SimpleDB

- A distributed relational database management system -

CSE 544 M Project Report

Armaan Sood

Contents

Usage **1**

1 System Architecture **2**
1.1 Overview Description 2
1.2 Components 4
1.3 Diagrams 9

2 Parallel Data Processing **11**
2.1 Overview 11
2.2 Implementation and Design 11
2.3 Parallel Evaluation 15

3 Discussion **21**
3.1 Performance 21
3.2 Moving Forward 21

Usage

At a high level, here is how SimpleDB can be used. Suppose you have a file, `data.txt`, comprising multiple lines of "x,y", where x and y are integers. Then, you can convert this into a SimpleDB table by running `java -jar dist/simpledb.jar convert data.txt 2 ''int,int''`. This will then create a file, `data.dat`. Now, create a catalog file, `catalog.txt`, that includes "data (f1 int, f2 int). This will notify SimpleDB that there is one table, `data`, stored in `data.dat`, with two integers fields, f1 and f2. Lastly, invoke the parser with `java -jar dist/simpledb.jar parser catalog.txt`. Then, you can run queries.

To run as a distributed system, modify `conf/server.conf` and change the address to one that is accessible by the workers through DNS, if the workers are on other physical machines. Otherwise, just use `localhost:port` with any port. Then, modify `conf/workers.conf` with the address of each worker. Lastly, execute `bin/startSimpleDB.sh etc/dataToUse.schema` and run queries.

Section 1

System Architecture

SimpleDB is a relational database management system (RDBMS) written in Java that can handle simple queries, joins, aggregate functions, ACID transactions, and a steal/no-force crash recovery with non-quiescent checkpoints. It also can run in parallel, distributing the query workload to multiple workers.

1.1 Overview Description

Storage Manager

At the lowest level of SimpleDB is the storage manager. This includes access methods, such as the heap files, the buffer manager, the lock manager, and the log manager. The access methods organize the data in OS files, or heap files, to support fast access to the requested subsets of tuples. Each heap file corresponds to a relation and breaks down the stored data into multiple pages (heap pages), which are the units of disk IO. Each heap page contains a page slot directory at the beginning, indicating which slots of the page are full and which are used, and then it contains the tuples after. The buffer manager caches accessed data (pages) in memory (the buffer pool) and supports writing to and reading from disk. After getting full, the buffer manager randomly evicts a page. SimpleDB implements a steal/no-force policy, which means the buffer pool can flush any page to disk (steal), and it doesn't require flushing all pages associated with a transaction to disk upon committing. These flexible policies require the log manager to implement both undo and redo logging. The log manager handles transactional recovery, including redoing transactions that have committing and undoing uncommitted and aborted transactions. It writes to a physical log file to ensure transactions aren't lost. In order to guarantee ACID transactions, the lock manager implements strict two-phase locking and grants shared and exclusive locks to transactions that wish to read or write a page. Whenever a transaction requests a specific page from the buffer manager, it must specify the permission level (either read or read and write). If it specifies read, it will be granted a shared lock and access to the page as long as that page has no exclusive locks on it. Otherwise, the transaction must wait for the lock to be freed. If it specifies read and write, it will

be granted an exclusive lock if there are no other locks on the page or the only lock on the page is a shared lock by the same transaction. In either case, the lock manager implements a deadlock detector using a wait-for-graph to prevent and abort any deadlocks.

Query Processor

On the next level of SimpleDB is the query processor. The query processor first parses queries into an internal format, performing various checks using the catalog (this contains references to access methods and pages). Then, it rewrites the query using view rewriting and flattening. Next, it finds an efficient physical and logical query plan to execute the query using the query optimizer. Lastly, the query is executed. Each operator implements a pull-based iterator interface, so the query is executed by calling the `open` method on the top-most operator of the query plan (usually a projection), which subsequently calls the `open` method on its children, until it reaches the access methods. After opening an operator, `next` can be called to retrieve the requested subset of tuples. Lastly, `close` is called to close any opened resources. Chaining operators is simple because of the iterator interface.

1.2 Components

The main components of SimpleDB include the buffer manager, the lock manager, the log manager, and various operators.

Buffer Manager

The buffer manager is the main access point for operators to retrieve data from data files. It acts as a much simpler way for operators to access data — it would be complicated to safely access OS file data directly from an operator, as operators would need to know the exact offset of the data, and would then need to manage transactions in a different way. The buffer manager also communicates with the lock manager and log manager to handle transaction processing and recovery. Internally, it stores a buffer pool, which, in SimpleDB, is a concurrent hash map mapping page ids to their pages.

If a transaction needs access to a page, it must first know the page id, which can be found in the catalog given the name of the page. The method that needs access to the page will call the `getPage()` method, with the transaction id, page id, and permission levels as arguments. The permissions can be either "read only" or "read and write", depending on whether the transaction will be writing or not. After the method is called, the buffer manager asks the lock manager for a shared lock if the permission level is "read only", or an exclusive lock if the permission level is "read and write". If the lock manager denies the lock, the buffer manager continues looping until it gains the lock. If there is a deadlock, the lock manager will detect it and abort the transaction. After obtaining the lock, the buffer manager checks if the buffer pool already contains the page id. If so, it returns the corresponding page. Otherwise, it checks whether there is space in the buffer pool. If there's no space, it will randomly evict a page. Then, it adds the page id as a key in the buffer pool, and it maps this key to the corresponding page. In order to get a reference to the corresponding page, it asks the catalog for the database file (usually a heap file) corresponding to the table id associated with the page id, and then it asks the database file to read the page corresponding to the page id. The database file then needs to read the corresponding page bytes from disk and return this page to the buffer manager, which then adds the page to the buffer pool and returns the page to the original caller.

The buffer manager also handles the insertion and deletion of tuples. If a transaction with transaction id `tid` wants to insert tuple `tup` into a table with table id `t`, then it calls `insertTuple(tid, t, tup` in the buffer manager. The buffer manager gets the table's corresponding database file and calls its `insertTuple(t, tup)` method. That method finds a page with space (if not, it constructs a new page), and then the page writes the bytes of the tuple to its in-memory data and marks the header slot as used. Deletion of tuples works in a similar way.

Lastly, the buffer manager also handles completing transactions and flushing pages to disk. If a transaction commits, then the buffer manager requests all of the transaction's associated pages from the lock manager and it tells the log file to write their before and after images to the log file on disk. If the transaction aborts, the buffer manager restores all of the dirtied pages associated with the transaction by re-reading them from disk. In both cases, all locks held by the transaction are

released. Whenever a page is evicted from the buffer manager, the buffer manager flushes it to disk by writing a log entry, similar to above, and using the heap file's `writePage()` method.

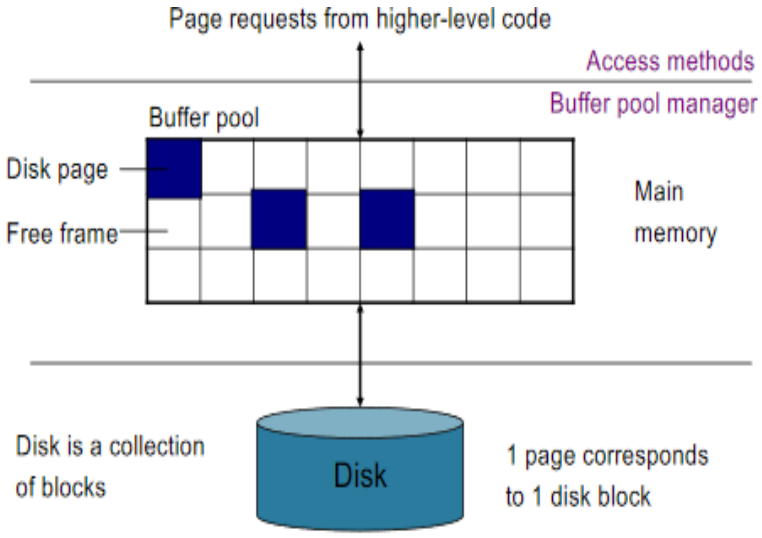

Figure 1.1: Buffer manager

Lock Manager

In order to provide ACID transactions, SimpleDB implements strict two-phase locking with shared and exclusive locks. The lock manager is only accessed via the buffer manager, and locks are only obtained in the buffer manager's `getPage()` method. The lock manager works by storing mappings from page id's to the transaction id's which hold exclusive locks and transaction id's which hold shared locks on the corresponding page. A mapping of transaction id's to a set of page id's is also stored to facilitate checking for locks, releasing locks from a transaction, and returning pages associated with a transaction. Lastly, a mapping of transaction id's to a set of transaction id's is stored to implement the deadlock detector's wait-for-graph.

If a transaction requests a shared lock, the lock manager checks to see if there are any exclusive locks held on the page. If so, the lock manager adds to the wait-for-graph, checks for a deadlock by finding a topological sort on the wait-for-graph, and returns false to the buffer manager (which then calls the method again). If a deadlock was detected, the lock manager releases all locks associated with the transaction id, removes the transaction from the wait-for-graph, and aborts the transaction. Once the transaction is able to obtain a shared lock, the lock manager adds it to the list of transactions holding a shared lock on the page.

If a transaction requests an exclusive lock, the process is similar to the above, but a lock isn't granted unless there are no other locks on the page or the only other lock is a shared lock that can be upgraded (a shared lock held by the same transaction).

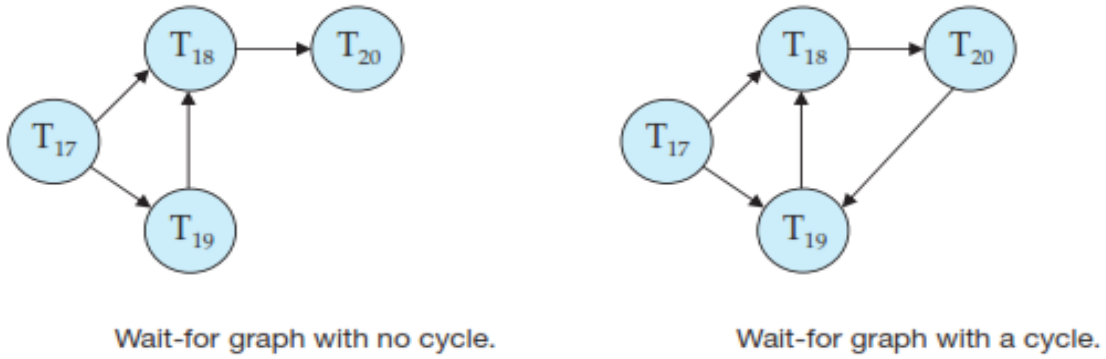

Figure 1.2: Example of a wait-for-graph without and with a deadlock

Log manager

SimpleDB implements a steal/no-force transaction policy, which means that pages of uncommitted transactions may be written to disk (steal) and pages of committed transactions don't need to be written to disk (no-force). These policies increase the throughput of SimpleDB, as pages won't be constantly written to disk, and it can write them in batches. In order to safely allow this to happen, our write-ahead log must be able to both undo and redo transactions. The main rule that must be followed for undo/redo-logging is that if a transaction T modifies X, then the log must write $< T, X, u, v >$ to disk (log file) before X is written to disk (heap file), where u is the old value and v is the new value.

In SimpleDB, recovery, rollback, and writing to the log is handled by the log file. At the very beginning of the physical log file is a long representing the offset of the last written non-quiescent checkpoint. Each log entry begins with an integer, representing the type of log entry, and a long, representing the transaction id. Then comes the log entry data (such as the page for an update record), and the log entry ends with a long integer representing the position in the log file where this log entry began — this makes traversing the log much simpler. There are six main log entry tags: commit, update, abort, begin, checkpoint, and CLR. These will be explained in more detail.

Say a transaction with transaction id, `tid`, aborts, and the rollback method is called on that. After getting a lock on the buffer pool and the log file (to prevent conflicting writes), and after pre-appending to the log file, it creates an empty hash set of transaction ids and adds the single transaction id, `tid`, to this set. Then, a function that will rollback every transaction in this set is called. This function first finds the earliest log record in the set by comparing them all from the map. This earliest record will be the stopping point, since the undo-log scans backwards. After finding the earliest record, the log manager seeks to to eight bytes before the end of the file. Since each log record has, at the end of the record, a long indicating where in the file the record begins, the log manager can read this (eight byte) long and seek to that location. At the beginning of each log record is a tag indicating the type of log record. If the tag says the record is an update record, the log manager can then read the transaction id, which is conveniently encoded as a long directly after the

tag, and the log manager can check if the argument set of transaction ids contains this transaction id. If so, the log manager calls a method to read page data. First, the log manager reads in the page data to get the "before" page image then reads in the page data again after the first page offset to get the "after" page. After getting these two pages, the log manager write a compensating log record to the log containing the transaction id, after image, and before image (this will be explained in more detail). Then, the log manager requests the heap file corresponding to the after page from the catalog, and it writes the before page to this heap file, thus correctly undoing changes. Then, the buffer manager discards the after page from the buffer pool, since it is outdated information. After this, and also in the case that the log manager didn't undo/write to disk (if the transaction or tag didn't match earlier), the log manager seeks to eight bytes before the original seek location (the place where it found the tag). This places it in a position to find the beginning of the next log record. It repeats the entire above process until the current offset in the file is smaller than the first log record (which is obtained from the map, as described above).

After deciding to undo a transaction, but before writing it to disk, the log manager writes, at the very end of the file, a unique CLR tag, then the transaction id, followed by the updated page and the page that had been undone. After this, the log manager writes a long indicator to where this CLR record had started. CLR records make undoing have less work, since the log manager doesn't need to worry about undoing stuff that has already been undone (like aborts, or if the database crashes while undoing something).

If the database crashes, the recover method is called. This method first redoes changes, then undoes changes that haven't committed. First, the log manager seeks to the beginning of the file. It reads in the first long of the file, indicating if and where the most recent checkpoint is. If there is no checkpoint, it starts redoing from the beginning. If there is a checkpoint, it goes to the checkpoint, reads in the list of transactions and where they begun, and starts redoing from the earliest transaction. Then, it creates two sets, one for the "loser" transactions and one for transactions that have committed. While doing this, it adds any committed and CLR transactions to the committed records set and adds any update records to the losers set. Once it finds an update or CLR record, it reads in the first page of the record, which is the page it will write on, and reads in the second page, which is the page it will read from to overwrite the first page with, and then it redoes the changes by overwriting the page. After it has gone through all of the update and CLR records, it removes from the losers set all the transactions from the committed set, and it calls the rollback method on the losers set.

Operators

In order to retrieve and filter data from disk, SimpleDB has various operators, some corresponding to the logical operations of relational algebra. These are responsible for the execution of the query plan. Operators implement a pull-based iterator interface, allowing query plans to be composed of a tree of chained operators — each operator (except for access method operators) store a given child operator upon construction. The operator interface has five main methods: `open()`, `close()`, `hasNext()`, `next()`, and `rewind()`. Most operators implement these methods around calling the child's corresponding method. The topmost level of the plan starts the query execution by

calling `open()`. This loads any resources needed to start retrieving tuples. Then, after checking if there are any tuples left to retrieve with `hasNext()`, `next()` is usually called, which retrieves the first tuple that matches the SQL query. Lastly, `rewind()` can be called to restart the iterator from the first tuple, and `close()` can be called to close any open resources.

At the bridge between the query plan and the disk is the heap file iterator access method. After calling the heap file's `iterator(tid)` method, a new heap file iterator is constructed that will iterate over the entire file. Calling `open()` gets the first heap page from the buffer manager (with read only access), gets the tuple iterator for that page (this iterates over all the tuples in the page) and stores these. Calling `next()` in the heap file iterator returns the next tuple by calling next on the heap page's tuple iterator. If there are no more tuples left on the page, the heap file iterator finds the next empty page and returns the first tuple on that page.

The root operator of the query plan is the sequential scan operator. This operator is an access method that reads each tuple from a table in the order they're laid out on disk. The operator is constructed with `SeqScan(tid, tableId)`, where `tid` is the transaction id associated with the query and `tableId` is the id of the table that will be sequentially scanned. The `open()` method for this operator asks the catalog for the database file corresponding to the table id, and then asks the database file for an access method iterator. Then, the `open()` method of the database file iterator is called. Calling `next()` simply returns the result of calling `next()` on the database file iterator — this is the same case for `rewind()`, `hasNext()`, and `close()`.

Any operator can hold sequential scan as a child — aggregate, delete, filter, insert, join, order by, project, and rename are some of the operators SimpleDB supports.

The join operator is a unique operator, in that, upon construction, it takes in two children, corresponding to the inner and outer relations to join. It also requires a join predicate, which can be used to check whether two tuples can be joined. Calling `open()` calls open on the two children. Depending on whether the join predicate is checking for equality (as opposed to inequality), the physical join plan will be either a hash join (better for equality, by the nature of hashing) or nested loop join (better for checking if a tuple is in a range). For hash join, all of the inner child relation is hashed on the join attribute into buckets (via a hash map). After hashing all the tuples (when the inner child's `next()` method is depleted), each tuple of the outer relation is retrieved, hashed on its join attribute, and checks for the first match in the corresponding bucket. If there is a match, a new tuple containing a concatenation of the matching tuples is returned. Subsequent calls to `next()` will continue searching this bucket until it is depleted. If there isn't a match, it keeps searching using the next tuple. For the nested loop join algorithm, every tuple of the inner relation is searched for each tuple of the outer relation to find a satisfied join predicate. Similar to every other operator, calling other methods (such as `close()`) just calls the corresponding method in the child operators.

1.3 Diagrams

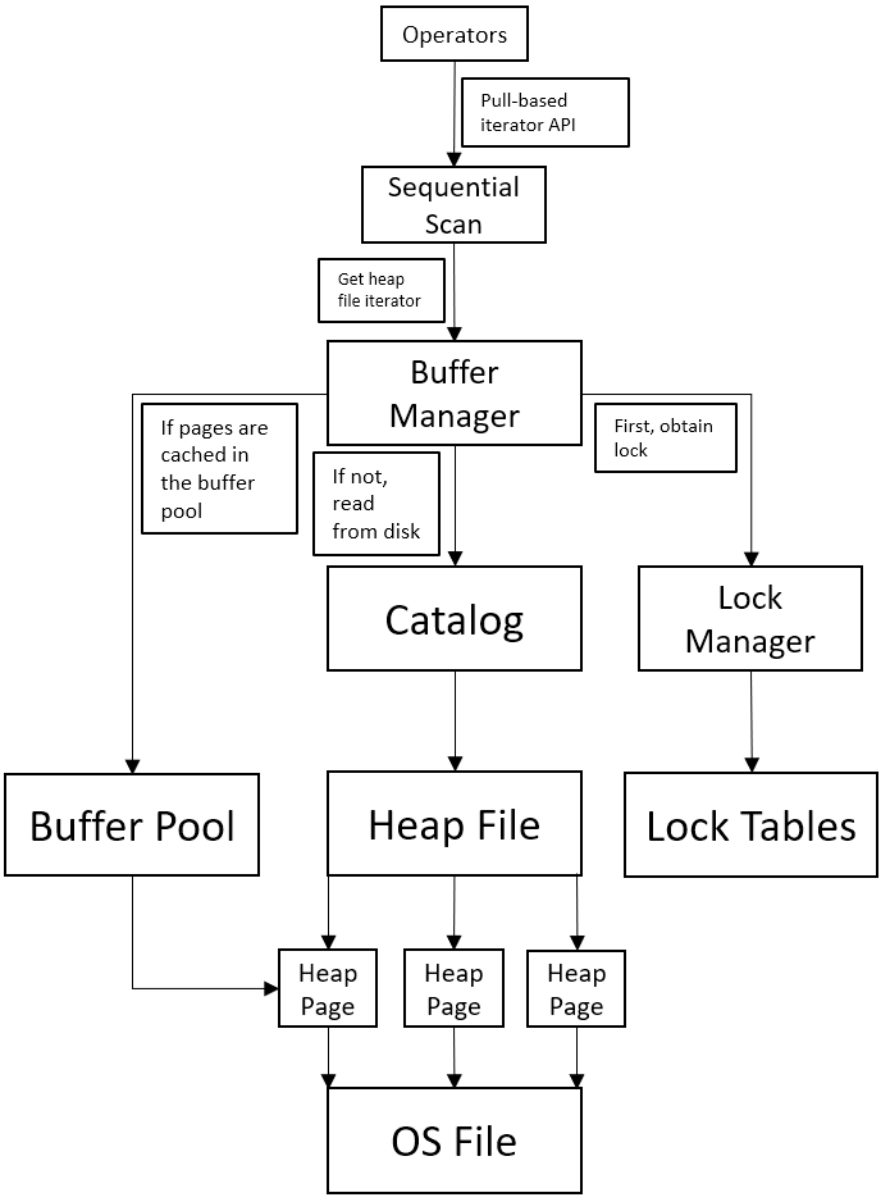

Figure 1.3: Overview of SimpleDB's architecture

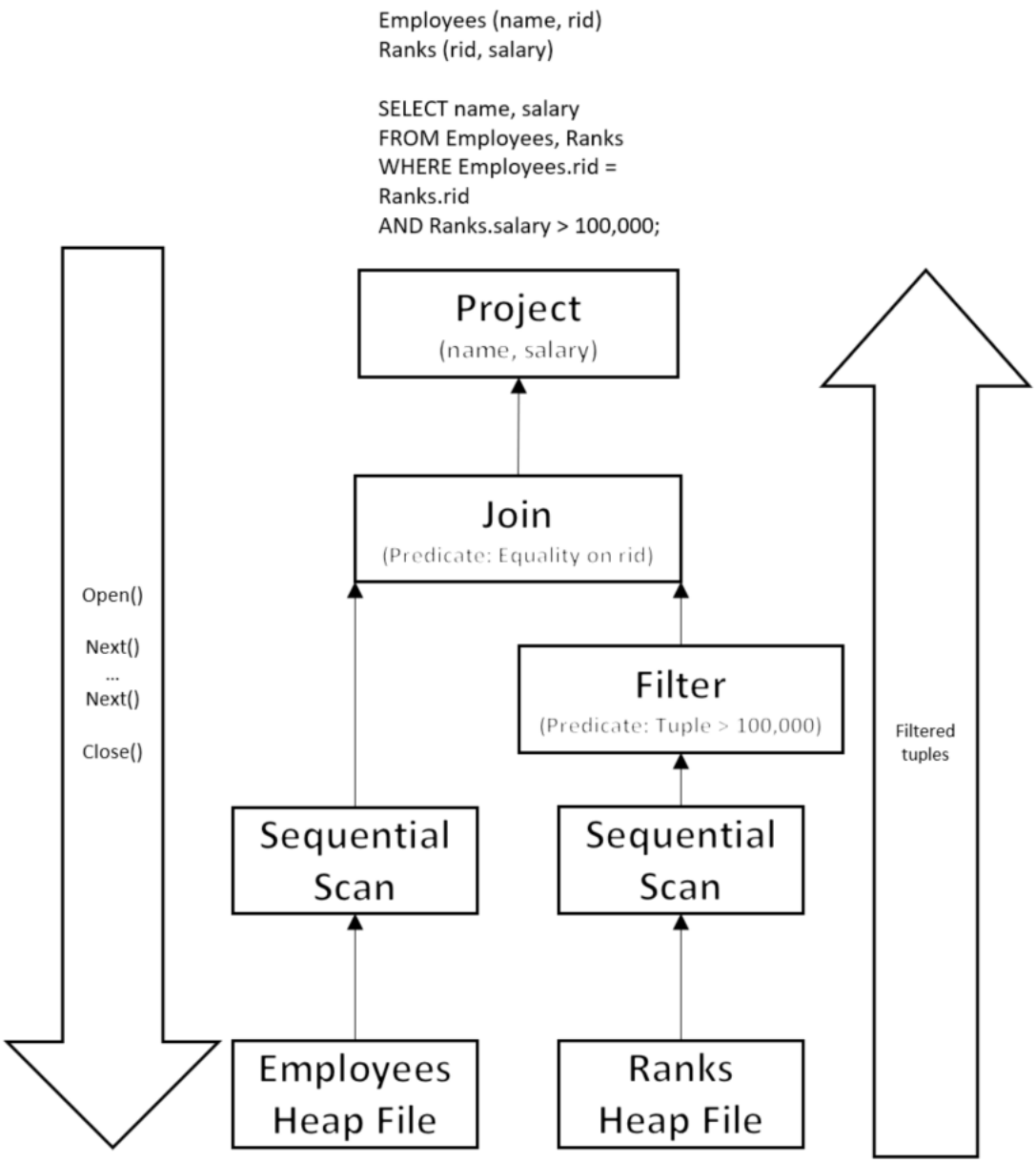

Figure 1.4: An example of a query plan. Notice that the iterator methods are called from top to bottom, and tuples are returned from bottom to top.

Section 2

Parallel Data Processing

SimpleDB also can run queries in parallel — either on a single machine or as a distributed system with shared-nothing architecture across multiple physical machines.

2.1 Overview

SimpleDB horizontally partitions data to each node. The addition of new operators, shuffle and collect, for producers and consumers, allow for tuples to be sent between nodes. Projections and simple selection operations can operate in parallel, each on their own node, and return their results to a single master node. For equi-join operations, each node hashes its tuples on the join attribute and sends each hash bucket to a certain node (each node uses shuffle producer to send the tuples and shuffle consumer to collect the tuples). From there, each node can perform a local join much quicker due to the hashing. For aggregate operations, if there is no grouping, each node can compute a partial aggregate and send the results to a single node (the nodes send via collect producer and the single node collects via collect consumer). If there is grouping, similarly to join, each node hashes on the grouping attribute and sends tuples (via shuffle producer) to other nodes corresponding to buckets.

2.2 Implementation and Design

The main components involve a basic worker (node) process which works with other workers running in parallel, the shuffle and collect operators that allow SimpleDB to run joins and aggregates in parallel, and an optimized version of the aggregation operator for parallel queries. These will allow SimpleDB to execute queries with multiple processes, exchange data between processes, and optimize operators for a parallel architecture.

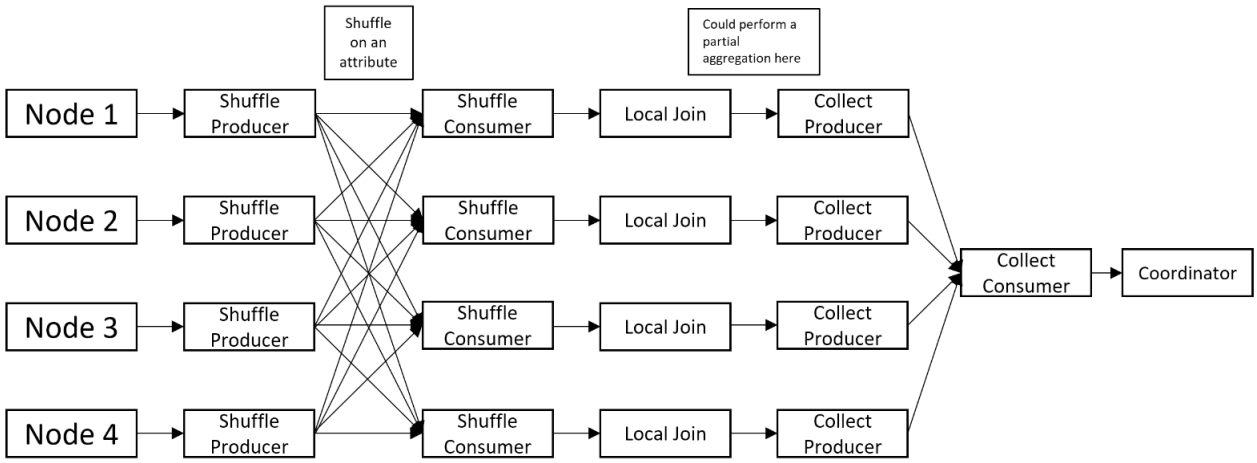

Figure 2.1: An example of how shuffle producer, shuffle consumer, collect producer, and collect consumer can be used to distribute queries over multiple machines.

Worker

One of the most important pieces to running SimpleDB in parallel is the worker. Any query received by the server (the coordinator) will be pre-processed and then sent to all the workers.

At a high level, the server waits for the client to enter a query, then the server generates a query plan and sends it to the workers, who store the query plan and localize it. Next, it notifies the server that it is ready to begin executing. The server waits for all workers to send this message, then the server notifies all workers to start executing. After receiving the start message, the worker executes the query plan.

On a lower level, the worker class runs with two command line arguments: the worker's address (worker id) and the server's address. The worker stores its address and parses it to find and store the hostname and port number. Then, it parses the server's address to find the hostname and the port number, and it attempts to create and store a socket address by resolving the hostname into an IP address (this should pass as long as the address is valid). Then the worker creates an acceptor — this binds to a TCP socket and waits for connections. The worker will receive query plans and control messages from the server and tuples from other workers during query execution through the acceptor. After creating this worker, SimpleDB loads a copy of the schema for the worker, since the schema is the same as the server's schema. Then, the worker gets prepared to receive network messages by setting up the handler for the previously initialized acceptor. Lastly, the message handler starts by binding the acceptor to a network socket (which has the same hostname and port as the worker), and then the worker starts. Periodically, SimpleDB will detect if the server is down — if so, the worker will stop running. Now, the worker listens for messages to arrive over the network.

After the worker's working thread begins to run, it continuously checks for a query plan. If it received a query plan, it knows that the topmost operator will be a collect producer, since, at the end

of the query execution, it needs to send all tuples to a single coordinator (the server). The working thread casts the topmost operator into a collect producer, opens it, and calls `next()`. At a high level, calling `open()` on the collect producer opens its child operator, executes the operator, and sends all of the child operator's tuples to the paired collect consumer (the collect producer operator stores the socket address of the paired collect consumer upon construction). The working thread sends tuples by first creating a network session between the consumer (using its socket address) and the producer (using its handler). Then, tuples from the child operator continuously fill up tuple bag buffers and are written to the session after reaching a certain capacity or time limit. Calling `next()` waits for collect producer's working thread to terminate and returns nothing — it just lets the worker's working thread know that there is nothing left to send. Lastly, the worker's working thread calls `close()`, which closes the child operator in turn.

However, before executing the query plan, it must first be localized with information local to the worker, rather than the server. The localize method checks for sequential scan operators in the query plan and resets the underlying table id to one that is local to the worker, using the table's name and the worker's catalog. For any producer operators, the localize method needs to update the worker that the producer runs on to this worker so that it can send data to the consumer process. Lastly, each consumer operator needs an input buffer, which, as described earlier, is what the producer writes to in a network session. The buffer is found using a map that maps the operator's id to get the I/O buffer where messages are sent to.

Some of the data sent over the network is serialized, such as a tuple's fields. However, a lot of the data should not be serialized and should instead be recalculated after being sent over the network. For example, the database file iterator (the access method) is stored in sequential scan, but after being sent over the network, it is no longer relevant since the iterator was local to the coordinator node, not the worker node. Therefore, sequential scan stores a transient iterator, which means it isn't sent over the network and it's recreated after being sent, using the local catalog.

Shuffle Operator

In order efficiently execute equi-joins and aggregations, the workers need to be able to send and collect tuples to other nodes based on a hash attribute. While the collect producer operator allow for the sending of tuples to a specific single node, shuffle producer sends to multiple.

At a high level, if the user has relations $X(a, b)$ and $Y(b, c)$ and wants to perform and equi-join on b, SimpleDB running in parallel will partition based on the value of b and each worker will get certain values of b. Then, each worker receives the bags of tuples with the same b values and each shuffle consumer passes the tuples to the join operator.

On a lower level, the shuffle producer operator is constructed with a child operator, the operator id (for the server and the workers to find out which operator is the owner of an incoming message), an array of socket information for the workers it will be sending data to, and the partition function (used to simplify figuring out which worker to send a tuple to). Upon calling `open()`, shuffle producer opens its child, and invokes a working thread to start. The working thread works similarly to collect producer, but with multiple network sessions. When the working thread runs, it creates an array of network sessions and an array of empty buffers. The array of network sessions is

created by using the the corresponding socket addresses from the array stored upon construction as the remote address, and using the I/O handler from the current worker as the handler for the other end of the session. Then, while the child of the shuffle producer operator has tuples left, the next tuple is stored (by calling the child's `next()` method), and the corresponding worker is calculated (using the partition function). The tuple is added to the worker's buffer (in the array of buffers), and it checks to size and time to see if the buffer should be written to the session. After all of the tuples have been emptied from the child, it writes all of the buffers to their corresponding sessions.

On the other end of the session is the shuffle consumer. Upon construction, this operator stores the child operator (which is a shuffle producer), the operator id, and an array of socket information for each worker. Calling `open()` essentially tells the shuffle producer (the child operator) to open. Calling `next()` reads in a single tuple from the incoming buffers that the shuffle producers sent.

Aggregate Optimization

Aggregate operators can also be optimized by running in parallel. There are two cases: either it is an aggregate operator with grouping or it is one without grouping. In the case that there is grouping, it works similarly to join — the shuffle producer operator is used with a partition function based on the group by attribute, and the shuffle producer operator passes tuples to the aggregate function. In the case that there is no grouping (such as `COUNT(*)`), each worker first performs a partial aggregation then sends the results to a single master worker via the collect operators. Finally, the master work performs the aggregation of the partial aggregates and sends the results to a coordinator. Note that this works for all non-holistic functions except for average — for this function, the partial aggregation is performed by calculating both the sum and count, and then the master worker calculates the average by summing the sums and counts and then dividing.

SimpleDB has an aggregate optimizer that replaces each aggregate operator within an unoptimized parallel query plan with two aggregate operators — a down-stream and an up-stream aggregate operator. Down stream is the aggregation done before sending the nodes between workers, and upstream is done after. For example, for a count operation, the downstream aggregation would be count and the upstream would be sum, since the sum of the counts is the overall count.

For each aggregate function, the optimizer replaces it with two. Sum is replaced with a down-stream and up-stream sum, min is replaced with a down-stream and up-stream min, and max is replaced with a down-stream and up-stream max. In the case of average, however, the down-stream is a sum-count function, and the up-stream is a sum-count average function. Normally, an aggregate function outputs a tuple with one or two fields, but the sum-count function outputs a tuple with two or three fields. The first field is the grouping attribute, if any, the second field is the sum of the aggregate values, the third field is the count of the aggregate values. Then, the sum-count average function sums up the sum and count fields and returns a tuple with the grouping attribute and the result of dividing the count field by the sum field.

2.3 Parallel Evaluation

To evaluate the performance of different queries on the parallel database, the timing of various queries with varying number of workers and trials will be presented. Specifically, time in seconds will be recorded from testing one of three queries on a SimpleDB instance with either one, two, or four workers, on two datasets — one (0.1) being 10 times larger than the other (0.01) — and three trials each. Testing one worker is roughly equivalent to running the non-parallel version of SimpleDB.

The first query is:

```
select * from Actor where id < 1000.
```

Here is the raw data:

Query 1	1 Worker	2 Worker	4 Worker
Trial 1	1.08	0.37	0.25
Trial 2	0.66	0.39	0.24
Trial 3	0.63	0.35	0.24
0.01 average	0.79	0.37	0.24333333
Trial 1	4.54	2.69	1.89
Trial 2	4.52	2.43	1.67
Trial 3	4.39	2.34	1.36
0.1 average	4.48333333	2.48666667	1.64

Figure 2.2: The raw data for query 1. The first three trials are for the 0.01 dataset, the second three are for the 0.1 dataset. All times are in seconds

Notice how the seconds taken to execute a query gets smaller as the trials go up — this is likely due to the data being cached.

Here is a graph comparing the speedup for the smaller dataset and the larger dataset.

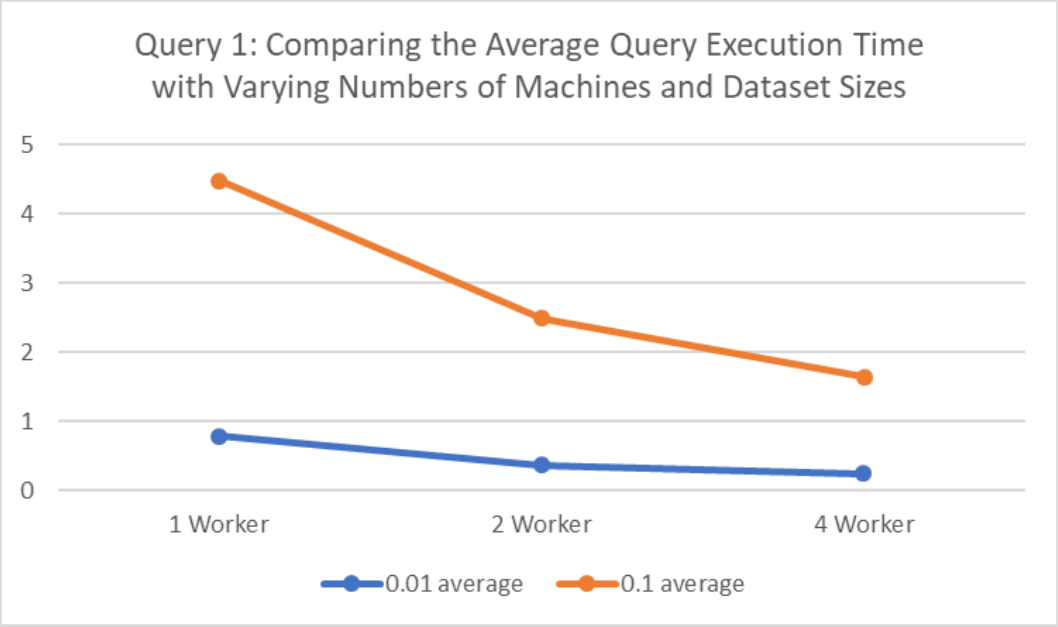

Figure 2.3: The graph representing the speedup and scaleup for query 1.

As evident in the graph, SimpleDB both speeds up and scales up decently well with simple queries. The speedup seems to asymptote around 4 workers, as seen by the small deviation in time from 2 to 4 workers, compared to the deviation from 1 to 2 workers. Clearly, for simple selection queries, running on multiple nodes is much faster than running on a single node, but adding more than 2 workers seems to have no effect, at least for these dataset sizes.

The second query tests parallel joins:

```
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.
```

Here is the raw data:

Query 2	1 Worker	2 Worker	4 Worker
Trial 1	0.3	0.89	0.44
Trial 2	0.17	0.37	0.44
Trial 3	0.15	0.27	0.45
0.01 average	0.206667	0.51	0.443333
Trial 1	2.03	1.46	2.18
Trial 2	1.53	1.16	1.3
Trial 3	1.51	1.02	1.31
0.1 average	1.69	1.213333	1.596667

Figure 2.4: The raw data for query 2. The first three trials are for the 0.01 dataset, the second three are for the 0.1 dataset.

As previously mentioned, the seconds it takes to execute each query goes down with the number of trials, likely due to caching.

Here is a graph comparing the speedup for the smaller dataset and the larger dataset.

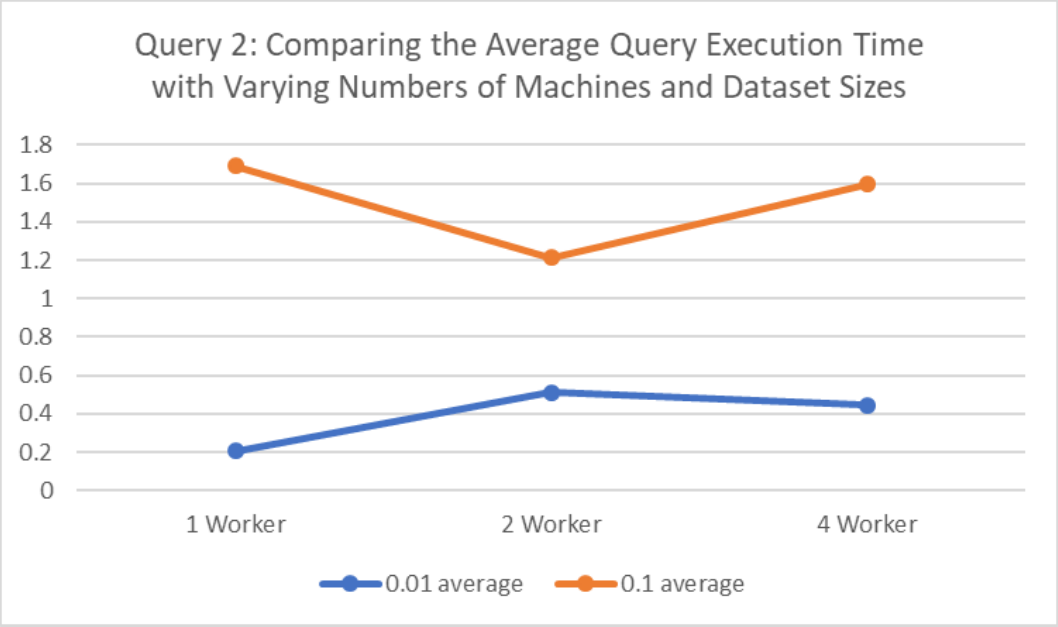

Figure 2.5: The graph representing the speedup and scaleup for query 2.

For the smaller dataset, it seems that 1 worker is faster than 2 workers. This is unlike the previous query, because this query uses a join, and therefore uses the shuffle producer and consumer operators. This point could be an outlier (it likely is not significant, as there is a 0.2 second difference between 1 and 2 workers). If the point is not an outlier, it could potentially be due to the slightly larger overhead of having 2 workers as opposed to having 1 worker (multiple network sessions will be opened, rather than a single, trivial one). For the larger dataset, it seems that 4 workers takes longer than 2, and this isn't the case for the smaller dataset. If this is significant, then it could be due to the overhead of the shuffle operator, as there are many more tuples to send between the four workers in the larger dataset. Overall, SimpleDB seems to speedup slightly worse than with simple queries, but the scaleup is still somewhat significant (although both datasets ran quickly).

The last query tests parallel aggregation:

```
select m.name,m.year,g.genre
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
```

Here is the raw data:

Query 3	1 Worker	2 Worker	4 Worker
Trial 1	0.84	0.7	2.4
Trial 2	0.83	0.67	1.3
Trial 3	0.8	0.74	1.24
0.01 average	0.823333	0.703333	1.646667
Trial 1	5.71	3.34	3.7
Trial 2	5.51	3.23	3.13
Trial 3	5.51	3.36	3.24
0.1 average	5.576667	3.31	3.356667

Figure 2.6: The raw data for query 3. The first three trials are for the 0.01 dataset, the second three are for the 0.1 dataset.

As previously mentioned, the seconds it takes to execute each query goes down with the number of trials, likely due to caching.

Here is a graph comparing the speedup for the smaller dataset and the larger dataset.

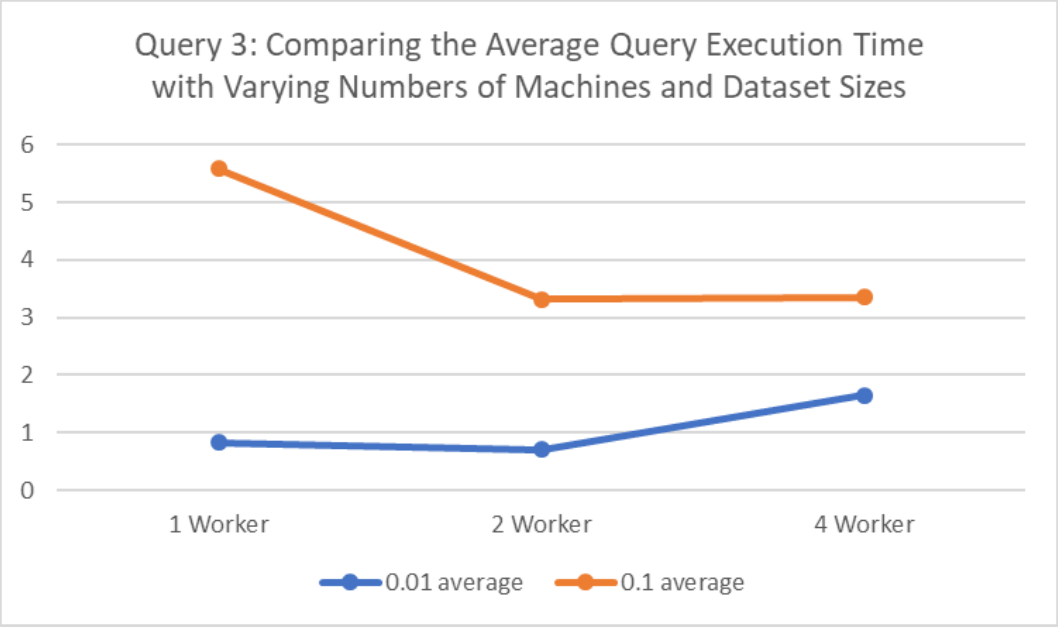

Figure 2.7: The graph representing the speedup and scaleup for query 3.

The larger dataset ran much more slowly than the small dataset, so the scaleup for aggregate queries is likely large. It seems like there isn't really a benefit from moving to a single node SimpleDB to a parallel one for the smaller dataset — the queries took longer to execute, likely due to the overhead of the parallel operators. For the large dataset, there was a significant speedup moving from one worker to two workers, but not so much when moving from two to four — again, this is likely due to the additional overhead of four workers on such a large dataset.

Section 3

Discussion

SimpleDB is nowhere near a perfect, or even complete relational database management system — let alone a distributed database management system. There are many features that are essential to contemporary database systems that SimpleDB lacks, such as replication and distributed transactions. However, as the name suggests, SimpleDB isn't meant to be anything but a simple database management system — it does contains many essential features, such as a basic storage manager, query executor, write-ahead log, and lock manager, but it cannot offer anything that another database management system lacks. Despite this, it was a great learning experience.

3.1 Performance

Performance is a very broad category — it could mean anything from the basic parallel time-based performance tested in Section 2, to usability, security, scalability for massive sets of data, network performance (for database systems in poor network-access zones) or even energy consumption. Personally, I think that energy consumption would be an interesting thing to test on a parallel database management system compared to a single-node database management system, and I think that the energy consumption of large-scale database systems is something to worry about. However, this cannot be easily tested in SimpleDB, so I'll discuss mainly time-based performance. It performs much better than I would've expected, almost as fast as SQLite. I'm especially surprised at how quickly it performed multi-way parallel joins.

3.2 Moving Forward

If I had more time, I would've added index pages and index scans, rather than having only heap pages and sequential scans; a more advanced statistical-based query optimizer; distributed transactions; a process manager that maintains concurrent client requests across a network and ensures clients have permissions; and shared utilities, such as configuring the amount of resources

SimpleDB can use, permissions, and replica creation, so that the database system can handle multi-tenant requests.