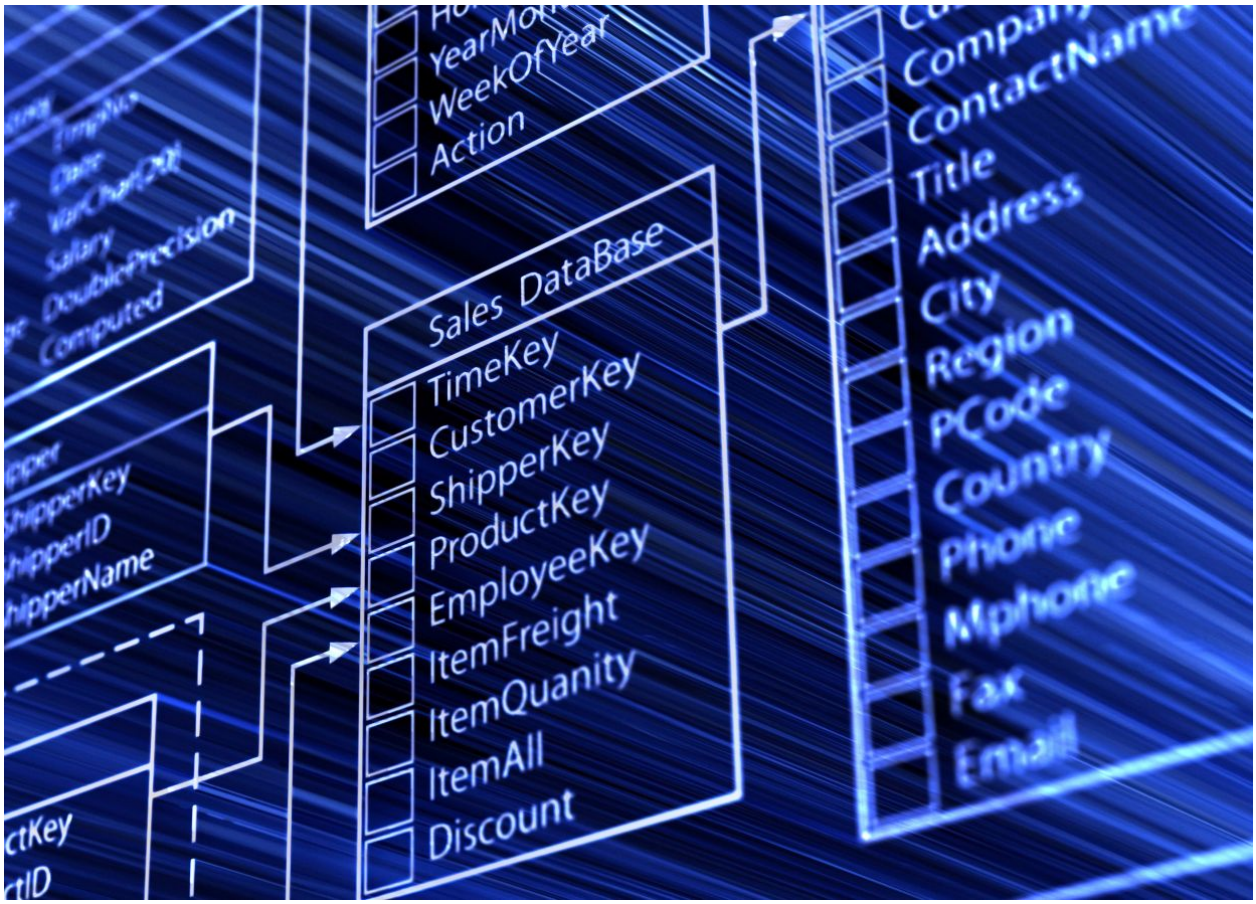
DATABASE SYSTEMS

PROJECT 2

TinySQL Interpreter



Submitted By:

Anavil Tripathi

Abhishek Nayyar

INDEX

INTRODUCTION

We have created a Database System basesd on TinySQL grammar provided in the project description. A database system's aim is to interact with user and database to manipulate the contained data so as to allow the user to analyze the data. TinySQL grammar was fairly simple, where we had to only handle a small subset of queries used in the real world database requirements.

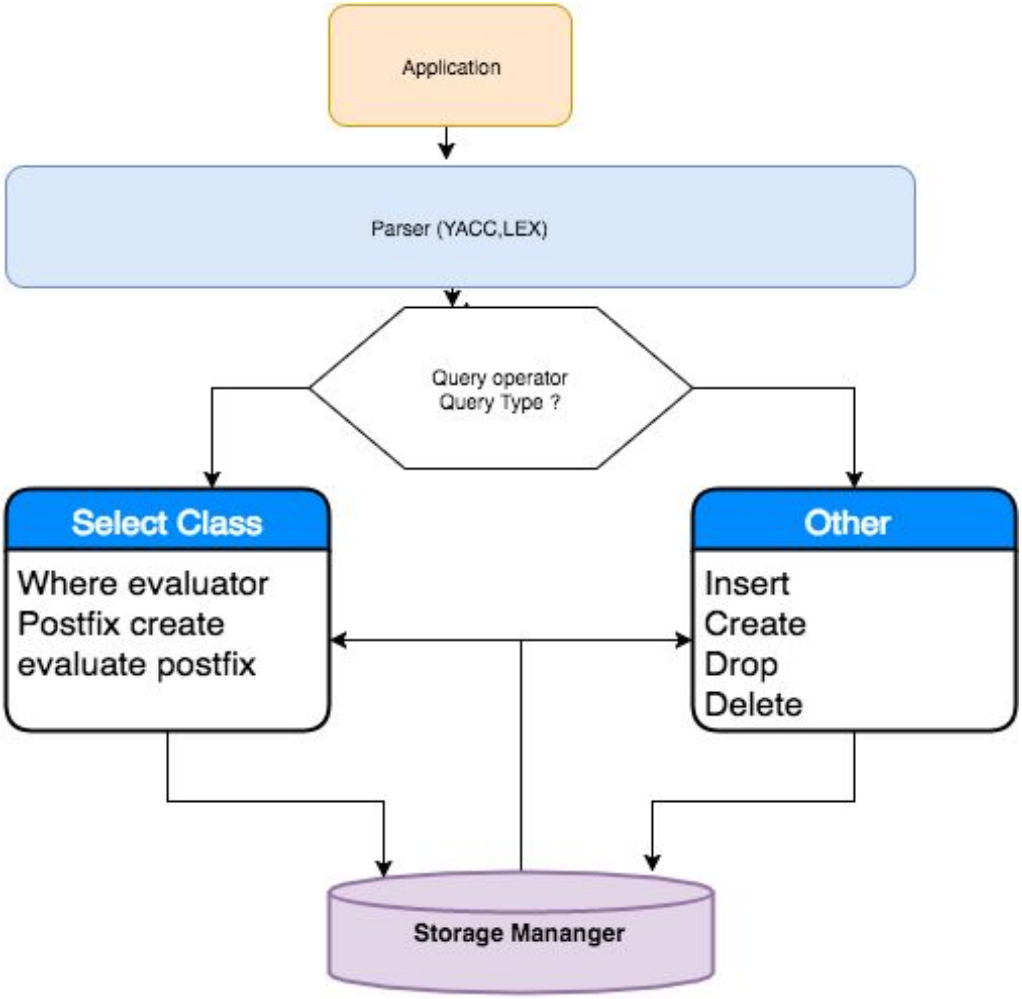
For our project we were given a StorageManager module which simulates the Disk and Memory. Our aim was to implement the structure to handle the user queries, manipulating the data in the disk accordingly. We had to make sure that we correctly handle the disk blocks for our queries.

Our interpreter broadly contains the following components:

- A parser: the parser accepts the input Tiny-SQL query and converts it into a parse tree.
- A logical query plan generator: the logical query plan generator converts a parse tree into a logical query plan.
- A physical query plan generator: this generator converts an optimized logical query plan into an executable physical query plan.
- Code Optimizer : This phase also includes any possible physical query plan optimization.

All the queries are handled through subroutine functions for each queries. Each subroutine interacts with optimizer and StorageManger for evaluating results.

Logical Flow of the project



Major steps for query executions are :

- **StorageManager Interface :**
  - Create SchemaManager object : This helps in handling relation and schema to be used.
  - Create Disk object : This helps in handling relation in secondary memory
  - Create MainMemory object : Helps in handling blocks in main memory simulation of Storage Manager

Create query_operator object:

- Parse the statement using Lex and Yacc.
    - Lex split the code into token
    - Yacc compare the token generated by lex with defined grammar
- Call execute_query() and depending upon the type of query we call different handler methods
    - CreateStatement:
        - Call execute_create_query():
            - Get table name from query
            - Get attribute name and type from query
            - Create schema and relation
    - Insert
        - Call execute_insert_query():
            - Get table name from query
            - Get attribute names and values from query
            - Make tuples using above data and Insert into table
    - Select
        - Call execute_select_query():
            - Parse the query to know whether it contains ORDER BY
            - Get column names that are to be output from query
            - Parse column names to know whether it contains DISTINCT or *
            - Get table names, on which select is to be performed, from the query
            - If there are more than 1 table the perform one-pass or two-pass join
    - DeleteStatement:
        - Call execute_delete_query():
    - Drop
        - Call execute_drop_query():
            - Get table names from query
            - Delete the relation

Implementation

User Interface: We have implemented a php based web-page to show get the input queries from user. We have two types of input method in our interface, where one is through a textarea and the other option for users is to input the file containing the queries.

Query parsing Implementation:

We have used lex and yacc for complete end to end parsing of the queries. Lex and yacc are used in conjuction for analyzing the keywords present in the queries to execute commands. We break the query using lex to generate keywords that are then used by yacc which has the grammar for our TinySQL.

Query parsing is done in accordance with the grammar specified in lex/yacc

Statement/Keyword Handling:

CREATE :

We get the schema details from the query. This is used to create a new schema and consequently a new relation is also created based on the new schema. This will be used in the future for insert, select, delete and drop queries by the user.

INSERT :

We create a new tuple setting all its fields according to the query. We get a memory block write the tuple to the block and then write that block back to disk. Here we also need to make sure that there are no gaps/holes in the disk blocks while inserting. We ensure this by implementing a method named "*appendTupleToRelation()*" which follows closely the method provided in the sample code. This method first checks the disk if the last inserted block can add another tuple to it.

SELECT

For select query , we first create query tree . Each keyword SELECT, FROM, WHERE, DISTINCT, ORDER BY represent keyword node and select_list,table_list, where_list specifies the user specified attribute node. For where_list we first break down entire query into where_condition_tree. This tree can have one child , where we have single expression to evaluate or three child node where leftmost and rightmost nodes specifies the expression or another where_condition_subtree to evaluate and middle node is AND/OR. In expression we handle all the comparator and other standard operations. Selection process is evaluated in two ways, one in optimized way and non-optimized way. For Non optimized way we first compute cross join and then we can simply apply the where clause on entire cross table. Select query is evaluated in following phases:

EVALUATE WHERE CLAUSE :

In this phase we do table loading from memory/disk based on the where conditions. In optimized version of code only the required table is brought and evaluated , whereas in non optimized all tables are cross joined before query evaluation.

Where_condition_tree evaluation process is broken down into two phases:

1. Form Postfix : Each expression is broken down into postfix expression in phase one which is used for ordered evaluation in next phase.
2. Evaluate Postfix : In this phase whenever we encounter operator we pop two elements from postfix stack and perform operation on two values and push back the result into postfix.

For optimized method, we load the table value from the memory/disk only during evaluation phase. During first phase we simply record the tokens specifies in query in postfix stack and during evaluate postfix phase values are actually brought and computed. If we compute the expression which involves cross join, we compute it and store it as new cross relation in disk. This re-enables us to reuse cross relations. ORDER BY clause is evaluated once we have entire results from where clause

PROJECTION PHASE

Once we have entire where clause evaluated , we apply projection operation on the basis of arguments specifies in select list of query. If DISTINCT is specified we take only, unique tuple.

DELETE

We have implemented two cases for this query. One is without any where condition where we remove all the tuples without removing the relation. In other case we check the tuples that satisfy the where condition and not include them in the temporary vector which will be later written to the disk. We also make sure to remove the extra blocks from the relation.

DROP

We remove the relation from disk when this is encountered. This is handled simply by calling "*deleteRelation()*" over the relation that is to be dropped.

DISTINCT

Once we have entire query evaluated , we create hashmap for each tuple values and pick only distinct values for each table.

ORDER BY

We have implemented a comparator over the tuple to handle the sorting of tuples. We sort the tuples based on the attribute provided in the query. We have used quick_sort to implement the actual sorting logic for the entries.

Join Handling:

OnePass: If at least one of the table size is smaller than the memory size we use OnePass algorithm as it allows for much faster execution. Here we bring the smaller table completely into the memory and then use the remaining blocks in memory to bring the other relation's blocks. We then compute the cross of the recently fetched block's tuples with all the tuples of the smaller table and write it back to disk.

We have implemented *onePassJoin()* which fetches all the blocks of smaller relation "R" to memory and then loops over "S" to fetch its blocks. We have then implemented a method

performVectorTuipleJoin() which works on two Tuple vectors to calculate cross product between the two and it writes the tuples to the new cross_table relation. We have also implemented a method "computeTupleJoin()" which computes the join between two single tuples and returns a new tuple.

TwoPass: If both the tables are larger than the memory then we use this method to loop over both the relations for calculating cross product. This is much slower than OnePass but allows for handling for much larger join sizes, which is only limited by the disk space and not memory size.

We have implemeted *twoPassJoin()* which brings two relations' blocks one by one to memory and performs the cross join using the two general methods already discussed for *onePassJoin()*. This method also makes sure that there are no holes created when inserting into the new cross_join table.

Optimizations Implemented

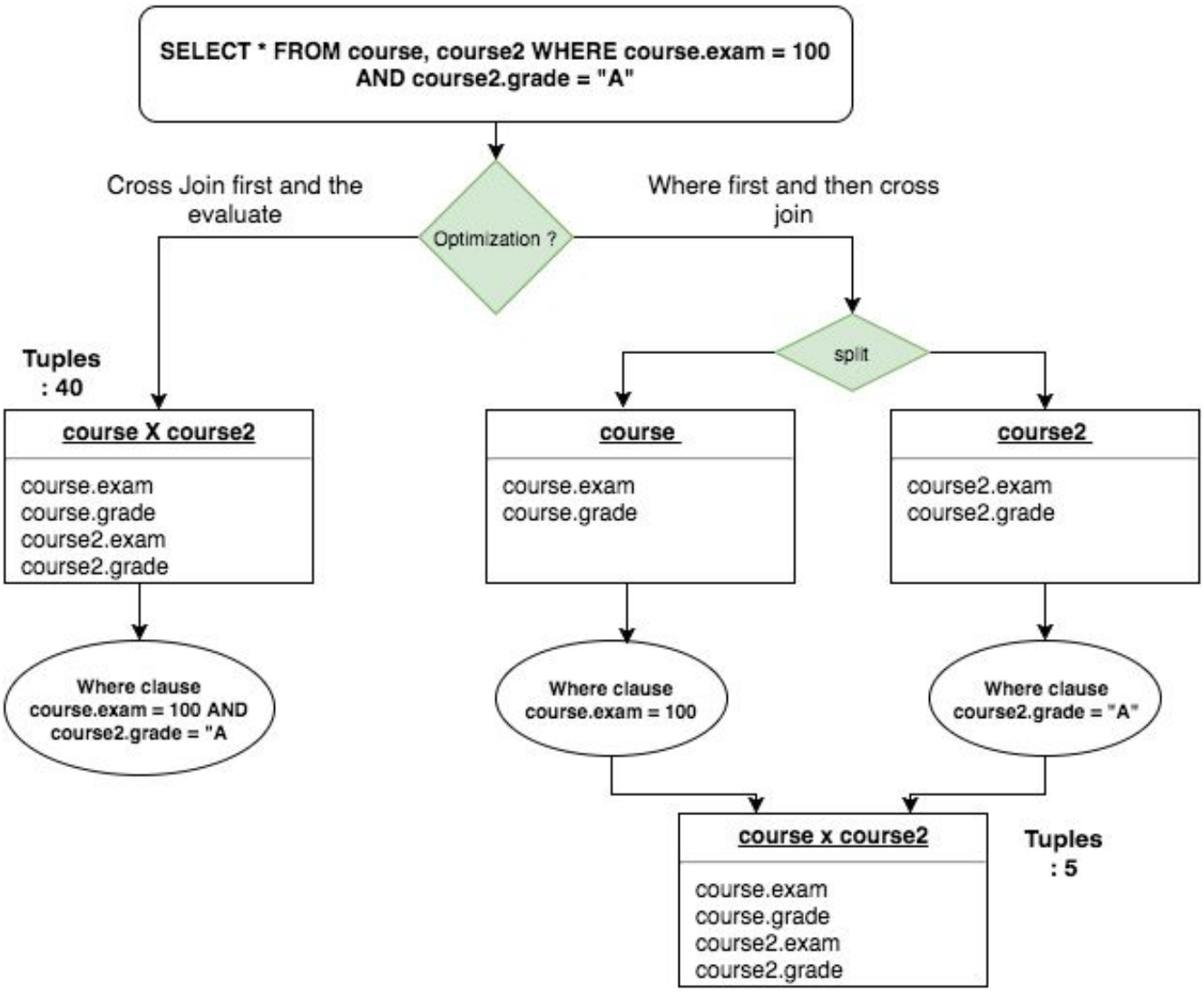
Join Cleanup:

Join relations that are written to the disk while executing the queries are later removed when there is no of use of those.

Implementing push down conditions:

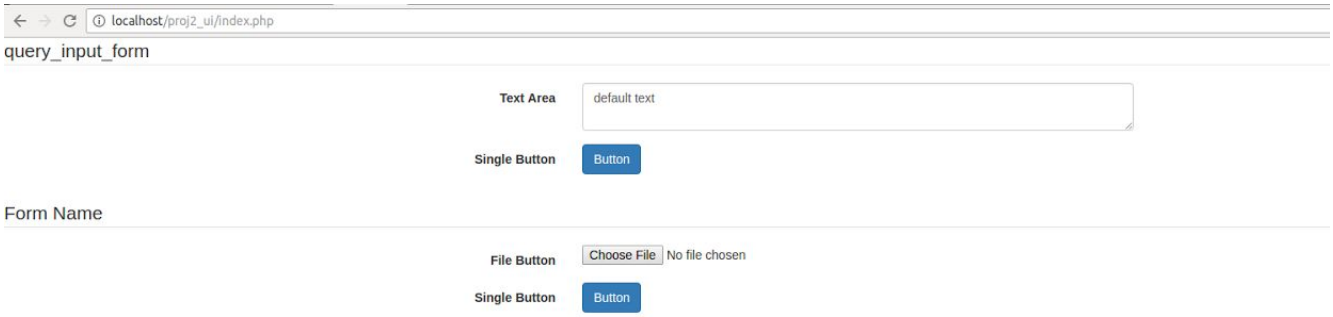
For this optimization , we load table from memory/disk for selected subquery . We have where_condition_tree where nodes reflects the expressions or AND/OR operator. When we do where condition separately, we reduce the size of cross join table. Entire optimization code flow is depicted by below diagram.

Implementing join tree optimization:

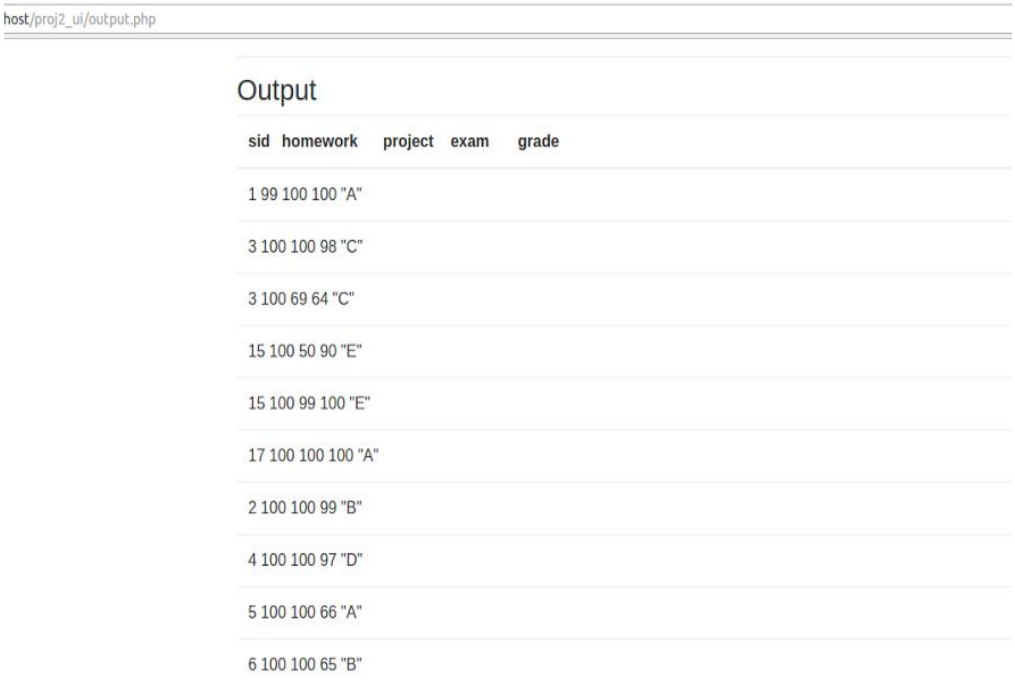


User Interface

This is the form to give the inputs:



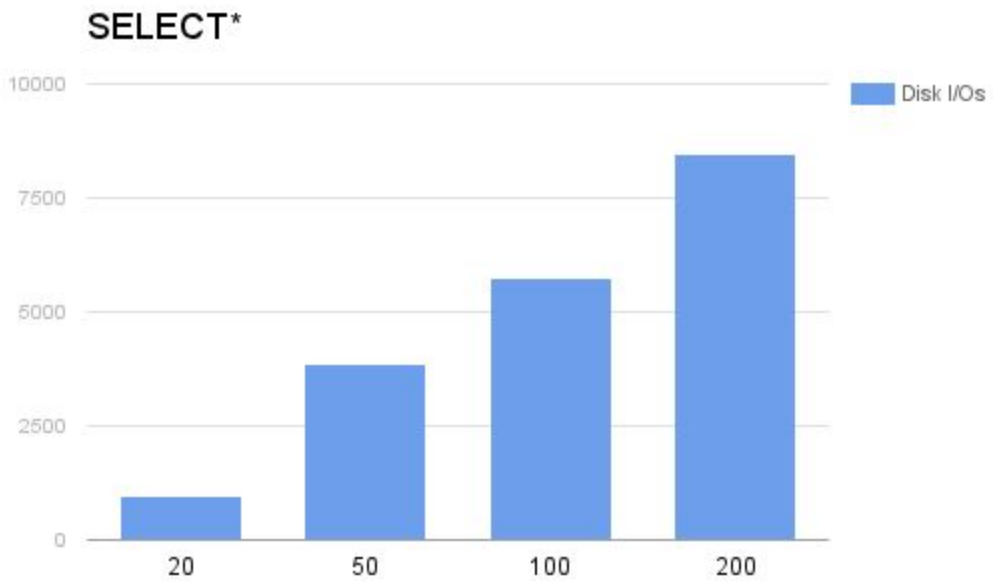
This is the output table.

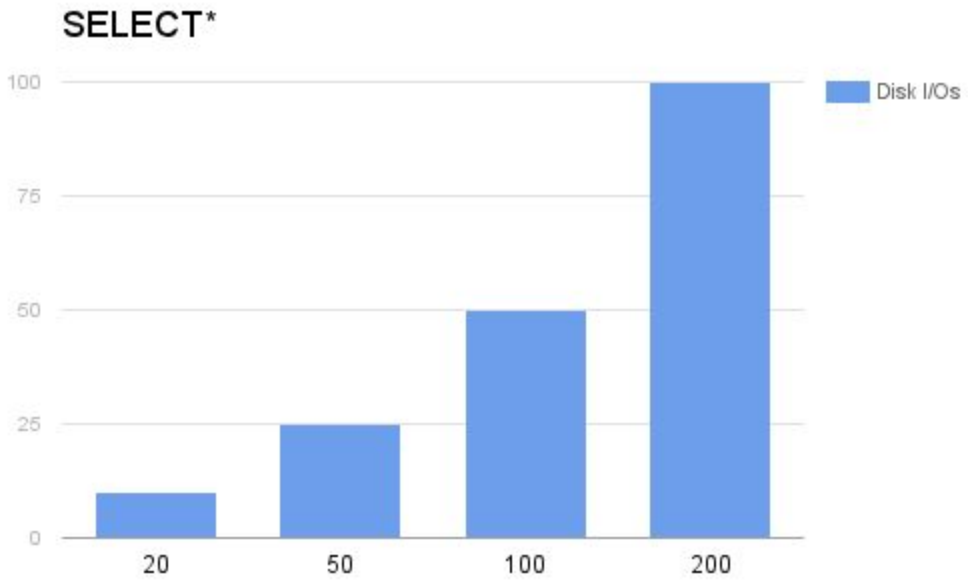


Testing and Results

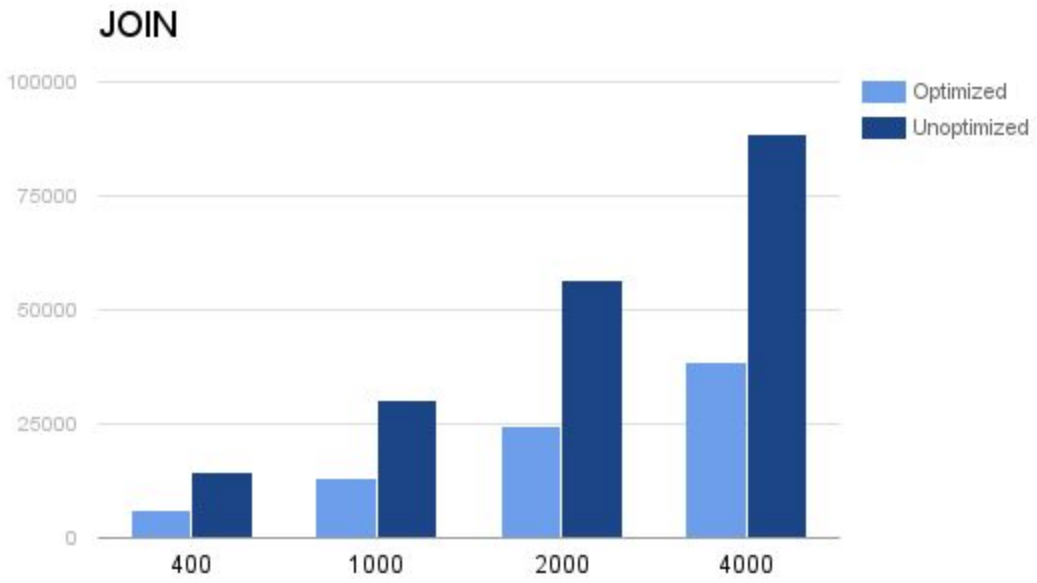
We tested our interpreter on the two tables very similar to the ones provided in the test query file. These tables are :
course(sid, homework, project, exam, grade)

course2(sid, exam, grade)





	Disk I/O's		**Execution Time**	
	B = 20	B = 40	B = 20	B = 40
A = 20	74	81	6227.0	9119.66
A = 50	165	182	13104.86	14597.46
A = 100	357	360	24523.86	26060.46
A = 200	756	780	38493.86	40986.46



Project1 data:

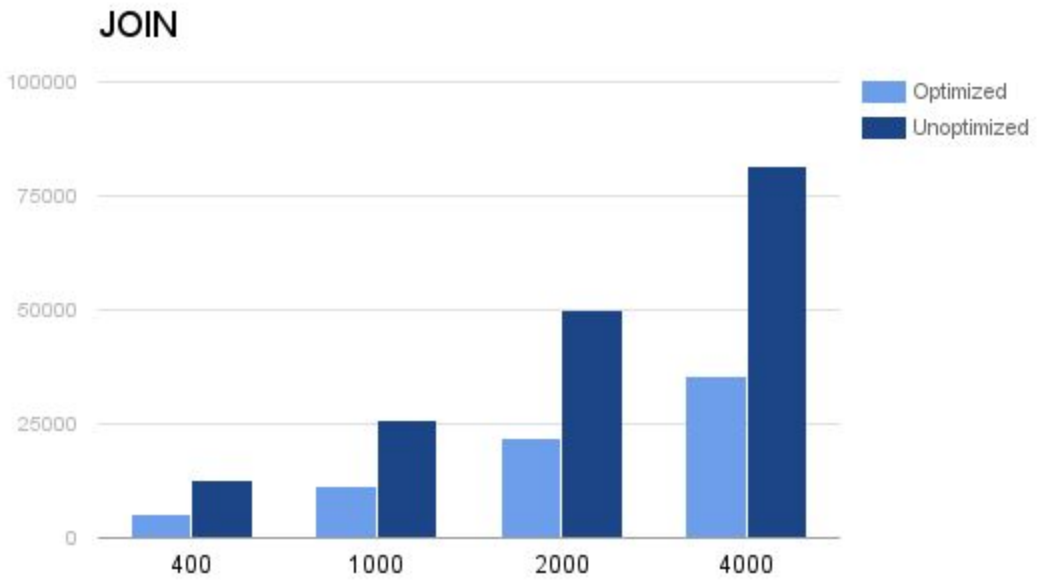
Our idea behind the project was to enable users to play puzzle solving games in groups where they can form groups and compete against each other by scoring better than others.

We tested on two of the tables of our previous project:

round(round_id, game_id, round_name)

score(score_id, score, round_id, user_id, game_id, group_id)

We evaluated some select queries on these where score table has more than 1000 tuples.



How to Run the Project

Follow these steps to run the project:

1. Unzip the submission
2. Run “./run.sh”
3. Now run
    a. “./dbparser <input_file_name>”
4. This will generate an output file with name *“output.txt”*