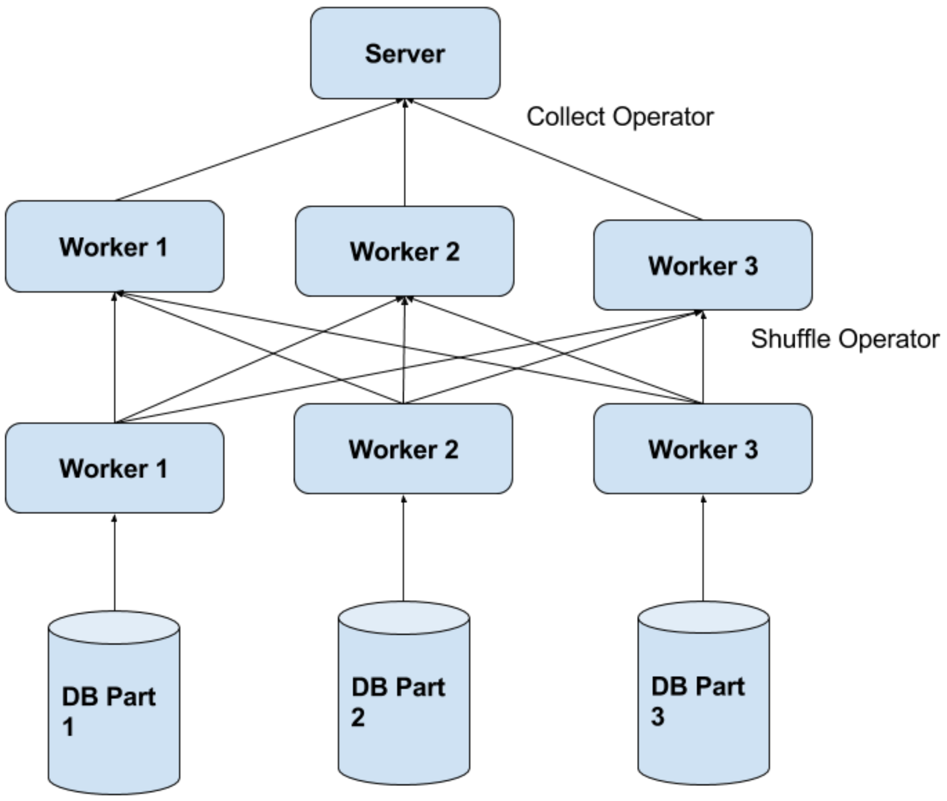
Log Manager
Log Manager is responsible both for writing logs and using the log for both recovery and rollbacks. Every log entry is marked with a transactionId of the transaction that is responsible for the action. The log is an append only log and is forced to disk. When a transaction X aborts, log manager finds all updates performed by X, extracts the before image (the state of the page before the update) and updates the on disk representation of the page to match the before image. When an update is rolled back the in memory representation of that page (BufferPool)

must be discarded. If the database system crashes the in memory state of the database is lost so the log manager must use the log to recover the state of the database. This is done using an undo and redo phase. From the last checkpoint in the log file all updates are redone using the after image. Then all loser transactions (transaction that did not commit before the crash) are undone.

Section 2


Server - Handles input from user, creates parallel query plan, optimizes query, dispatches plan to workers and returns results to user.
Worker 1 - Runs query locally on local partition of database and returns results to coordinator
Worker 2 - Runs query locally on local partition of database and returns results to coordinator
Worker 3 - Runs query locally on local partition of database and returns results to coordinator
DB Part 1 - Local DB partition for Worker 1
DB Part 2 - Local DB partition for Worker 2
DB Part 3 - Local DB partition for Worker 3

The diagram above depicts the high level architecture of a parallel deployment of simpleDB. A parallel instance executes a query using a set of concurrently executing processes. One of these processes gets designated the special role of coordinator. The coordinator is responsible for handling user input, generating/optimizing the parallel query, dispatching the query to workers, collecting results from workers and returning the final results to the user. The workers are each running in their own process and operate on a horizontal partitions of the full database. Each worker receives a query plan from the coordinate runs it locally and returns its results back to the coordinator. Server.java is the class that acts as the coordinator. It uses a set of optimizers (ParallelQueryPlanOptimizer.java) to produce a query plan that will run well in parallel. Server.java also inserts Shuffle and Collect operators to paralize the query (this is explained in more detail below). Worker.java provides functionality for localizing query plans and running a query plan. Since each worker operates on a local partition of the database features such as tableId are different between the coordinator and the worker. Therefore before a worker can run a query they must replace all instances of tableId with their local version. Workers run queries simply by iterating over the tuples. A query plan is nothing more than a chain of DbIterators strung together. Worker opens up the root DbIterator and calls next until all tuples have been returned.



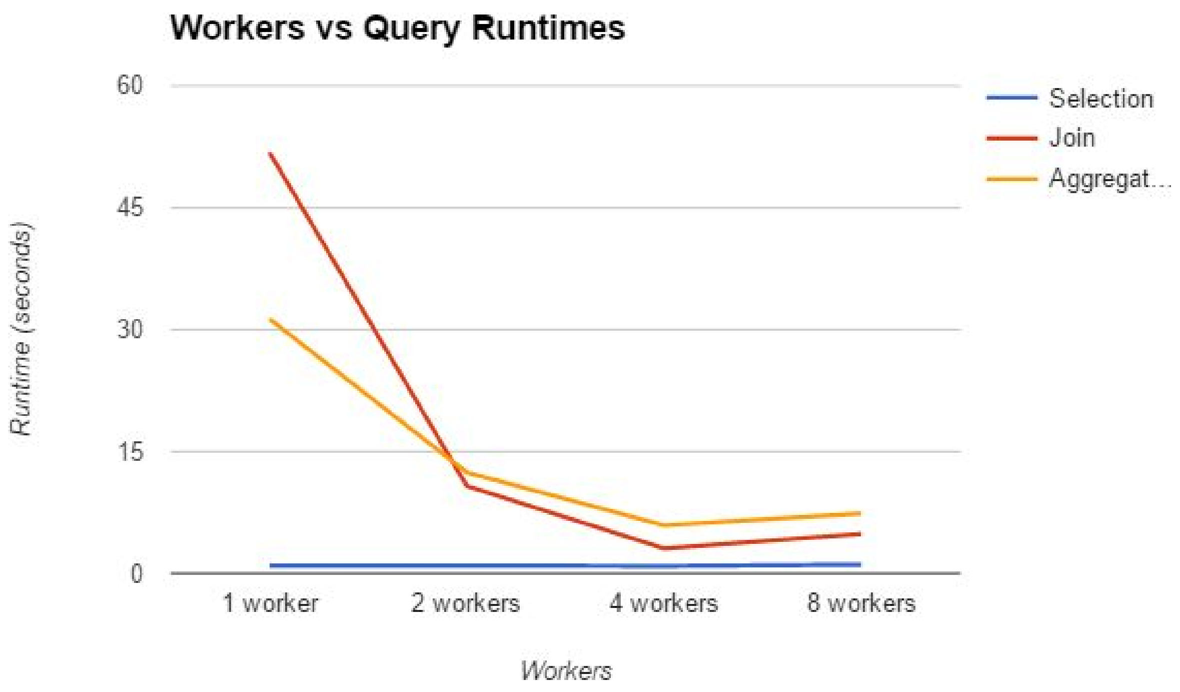
The following steps are ran to perform a query execution.

1. The coordinator receives a query from the user
2. The coordinator generates a sequential execution plan. This query is not yet parallelized and cannot run on multiple processes.
3. The coordinator parallelizes the query plan by inserting Shuffle and Collect operators. In order to execute a query in parallel workers must have a way to send data to several other workers (Shuffling) and workers must have a way to send data to single machine (Collecting). Both the shuffle operator and the collect operator have two pieces - a producer and a consumer. The producer is responsible from sending tuples from one machine to another; while the collector is responsible for receiving tuples sent by a producer. Shuffling is needed in the case of Joins. In a Join all tuples matching a predicate should be shuffled onto a machine. Collecting is needed to perform aggregates - in order to aggregate a result all the sub-results must be accumulated onto a single machine. In order to make aggregates more efficient local sub-aggregate results are computed locally on each worker and these sub computations are sent to the coordinator rather than sending the whole worker's set of tuples. For example if average is being computed all workers compute local sum_count values over their partition of the data, send these sub-results to the coordinator and the coordinator uses these sub-results to quickly compute the average.
4. At this point the query still is in the coordinator and nothing has been run yet. The query could be run in parallel at this point, however, it is not yet optimized for parallel execution. The coordinator runs the query through a series of optimizers that will modify the query plan to perform better in a parallel execution environment.
5. The coordinator now send the query plan to the workers. Each worker runs the query locally and returns the results back to the coordinator. If there are any aggregations to be done the coordinator will perform the aggregations. Finally the results are returned to the user.

My extension was to explore how varying the number of workers affected query runtimes. In order to do this I varied the number of workers and ran 3 types of queries (selection, join and aggregate). All timings are cold runs.

Experiment Runtimes for Variable Workers

	1 worker	**2 workers**	**4 workers**	**8 workers**
Selection	0.97 seconds	0.98 seconds	0.92 seconds	1.11 seconds
Join	51.78 seconds	10.76 seconds	3.08 seconds	4.87 seconds
Aggregation	31.31 seconds	12.43 seconds	5.92 seconds	7.38 seconds



These results show that as more workers are added the runtimes improve up to a point. This makes a lot of sense because with more workers we expect to get parallel speed up. However with too many workers the overhead of distributing the query and recombining the results will be a bottleneck. It is interesting to me that we see such good results for Join. This is surprising to me because Join requires data to be shuffled across many machines which is an expensive network operation - this would lead me to believe the overhead of many workers would dominate the runtime.

Section 3

The overall performance of the SimpleDB engine is good. There is almost a linear speedup seen as more workers are added. The system is recoverable and provides ACID guarantees for transaction support. Additionally the quality of code in SimpleDB is good. The code is clear and well commented.

If I had more time I would be interested in implementing support for indexing. Indexing would allow queries to run faster. It would be interesting to see the performance gains we get from adding indexes. I suspect that we would see similar types of speedups to the workers - a few indices would help runtimes but too many indices would slow down performance. It would also be interesting to see which types of queries benefit most from indices.

Overall I found this lab educational. It would nice to work with a large code base. It required a solid understanding of a large architecture and it made debugging interesting and challenging.

Appendix

In order to perform timings a script was written. The script automates varying workers and running different query types. In order to run the script make sure the script has execute permissions and run ./lab6-extension.sh. All the runtimes will all print to standard out with labels.