# CSCE 608 Project 2

* Team Member

Qianzhen Li 226000842 qzli@tamu.edu

Minrui Wang 525000637 [wangminruitamu@tamu.edu](mailto:wangminruitamu@tamu.edu)

* Language

Java

* Changes to the StorageManager Library

Add compareTo function in Tuple class.

Change one of constructors of Tuple class ,Tuple(Tuple t) from protected to public

# Main Topic

In this project, we design and implement the Tiny-SQL interpreter, which can execute SQL queries with the aid of StorageManager Library. We read the SQL queries sentence from Main class line by line. After that use the Parser class to convert it into parser tree. We also use create, insert, select, delete and drop class in the Main class to execute each kind of statement and output the result.

# Program Flow

## 1. Initiation

First, run Main.java to load in the test file with queries, the Main.java will scan each line in the test file, and use parser.java to parse the query into a parse tree. According to the parsing result, the main class will call Create, Drop, Delete, Select or Drop class to execute corresponding operations. The whole process ends until the main class scan the last line of test file.

## 2. Five Operation Classes

* 1. Create:

According to the parsing result, create corresponding table in the schema manager with right field names and field types

* 1. Drop:

According to the parsing result, drop the corresponding table if existed in the schema manager and erase the track in the disk assigned to this table.

* 1. Insert:

There is two different situations. One is the common situation: insert one tuple a time, and the other is inserting a list of tuples by using select subquery. For the second one, we need to call Selection class first to get the list of tuples needed to be inserted, and then using a loop to insert every tuple in the lists the way like the common situation, i.e. inserting one tuple a time.

* 1. Delete:

According to the parsing result, read the first and last block of corresponding relations into main memory(if there is only one block, then read in the same block but use different pointers referring to it). Using two pointers to record the indexes of the two blocks and another two pointers to record the tuple index in the two blocks. Scan the first block tuple by tuple. If the tuple satisfies the where condition, use the last tuple of last block to replace it and keep the tuple index for first block still and move the tuple index one step backward. If the tuple does not satisfy the where condition, the tuple index for first block one step forward. During the process, once the front block is done, write it back to disk. On the other hand, if the last block is empty because of the replacing, write this empty block to disk and read in another block from the back end of the disk. The process keeps going until the two pointer meets each other. The main purpose is preventing holes in the disk because of delete operation.

* 1. Selection Operation.

We decided to divide the selection query into different categories based on the number of relations involved in this query.

* + 1. One table.

If there is no distinct or order by keywords, no matter how large the relation is, we can read one block a time into memory and print out those tuples satisfying where condition. If there is distinct or order keyword and the relation is smaller enough to fit in the main memory, we can read in the whole relation into memory and sort them according to the distinct or order by keyword to eliminate duplicates and print them in order. If unfortunately, the relation size is larger than the main memory, we can read one blocks a time from the relation and during reading determine whether this tuple satisfy the where condition. If the tuple does not satisfy the where condition, skip this tuple. If the number of tuples still can be fitted into the memory after all the blocks has been read in, we can just use the one-pass algorithm mentioned above for smaller relation. Otherwise, we need to create a temporary relation to hold this sorted sublist of relation, and keep store those sorted sublist in the temporary relation until all blocks of the relation have been read in. Then, read one block from each sublist into a minheap which is constructed base on the distinct keyword or order by item. Each time pop a block with a smallest tuple. If there is only order by keyword, just print out the smallest tuple and update the tuple index of the block and put it back to the minheap. If this block is finished, read in another block from the sublist until the sublist is finished. For distinct keyword, just keep the previous smallest, skipping until the next tuple is different from the previous smallest, and then print it and replace the previous smallest with it.

* + 1. Two tables.

If this is cross product, just using nested loop. For natural join, if smaller relation can be fit into the main memory, read them into the main memory. Then read one block a time from larger relation into memory, join them with all tuples from smaller relation and output those tuples with join attributed of same value. If the even the smaller relation cannot be fit into the main memory, we should use two-pass algorithm here. We can optimize the process here by pushing down the selection to change a large relation into a small relation(detail explanation in below section.) Like one table two-pass algorithm, create two temporary relation to keep sorted sublists for both relations and use two minheap to get the block with smallest tuple of both relations. Through comparing them, we can output those tuples with join attributed of same value. If there is distinct or order by keyword, we need to store the join results in another temporary relation and apply the selection method for one table mentioned above.

* + 1. Multi-table.

Using dynamic programming to determine the join sequence with the minimum intermediate cost of disk I/O.

# Software Architecture

|  |  |
| --- | --- |
| Main Class | |
| - Base base | - Create create\_executor |
| -Main sql | - Drop drop\_executor |
| - File file | - Insert insert |
| - Scanner scanner | - Selection select |
| - Parser parser | - Delete delete |
| - Statement statement |  |

|  |  |
| --- | --- |
| Parser Class | |
| + Statement parse(String s) | + Statement parseSelect(Statement s) |
| + Statement parseCreate(Statement s) | + Statement parseFrom(Statement s) |
| + Statement parseDrop(Statement s) | + Statement parseWhere(Statement s) |
| + Statement parseDelete(Statement s) | + Statement parseOrder(Statement s) |

|  |  |
| --- | --- |
| Base Class | |
| - MainMemory mm | - SchemaManager sm |
| - Disk disk |  |
| Statement Class | |
| - ArrayList<Node> nodes | |
| + ArrayList<Node> gerNode() | |

|  |
| --- |
| Delete Class |
| + void deleteTuple(Statement statement, Tuple deleted, Base base) |

|  |
| --- |
| Create Class |
| +Relation createRelation(Statement statement, Base base) |

|  |
| --- |
| Drop Class |
| + void  dropRelation(Statement statement, Base base) |

|  |
| --- |
| Insert Class |
| + void insertTuple(Statement statement, Base base) |

|  |  |
| --- | --- |
| Selection Class | |
| - ArrayList<Node> select\_items | + ArrayList<Tuple> select(Statement statement, Base base) |
| - ArrayList<Node> from\_items | + ArrayList<Tuple> selectOneOne() |
| - ArrayList<Node> where\_items | + void selectOneTwo() |
| - ArrayList<Node> order\_items | + void selectMultiTable(Base base) |

\* The selectOneOne function deal with one table one pass situation, selectOneTwo function deal with one table two pass situation and selectMultiTable function deal with multiple table situation.

|  |  |
| --- | --- |
| Node Class | |
| - String label | + String getLabel() |
| - ArrayList<Node> children | + void setLabel(String label) |
|  | + ArrayList<Node> getchildren() |
| + void setChildren(ArrayList<Node> children) | |

# Algorithm

## One Pass

When we select tuples from one relation and the table size is smaller than main memory size or one of the table size is smaller than main memory size, and we need to sort in the main memory, thus we can use one pass algorithm to output the result.

To be specifically, when the SQL query has ORDER BY or DISTINCT, we need to sort the tuple in the memory. In addition, DISTINCT keyword ask us to remove the duplicate tuple after sorting in the main memory. In the end, we use conditions to output the result.

For two table, when we meet where condition has equal, we implement the two pass natural join, which means, the size of smaller table is smaller than main memory size, we can read the whole smaller relation in the main memory and read the larger relation block by block, compare the block of larger relation with every block in the smaller relation, output the same tuple.

For multi-table, we use dynamic programming to optimize the join sequence, we will illustrate it in the optimization part.

## Two Pass

Obviously, when the table size or both of the table size are larger than the main memory size, we need use two pass algorithm to solve this problem.

For single table, we divide the relation into several sub-lists which can fit into the main memory. Sorting these sub-lists one by one and write back to disk, read the first block of each sub-lists in disk and then output the smallest one, which implement by min-heap.

For multi-table, the process is similar to single table, however, we need to optimize the process according to where condition, we will also discuss it in the optimization part.

# Optimization Technique

## Selection Operations in the tree

As we mentioned above, for example:

SELECT \* FROM course, course2 WHERE course.sid = course2.sid AND course.exam = 100 AND course2.exam = 100

In this situation, we should first select tuples from course which exam = 100 and select tuples from course2 which exam = 100. By applying selection first, we can reduce the number of tuples need to be joined, effectively reducing the disk I/O.

## Join Operations

For multi-table, the sequence of join will also have influence to the cost. We run DP algorithm to determine the order of join. For example:

SELECT \* FROM t1, t2, t3, t4, t5, t6

In this situation, we need to use DP to determine the order of cross join.

## Fixed the black hole problem

When delete tuple in block, we first read the first block and the last block from disk to main memory, scan the first block in main memory and delete the target tuple. Now, we out the last tuple of last block into that target tuple position. When the scanning is done, write the first block back to disk. Then read the next block. What’s more, when the last block in the main memory is empty, write it back to disk and delete it using deleteBlocks function, then read the second last to main memory.

In this way, we will avoid black hole in each block and also avoid emptys block existing between blocks.

# Experiments and Results

## Simple Operation

CREATE TABLE course (sid INT, homework INT, project INT, exam INT, grade STR20)

INSERT INTO course (sid, homework, project, exam, grade) VALUES (1, 99, 100, 100, "A")

INSERT INTO course (sid, homework, project, exam, grade) VALUES (2, NULL, 100, 100, "E")

INSERT INTO course (sid, grade, exam, project, homework) VALUES (3, "E", 100, 100, 100)

SELECT \* FROM course

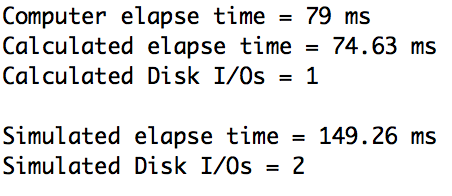
DELETE FROM course WHERE grade = "E"

INSERT INTO course (sid, homework, project, exam, grade) VALUES (4, 99, 100, 100, "B")

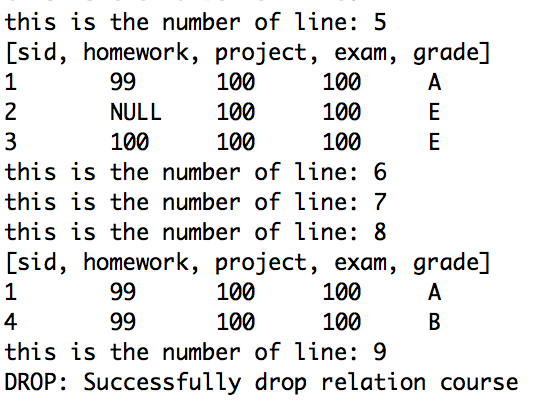
SELECT \* FROM course

DROP TABLE course

**Running time and disk I/O’s:**



**Result:**



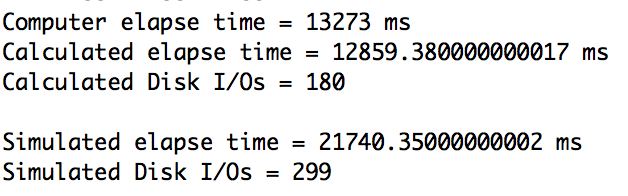
We can see that we create table, insert tuple into table and delete tuple in the table successfully.

## Single Table Two Pass

We insert 60 tuples into one relation, and we run the following query:

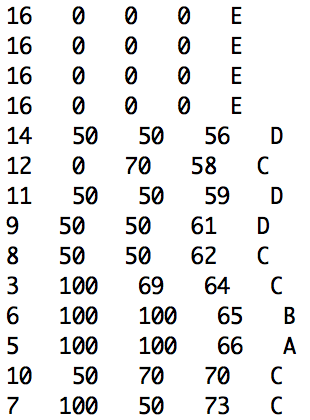
SELECT \* FROM course ORDER BY exam

**Running time and disk I/O’s:**



**Result:**

Sid homework project exam grade



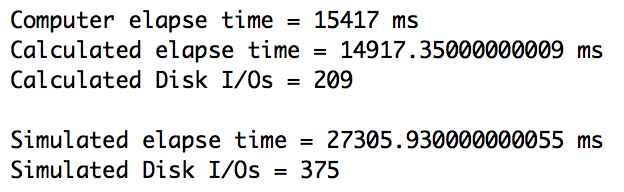
This is the part of the result, we can see the order by result.

## Multi-Table Two Pass

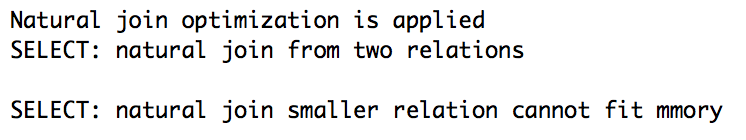
We also run the following query and show the partial result.

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

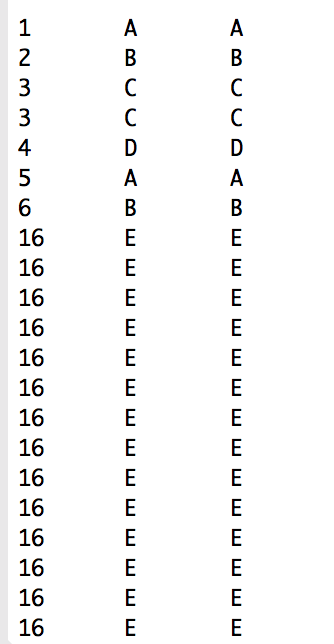
**Running time and disk I/O’s:**



**Result:**



course.sid course.grade course2.grade

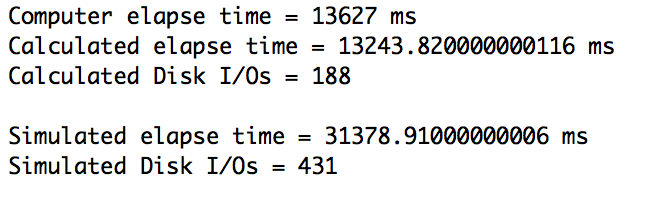


## Multi-Table Natural Join

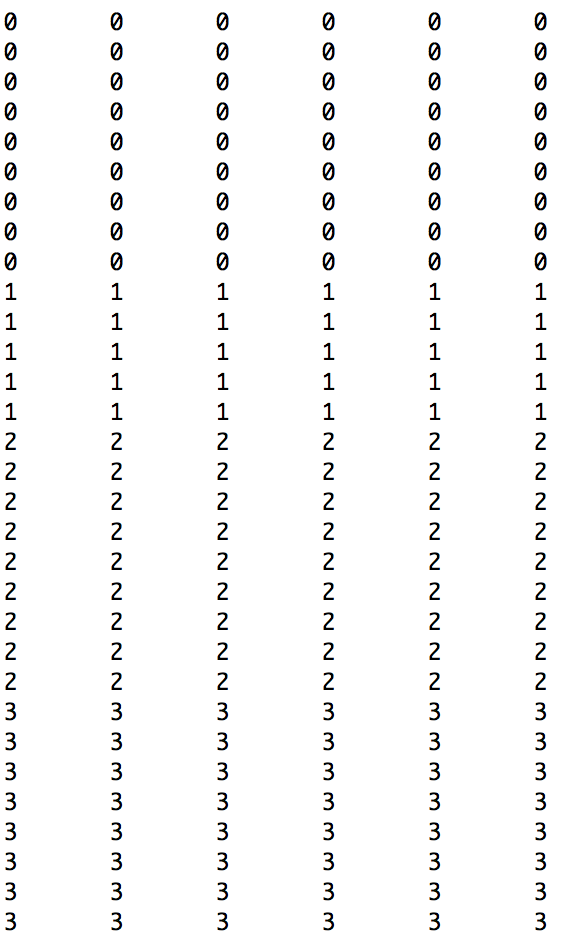
For the following query:

SELECT \* FROM r, s, t WHERE r.a = t.a AND r.b = s.b AND s.c = t.c

**Running time and disk I/O’s:**



**Result:**



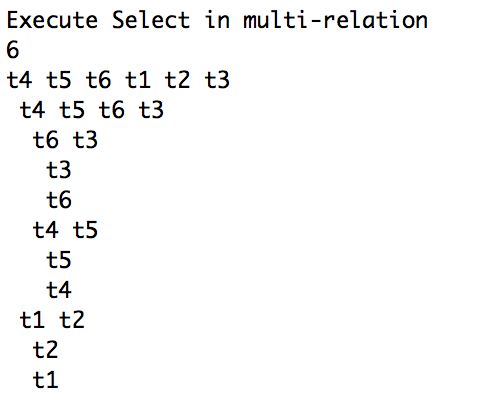
## DP Algorithm for Multi-Table Join

Query:

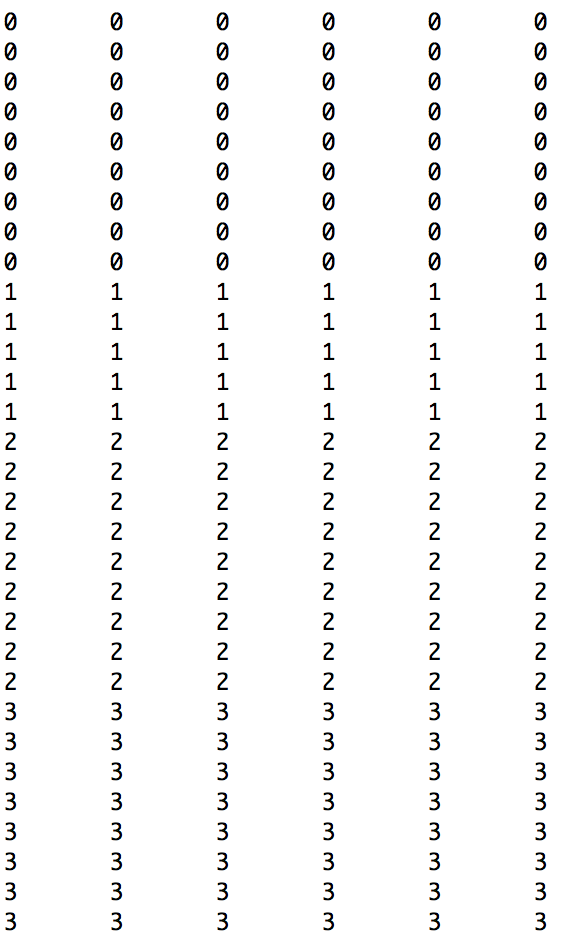
SELECT \* FROM t1, t2, t3, t4, t5, t6

**Result:**

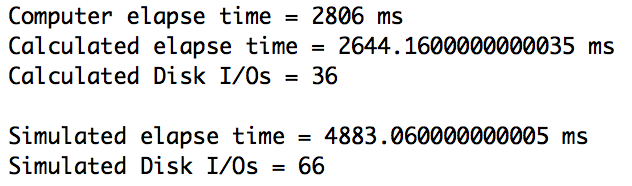
Join Sequence:



Partial Result:



**Running time and disk I/O’s:**



# Summary

In this project, we learned how to built parse tree, how to implement one pass, two pass and DP algorithm to optimize select and join operation and how the data manipulate between dick and main memory.

From the running time and disk I/O’s result, we can see that our programming works pretty well after adding one pass, two pass, DP algorithm to select and join optimization.