Database Systems

CSCE 608

Project 2

December 6, 2017

Xichao Chen
chenxichao@tamu.edu
127002358

Ruosi Lin
rlin225@tamu.edu
826009602

1 Project Description

1.1 Overview

Our TinySQL project is implemented in Java. We used the provided StorageManager library to simulate the physical execution of SQL queries. Figure 1 below demonstrates how a SQL query is processed in our system.

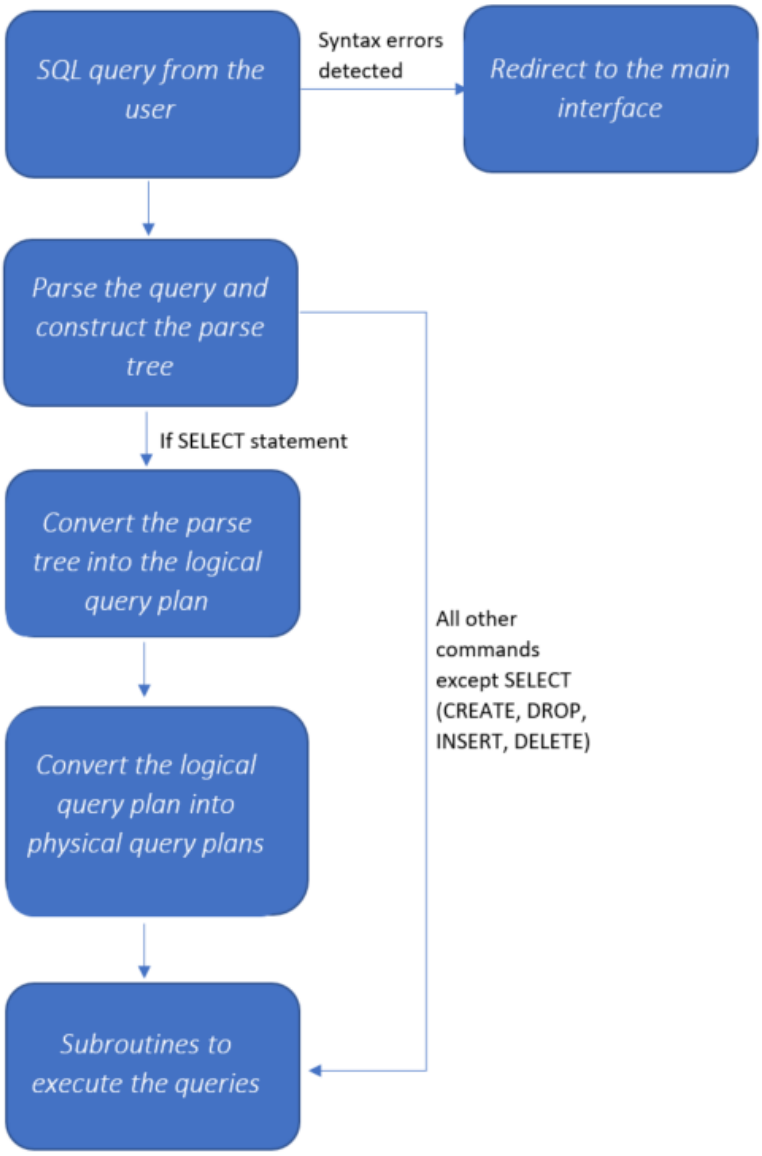

Figure 1: TinySQL interpreter workflow.

Our interpreter consists of the following four major parts:

- Parser: Given the TinySQL grammar, we fed it to a lexical analysis tool called ANTLR. We converted the TinySQL grammar into the ANTLR-acceptable format, and have tested that all components are identified as expected through the parser. In our interpreter, the parse tree is

printed out as a collection of strings. Each string represents a node; terms that are surrounded by a pair of parentheses are considered to be a child to the previous node. We utilized the defaulted ANTLR listener to traverse the parse tree and get elements for each subroutine.

- Logical query plan: This is only needed for the SELECT statement, as other statements are not part of the relational algebra. We wrote a class called LQP that is dedicated to handling the logical query plan piece. LQP generates an expression tree first, then optimizes it through traversal. The tree contains the following operators, evaluated in order: distinct, order, projection, and join.

- Physical query plan: Similar to logical query plan above, this is only needed for the SELECT statement simply because SELECT is the only statement that exhibits the characteristics of relational algebra. It is handled in the LQP class as well. Once an optimized logical query plan is generated, the physical query plan would be kicked out and traversed through the expression tree in a bottom-up manner. We do print it out on the screen so that users may see how the execution sequence changes before and after optimization.

- Subroutines: For each SQL statement, a subroutine is written to execute it. Except for SELECT, all other queries are redirected to their corresponding subroutines after the parsing step. Required elements (for example, table name and attributes in the CREATE subroutines) are gathered through ANTLR listeners. These elements are then fed into corresponding objects in the StorageManager library to simulate actions in a real DBMS. As for SELECT, a temporary relation is created at the time of generating the logical query plan to save the result set. We then print out each tuple from the result set to the screen.

1.2 Limitations and Assumptions of Our Work

- In the delete operation, currently we only support three math operators (plus, minus and times) under three situations: 1) both ends are valid column names, e.g. a + b = c + d; 2) left ends are the column names, right end is the integer for comparison, e.g. a + b ¿ 100; 3) the left end contains a single column and the right ends are column names, e.g. a ¡ c - d. Other scenarios are not included at this moment.

- ORDER BY clause could only sort on one relation. Multiple columns from different joined tables are not supported. DISTINCT supports duplicate removal in multiple relations as well as multiple attributes. However, in order to have it run correctly with multiple relations, the column names must be in the "table name.column name" format. (e.g. SELECT DISTINCT t.a, t.c FROM t, s WHERE t.c = s.c). There is no restriction on single relation duplicate removal.

- Whenever multiple relations are joined together by a common attribute, the common attribute would occur multiple times in the final result table (unless it is not part of the final selected columns). We have decided not to merge the common key together.

1.3 Algorithms

Our main algorithms for SELECT are implemented in the Algo class. One-pass and two-pass sorted based algorithms are implemented to handle the sort and distinct scenarios. Our interpreter naturally supports selection from up to two tables simultaneously; for queries with selections from more than two tables, a left-join tree is built. If there is no condition specified in the WHERE clause, we used the nested-loop algorithm (block-based) for the cross join to further reduce disk I/Os. Whenever the join condition is explicitly stated in the WHERE clause, the interpreter would execute the two-pass sorted based algorithm for natural joins. More details could be found in the discussion section.

1.4 Data Structures

We implemented a heap structure in the simulated main memory whenever natural join is required. It organizes tuples based on the designated attribute specified in the WHERE clause and pops the tuple with the minimum value. It is a wrapper around the Java PriorityQueue structure. Within the heap class, we have also implemented Map.entry, namely MyEntry, so as to store relationships between multiple elements. Specifically, a heap would contain a MyEntry record that stores the heap block id, the tuple, and the sorted key (for multiple relations) or the field id (for single relation).

As for the expression tree, we constructed it by utilizing the customized Node class. A node contains the following attributes: type (name of an

operator, e.g. DISTINCT, SORT), parameters (associated values stored in an ArrayList, for instance: from_list stores all relations names), relation name (optional, defaulted to be null), associated children (not limited to one child only, could have multiple children), and level within the tree. This structure is useful for logical query plan optimization as well as physical query plan generation.

To handle the general condition in the WHERE clause (i.e. no math operators are included), we have a class called CondElem. It is a customized list that stores the terms and compares operators in the WHERE condition.

2 Experiments and results

2.1 Standard TinySQL Test Cases (TinySQL-TextWin.txt)

Computer elapse time = 826243 ms
Calculated elapsed time = 732312.5200000405 ms
Calculated Disk I/Os = 9814
this test set contains 292 commands in total, and all of them passed with no issue.

Among all test cases, cross join is the operations that take the longest time to run, whereas create/drop table, insert tuples into tables have nearly no latency.

When conducting select statement, because of the limitation of Main Memory, a lot of disks I/O are cost in a temporary table read and write, which greatly increases the IO and elapse time.

A lot of function calls may also contribute to the elapsed time, although we have no idea how much it weights in the result.

Multiple Thread may greatly speed the program up but would increase the complexity of the program. Because of the time limit, we did not conduct this part of the program.

2.1.1 Performance Analysis

INSERT

We ran 10 test cases with the number of tuples uniformly increased from 10 to 100 and captured the calculated elapsed time (in milliseconds) as well

as the number of disks I/Os. We used the test queries on relation r. As figure 2 below shows, as the number of tuples increases, both the elapsed time and the number of disks I/Os uniformly increase as well. The relationship is linear because INSERT is a tuple-based operation.

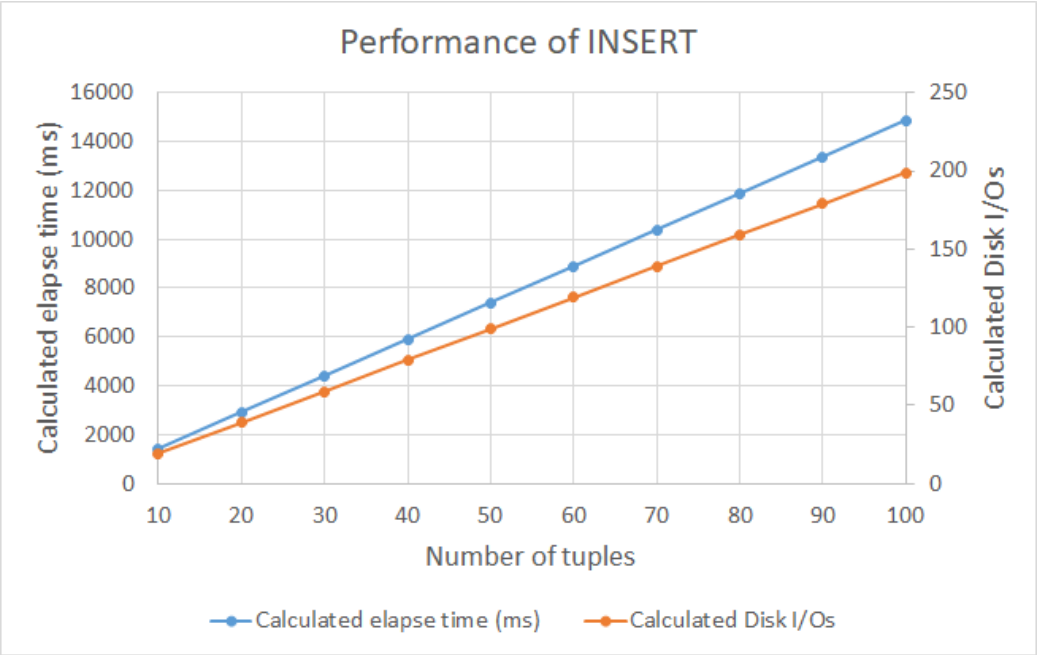

Figure 2: How number of tuples affect the calculated elapsed time and disk I/Os on insertion.

DELETE

Similar to INSERT, we also checked to see how DELETE performs. In our 10 test cases, each time we first inserted 10 to 100 tuples, then run the DELETE statement with condition (i.e. DELETE FROM r WHERE a = 0 OR a = 1 OR a = 2 OR a = 3 OR a = 100) to remove all tuples. This would result in holes in the final table structure, but that could be handled in the holeTracker as mentioned in the next section. We would like to test if conditions in the WHERE clause works as expected, though at the expense of adding additional disk I/Os.

Figure 3 below shows a similar pattern as INSERT; that is, as the number of tuples grows, the time it takes to remove all the tuples that satisfy the condition as well as the number of disks I/Os increase. Notice that we only plot the "actual" elapsed time and the "actual" disk I/Os, where the

INSERT part is isolated from consideration. One interesting side note is that deleting 100 tuples only required 50 disks I/Os, whereas inserting the same # of tuples would need 199 disks I/Os. That is because whenever inserting a new tuple into a block, our interpreter needs to read the block from the disk and do a write back. This takes roughly 2 disks I/Os per tuple inserted. But for DELETE, our system directly operates on the number of blocks existed in a relation. Therefore the disk I/Os purely depends upon the relation size. We used relation r in our test case, which contains only two fields (so 5 tuples per block). In that sense, delete 5 tuples would only take 2 disks I/Os (read from the disk and write back to the disk), so it saves a lot of time.

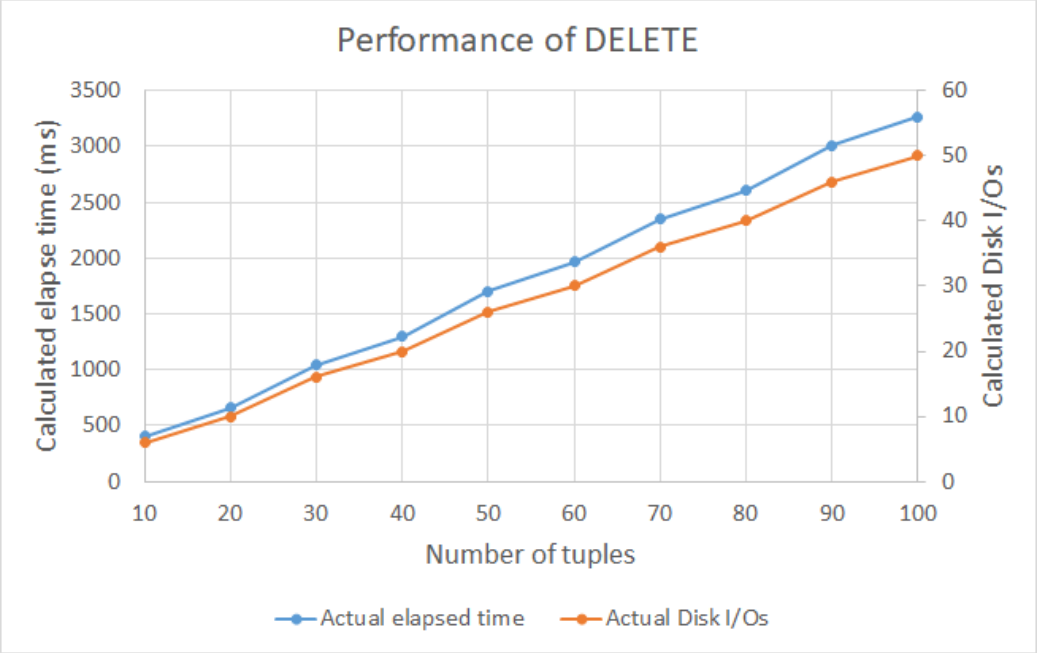

Figure 3: How number of tuples affect the calculated elapsed time and disk I/Os on deletion.

JOIN

We test the performance of CROSS JOIN and NATURAL JOIN of TinySQL and captured the calculated elapsed time (in milliseconds) as well as the number of disks I/Os. The test cases can be found in the appendix. The relations are r and s, here are the tuples we use. As figure below shows, whether the size is 10 or 15, both the elapsed time and the number of disks I/Os of CROSS JOIN do not change a lot. But that is totally different for NATURAL JOIN.

Without optimization, the number of disk I/Os of tables of size 15 is 1188, and its elapsed time is 88660.44ms; the number of disks I/Os of tables of size 10 is 531, and its elapse time is 39628.53ms. After optimization, the number of disks I/Os of tables of size 10 reduce to 67 (87.39% decrease) and its elapsed time reduces to 5000.21ms (87.39% decrease); the number of disk I/Os of tables of size 15 reduces to 157 (86.78% decrease) and its elapsed time reduces to 11716.91ms (86.78% decrease).

In optimization program, we not only push down the SELECT node, but also sort tables once the operation on them is NATURAL JOIN, so we can see optimization does not affect CROSS JOIN greatly, but reduces the running time and disk I/Os dramatically.

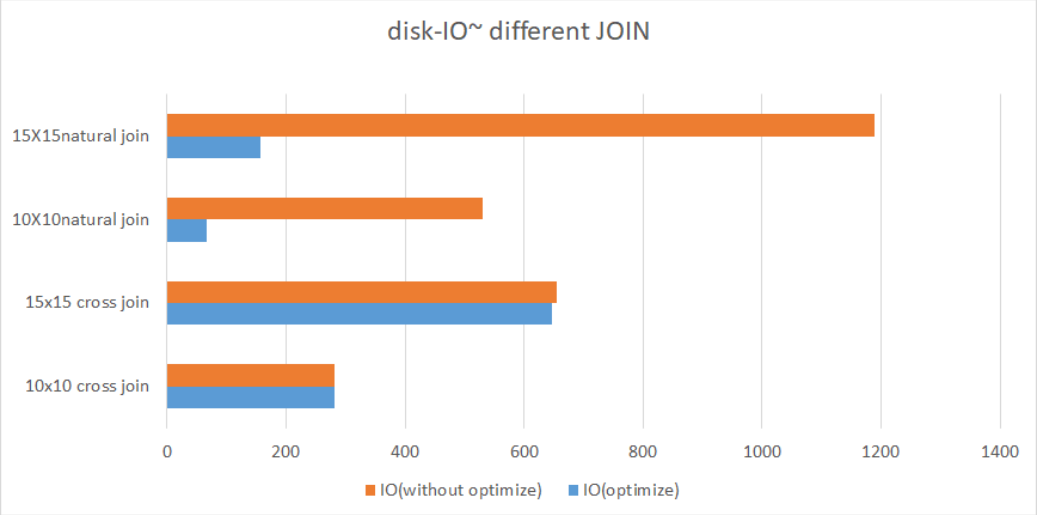

Figure 4: How disks I/Os change before and after optimization.

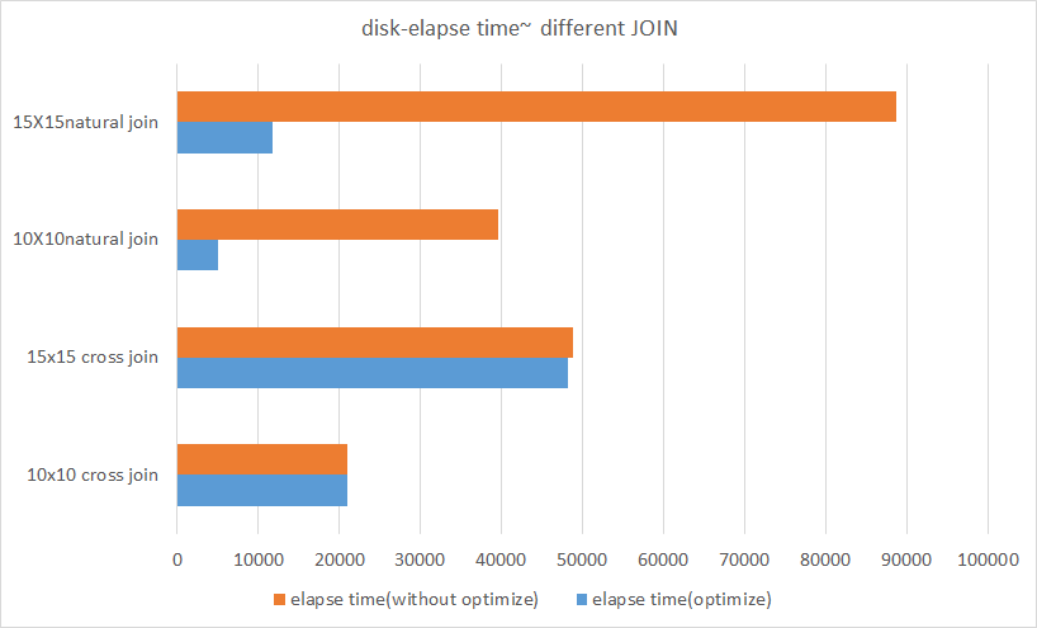

Figure 5: How calculated elapsed time changes before and after optimization

2.2 Customized TinySQL Test Cases

2.2.1 TinySQL-mytest100.txt

This test set contains 100 tuples in total, randomly permuted from three tables: restaurants, category and review. See the text file in the zip bundle for details. All test cases passed with no issue.

Runtime result:
Computer elapse time = 453974 ms
Calculated elapsed time = 299458.5200000135 ms
Calculated Disk I/Os = 4014

Among all test cases, cross join is the operation that takes the longest time to run, whereas create/drop table, insert tuples into tables have nearly no latency.

2.2.2 TinySQL-mytest200.txt

Similar to the mytest100.txt, this test set contains 200 tuples in total, randomly permuted from three tables: restaurants, category and review. See the text file in the zip bundle for details. All test cases passed with no issues.

Runtime result:
Computer elapse time = 1408219 ms
Calculated elapse time = 1250639.5699997598 ms
Calculated Disk I/Os = 16761

Among all test cases, cross join is the operation that takes the longest time to run, whereas create/drop table, insert tuples into tables have nearly no latency. Also, the time it takes to execute the queries is significantly longer than the 100 tuples cases. Cross join mostly contributes to the increase in time, as it created a temporary relation with 5144 tuples while the other test set only has 989 tuples in the intermediate relation. Through this experiment, we realized why natural join is always preferable.

3 Discussion

We implemented most of the optimization techniques in the Algo class, with the exception of empty block handling due to deletion. It is saved in the Main class.

Notation used:

$B(R), B(S)$ - number of blocks in relation R/relation S

$T(R), T(S)$ - number of tuples in relation R/relation S. $B(R) <= T(R), B(S) <= T(S)$.

3.1 Hole handling and clustered relations

When tuples are removed from a relation, holes may appear due to the block being empty after the deletion. Initially, our solution was to ignore all holes and insert tuples at the end of the relation. This, however, caused a relation to be really sparse. Moreover, a table might not fit into the main memory even if it only contains few valid records (with the whole table size less than the maximum size of the main memory). We eventually decided to avoid as many holes in our relations as possible. This was achievable

by a hash map structure called holeTracker in our program. In the given operations, only would deletion introduce holes to the database. Therefore, whenever a tuple is wiped from a block of a relation, the holeTracker records the name of the relation as well as the block number where the removed tuple belonged to. Upon completion of a delete action, if a block still holds some tuples, it will be inspected and reset to ensure that no hole would occur within this block. On the other hand, if the deletion causes a block to be empty, all tuples within that block will be invalidated with holes written back to the disk. When insertion into the same relation is called afterward, it will check against the holeTracker to see if blocks within the table have spaces for inserting new tuples. If so, these empty slots will be filled first. Our goal is to reduce as many holes/unused space within a block as possible. Using the holeTracker instead of scanning the whole relation to look for an empty space every time saves a lot on disk I/O. Previously, inserting a tuple into a 5-tuples relation (assuming each tuple takes a block and a hole in the first block) would take 65 disks I/O. Now the number is reduced to 21 in our program.

3.2 Expression tree: pushdown selection

When optimizing the logical query plan, we attempted to push down selections further down on the expression tree along the process. However, there are cases that we could not push down selection when joins are presented. In that case, our interpreter would evaluate if it is possible to combine the selection with joins so as to turn a cross join into either theta join or natural join, accordingly. This can greatly reduce the intermediate relation size.

3.3 One-pass algorithms

3.3.1 One pass sorting

One pass sorting is used when there is an ORDER BY keyword in the select query, and the relation of interest fits into the main memory. The whole relation would be read into the main memory and inspected on a block by block basis. Our interpreter would write out the new sorted relation to the disk and return the number of disk blocks used, which is equivalent to the cost of this sorting operation.

3.3.2 Optimization on Distinct

Similarly, when the DISTINCT keyword is detected in a select query, our interpreter would attempt to read the whole relation block by block and remove duplicate copies using HashSet. The result would be saved into a new relation on the disk. This optimization is applicable disregarding the relation size. The number of disks I/O is reduced from $T(R)$ to $B(R)$.

3.4 Two-pass algorithms

3.4.1 Two pass sorting

When the ORDER BY keyword is detected but the requested relation(s) cannot fit into the main memory, two pass sorting is called. It first sorted the relation(s) and save the resulting sublists to the disk as the one pass sorting does. An ArrayList is used to keep track of the sublist head positions in the disk. It then merges the sublists in the main memory and exports a temporary relation. Since sorting is a unary operation, the hardware cost is reduced to $3B(R)$ instead of $T(R)$.

3.4.2 Two pass sorted based natural join

Since we completed the two-pass algorithm for sorting, it is natural to use it for the natural join. Natural join is preferable than cross join due to the smaller size of the intermediate relation. However, this is not always achievable in TinySQL. If users do not specify the join condition, then cross join has to be used. Optimization for cross joins can be found in the next section. Our interpreter would switch to the natural join mode if there are more than one relation in the "from" list and the join condition is given under the where clause (e.g. SELECT $*$ FROM r, s WHERE r.a = s.a). The algorithm first checks if the products of the two relations sizes are less than the number of blocks available in the main memory. If not, then there is no need to use two-pass and the interpreter would return to the main interface. Once tuples from both relations are joined through the common attributes, a temporary relation is created. The naming convention for the columns in the temporary relation would be original_table_name.original_column_name, e.g. r.a. The only exception would be the joined attribute, whose name stays the same. The "sorted-merging" nature makes the natural join more appealing than the cross join, as its total cost is only $3(B(R) + B(S))$.

3.5 Nested Loop for Cross Join

Cross join is the most time-consuming operations in our TinySQL interpreter so far, as it concatenates each tuple from the left relation with every single tuple from the right relation (assuming that only two relations are cross joined in this case). Since the size of the joined relations is too big to fit into the main memory, one pass algorithm does not work. Neither the hash-based nor sort-based two pass algorithms help with the cross join operations as well, based on the definition of the cross join. Therefore, we implemented nested loop algorithm to handle the cross join case. At the extreme case, the cost can be reduced to $B(R) * B(S)/M + B(R)$ when the smaller relation can have as many blocks to fit into the main memory as possible. We, however, did not implement this. Our interpreter would loop through two relations on a block basis. The outer loop would inspect each block of the smaller relation, then check blocks of the larger relation in the inner loop. In this way, our disk I/O has been reduced from $T(R) * T(S)$ to $B(R) * B(S) + B(R)$.

4 Appendix

4.1 Performance Analysis - JOIN test cases

We used the following test cases to check how JOIN behave in our interpreter:

```
S: [b, c]
(0)    0    0
(1)    2    2
(2)    3    3
(3)    0    0
(4)    0    0
(5)    1    1
(6)    1    1
(7)    2    2
(8)    2    2
(9)    3    3
(10)    3    3
(11)    3    3
(12)    101    101
(13)    101    101
(14)    101    101
R: [a, b]
(0)    0    0
(1)    1    1
(2)    1    1
(3)    2    2
(4)    3    3
(5)    0    0
(6)    1    1
(7)    2    2
(8)    100    100
(9)    100    100
(10)    100    100
(11)    100    100
(12)    100    100
(13)    100    100
(14)    100    100
```