CSCE-608 Database Systems

Design and Implementation of Tiny-SQL interpreter

(Medium of implementation : JAVA)

Submitted By

AKASH
UIN : 823008707

NIKHILESH PANDEY
UIN : 224002412

The objective of this project is to design and implement a simple SQL (called Tiny-SQL) interpreter according to the grammar provided as a part of the project statement. The interpreter accepts SQL queries that are valid in terms of the grammar of Tiny-SQL, execute the queries, and output the results of the execution.

1. Structure and working

Structure adheres to the components suggested in the project description.
The general flow of program is shown in fig 1.

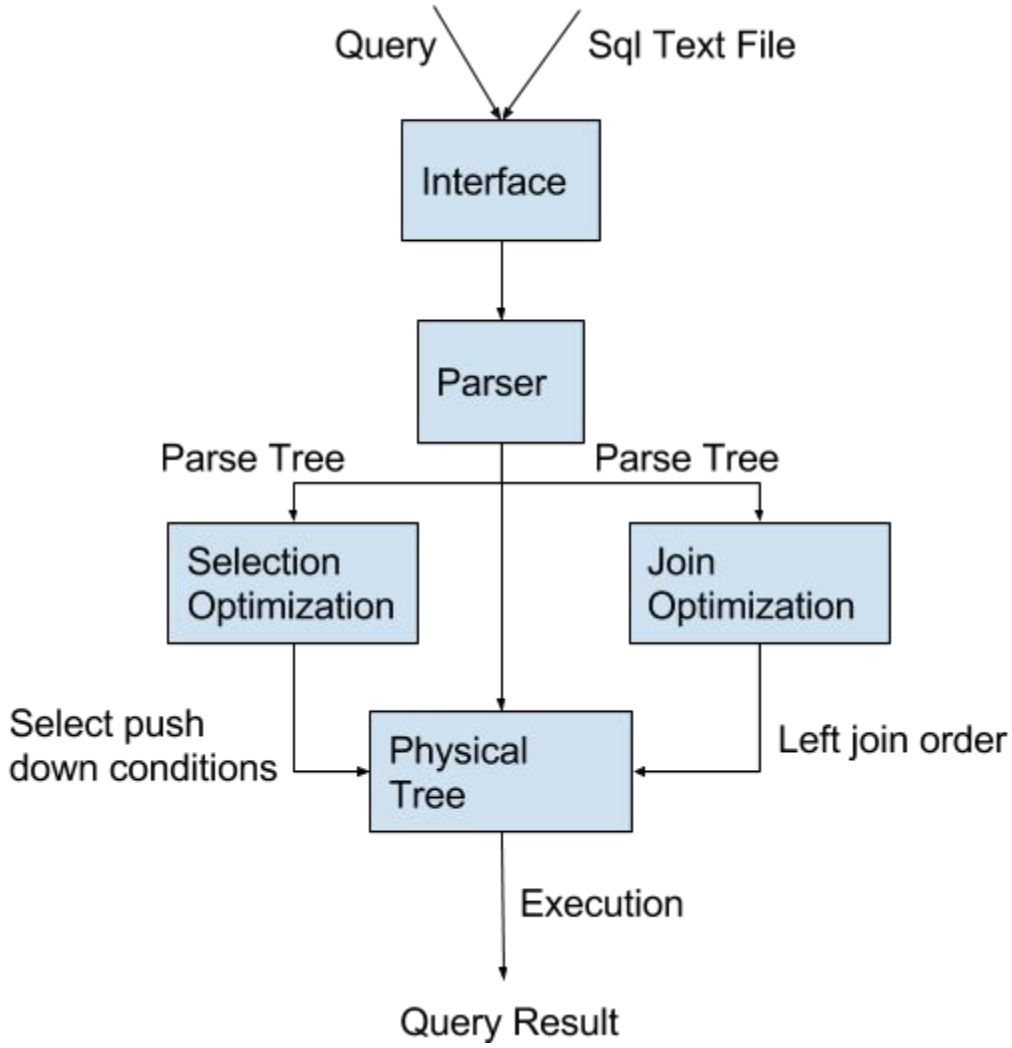


Figure 1: General Program Flow

Components of the program are:

1. **Interpreter**
   Interpreter is that starting point of the code. Interpreter is configurable to accept either a single query or accept multiple SQL queries from INPUT.txt file. It also writes the output of the queries to both the console and in OUTPUT.txt file.
   Interpreter then begin calling the parser on the given query.
2. **Parser**

Parser parses the given query based on the type of statement (INSERT, SELECT, DELETE, DROP and CREATE). Parser construction is then followed by physical tree construction for SELECT and INSERT statement as only these statements are complicated enough to have physical tree. Rest of the statements are executed in the parser phase only.

3. **Selection Optimization (Logical Query Optimization)**
   Select Optimization phases generated a map between tables and search condition applicable to tables. Select optimization phase helps in pushing selection down the tree, thus, helping in decreasing I/O operations. Selection conditions extracted per table is shown ahead.
4. **Physical query plan generator**
   Physical query plan generator constructs an executable physical query tree. The nodes of the tree are the various operators possible. All the possible workflow of the operator is shows in Fig 2.

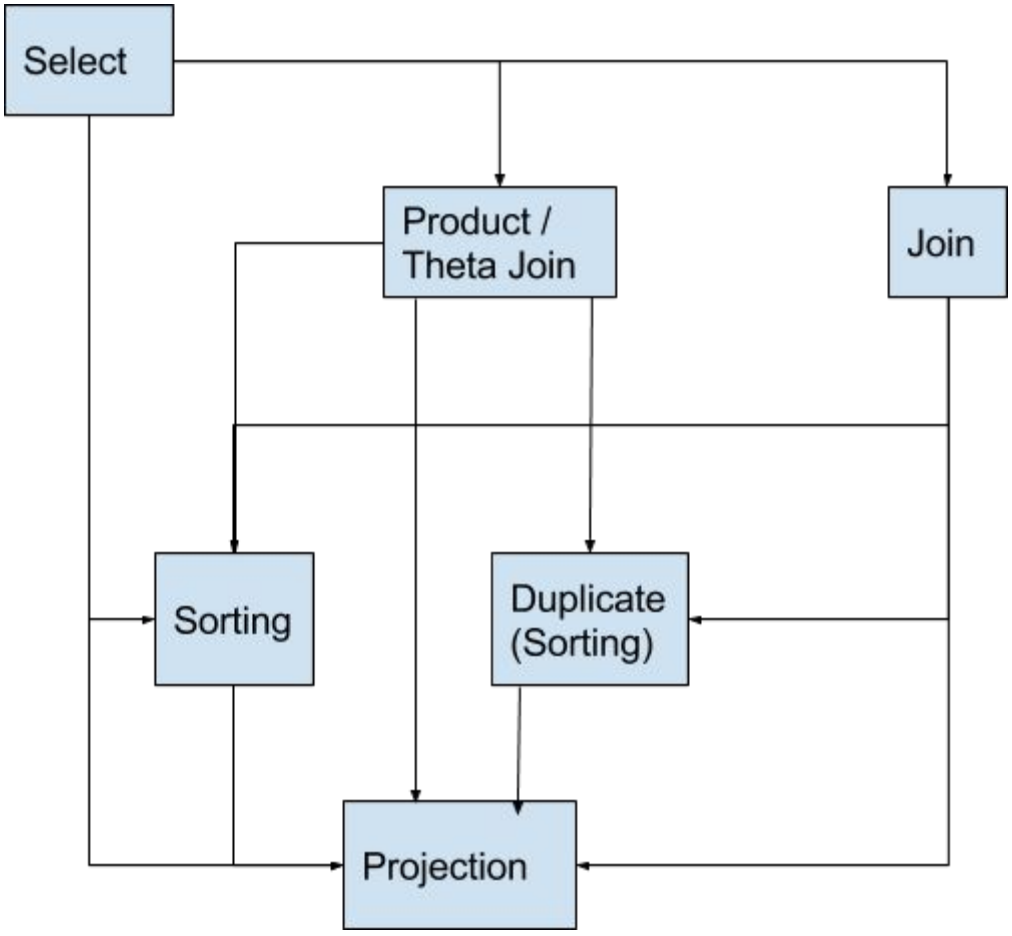


Fig 2. Physical Operator Flow

2. Key techniques

2.1. Interpreting Where Condition

For interpreting WHERE clause, where condition is parsed and stored as a tree with leaves of the tree being factor as given in the Tiny-SQL Grammar.

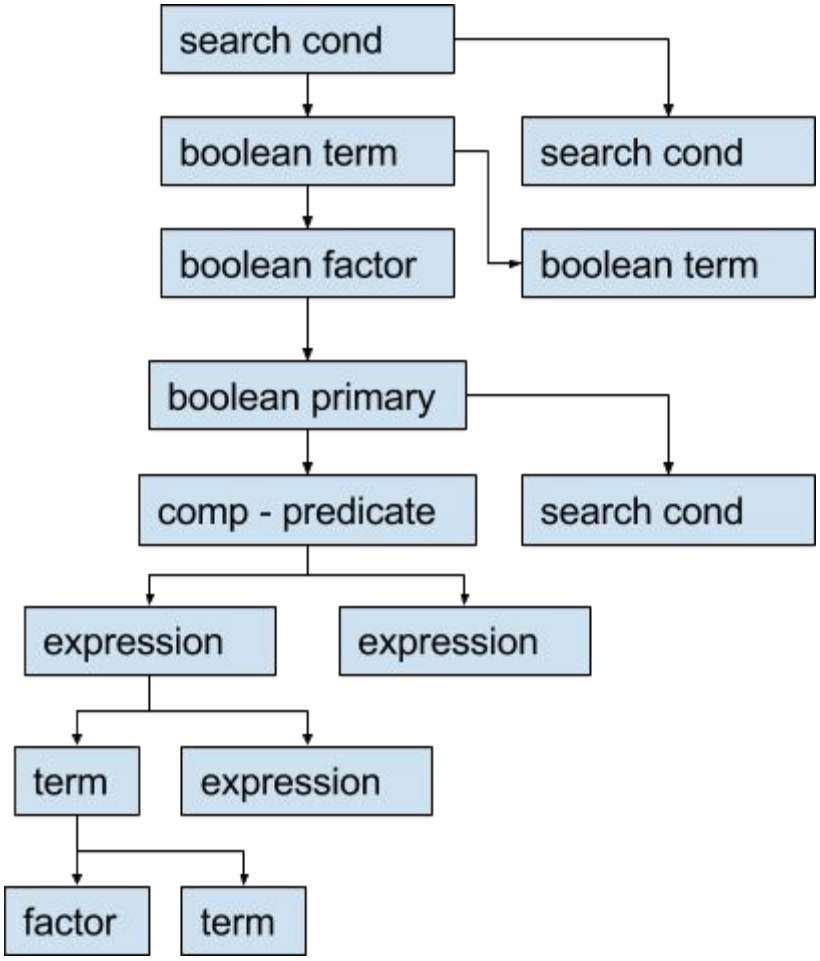


Figure 3: Search Condition Tree

Storing the search (where) condition as a tree gives a lot of flexibility in handling various search conditions easily. It also helped in breaking the search condition into multiple smaller table based search condition which was used for pushing the selection down which is explained further.

2.2. Join Operations

2.2.1. We have considered **memory constraints on the size of joinable tuples** in our implementation. Hence, in several scenarios (too many joinable tuples), two-pass join should and will not work. We have also come up with an optimization technique to minimize this problem by efficient use of memory blocks and conserving as many memory blocks(empty) as possible. We use these empty blocks to store the joinable tuples and declare two-pass not possible only if the joinable tuples can't be accommodated in these empty blocks. See section "**3.10.2 Two-pass Join**" for more details on the issue and our solution.

2.2.2. **One-pass :** We check if the smaller relation can be accommodated in the main memory

2.2.2.1. after keeping a memory block for the second relation's block and

2.2.2.2. one extra memory block for conditions when we need to write back the (intermediate) output to disk.

In case this is possible, we proceed with one-pass algorithm.

Note : Similar logic has been applied while deciding one-pass for sorting and duplicate removal operations on single relation.

2.2.3. **Two-pass**

We are using **"Two-pass sort-based algorithms"** for implementing two-pass.
We check if number of blocks required by the two relations in Phase 2 can be accommodated in the memory. This is
number of sublists for Relation1 + number of sublists for Relation2
This value should be <= main memory size OR
This value should be <= main memory size - 1 // in case, we need to write back the (intermediate) output to disk.

Note : Similar logic has been applied while deciding one-pass for sorting and duplicate removal operations on single relation.

2.3. **ORDER BY**

2.3.1. **One-pass**

2.3.1.1. Check if one-pass possible (2.2.1 above)

2.3.1.2. Bring the relation to memory and sort on the given sorting columns.

2.3.2. **Two-pass**

2.3.2.1. Check if two-pass possible (2.2.2 above)

2.3.2.2. Apply two-pass sorting on the given sorting columns.

2.4. **DISTINCT**

2.4.1. **One-pass**

2.4.1.1. Check if one-pass possible (2.2.1 above)

2.4.1.2. Bring the relation to memory and remove duplicates.

2.4.2. **Two-pass**

2.4.2.1. Check if two-pass possible (2.2.2 above)

2.4.2.2. Apply two-pass duplicate removal along with optimization mentioned in section "**3.8 Duplicate removal : Two - pass**"

3. Optimizations

3.1. Delete Optimizations

Instead of deleting/invalidating the tuples-to-be-deleted on the relation blocks and then writing the blocks back to the disk, we use an efficient optimization (below reads and writes for tuples are done block by block. Term "tuple" is used for clarity in the algorithm)

- we copy disk blocks one by one into main memory starting from Relation block 0 on disk.
- save tuples-**not**-to-be-deleted back to the disk block by block (start from Relation Block 0 on Disk - Clearly this is overwriting the existing data. But only tuples-to-be-deleted are getting lost).
- Invalidate the remaining blocks (not overwritten) in the relation - overwritten blocks contain all tuples-**not**-to-be-deleted.

Below are the benefits for this technique.

3.1.1. No holes in the relation

We never have any holes in any of the relations in the database at any given instance of time.

3.1.2. Improvement in I/O complexity

- We will require lesser writes in writing back this relation blocks after deletion.
- All subsequent read/writes for this relation will take lesser I/Os as the relation is now stored in lesser number of blocks (no holes)

3.2. We decide between **one-pass, two-pass or simple logic** (iterating through block pairs) at run time using the stats available to us.

We first check if one-pass is possible, if not *(see 2.2.1 above)*
We then check if two-pass is possible, if not *(see 2.2.2 above and 3.2 below)*
We do for block pair iteration method

3.3. Join Operations and vTable (Statistical data for table joining) :

We need to improve the logic plan to optimize the I/Os in case of joins involving more than two relations. The best order for the join operations is obtained by implementing **Left-deep join tree optimization**.

For achieving this
We have the
B(R) - Number of blocks containing the tuples of R and
T(R) - Number of tuples in R
But we do not have
V(R,A) - Number of distinct values on attribute A of R

Solution : Created **vTable** to keep track of V(R,A)

<table>
<tr><td>Relation 1</td><td><table>
<tr><td>Attribute 1</td><td>[value 1, value1 count]
[value 2, value2 count]
.
.
[value K, valueK count]</td></tr>
<tr><td>.
.</td><td>.
.</td></tr>
<tr><td>Attribute n</td><td></td></tr>
</table></td></tr>
<tr><td>.
.</td><td>.
.</td></tr>
<tr><td>Relation m</td><td></td></tr>
</table>

Here, V(Relation 1 , Attribute 1) = K

Note that we are also storing the frequency of an attribute value. This will be used for

- **Delete :** To find out when an attribute value needs to be removed from the vTable. We keep incrementing the count every time a value is inserted and decrement it on each delete. We remove the entry of value { and hence decrease V(R,A) } if the count becomes 0.
- **Two-pass natural join optimization :** If we have too many joinable attributes, Two-pass natural join is not possible. We use the count of matching values on the join attributes to find out if or not we have enough memory (main memory) to perform this join.

  e.g for a particular attribute value X, we can get total number of joinable matching tuples as
  count for value X from Relation1 + count for value X from Relation2
  This is the extra memory needed in the memory.

  We can check the memory blocks already in use as
  number of sublists for Relation1 + number of sublists for Relation2

  We know the total memory size. In case the sum of above stats is more than main memory size, Two-pass is not possible. Check section **3.10.2 Two-pass Join** (optimization) for more details.

3.4. Pushing selections down tree

Search condition is broken into various conditions depending on the table and combination of tables satisfying what part of search condition.
For Ex: SELECT * FROM course, course2 WHERE course.sid = course2.sid AND course.exam > course2.exam AND course.homework = 100
is broken into 3 search conditions:

Tables	Search Condition
course	course.homework = 100
course2	Null
course, course2	sid = course2.sid AND course.exam > course2.exam AND course.homework = 100

Once the search conditions are known, then Product/Theta operation can push the selection condition on each table before combining the tuples of tables for product operation.

3.5. Size based Product optimization

Product/Theta Join of tables is optimized by deciding the order of product based on the size of the tables. The smallest size tables are joined first so that if possible, result join can be stored in memory instead of disk, if possible. And, perform one pass product on other table saving a lot of disk I/O's.

SELECT * FROM t1, t2, t3, t4, t5, t6 where t1(c), t2(c), t3(c) ,t4(c), t5(c), t6(c)

Query 1

Table	t1	t2	t3	t4	t5	t6
Size	5	4	4	2	2	1

Table 1: Size of various relations

We can see the runtime and I/O comparison between two techniques in table 2 for Query 1. Size Optimization leads to nearly 450% improvement in I/O and runtime for this particular query.

No Optimization		Size Optimization	
I/O	Runtime (ms)	I/O	Runtime (ms)
1337	99780	227	16941

Table 2: I/O and Runtime comparison for Size Optimization

3.6. Avoiding unnecessary tables creation and storage in disk for Selection, Product, Sorting, Duplicate Removal

All the operators are intelligent enough to decide whether to keep the resultant table in memory or store in a disk. If the results generated can be kept in memory then the next physical operator is informed with the block numbers storing the table in memory and next operation can be performed in one pass.

For Query 1, as discussed in above section, if we use the above optimization technique

No Optimization		Table Creation Optimization	
I/O	Runtime (ms)	I/O	Runtime (ms)
1337	99780	886	66122

Table 3: I/O and Runtime comparison for Table Creation Optimization

We can see that there is more than 50% improvement in I/O and runtime for above query.

No Optimization		(Table Creation + Size) Optimization	
I/O	Runtime (ms)	I/O	Runtime (ms)
1337	99780	139	10373

Table 4: *I/O and Runtime comparison for Table Creation and Size Optimization*

Combining above optimization techniques have a nearly 860% improvement in both runtime and I/O as shows in Table 4,

3.7. Calling projection on tables directly instead of storing as tables

All the physical operators(Product, Join, Sorting, Duplicate, Select) check if next operator is projection then the intermediate tuple generated is passed directly to projection operator instead of storing it as a table saving a lot of disk I/O.

SELECT t.a FROM r, s, t WHERE r.a=t.a AND r.b=s.b AND s.c=t.c where r (a, b), s (b, c), t (a, c)

Query 2

Table	r	s	t
Size	41	41	41

Table 5: Size of relations

Performing Query 2 with calling projection directly on intermediate results instead of storing in a table has decreased the I/0 and runtime by 33% as evident from table 6.

No Optimization (Storing intermediate results in disk)		Projection Optimization	
I/O	Runtime (ms)	I/O	Runtime (ms)
367	27389	275	20523

Table 6: I/O and Runtime comparison for calling projection directly

SELECT course.sid, course.grade, course2.grade FROM course, course2 WHERE course.sid = course2.sid where course (sid, homework, project, exam, grade) and course2 (sid, exam, grade)

Query 3

Table	course	course2
Size	60	24

Table 7: Size of relations

Performing Query 3 with calling projection directly on intermediate results instead of storing in a table has decreased the I/0 and runtime by 12% as evident from table 8.

No Optimization (Storing intermediate results in disk)		Projection Optimization	
I/O	Runtime (ms)	I/O	Runtime (ms)
880	65674	780	58211

Table 8: I/O and Runtime comparison for calling projection directly

3.8. Bulk Insert Optimization

In the case of INSERT statement which takes values from a SELECT statement, if we simply insert the values one by one then it will take much more higher I/O as compared to handle the case separately.

For table course (sid, homework, project, exam, grade) if it already has 80 tuples and let say we execute query

```
INSERT INTO course (sid, homework, project, exam, grade) SELECT * FROM course
```

Query 4

The I/O and runtime of the query improves by nearly 50% in I/O which is a significant boost in the performance.

No Optimization (Appending one by one)		Optimization (Bulk Insertion)	
I/O	Runtime (ms)	I/O	Runtime (ms)
240	17911	161	12015

3.9. Duplicate removal : Two - pass

In phase 1 of two-pass algorithm, instead of converting the relation entries in disk into sorted sublists, a new temporary relation can be created (for storing the sublists of phase 1) and duplicate removal can be coupled with sorting in Phase 1 and this modified sublists can be stored to the said temporary relation. We can read this temporary relation for phase 2.
This optimization will reduce writes at the end of Phase 1 and reduce reads at the beginning of Phase 2 of two-pass algorithm. It will be highly effective in cases where many duplicates are present.

For table course (sid, homework, project, exam, grade) when it has 60 tuples (as given in TinySQL_linux.txt)and let say we execute queries

SELECT DISTINCT * FROM course

Query 5

No Optimization (Phase 1 sorting of original relation)		Optimization (Phase 1 temp relation creted)	
I/O	Runtime (ms)	I/O	Runtime (ms)
180	12285	131	9213

SELECT DISTINCT grade FROM course

Query 6

No Optimization (Phase 1 sorting of original relation)		Optimization (Phase 1 temp relation creted)	
I/O	Runtime (ms)	I/O	Runtime (ms)
180	12285	101	6974

The I/O and runtime of the query improves by nearly 40% on average in I/O which is a significant boost in the performance. This performance will further improve with more duplicates.

3.10. Natural Join optimizations

3.10.1. One-Pass Join

Runtime decision is taken for choosing if one-pass is to be used. Check section (**3.2**) for details.

3.10.2. Two-pass Join

Issue : If we have too many joinable attributes join is not possible. For doing R ⋈ S, in phase 2,

IF R_{min} = S_{min} THEN

collect all R_{min}-tuples and all S_{min}-tuples, and send their join to the output;

We need to optimize our algorithm so that we can spare maximum amount of memory possible for keeping "R_{min}-tuples and S_{min}-tuples"

Our Solution :
We need
MinMemRequirement = (R_{size} / memory_size) + (S_{size} / memory_size) + 1 (as in-memory buffer for writing back output to disk)
blocks for implementing any two - pass algorithm. We use the remaining "memory_size - MinMemRequirement" as additional buffer for storing "R_{min}-tuples and S_{min}-tuples".

This additional buffer can highly improve the efficiency and make two-pass join possible for relations with more joinable attributes.

3.11. Updating main memory size :

We keep updating the main memory available at run time.

- If this step in the query execution is final and output needs to be printed to standard output,
  available main memory size = actual main memory size
- If this step in the query execution is intermediate and output needs to be written back to disk
  available main memory size = actual main memory size - 1 // one block for writing output back.

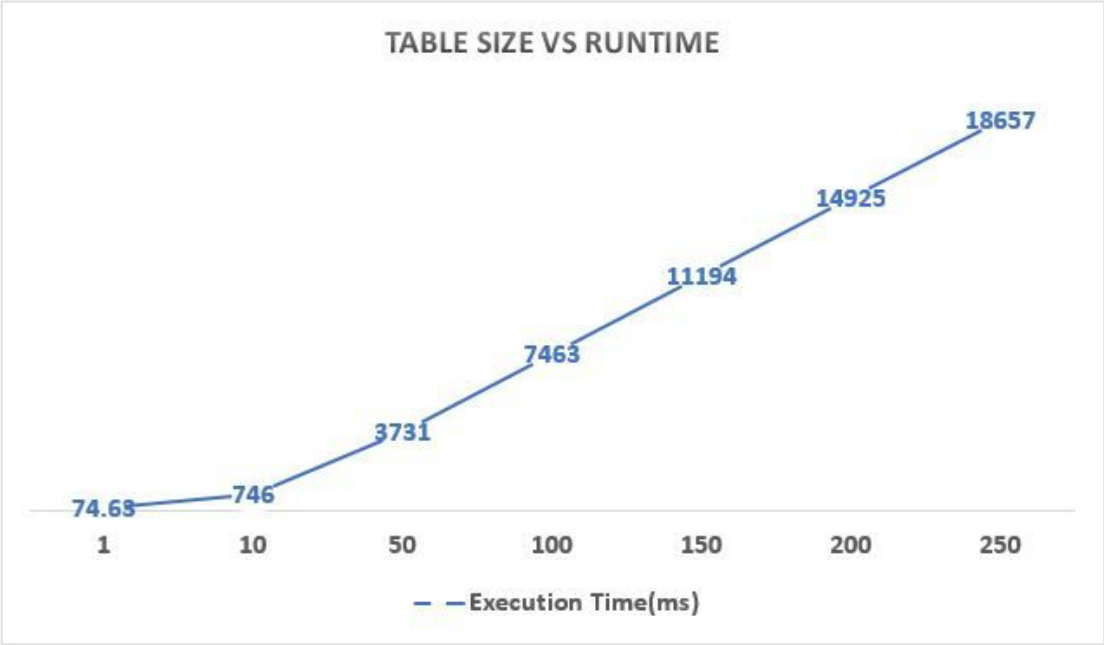
3.12. Keeping track of and deleting temporary relations

- We keep a global list with the names of all the temporary relations created on the disk as a part of a query execution.
- We delete all such relations on query completion
- This avoids the disk running out of space : The disk can accommodate at max 100 relations at a time.

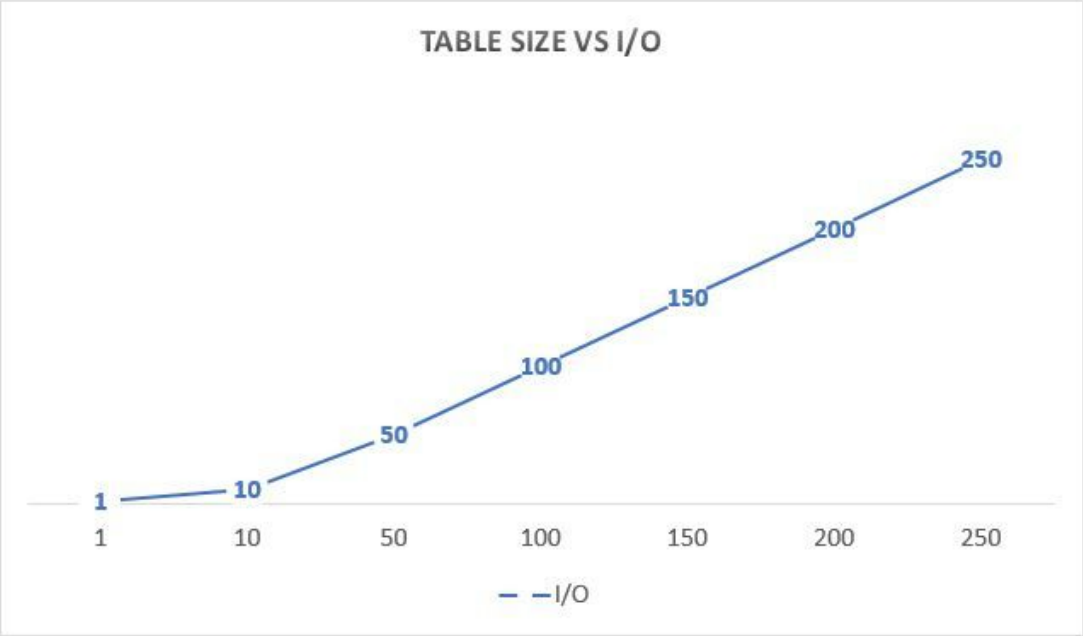
4. Runtime and I/O complexity

The I/O and execution time complexity of the system was tested by running the SELECT statement against the size of the table. The increase in the I/O and Runtime is both **LINEAR** with respect to size of the table as evident from graph 1 and 2 shown below.
Query used for testing was:

SELECT * FROM course WHERE sid = 18 ,
where course (sid INT, homework INT, project INT, exam INT, grade STR20)



Graph 1: Execution Time of SELECTION wrt Size of Table

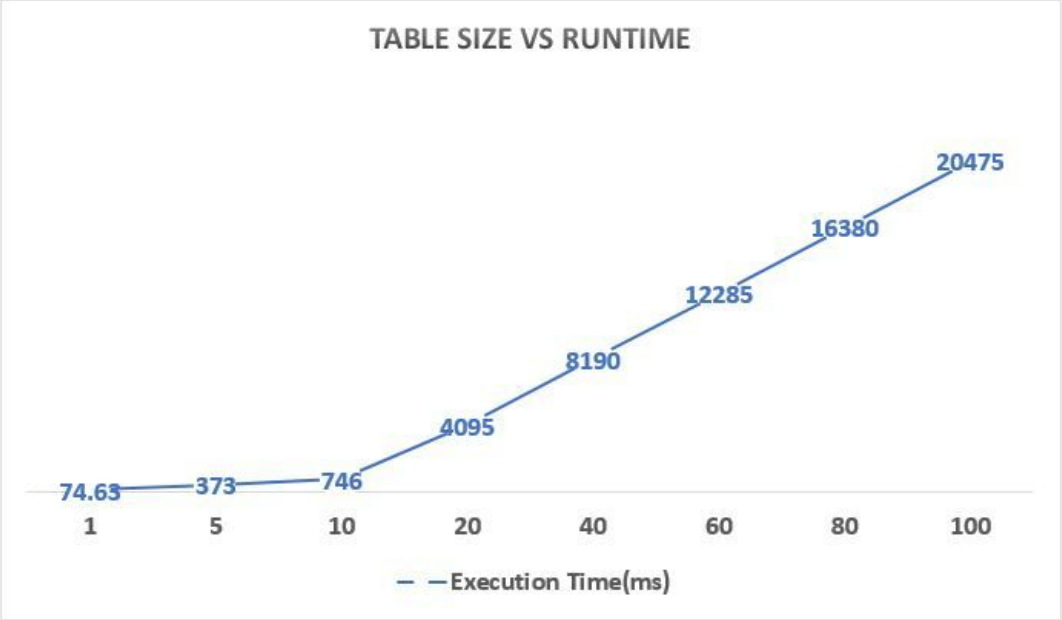


Graph 2: I/O of SELECTION wrt size of relation

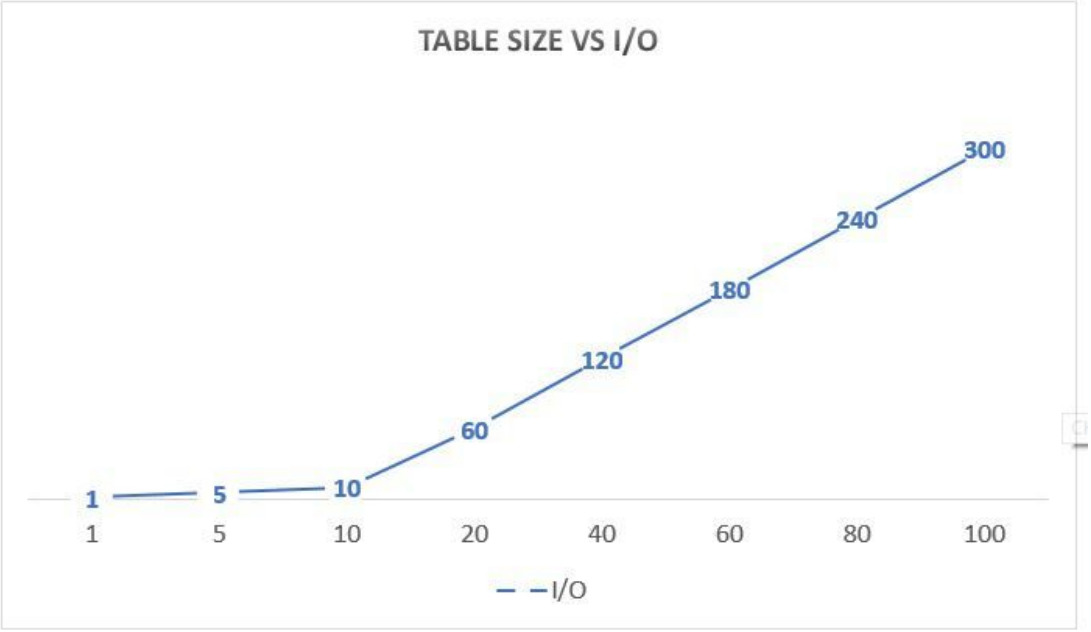
Similar analysis was done for SORTING query. Query used was

SELECT * FROM course ORDER BY exam
where course (sid INT, homework INT, project INT, exam INT, grade STR20)

The runtime and I/O of the query against the size of the relation can be seen in the graph 3 and 4. We can clearly see that the trend is **LINEAR**.



Graph 3: Execution Time of SORTING wrt Size of Table

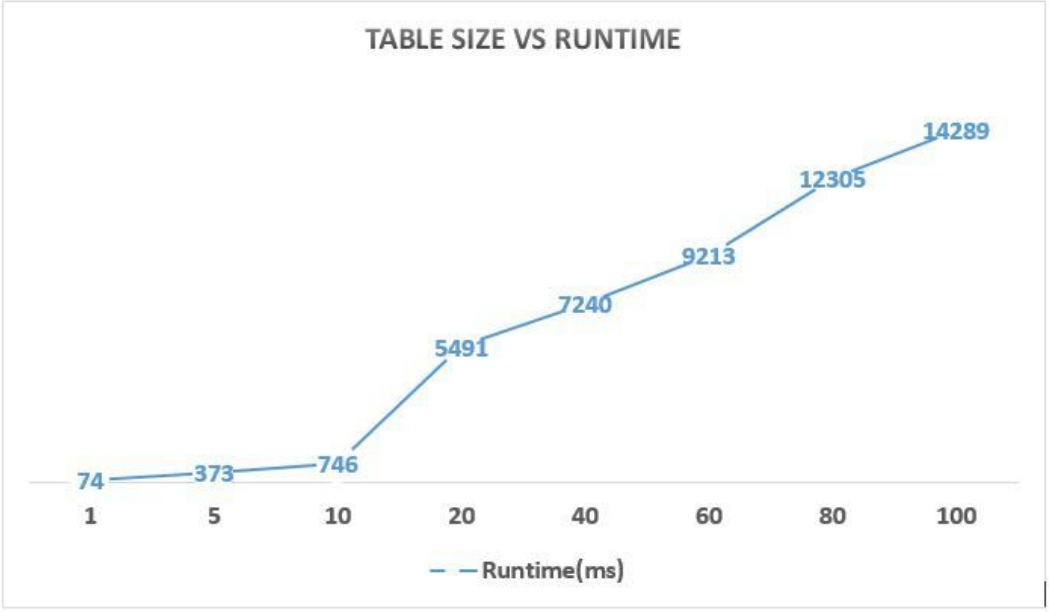


Graph 4: I/O of SORTING wrt size of relation

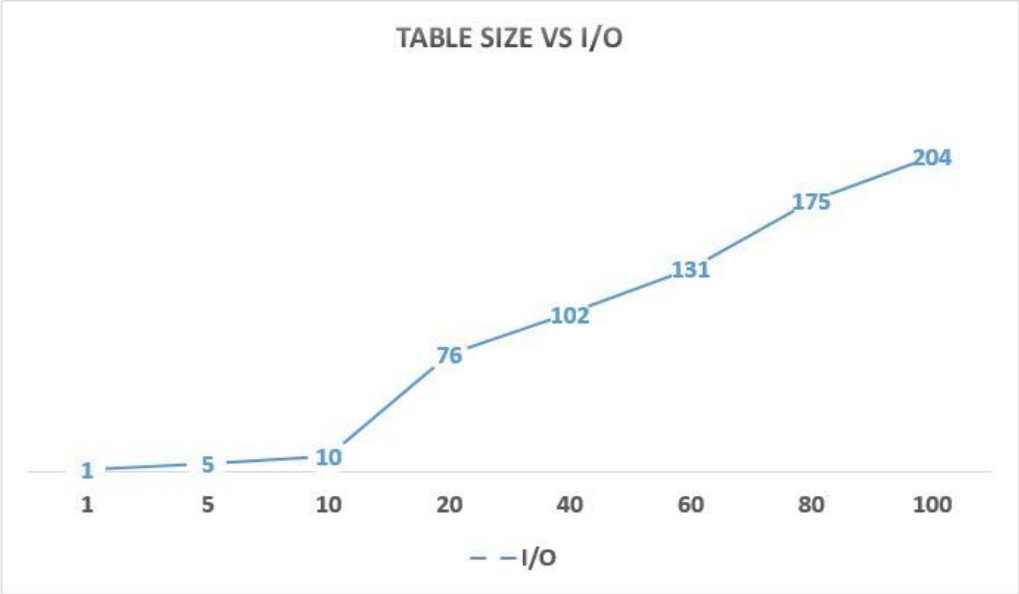
For Duplicate analysis, following query was used

SELECT DISTINCT * FROM course

The analysis of the duplicate query wrt to relation size is shown in Graph 5 and 6. Again, we can see that the relation is LINEAR.



Graph 5: Runtime of DUPLICATE wrt size of relation



Graph 6: I/O of DUPLICATE wrt size of relation

5. Acknowledgement

Implementing even a TINYSQL grammar based MYSQL database was a very challenging task and we believe that we have done justice with the project. We are really thankful to Dr.Jianer Chen for giving us such a wonderful opportunity to work on

such an interesting project. We are heartily thankful to our TA, Yi Cui, for guiding us and promptly answering our queries.