Fall 2015 CSCE 608 Database Systems
Project 2 Report

Team Members:

1) **Name:** Deep Desai
   UIN: 124001412
2) **Name:** Jasmeet Singh
   UIN: 523005618

Table of Contents

1. Introduction

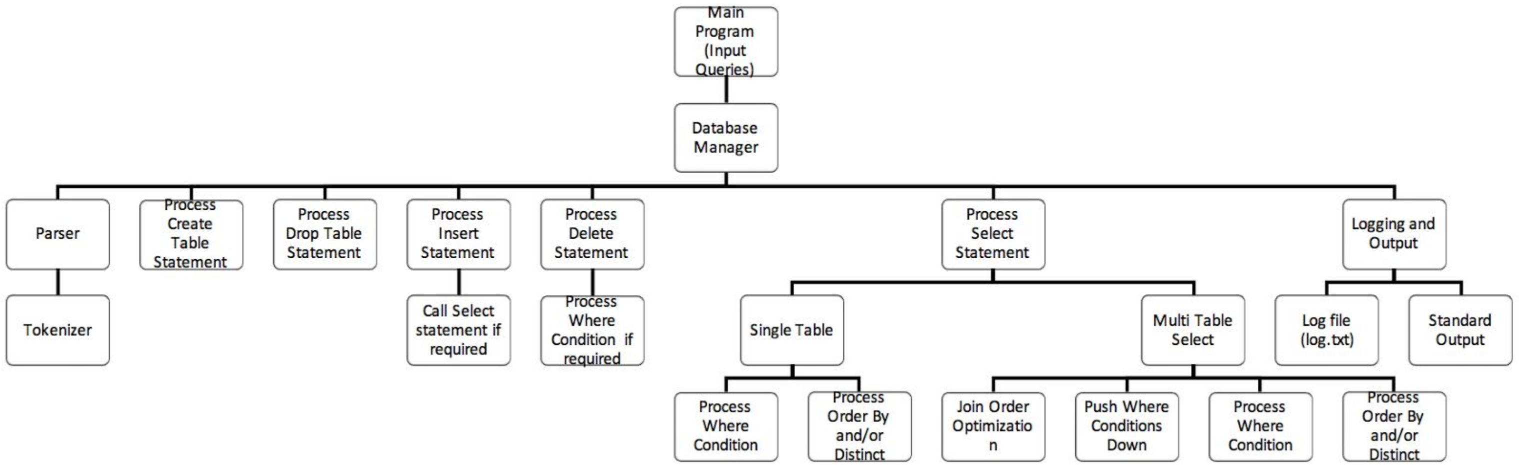
This project implements a mini database management system compliant to the TinySQL grammar. The system interprets the input SQL query and executes the query using provided StorageManager library as an abstraction for Disk and Main memory. The StorageManager library helps simulating the Disk I/Os and execution times. This shifts the focus of the project on implementing efficient algorithms to process the SQL queries minimizing the Disk I/Os and ultimately the execution time. The system is then tested on different types of valid queries in TinySQL grammar to get the Disk I/Os, execution time and the output of the query. The system has limited number of Main Memory blocks, which it has to use efficiently. The next section discusses the implementation details and the different optimization techniques at various steps. Different one pass or two pass algorithms are used to execute the queries minimizing the disk I/O's. The third section discusses experimental results and the report concludes in the fourth section.

2. Implementation

The project is implemented in C++ and uses G++ (tested on version 4.2.1) to compile the code. We are using some C++11 constructs in our code, so we change certain const declarations in StorageManager library to constexpr which is the standard in C++11. The project is built using a Makefile with all or main rules which compile all the files (including StorageManager) individually to construct object files and links each file and main file to get an executable output in file 'a.out'. The whole project follows a very elegant Object Oriented Design and appropriate testing using gtest testing library. The whole system is divided into different modules independent of each other and reduce coupling by passing any outside object inside the constructors of the classes.

2.1 Program Structure

At the high level the program consists of a main file which inputs string queries line by line till you reach EOF. Main program also initializes a Database Manager with Disk and Main Memory objects. The input query is then passed to the Database Manager which first creates a Parse Tree from the query using the Parser. Then it processes the query with its different modules depending upon whether query was of type Create Table, Insert, Drop Table, Delete or Select. The database manager also takes care of logging output to a log file and standard console output. The program structure is depicted in the block diagram shown below:



We will now discuss each of these subsystems in detail.

2.2 Tokenizer

The tokenizer takes an input string query and breaks it into token strings. This is an essential step before constructing a parse tree of the input query. The tokenizer breaks the input statement on spaces, commas, braces, quotes and any special symbol as required. It takes special care of not breaking the query when inside a quoted text. The tokenizer returns token strings as a vector of strings. These tokens are then used by the parser to construct the parse tree for the query.

2.3 Parser

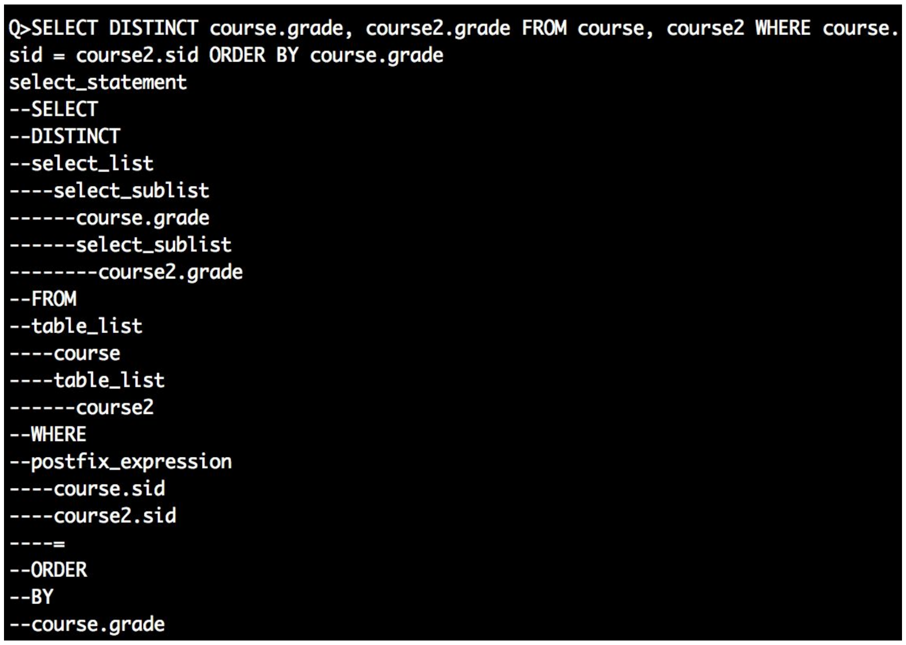
The main job of the parser is to take input query tokens constructed using the tokenizer and construct a parse tree from it. The parse tree constructed follows the TinySQL grammar structure strictly except for processing where clause, in which the whole where clause is stored as a single node.
The parser first checks from the tokens what is the type of the input query from among create table, insert, drop table, delete or select statements. Once the type of the query is determined the tokens are passed to appropriate functions which specialize at creating parse tree for the specific type of query. The parser uses a Parse Tree Node as a basic structure which stores the Node type which are the different types in the TinySQL grammar, a string value and a vector of pointers to it's children.
If there is a where clause inside a query, it is converted into a postfix expression by using a stack and appropriate precedence order decided by the TinySQL grammar. The possible operators as defined by the TinySQL grammar are +, -, *, /, <, >, =, AND, OR, NOT. The algorithm is as follows:

1) If the token is not an operator or an opening or closing bracket, the token is directly pushed at the back of the postfix expression.

2) If the token is an opening bracket, the token is pushed into the stack.
3) If the token is a closing bracket, pop the stack and push the popped token in the postfix expression till the stack becomes empty or an opening bracket or the same type is encountered.
4) If the token is an operator, pop the stack till becomes empty, or the top of the stack is an opening bracket or the top of the stack is an operator of higher precedence. The popped tokens are simultaneously pushed into the postfix expression.
5) At the pop the stack till it becomes empty and push all the popped tokens into the postfix expression.

This algorithm very efficiently builds the postfix expression in a single pass on the tokens. At this stage the only details we know of the token is if it is an operator or an operand. The Operator precedence as defined by the TinySQL grammar is as follows:
(*, /) greater than (+, -) greater than (<, > , =) greater than (NOT) greater than (AND) then (OR).

An example parse tree output for a complex input query is shown below (parse tree starts from 3rd line):


```
Q>SELECT DISTINCT course.grade, course2.grade FROM course, course2 WHERE course.
sid = course2.sid ORDER BY course.grade
select_statement
--SELECT
--DISTINCT
--select_list
----select_sublist
------course.grade
------select_sublist
--------course2.grade
--FROM
--table_list
----course
----table_list
------course2
--WHERE
--postfix_expression
----course.sid
----course2.sid
----=
--ORDER
--BY
--course.grade
```

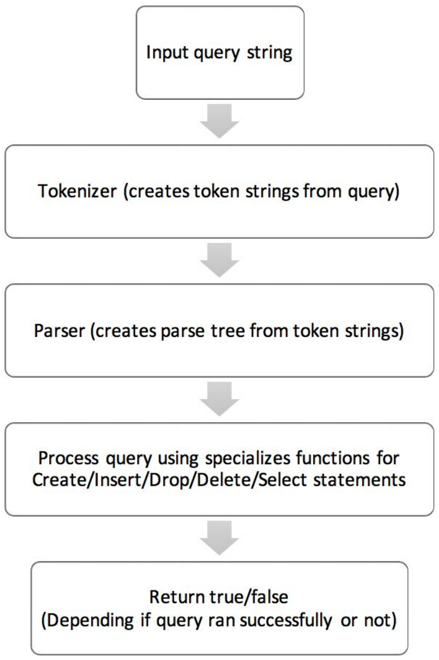
2.4 Memory Manager

The memory manager performs an important task of assigning and releasing memory blocks of the memory. The database manager initializes a memory manager with the main memory object. The memory manager then creates a stack with indices of the memory blocks according to the size of the main memory object. When a memory block is requested, it pops the free block index from the top of its stack and returns the index of the main memory block after clearing the memory block. The caller function can then use the memory block and the same

memory block index will not be assigned to any other caller as the index is not on stack. If the memory manager is out of free blocks it returns -1 indicating the same.
Once a function is done with using a memory block it calls the release block function which essentially pushes the memory block index back into the free blocks stack, ready to be assigned to any other caller function. To prevent any sort of main memory block leaks, the database manager initializes the memory manager after processing each query.

2.5 Database Manager

The database manager is the center of the system. It is only initialized once from the main program by providing a disk and main memory object. Then the database manager initializes a memory manager on the main memory object. The database manager provides a single public function accessible from the main program, that is to process a query input in the form of string.
The process query function of the database manager takes in the input query, then calls the parser to construct the parse tree and then calls the appropriate query processing modules depending upon what is the type of the input query. This functions returns a true or false depicting whether the query was successfully executed or not respectively. The database manager also logs and prints the output to a file named 'log.txt' and the standard console output respectively.
The flow of a query inside the database manager is as follows:


Input query string
Tokenizer (creates token strings from query)
Parser (creates parse tree from token strings)
Process query using specializes functions for Create/Insert/Drop/Delete/Select statements
Return true/false
(Depending if query ran successfully or not)

Next we discuss the different fourth step in the above flowchart in detail, that is to call specialized functions depending upon the type of the input query.

2.6 Create Table Statement

First of all the table name is retrieved from the parse tree created for the query. After that a vector of column names and their data types is created from the parse tree by recursively traversing the parse tree. Then a new schema is created using these column names and data types. After the schema is created a new relation is created using the SchemaManager from StorageManager library. If the created relation is not a null pointer it returns true otherwise it returns false.

2.7 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 deleted using the SchemaManager from the StorageManager library.

2.8 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.

2.8.1 Values

If there is keyword VALUES in the query, a vector of strings is created that are the attributes in the order it is written in the query. Then another vector of strings is created that are the values in the order it is mentioned in the query. Once these vectors are ready, a tuple of these values is created according to the relation's schema. If the field type for any field is INT, the string value is converted to integer and then set into the tuple. This tuple is then set in the relation.

2.8.2 Select Subquery

If the query doesn't have the keyword VALUES, that means there is a select subquery in the query. The select subquery needs to be processed to retrieve the values. The select subquery either stores the result in the main memory or into a temporary relation depending on the size of the results and the available main memory. The tuples are then retrieved from either main memory or the temporary relation, whatever the case, and set into the table.

2.9 Condition Evaluator

Whenever there is a WHERE clause in a query, it uses the condition evaluator class to process the where clause. Only Delete and Select statements can have where clauses and sometimes Insert statements when a Select statement is called recursively.
The condition evaluator 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 evaluator 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 evaluator does the following checks:

1) If the token in the postfix expression is an operator (+, -, *, /, >, <, =, AND, OR, and NOT are possible operators).
2) 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.
3) Otherwise the string is assumed to be a constant, and it is converted to appropriate integer or string depending on the variable on the other side of the operator is an integer or string

Once the condition evaluator has been initialized, the calling function calls the evaluate function again and again as required passing just the tuple as input. The input tuple can then be efficiently evaluated using a stack to check if it passes the where condition or not. A true or false value is returned accordingly. The algorithm to evaluate the where postfix expression on an input tuple is as follows:

1) Start a loop from first token of the postfix expression to the last:
    a) If the token is a variable, fetch the appropriate field value from the tuple using the stored field offset and push it onto the stack.
    b) If the token is a constant, push the constant value onto the stack.
    c) If the token is an operator, it can be a unary or binary operator:
        i) Unary Operator: Specifically NOT operator, pop the top value from the stack, take it's logical negation and push it back onto the stack.
        ii) Binary Operator: Pop two values from the stack and apply the operator on these two values depending on which operator (+, -, *, /, <, >, =, AND, OR) and push back the result onto the stack
2) Pop the top value from the stack, this is the result to be returned, if it is a true value return true else return false.

This algorithm very efficiently evaluates a tuple for the stored where condition in the form of a postfix expression. The initialization step does optimizations like storing the offset of the variable in the tuple right at the initialization step, so that the value of the variable can be quickly fetched from the tuple during the evaluation step. Another optimization is converting constants to integer values if the variable on the other side of the operator is an integer, thereby speeding up the comparisons during the evaluation step.

2.10 Delete Statement

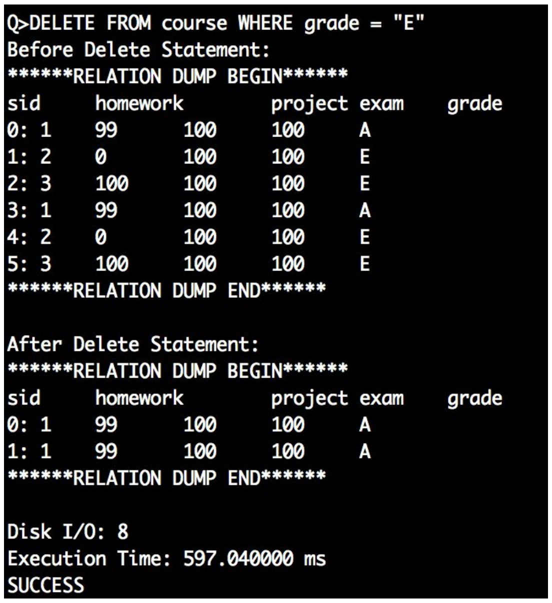
Delete statement has a relation name from where the tuples need to be deleted and can have a WHERE clause to check if the tuple has to be deleted or not. Delete statement can also be without a WHERE clause in which case all tuples are deleted. If we just delete a tuple from the disk block it creates 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 where the next block has to be read from, and second where the next output block has to be written. The algorithm is as follows:

1) Initialize read_index, write_index to 0

```
2) Get two memory blocks viz input_mem_block and output_mem_block
3) Initialize condition evaluator if WHERE clause present
4) If no WHERE clause present
    a) Clear all blocks starting from 0 in the relation
    b) return
5) Start from 0 to number of blocks in relation
    a) Read a block at read_index into input_mem_block and increment read_index
    b) For each tuple in input_mem_block check for validity of WHERE clause
    c) If WHERE clause fails
        i) If output_mem_block is full,
            (1) write output_mem_block to write_index block in disk
            (2) increment write_index
            (3) clear the output_mem_block
        ii) Write the tuple to output_mem_block
    d) Else do nothing, as this tuple has to be deleted.
 4) Clear all blocks from the relation starting from write_index.
```

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. Optimization done at this step is no disk reads when no WHERE clause and hole removal while deleting as described above.
The sample output below shows relation dumps before and after a delete query and the disk I/Os in each case.

1) Delete statement with WHERE clause. It can be seen there are no holes in the relation dump.

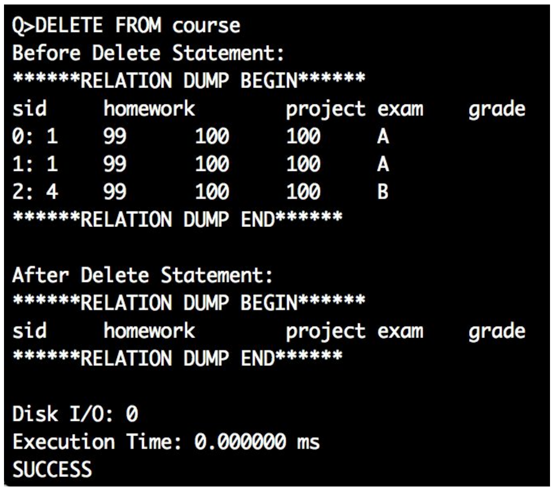


```
Q>DELETE FROM course WHERE grade = "E"
Before Delete Statement:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 2    0       100     100     E
2: 3    100     100     100     E
3: 1    99      100     100     A
4: 2    0       100     100     E
5: 3    100     100     100     E
******RELATION DUMP END******

After Delete Statement:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 1    99      100     100     A
******RELATION DUMP END******

Disk I/O: 8
Execution Time: 597.040000 ms
SUCCESS
```

2) Delete Statement without WHERE clause. It can be seen in the image that there were no Disk I/Os done.


```
Q>DELETE FROM course
Before Delete Statement:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
0: 1    99      100     100     A
1: 1    99      100     100     A
2: 4    99      100     100     B
******RELATION DUMP END******

After Delete Statement:
******RELATION DUMP BEGIN******
sid     homework        project exam    grade
******RELATION DUMP END******

Disk I/O: 0
Execution Time: 0.000000 ms
SUCCESS
```

2.11 Select Statement

To process the select statement three vectors are created: table-list, select-list and projection-list. The table-list is the list of all the tables mentioned in the query. The select-list is the list of all the column-names that are to be projected by the query. The projection-list is the list containing all the column-names from the select-list, all the column-names that are in the where clause and the column-name in order by. The select statement can have a single table select statement or it can have multiple tables in the table-list.

2.11.0 Optimization: Only fields in projection list stored in output relation

Only the fields present in the projection-list needs to be stored in output relation or main memory, the other fields are anyways not useful for processing this query. So the tuples for the temporary output relation or that stored in main memory will contain only that many fields that are in the projection-list. This optimization helps in reducing the size of the temporary relation or the number of blocks used in main memory.

2.11.1 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 evaluator if applicable. It makes a temporary relation if there is any of the DISTINCT or ORDER BY keywords present in the query and applies the corresponding algorithm on it. If there are no such keywords it tries to fit the results in main memory and return the main memory indices for the corresponding blocks or else it again makes a temporary relation with the results and returns it to the calling function. If it needs to print to stdout, it simply outputs it to stdout without creating any temporary relation or adding the results to main memory.

2.11.2 Multi Table Select

If the select statement contains more than one tables in the table-list, multi table select applies. Cross product order optimization as mentioned in section 2.11.2.2 is applied if number of tables are more than 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 as mentioned in section 2.11.2.1, 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. The final temporary relation is then returned to the DISTINCT or ORDER BY if required else it outputs to stdout. The multi table select can be one pass or two pass depending on the size of the temporary relations that gets created.

2.11.2.1 Optimization: Push Where Conditions Down

A very important optimization is done before doing a cross product of two tables. If any condition in the WHERE clause which can be separated and can be applied to a cross product is pushed down into that cross product and is applied when the cross product is done.
First of all a WHERE clause is broken into smaller conditions separated by an AND operator. Each of these smaller conditions can be evaluated separately and can be pushed down to cross products. The following is the algorithm:

1) Create a linked list of where conditions broken by AND operator only. If there is an OR operator in conjunction with AND operator, the condition cannot be broken.
2) For the two tables being joined:
    a) Create an empty linked list push_down to hold any pushed down Where conditions
    b) For each smaller Where condition in linked list:
        i) Check if all variables in the condition belong to the two tables
        ii) If true
            (1) Delete the current condition node from the linked list
            (2) Insert the deleted object into the push_down linked list of the current tables being joined.
    c) For each Where condition in the new push_down linked list
        i) Add AND operators in postfix form after two conditions to obtain postfix_condition
    d) Do the join operation with the pushed down postfix_condition Where condition

2.11.2.2 Optimization: Cross Product Order

The cross product order optimization 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.

2.11.2.3 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 mentioned in section 2.11.0 is applied.

2.11.2.4 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 mentioned in section 2.11.0.

2.11.3 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.

2.11.3.1 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 a bubble sort 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.

2.11.3.2 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 algorithm for sorting is as follows:

1. create a vector of queue of integers called sublists and a temporary relation called sublist_rel
    a. bring as many blocks into the main memory as possible from original relation
    b. sort these blocks using one pass algorithm
    c. add the sorted tuples to the temporary relation "sublist_rel"
    d. make a queue of these block indices in the sublist_rel and add it to sublists
2. make a vector of memory blocks used for sublists' blocks
3. bring 1 block from each sublist in memory
4. create a min-heap of the first tuple from each block
5. create a temporary relation called final_rel
6. get a main memory block which will act as the output block
7. while heap size != 0 do
    a. pop a tuple from heap and add it to the output block

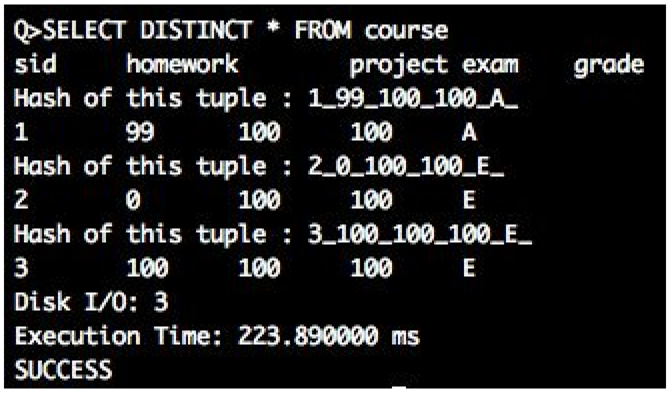
```
    b. if output block is full then
        i. add the block to final_rel
        ii. clear output block
    c. if the main memory block corresponding to the tuple is done
        i. if corresponding sublist is not empty
            1. clear the main memory block
            2. pop a block from sublist and put it into this main memory block
            3. push the top tuple from this main memory block to the min-heap
        ii. else
            1. push the next tuple from this main memory block to the min-heap
8. return final_rel
```

2.11.4 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.

2.11.4.1 One Pass Distinct

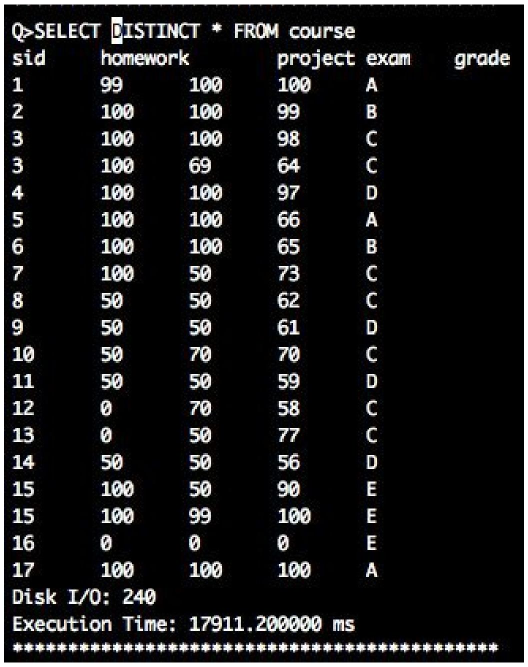
The one pass distinct can be done if the number of tuples in the relation is less than what can be accommodated in main memory. This algorithm takes in a vector of main memory block indices and first of all applies a bubble sort on the tuples in these blocks, the comparison is done based on a column chosen at random. It keeps a set (hash) of strings to keep a track of seen tuples. The tuples are converted to string by concatenating all the fields separated by "_" and then stored in this set. First tuple is sent to the output block. From the second tuple, it checks if the set contains the string representation of the tuple then it is ignored otherwise it is sent to the output block and its string representation is added to the set. If the output block gets full, it is sent to the output relation or printed to the stdout depending on the requirement.


```
Q>SELECT DISTINCT * FROM course
sid     homework        project exam    grade
Hash of this tuple : 1_99_100_100_A_
1       99      100     100     A
Hash of this tuple : 2_0_100_100_E_
2       0       100     100     E
Hash of this tuple : 3_100_100_100_E_
3       100     100     100     E
Disk I/O: 3
Execution Time: 223.890000 ms
SUCCESS
```

2.11.4.2 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. The duplicate removal is same as the two pass algorithm for sorting as mentioned in the

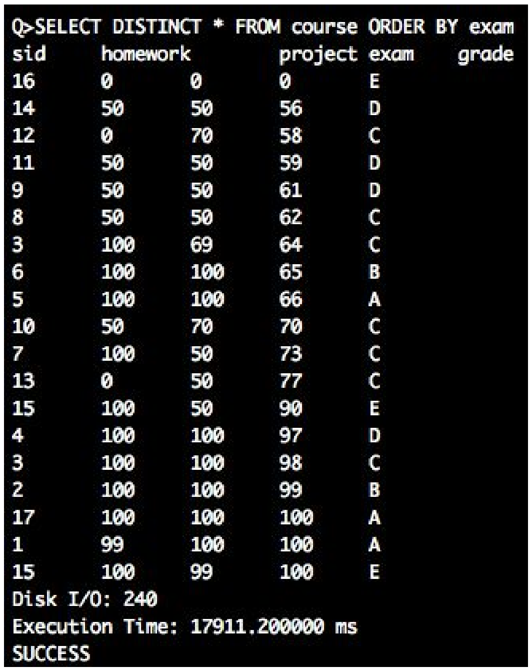
2.11.3.2 section except when any tuple is removed from the heap, it checks if the set contains the string representation of the tuple then it is ignored otherwise it is sent to the output block and its string representation is added to the set. In this way the temporary relation, final_rel, will have no duplicates and it is then returned to the calling function.



```
Q>SELECT DISTINCT * FROM course
sid     homework        project exam    grade
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
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
Disk I/O: 240
Execution Time: 17911.200000 ms
********************************************
```

2.11.5 Distinct and Order By

Distinct and Order By also can be done in two ways, one pass or two pass depending upon the size of the relation. It works in the same way as the Distinct works except for choosing a random column to sort, the column provided by the Order By is used to sort.



```
Q>SELECT DISTINCT * FROM course ORDER BY exam
sid     homework        project exam    grade
16      0       0       0       E
14      50      50      56      D
12      0       70      58      C
11      50      50      59      D
9       50      50      61      D
8       50      50      62      C
3       100     69      64      C
6       100     100     65      B
5       100     100     66      A
10      50      70      70      C
7       100     50      73      C
13      0       50      77      C
15      100     50      90      E
4       100     100     97      D
3       100     100     98      C
2       100     100     99      B
17      100     100     100     A
1       99      100     100     A
15      100     99      100     E
Disk I/O: 240
Execution Time: 17911.200000 ms
SUCCESS
```

2.12. Logging, Printing and Garbage Collection

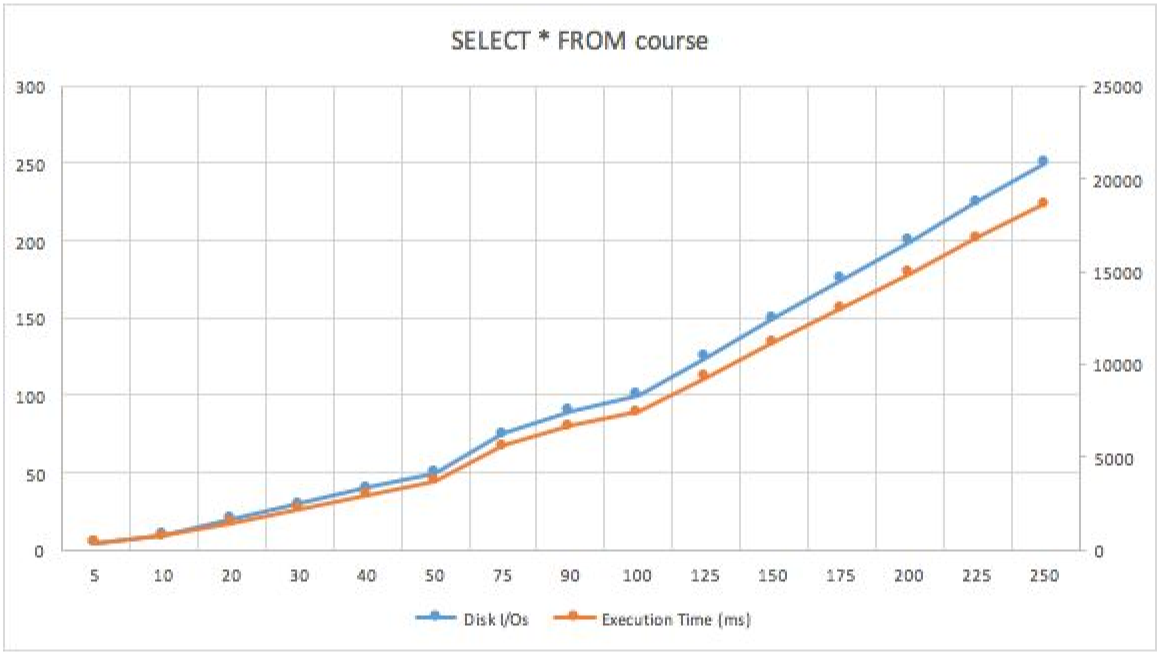
A file output stream is created that logs to a file. A stdout stream is used along with the file output stream in the same function to print to stdout as well as log to a file. After each query is run, garbage collection takes care of removing the temporary relation that are created during the execution of that query.

4. Experiment Results

For all the graphs shown below :
Left Y-axis: Disk I/Os; X-axis: number of tuples; Right Y-axis: execution time.

1) SELECT * FROM course

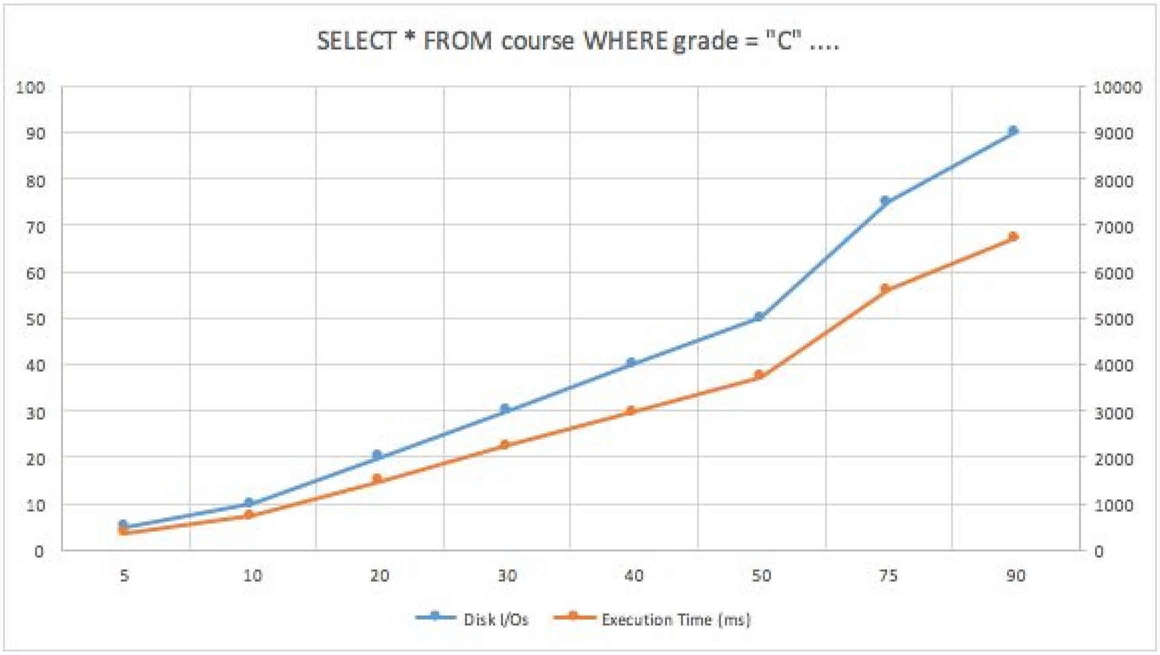
Table Size	Disk I/Os	Execution Time (ms)
5	5	373.15
10	10	746.3
20	20	1492.6
30	30	2238.9
40	40	2985.2
50	50	3731.5
75	75	5597.25
90	90	6716.7
100	100	7463
125	125	9328.75
150	150	11194.5
175	175	13060.25
200	200	14926
225	225	16791.75
250	250	18657.5



For single table select operations, the relation is read block by block and hence the disk I/Os increase linearly with increase in the number of blocks in the relation. Here each block contains only one tuple hence the increase is linear with increase in tuples. The execution time also increases linearly.

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

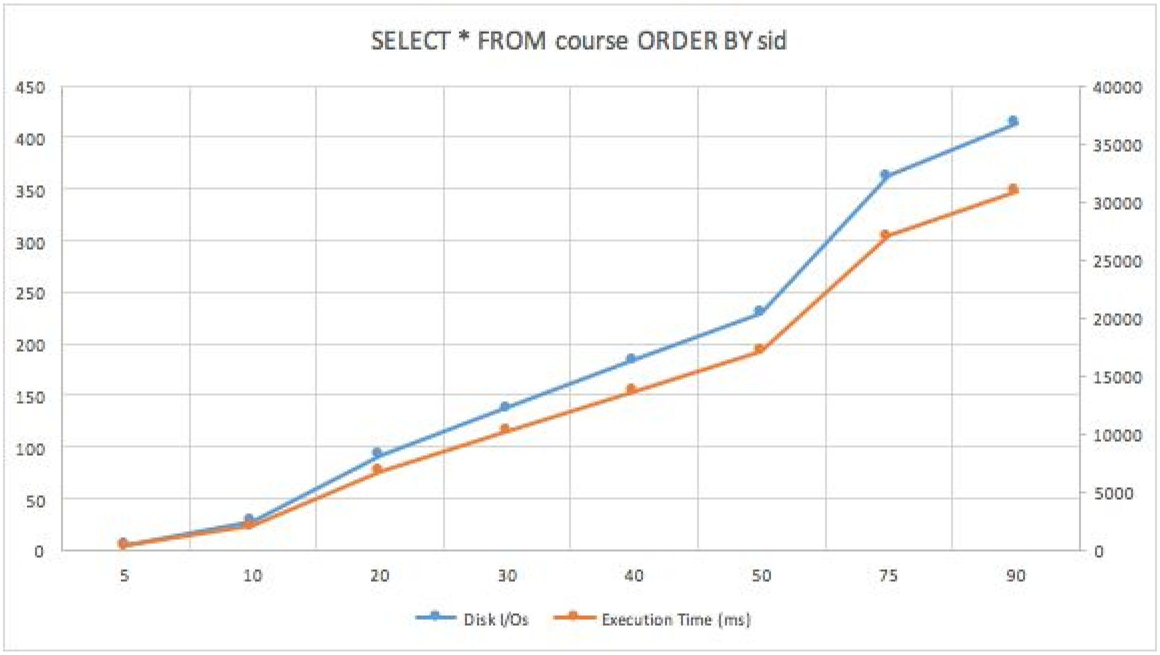
Table Size	Disk I/Os	Execution Time (ms)
5	5	373.15
10	10	746.3
20	20	1492.6
30	30	2238.9
40	40	2985.2
50	50	3731.5
75	75	5597.25
90	90	6716.7



For single table select operations, the relation is read block by block and where condition is applied to each tuple, hence the disk I/Os increase linearly with increase in the number of blocks in the relation. Here each block contains only one tuple hence the increase is linear with increase in tuples.

3) SELECT * from course ORDER BY sid

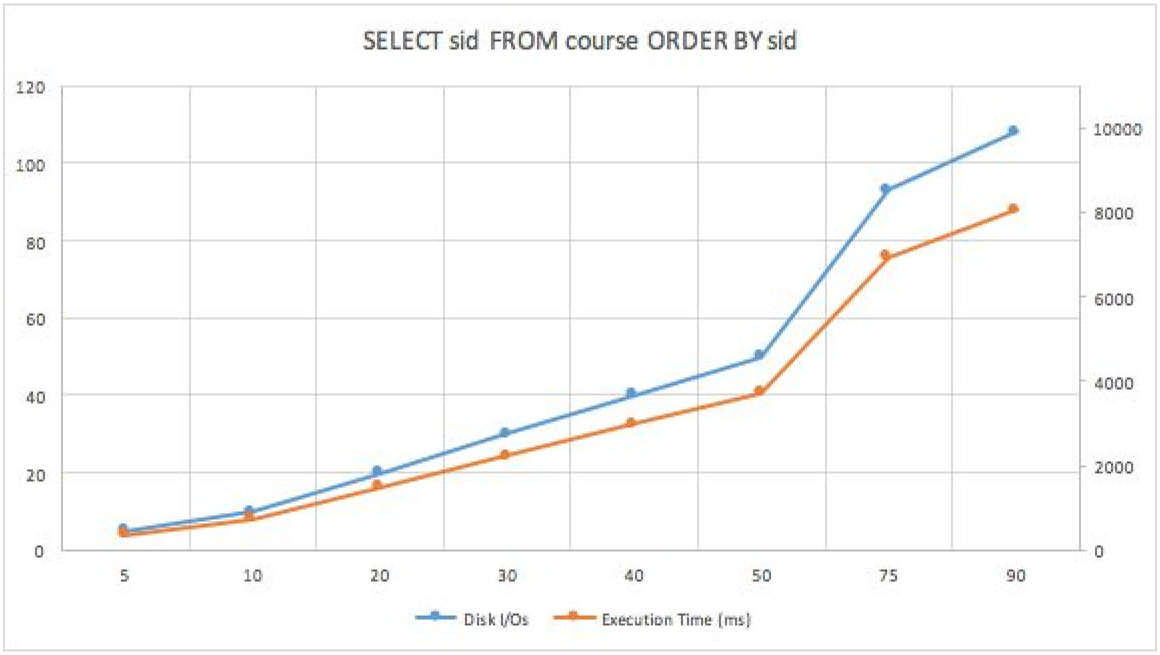
Table Size	Disk I/Os	Execution Time (ms)
5	5	373.15
10	28	2089.64
20	92	6865.96
30	138	10298.94
40	184	13731.92
50	230	17164.9
75	363	27090.69
90	414	30896.82



The one pass works until there are 9 tuples of this schema in the relation. After that the two pass algorithm applies and hence the disk I/O's increase. The execution time increases accordingly.

4) SELECT sid from course ORDER BY sid

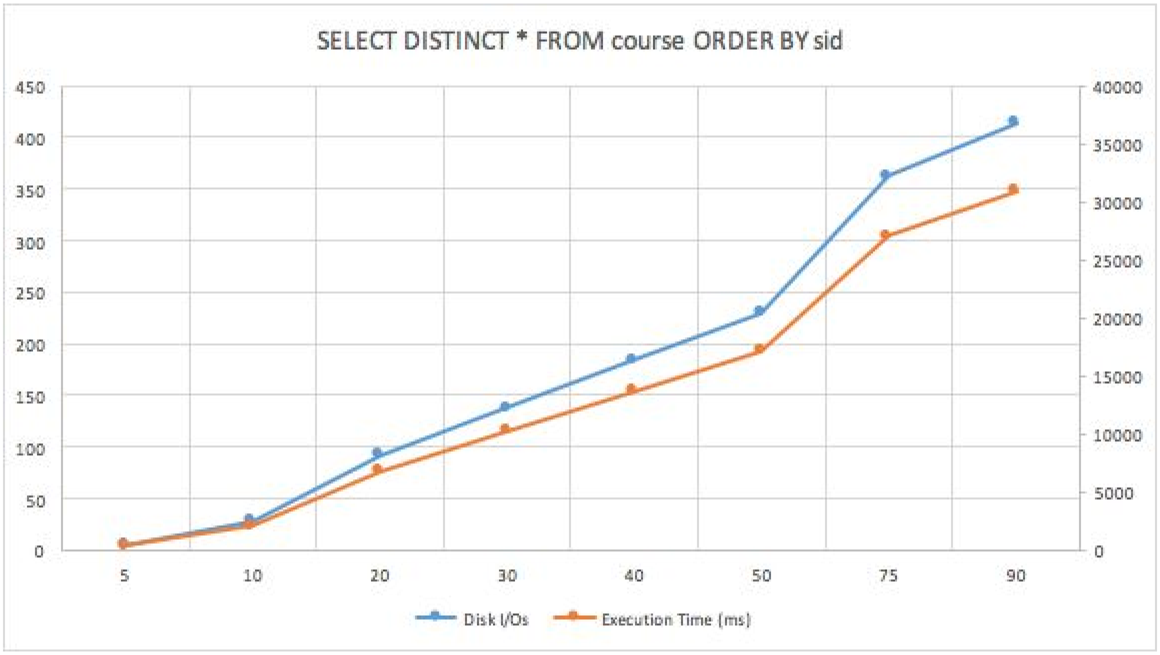
Table Size	Disk I/Os	Execution Time (ms)
5	5	373.15
10	10	746.3
20	20	1492.6
30	30	2238.9
40	40	2985.2
50	50	3731.5
75	93	6940.59
90	108	8060.04



Here the projection list has only one column hence the tuple stored in main memory will have only one field. So the Disk I/O increases linearly with increase in number of tuples until 8*9 = 72 tuples and after that it increases more than the increase in the number of tuples because of two pass algorithm.

5) SELECT DISTINCT * from course ORDER BY sid

Table Size	Disk I/Os	Execution Time (ms)
5	5	373.15
10	28	2089.64
20	92	6865.96
30	138	10298.94
40	184	13731.92
50	230	17164.9
75	363	27090.69
90	414	30896.82

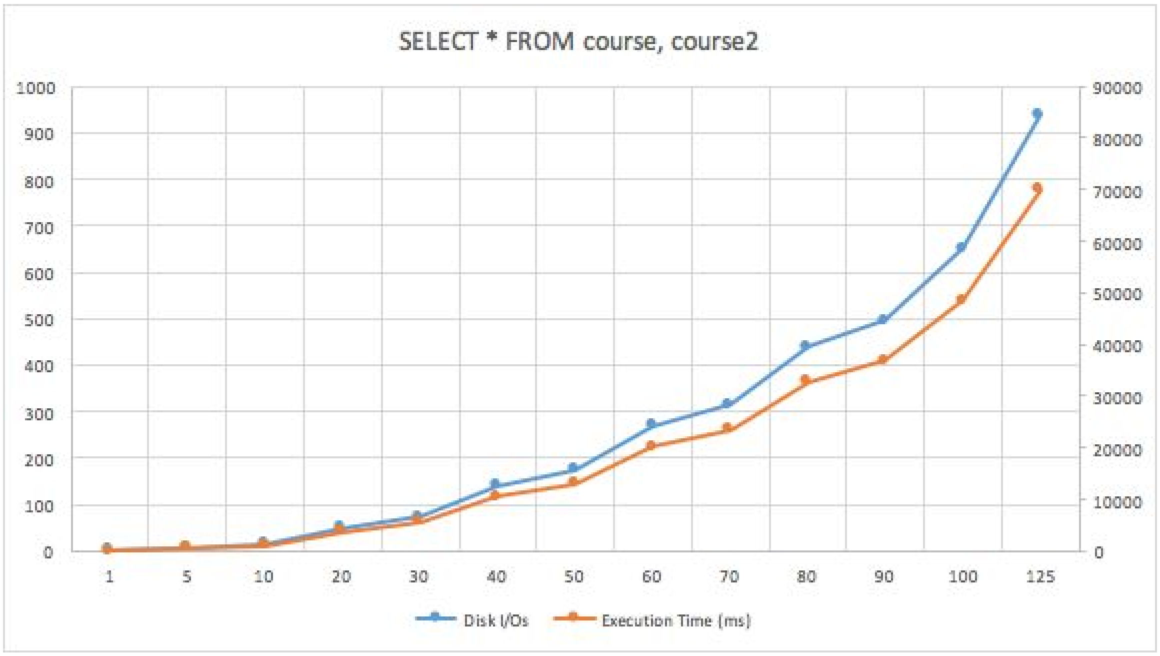


The one pass works until there are 9 tuples of this schema in the relation. After that the two pass algorithm applies and hence the disk I/O's increase. The execution time increases accordingly.

6) SELECT * FROM course, course2

Both Table Size	Disk I/Os	Execution Time (ms)
1	2	149.26
5	8	597.04
10	15	1119.45
20	50	3731.5
30	75	5597.25
40	140	10448.2
50	175	13060.25
60	270	20150.1
70	315	23508.45
80	440	32837.2

90	495	36941.85
100	650	48509.5
125	938	70002.94



The increase in Disk I/Os is much more compared to increase in the number of tuples because the cross product will have N*M tuples. One pass works until both tables have 3 tuples each. Then two pass algorithm applies. The execution time increases accordingly.

5. Conclusion

In this project we learned how a database system is implemented and how different algorithms are used to do the desired work. A lot of other database functions like indexing, logging, crash recovery, scheduling are key to a good database management software, but are out of the context of this project. We conclude that a lot of Disk I/Os and execution time can be reduced using optimizations as mentioned above. Increase in main memory helps reduce Disk I/Os and execution time. Cross product is expensive but the size of cross product can be reduced using optimizations.

****