CSCE 608 Project Report

Mian Qin
UIN: 725006574
Songyuan Tang
UIN: 226002535

ABSTRACTION

This project implements a mini database manager system to design and implement a simple Tiny-SQL interpreter. After obtaining every SQL query, the system interprets the query and executes the query based on Tiny-SQL Grammar. And the system is constructed by a combination of Disk and MainMemory. An efficient algorithm is designed to process SQL queries that passes it to a parse tree and then execute the parse tree with different modules depending on the type of query. And the system can also provide with Disk I/Os and execution time for every query.

IMPLEMENTATION

This project is implemented in C++. The system takes queries into parse tree, then executes one query by one with its different modules depending on whether the type is Create Table Statement, Drop Table Statement, Delete Statement or Select Statement. After execution the query, the system provides a standard output.

1. Software Architecture

In the main.cpp, we first construct a Database Manager with Disk and MainMemory objects provided in the StorageManager library, which are used through the whole system to store tuples and process tuples. And we also construct SchemaManager to get the relation and the schema of one relation.

The system obtains one query once a line and passes it into the Database Manager, then splits the query into token strings using Tokenizer and passes into a parse tree by the Parser in order to execute the query depending on the type of the parse, such as Create Table, Drop Table, Delete or Select, by searching for the key word. The Database Manager can obtain standard output to determine whether the query is process successfully or not.

2. Tokenizer (Parser.h)

The Tokenizer splits the input queries into token strings in order to construct a parse tree for the next step. The Tokenizer get the input of SQL query line by line, and breaks the input query statement on spaces, commas, braces, quotes and other symbols, which returns token strings in a vector of strings without space. Notes that even if some tokens are stored as strings, but are actually integers, we change the type of these tokens from string to integer.

3. Parser (Parser.h)

The Parser takes advantage of token strings created by Tokenizer to construct a Parse Tree. After a Parse Tree is constructed, it helps the Database Manager system process the query and read or write in the Disk.

The Parser follows the TinySQL grammar provided by appendix of project description. We first define a **ParseTreeNode** class as a basic structure to store its type, value and siblings for every node in the Parse Tree. Then we define a **ParseTree** class on the basis of Parse Tree Node, the Parse Tree is constructed by a single Parse Tree Node. And some Parse Tree Node is connected recursively that is stored in the parent node. For example, the query is to create a new table. The type of root in the Parse Tree is CREATE_TABLE_STATEMENT, which the tokens will pass to create parse tree function to construct a Parse Tree.

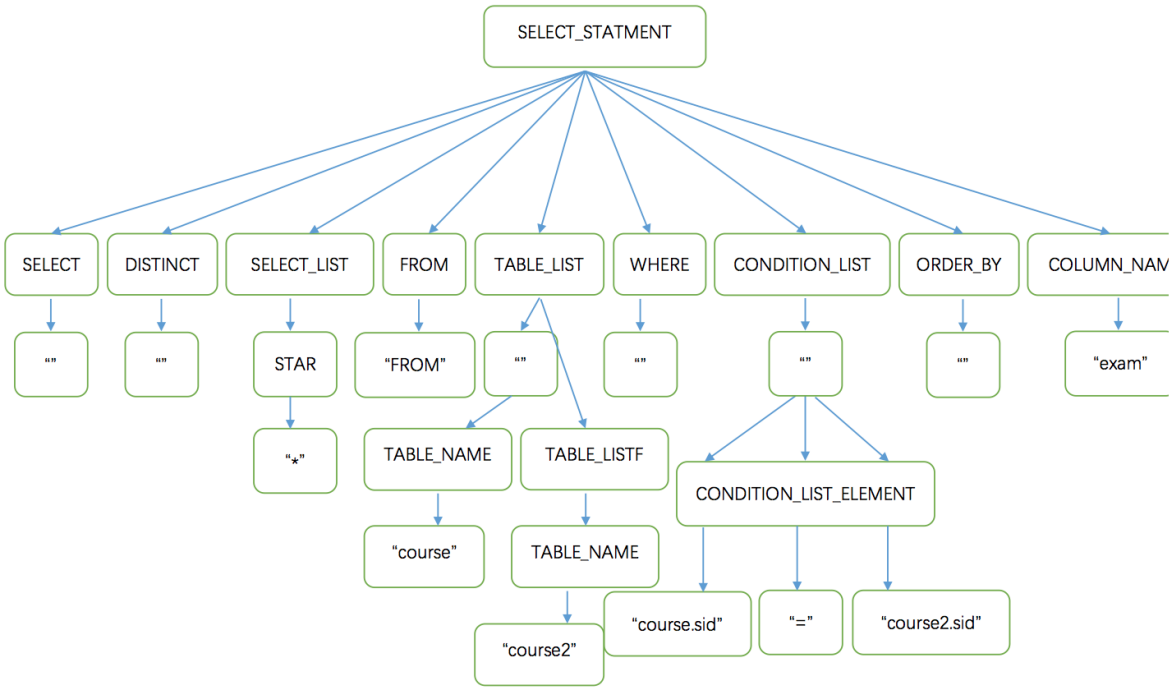
If there is a where clause in the query, it is converted into a postfix expression by using a stach and a priority order decided by the TinySQL grammar. The operators as defined by the TinySQL grammar are +, -, *, /, <, >, =, AND, OR, NOT. And the priority order of above operators is (*, /) greater than (+, -) greater than (<, >, =) greater than (NOT) greater than (AND) greater than (OR).

We create **ConditionParser** class to obtain the appropriate order to deal with the condition element strings. The main algorithm is as follows:

a. If the token is not an operator or a bracket, the token is directly pushed into the back of the postfix expression.
b. If the token is a left bracket, the token is pushed in the stack.
c. If the token is a right bracket, pop the top token from the stack and push in the postfix expression until the stack is empty or left bracket.
d. If the token is an operator, pop the stack until it is empty, or the top token in the stack is a left bracket, or the top one in the stack has a higher priority than this. And the popped tokens are pushed into the postfix expression at the same time.
e. Pop all tokens until the stack is empty and push all popped tokens into a postfix expression.

To better illustrate the structure of parse tree, a parse tree based on an example query is shown below. The example query is **SELECT * FROM course, course2 WHERE course.sid = course2.sid ORDER BY exam**.

We learn from the picture that the SELECT_STATEMENT is the type of root, which is used to further process. Every tokens obtained from Tokenizer can be formed to a Parse Tree Node as a sibling of parent root node. For every Parse Tree Node, there is a value according to its type. And some Parse Tree Node uses recursive way to store more tokens, such as TABLE_LIST.



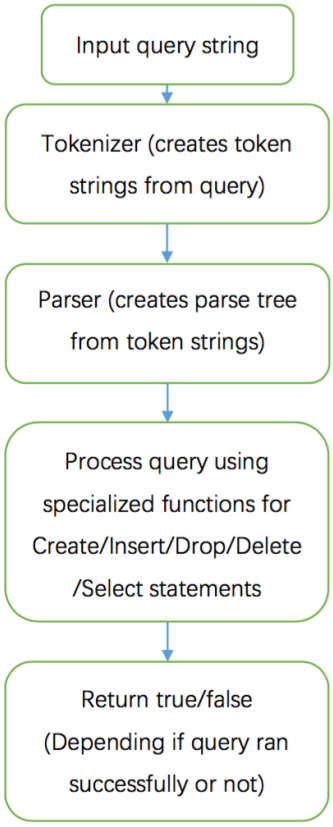
4. Database Manager (DatabaseManager.h)

The Database Manager is the core part of the system. It is initialized with a Disk and MainMemory objects. Disk is used for storage where each relation resides on a single track of blocks. Main memory stores tuples that need to be operated.

The Database Manager provides **ProcessQuery** function to process the query. The function is used to process a query input to a vector of token strings first, then a Parse Tree is generated from token strings with the help of the Parser. And it calls an according query process

module next depending on the type of the input query. At last, the function returns a true or false value indicating that the query was executed successfully or not.

The work flow of the Database Manager processing a query is shown below.



5. Create Table Statement

After obtaining a Parse Tree, we can retrieve all data from the tree. When the type of root equals to CREATE_TABLE_STATEMENT, a vector of table name, attribute name and attribute type list will be retrieved by recursively traversing the parse tree. Table name means the name of relations, and a new schema is created via attribute names and types. A new relation is created by SchemaManager.

The output in the console is shown below.

```
> CREATE TABLE course (sid INT, homework INT, project INT, exam INT, grade STR20)
********** Statistics **********
*  Disk I/O:         0         *
*  Execution Time:   0         *
*******************************
SUCCESS
```

6. Drop Table Statement

To execute a Drop Table Statement, the name of the table is retrieved from the parse tree and then the relation is delete using the SchemaManager from the StorageManager library.
The output in the console is shown below.

```
> DROP TABLE course
********** Statistics **********
*  Disk I/O:         0         *
*  Execution Time:   0         *
*******************************
SUCCESS
```

7. Insert Statement

The Insert statement can either have values explicitly mentioned using the VALUES keyword or there can be a select subquery which gives the values to be inserted into the table.

At first, we retrieve schema from the relation that means a vector of attribute list is created that are the attributes in the order it is written in the query. If there is a keyword VALUES in the query, a vector of value list is created, a tuple of these values is obtained from the parse tree. Note that since we record type of every value to be INT or STR20, we confirm each type of the value before it is inserted into a tuple.

After creating a tuple, we need to insert it into Relation. There are three situations for the insertion:

a. When the relation is empty, we obtain an empty block from MainMemory and write the block back to at the beginning of Disk.
b. When the relation is not empty, but the last block is full, we obtain an empty block from MainMemory and append the block to the end of Disk.
c. When the relation is not empty and the last block is not full, we obtain the last block in Disk to MainMemory and append a tuple to the end of block, besides, write the block into

original block.

The output in the console is shown below.

```
> INSERT INTO course (sid, homework, project, exam, grade) VALUES (1, 99, 100, 100, "A")
********** Statistics **********
*  Disk I/O:         1         *
*  Execution Time:   74.63     *
********************************
SUCCESS
```

8. Condition Parser

When there is a keyword WHERE in the clause, the **ConditionParser** class is created to process the where clause. And WHERE clause is only applied in Delete statement and Select statement, so we discuss two different situations in these two statements.

The Condition Parser object is initialized by giving a postfix expression root node and a relation as input. The postfix expression is built by the parser and stored as a single node with all the postfix expression nodes as its children. Condition Parser uses this postfix tree to create another internal postfix expression by specifying which tokens in the postfix expression are operators, variables and constants. During the initialization step Condition Parser does the following checks:

a. If the token in the postfix expression is an operator.
b. If a possible variable is a column inside the relation schema. If so it is marked as a variable and its field type is stored.
c. Otherwise the string is assumed to be a constant, which could be either integer or string type.

The input token passes into Condition Parser with the help of a stack to obtain a ordered vector strings. The algorithm evaluates the where postfix expression for a vector of input tokens that is based on the priority order for operators, columns and constant. And at last, a Boolean value is returned depending on whether the function is executed successfully.

An example is needed for better illustration. At first, we obtain the sub-parse tree of query which is the next sibling of “WHERE” parse tree node in the parse tree. Then we split it into a vector of strings and reorder it based on priority order. For example, "(homework + project)

= 100" in the query after WHERE, we acquire a vector of tokens like “homework”, “+”, “project”, “=” and “100”. A new vector is created based on priority order, which is “homework”, “project”, “+”, “100” and “=”.

9. Delete Statement

If there is a DELETE in the clause, it means tuples need to be deleted. The delete statement may have a WHERE clause to check which tuples need to be deleted, and it may not have a WHERE clause which means all tuples in the relation need to be removed.

If we just simply delete a tuple from the disk block, it creates holes in the disk blocks. Our algorithm deletes tuples intelligently such that the resulting relation doesn’t have any holes in the blocks. The algorithm works by keeping two pointers, one pointer points to the index of tuple in the block where it needs to be removed, the other pointer points to the last index of tuple in the last block. When the tuple pointed by the first pointer is deleted that it will create a “hole”, the tuple in the last block will be moved to the “hole”.

This algorithm efficiently deletes the tuples which match the WHERE clause or deletes all tuples if there is no WHERE clause. When there is no WHERE clause not a single read is done from the disk. When there is WHERE clause, **optimization** done at this step is moving the last tuple in the last block to the position where the tuple is deleted, which will not create a hole for every block.

The output in the console is shown below, including delete information and display of the remaining relation.

```
> DELETE FROM course WHERE grade = "E"

********** Statistics **********
*  Disk I/O:          12        *
*  Execution Time:    895.56    *
********************************
SUCCESS
> SELECT * FROM course

-----------------------------------------
sid      homework project  exam     grade
-----------------------------------------
1        99       100      100      A
1        99       100      100      A
-----------------------------------------
********** Statistics **********
*  Disk I/O:          6         *
*  Execution Time:    447.78    *
********************************
SUCCESS
```

10. Select Statement

To process the select statement three vectors are created: table_list, select_list and condition_list. The table_list is the list of all the tables mentioned in the query. The select_list is the list of all the attribute names that are to be projected by the query. The condition_list is the list containing all the attribute names from the select_list, all the attribute names that are in the where clause and the attribute name in order by. The select statement can have a single table select statement or it can have multiple tables in the table_list. There is different when table_list only contains one table or multiple tables.

10.1 Single Table Select

If the select statement has only one table in the table_list, single table select applies. If the select_list has any entries of the format **<table-name>.<column-name>** then it is converted to only column name. Then it works on the table using the Condition Parser if applicable. If there are no DISTINCT and ORDER BY keywords present in the query, read a block in one time from Disk to Memory and print the tuple one by one from Memory. If there is any of the DISTINCT or ORDER BY keywords, it makes a temporary relation and applies the corresponding algorithm on it.

10.2 Multiple Table Select

If the select statement contains more than one tables in the table_list, multiple table select applies. Cross join is applied if number of tables are more than or equal two. The above mentioned optimization makes a relation_list, a pair of tables is selected in the order of the relation_list and a cross product is created using the push where conditions optimization, it creates a temporary relation and then next table is taken for cross product with this relation. This kind of cross product chain goes on until no other table is left in the relation_list. For optimization, we also apply natural join to make two table select. The final temporary relation is then returned to the DISTINCT or ORDER BY if required. The multiple table select can be one pass or two pass depending on the size of the temporary relations that gets created.

10.2.1 Cross Join

We apply cross join to join multiple tables in the order of relation_list, which the first two tables join, then the third one, the forth, etc. joins. The cross join is done by sorting the relations in increasing order based on their size. This way the temporary intermediate relation is smaller in size and can be accommodated in the main memory or one pass algorithms can be applied in later stages of the query execution.

10.2.3 Theta Join

Theta join is similar with cross join, which is also done by sorting the relations in increasing order based on their size. We apply theta join when there is a WHERE clause.

10.2.3 Natural Join

A natural join compares all columns of two tables which have the same column name and the resulting joined table contains those columns once which are same in name in both the tables. The cross join is done by sorting the relations in increasing order based on their size. This way the temporary intermediate relation is smaller in size and can be accommodated in the main memory or one pass algorithms can be applied in later stages of the query execution.

10.2.3.1 One Pass Algorithm

If any one of the relations is small enough to fit in main memory, the system applies one pass algorithm where the small relation is in main memory and the large relation is iterated block by block and crossed with the tuples of small relation to output it to the temporary relation that needs to be returned to the calling function. Here also the optimization is applied. The main algorithm is as follows:

a. Read a block of relation1 once, which is the smaller relation than relation2, from Disk to Memory, then sort the tuples in the block based on the column name provided.
b. Read a block of relation2 once from Disk to Memory, then sort the tuples in the block based on the column name provided. Write this block back to Disk.
c. Repeat b. until all blocks in the Disk is sorted.
d. Read smallest tuple from relaton2 from Disk to Memory, join tuples with same value.

10.2.3.2 Two Pass Algorithm

If none of the relations is small enough to fit in main memory, the system applies two pass algorithm. It tries to bring as many tuples from small relation into the main memory and then apply one pass on those tuples crossing it with the large relation. After one loop it again tries to bring as many tuples into the main memory as it can and keeps repeating until all the tuples from small relation are done. Here also it applies the optimization. The main algorithm is as follows:

e. Read a block of relation1 once from Disk to Memory, then sort the tuples in the block based on the column name provided. Write this block back to Disk.
f. Repeat a. until all blocks in the Disk is sorted.
g. Read a block of relation2 once from Disk to Memory, then sort the tuples in the block based on the column name provided. Write this block back to Disk.
h. Repeat c. until all blocks in the Disk is sorted.
i. Read some smallest tuples from relation1 and relaton2 from Disk to Memory, join tuples with same value.

10.3 Order By

Order by is basically sorting according to a particular column. The sort can be done in two ways, one pass or two pass depending upon the size of the relation.

The output in the console is shown below.

```
> SELECT * FROM course ORDER BY exam
-----------------------------------------
sid      homework project  exam     grade
-----------------------------------------
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
16       0        0        0        E
```

10.3.1 One Pass Sorting

The one pass sorting can be done if the number of tuples in the relation is less than what can be accommodated in main memory. This algorithm takes in a vector of main memory block indices and applies sort algorithm on the tuples in these blocks, the comparison is done based on the column name provided. After the execution of this algorithm the tuples in these main memory blocks are sorted.

10.3.2 Two Pass Sorting

The two pass sorting is required when the number of tuples in the relation is more than what can be accommodated in main memory. The two pass sorting algorithm is as follows:

j. Read a block once from Disk to Memory, then sort the tuples in the block based on the column name provided. Write this block back to Disk.

k. Repeat a. until all blocks in the Disk is sorted.

l. Use pointers pointed to the beginning of every block in the Disk and apply Merge Sort to find out the minimal value among tuples of all blocks. Print the minimal value tuple and move the pointer to the next tuple.

m. Repeat c. until all tuples in the Disk has been printed.

10.4 DISTINCT

Distinct is basically sorting according to a random column and then remove duplicates. This can be done in two ways, one pass or two pass depending upon the size of the relation.

The output in the console is shown below.

```
> SELECT DISTINCT * FROM course
-----------------------------------------
sid      homework project  exam     grade
-----------------------------------------
10       50       70       70       C
11       50       50       59       D
12       0        70       58       C
13       0        50       77       C
14       50       50       56       D
15       100      50       90       E
15       100      99       100      E
16       0        0        0        E
17       100      100      100      A
1        99       100      100      A
2        100      100      99       B
3        100      100      98       C
3        100      69       64       C
4        100      100      97       D
5        100      100      66       A
6        100      100      65       B
7        100      50       73       C
8        50       50       62       C
9        50       50       61       D
-----------------------------------------
********** Statistics **********
*  Disk I/O:         174       *
*  Execution Time:   12432.9   *
********************************
SUCCESS
```

10.4.1 One Pass Distinct

The one pass distinct can be done if the number of tuples in the relation is less than what can be accommodated in main memory. This algorithm takes in a vector of main memory block indices and first of all applies sort algorithm on the tuples in these blocks and stores the sorted tuples into a new relation, the comparison is done based on the column chosen from select_list. It keeps printing out the tuples with different values of one column name, if two tuples with same value of column name, it continues until find a different one.

10.4.2 Two Pass Distinct

The two pass distinct is required when the number of tuples in the relation is more than what can be accommodated in main memory. Like one pass algorithm for duplicate removal, here also a set of the string representation of tuple is maintained to keep track of the seen tuples. At first, two pass sorting is required for distinct. A temporary relation, temp_table, will be stored all tuples sort from Disk with the help of Memory. Print distinct tuples in the new relation, by continuing the same value tuple and printing the tuple with different values.

EXPERIMENTS RESULTS

1. Single table query

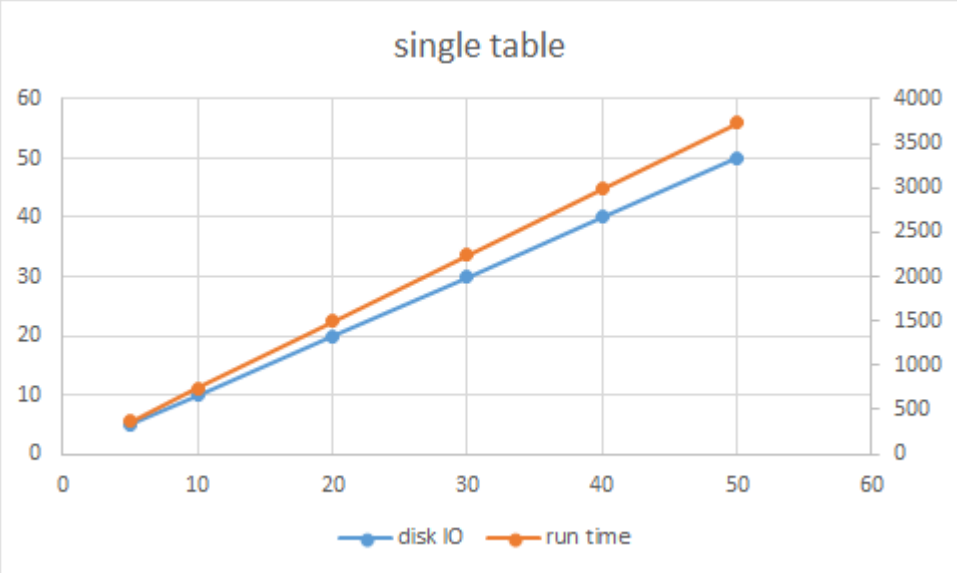
For performance evaluation, we have done two set of experiments. First set is for single table queries, which include simple table scan, with condition, duplication removal, sorting and duplication removal with sorting. Those are major operations for single table queries and can generally show performance of single table queries.

For five individual experiment below, the x-axis is block size of the table, two y-axis are disk IOs and run time of the query.

For test script, please check report_single_table.txt, it contains all the table and queries information.

a. Simple Query (Table Scan)

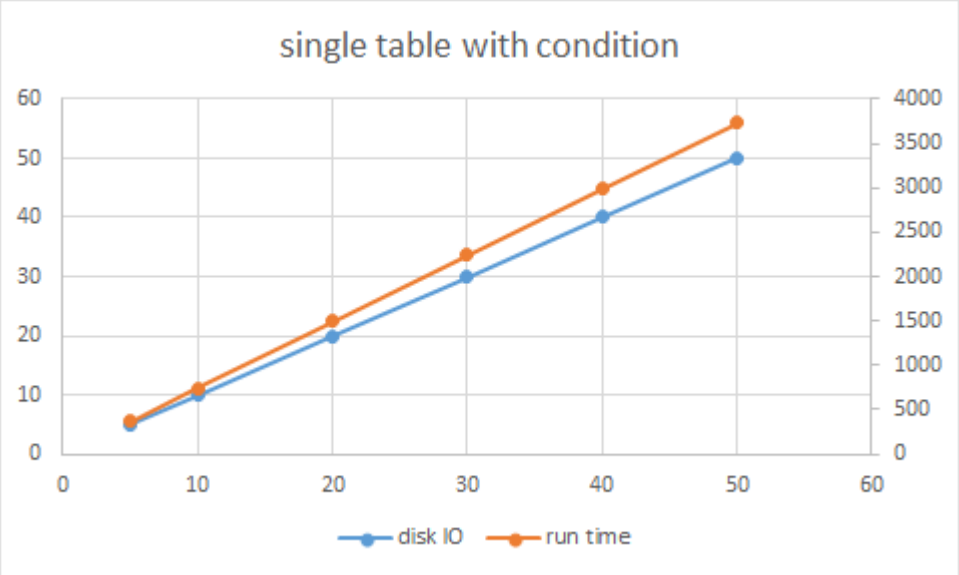
SELECT * FROM course



For simple table scan as you can see from the result, the disk IOs are equal to table block size.

b. Single Table Query With Condition

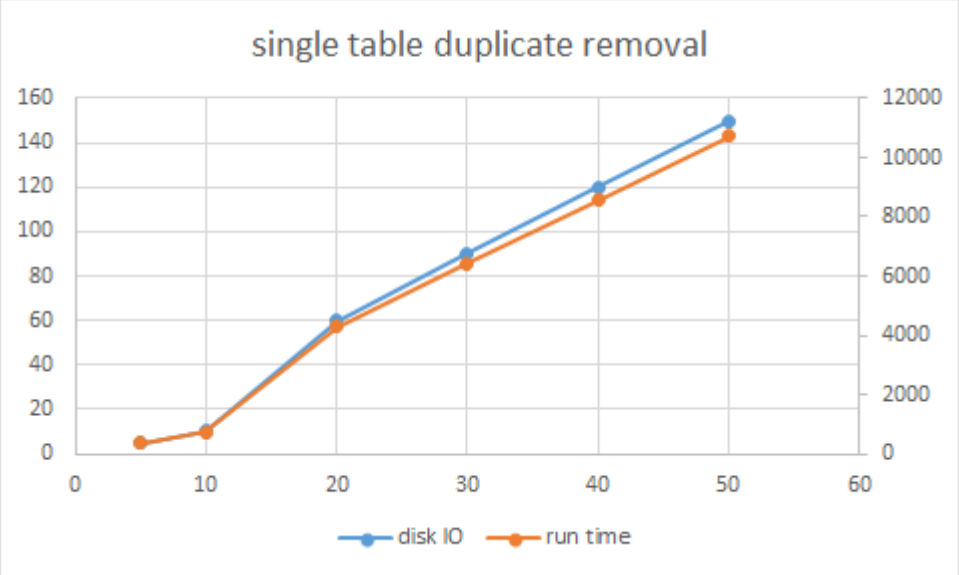
SELECT * FROM course WHERE grade = "C" AND [exam > 70 OR project > 70] AND NOT (exam * 30 + homework * 20 + project * 50) / 100 < 60



For simple table scan with condition as you can see from the result, the disk IOs are the same as simple table scan which is obvious since the condition evaluation are down in memory.

c. Single Table Query with Duplication Removal

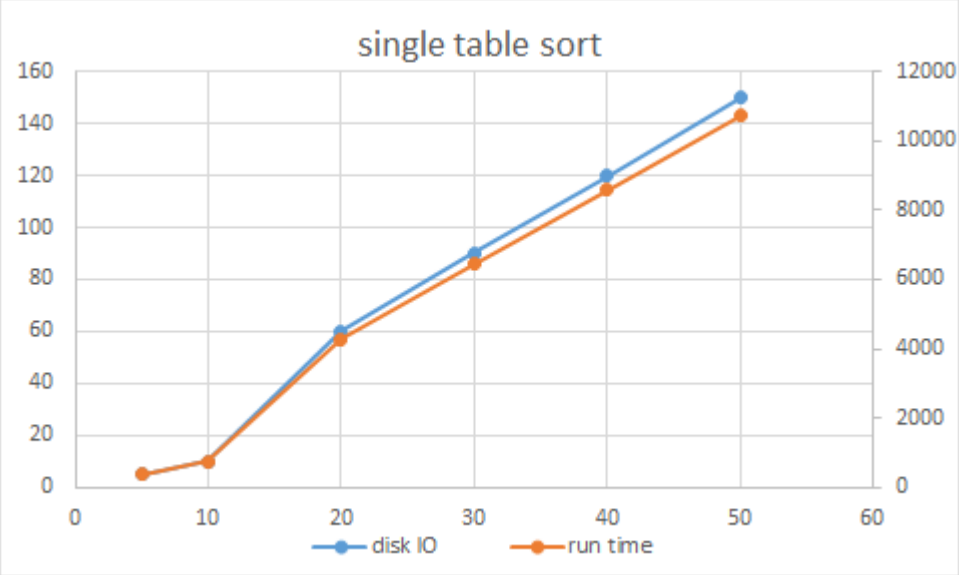
SELECT DISTINCT * FROM course



For duplication removal it starts to become different. Here we implement two kinds of algorithms for duplication removal, i.e. onepass and two pass. If the table can fit in the main memory (10 blocks), we perform the onepass algorithm. If the table cannot fit in the main memory, we perform the twopass algorithm. As you can see from the results, the first two experiments which table size is 5 and 10 blocks, onepass algorithm is performed, so the disk IO is the same as the table block size. However, for the rest of the experiments where twopass algorithm is performed, the disk IO is 3*B(table).

d. Single Table Query with Sort

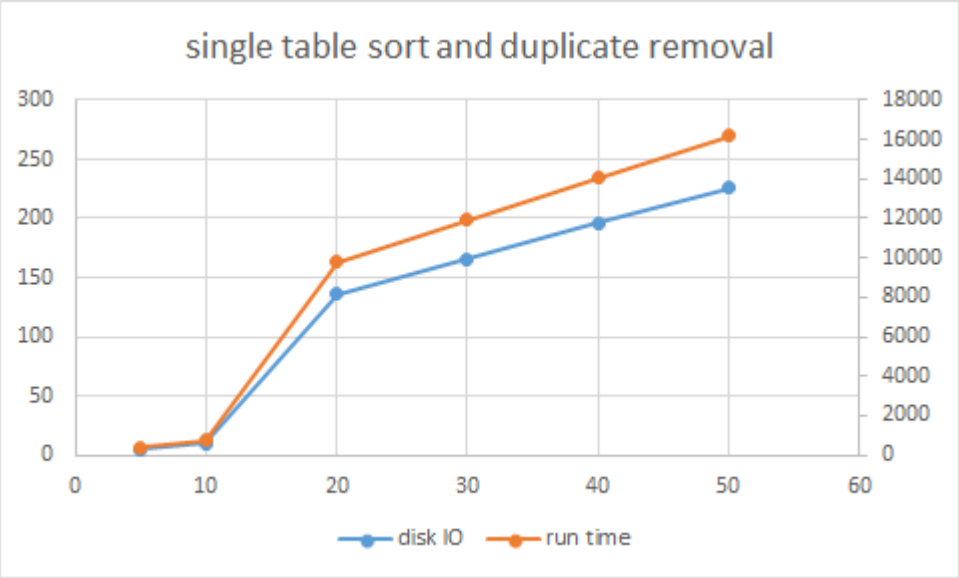
SELECT * FROM course ORDER BY exam



For sorting algorithm for single table, we also implemented two type of algorithms, onepass sorting and twopass sorting. The results is similar to the duplication removal, we don't explain again here.

e. Single Table Query with Duplication Removal and Sort

SELECT DISTINCT * FROM course ORDER BY exam



For duplication removal with sorting operation, it's actually more complicated. Same as above, when table can fit in the main memory, we still only need to read the table once and do the sort with duplication removal in memory. However, for table that cannot fit in main memory, we need to perform twopass algorithm for sorting and duplication removal twice, since duplication removal attributes and sorting attributes are not the same. What we did is first do twopass duplication removal and then save the results into a temporary table and read again to perform sorting for the sorting attribute. So the disk IO depends on the table size after duplication removal which reflect on the results.

2. Multi-table Join Query

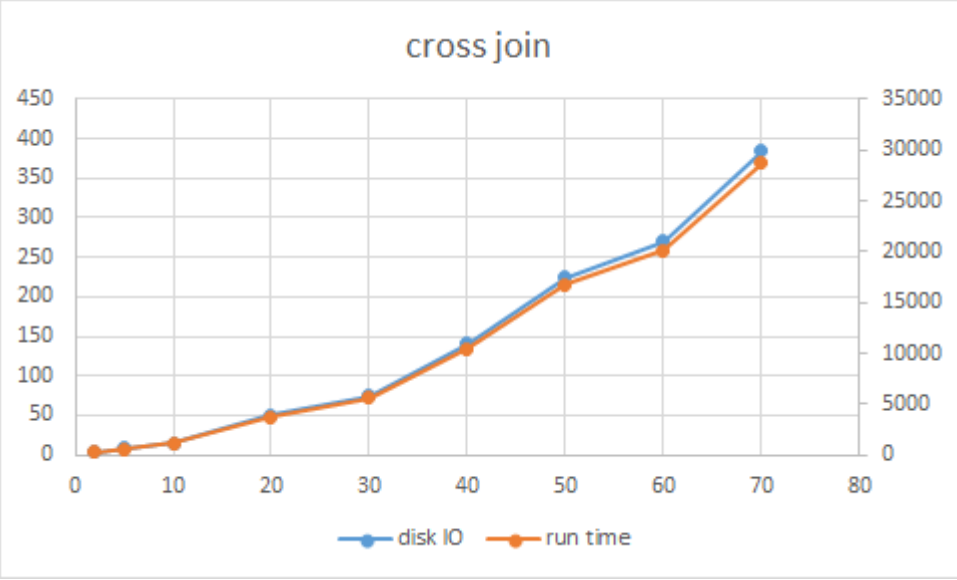
Second set of experiments are join operation which are major operations in database queries. Here we conduct three kind of join operations cross join, natural join and theta join for two tables. For multiple tables the performance should be similar, see the discussion of multiple table join optimization in the next section.

Noted here in all the figures below, x-axis is tuple size of each table (two tables have same number of tuples, but tuple size are different as blow). Besides, for natural join, due to the limitation of twopass algorithm, the last row configuration of experiment is not conduct. Detailed table information please check report_join.txt.

table1 tuple size	table2 block size	table2 tuple size	table2 block size
2	2	2	1
5	5	5	3
10	10	10	5
20	20	20	10
30	30	30	15
40	40	40	20
50	50	50	25
60	60	60	30
70	70	70	35

a. Cross Join

SELECT * FROM course, course2



For cross join, the operation complexity (in case of disk IO) is fixed as (B(T1)*B(T2))/M + B(T1), M is the main memory blocks. So, the implementation of cross join is nesty two loops. As you can see from the result, the disk IOs can reflect the implementation of the algorithm. However, why the plot is not a straight line? That is because the table size (in case of block) is not exact multiple times of the main memory block size (here is 8, since we need a block for read in table 2 and another block for output).

b. Natural Join

SELECT * FROM course, course2 WHERE course.sid = course2.sid

For natural join is actually much more complicated than cross join. Here we implement two algorithms for natural join, i.e. onepass and twopass natural join. When the smaller table can fit in main memory, we first read smaller table into memory and read larger table block by block to match tuples in memory as onepass algorithm. If the smaller table cannot fit in main memory, we need to perform two pass algorithm by presort the two table first to multiple sorted segments (as long as 10 blocks) and the read each segments by one block and get the smallest tuple of the two table and do natural join. The onepass algorithm disk IOs is B(T1)+B(T2) and twopass algorithm disk IO is 3(B(T1)+B(T2)) which can be reflected by the experiment results.

c. Theta Join

SELECT * FROM course, course2 WHERE [course.exam > course2.exam OR course.grade = "A" AND course2.grade = "A"]

For theta join, the implementation is pretty much same as cross join by with condition evaluation, so the performance is basically the same with cross join.

OPTIMIZATION

For optimization, we focused on reducing the number of disk IOs not optimize the in memory algorithms.

1. Delete optimization

For delete query process, we do optimization to remove holes in disk block. As mentioned above, when a tuple is deleted, we will replace with the last tuple in the table (if the last tuple doesn't need to be deleted) and then write back the block and also may need to delete the last block of the relation (if last block is empty). If the last tuple also need to be deleted, we will keep delete the last tuple (and the block) until we find a tuple after the initial deleted tuple that doesn't need to be deleted and replace with it.

The effect of this optimization is we may sacrifice some IOs during the delete (by reading the last block of the table), but we can get rid of the holes in the table. For example, if we use the naive delete algorithm (leaving a hole), after delete all the tuples and then do a select query, you still need to read in all the blocks which is full of holes in to main memory. However, with our enhanced delete algorithm, the disk IOs of select after full delete is 0.

2. Use memory as much as possible

Another optimization we did is use as much memory as possible. This related to the detailed implementation. For example, for cross join, if we only use three blocks, one for read in table1, one for read in table2 and one for output, it is very easy to implement, but disk IO will be much more (B(T1)*B(T2)) + B(T1). In order to get (B(T1)*B(T2))/M + B(T1), we need to use 8 blocks to hold table1, and one block for reading in table 2, and do some in memory operations for join, then output.

For certain operations, use main memory as much as possible can significantly reduce disk IOs, (for the above example, reduced 10 times). However, for some operations like simple tablescan, it doesn't matter.

3. Choose between onepass and twopass

Another optimization we did is wisely choose onepass and twopass algorithm. In general, all the situations can be handled by twopass algorithm, however, the disk IO differs largely, as the experiments results of sort duplication removal and natural join shows. This is a key point for database management system. also, this is the same philosophy of using as much as main memory as possible.

4. Multiple tables join

For multiple table join, we didn't implement the dynamic programming algorithm that optimize the join tree in class. In our implementation, we fixed the join tree, for example if we want to join t1, t2, t3, t4, we will first join t1 and t2, then join t3, then join t4 and output. However, we do a small trick to optimize by always finding a small table in the two table join. For cross join, this can reduce some disk IO, since the number of disk IOs is (B(T1)*B(T2))/M + B(T1), since we always let T1 be the smaller table.

For natural join, this optimization actually benefits more, since the smaller table may fit in the main memory and we can perform onepass natural join. So when we join multiple tables, for every intermediate two table join, we will always find the smaller table and choose the algorithm wisely.

CONCLUSTION

In this project, we learned how a database system is implemented. Besides, we implemented the database management system with different algorithms and even some algorithm to make some optimizations. Since it is only a tiny SQL interpreter, we really do a great amount of job, we cannot image how large a huge database management system could be.

Although a lot of optimization algorithms has been designed in the system, some other functions is still to be finished in the future to make the system more efficient. For example, cross join is quite expensive and the size of cross product can be reduced using optimizations.