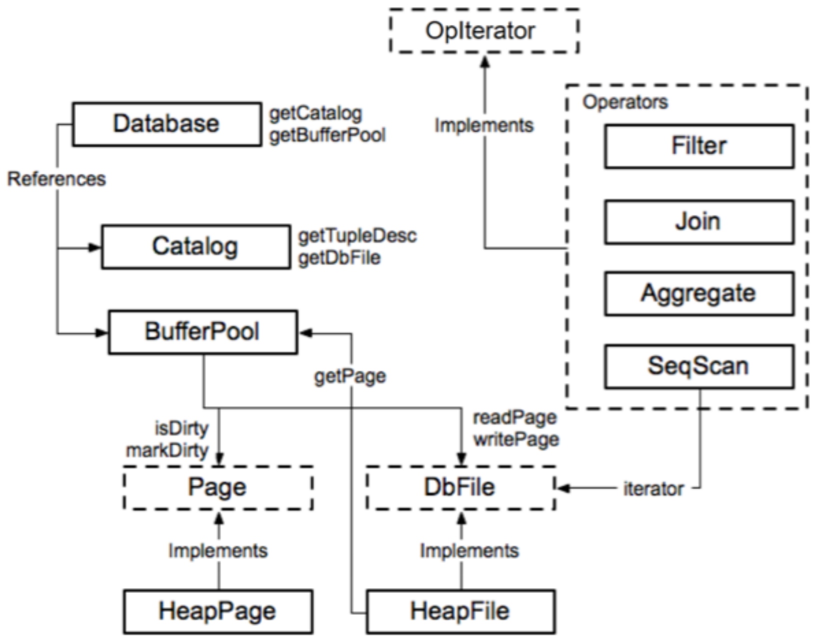
Final report

Name: Yuer Jiang
UW ID: 1771830

Section 1

SimpleDB is a relational database management system which realizes basic functions that a standard DBMS should have. It contains several basic classes including Database, Catalog, BufferPool, HeapFile, operators like Filter, Join, SeqScan and Aggregate. Besides, it also supports recovery from crash, locking, parallel running. Database is the overall schema that provides the reference to major classes, like Catalog and BufferPool. The Catalog class is a directory for references to all the database files. The BufferPool is a temporary memory to store pages that are requested by operators to read in. The HeapFile help support operation to read HeapPages from files as well as modifications on the files. All the operators implement OpIterator interface. They all have standard methods that an iterator should contain. These operators take in certain tuples from iterator and output the results after certain operators also as the form of iterator. In lab3 we implemented lock using a LockManager class written in BufferPool. It is a page level lock that enable us to set and release pages' locks when a transaction has access to certain pages and end up using those pages. Also, we do deadlock detection and throw exception when a deadlock is detected. We also support rollback and recovery by using LogFile. When a transaction aborts, we run rollback method to undo all the updates made by that transaction. When a crash happens, we use the written logfile to recover to the state right before the crash. By reading the logfile, we can redo all the transactions that have completed and undo all the transactions that are not committed.



(From section 1 slides)

Buffer manager

The BufferPool class realize the function for buffer manager. Its main job is to allocate temporary space for storing pages. All the pages read must be first stored into buffer pool if not existing and then read from the buffer pool. By doing this, we can achieve less cost of I/O from disk by reading pages already stored in buffer pool fast. It also provides a platform to perform locking. By involving LockManager class which will be discussed later, it realizes page level locking when a transaction requests access to certain pages in getPage method. To make this class more functional, it also implements some other methods like transactionComplete, flushPage, evictPage, insertTuple and deleteTuple. In details, transactionComplete releases related locks and commits a transaction. Here we use no force policy on this method. FlushPage method help write dirty page to disk. evictPage is called when no more empty memory can be used to store new page and is used to evict some pages stored in buffer pool. Here we use steal policy on this method. insertTuple and deleteTuple help do modification on pages.

Operators

Operators include aggregate, filter, insert, join and SeqScan. All of them implements OpIterator interface and have characters of iterators, i.e., hasNext, next, open, close, etc. Since each of them works different, we need to rewrite some of those methods to override the old ones. All of them, except for SeqScan, take in tuples and do certain operators on them. SeqScan, specially, iterator over all the tuples of given table.

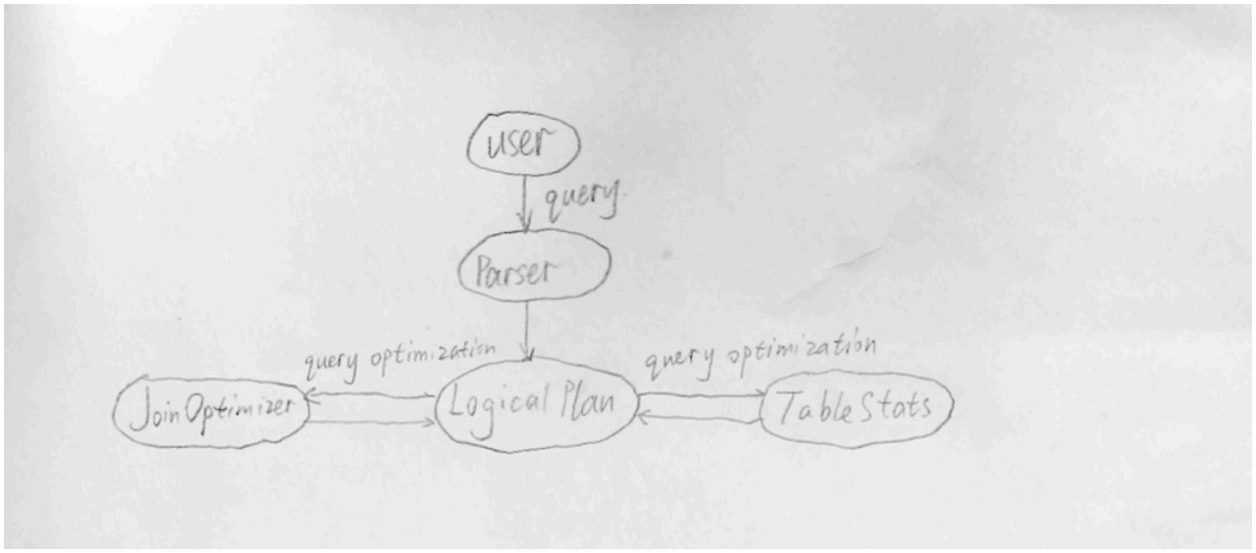
Lock manager

Lock manager is mainly based on LockManager class written in BufferPool class. As mentioned above, the main function of it is to be used in buffer manager to do page level locking. When a transaction get access to a page, it locks that page to that transaction. And if the page is already locked, the transaction has to wait until the lock is released and then proceed. When a transaction is completed/committed or aborted, it will release all the locks related to this transaction. Also, the LockManager supports deadlock detection. In details, for every transaction that is waiting for a lock to be released, do deadlock detection every time it tries to get the lock. If there is a cycle detected among those waiting transactions, it will throw an exception.

Log manager

The logfile class is mainly responsible for log manager. By maintaining a log file, it helps do rollback when a transaction is aborted and recovery when a crash happens. Whenever a transaction updates, commits, aborts or even performs rollback and recovery, the log manager needs to write the process into log file with shortened information. In my implementation, every entry needs to records both the value before write and the value after write to support redo/undo.

Section 2



High level design

The SimpleDB system uses a set of processes to deal with the queries input by users. After the processes, the system will come up with the best physical plan for the input query and use that plan to excute. The function of the query optimizer is to deal with the set of processes to get the best query plan. First it reads in the query and parse it into subqueries that are readable by the system. Then it will calculate costs and cardinalities for each subplan and combine it to the final whole plan. For now, the system can only support selection optimization and join optimization. For join optimization, the system uses Selinger Optimizer to acquire the best join plan.

Exercise 1 description is in write up text file.

IntHistogram

This is a helper class storing some information and maintaining some function for TableStats class to calculate and estimate selectivity. In details, it is specifically designed for Integer type field, while the StringHistogram is designed for String type field. For this class, it is like a histogram that distribute different values into different buckets. Also, it contains the boundary for the histogram, i.e., the maximum value and the minimum value. Besides, it supports to add new values into the histogram with addValue method, which will be called in TableStats to add all the corresponding field's value into the histogram. Then it will estimate selectivity with different formulas based on different operators.

TableStats

This class is used to calculate data statistics related to optimization. There are three components that are needed to be calculated in this class, i.e., selectivity, scan cost and table cardinality. To build this object, it requires input of table ID and I/O cost per page so that it could find the related table and calculate I/O cost of the whole table. Estimating scan cost and table cardinality is simple with direct formulas, and estimating selectivity is more complex that it needs to first figure out the type which is either Integer or String and then builds IntHistogram or StringHistogram with given data. Then it is able to use these two objects' method to calculate selectivities. With the estimated data, the system is then able to optimize by choosing less cost plan.

JoinOptimizer

This class consists two parts finished by two exercises. The first part is to estimate join cost and join cardinality with given data, which is similar to TableStats. The difference is that the latter one computes selection's while this class computes join's. This part is simple to implement since all the formulas have been given and the only thing I need to do is to consider different situations and apply corresponding formulas.

After finishing the first part, here comes the second part which aims to get the sequence of join order corresponding to the best join plan and this will make use of the two methods implemented in part 1. Here we are asked to comply with the Selinger Optimizer, the core of which is to compute best plans for every subset plan, i.e., start from length 1 to the length of the sequence and finally come up with the best plan for the joins. Here we use enumerateSubsets method to generate every set of subplan with given size, which is not very efficient for now. And in extension, I chose the second one to implement which will then improve this method and make it more efficient. And I will introduce this extension in the next part.

Extension

Remark: Test is added in JoinOptimizerTest which is named bigOrderJoinsTest20 which performs 20 joins.

For extension, I chose the second one in 2.5 to improve enumerateSubsets to make it more efficient. In order to keep the original one, I didn't modify this method and write two new methods instead and name them subsetsIterator and BitsetIterator separately.

BitsetIterator

This method performs a iteration over bit changes of a list of 0, 1 numbers. For example, when we are given total size of 9 and subsize of 4, we first create a list of number: 111100000. Then after one call of next, it becomes 111010000, then 111001000, etc. Finally, it becomes 000001111 and then return null. This class is a helper class for subsetsIterator to transform such bit numbers into corresponding sets of join nodes.

subsetsIterator

By iterating over the BitsetIterator, every time we call next, we get a bit number and transform it into a set of nodes and return that set. To be specific, each number in the list of bit number corresponds to one node sequentially. If current number is 1, that means we need to add relating node to set. On the contrary, if current number is 0, we don't need to add relating node. After iterating over every component in BitsetIterator, we get all the subsets we need with given subset size.

Evaluation

The main difference between the original method and the modified method is that the original methods create all those sets at once which could use fairly much memory while the modified one create one set at a time, which greatly improves efficiency especially when the number join members becomes large. To give some examples, in the JoinOptimizerTest, if we choose the original enumerateSubsets, the runtime for bigOrderJoinsTest is 6.425s, and the runtime for bigOrderJoinsTest20 is time runout, which performs 20 joins. However, the runtime for modified method is 0.377s and 7.088s relatively. In this regard, the runtime become much smaller.

Section 3

The overall performance of my SimpleDB is not bad, since it meets all kinds of requirements by each lab. But surely there are many places I can improve. For example, when implementing join class, I choose to use a simple nested loop join which is quite inefficient. If time allows, I would try to perform hash-join algorithm and index-based join algorithm in the future to improve efficiency. Also, I only implemented one extension, there are some others mentioned that could be improved in the future.

If I had more time, I might try to explore more on parallel running. Since this class doesn't have very strict requirements on multi-threads performing, and this is also something I am confused about all the time, I believe it is important for me to have a better understanding for such knowledge and that's quite essential for my understanding towards the CS field.