Advanced-DBMS-Implementation

Project for course K23a "Software Development for Information Systems".

Authors

Lampros Smyrnaios
Petros Morfiris
Giorgos Theodosopoulos

Description

The task was to evaluate batches of join queries on a set of pre-defined relations.
Each join query specifies a set of relations, (equality) join predicates, and selections (aggregations).
The challenge was to execute the queries as fast as possible without (much) prior indexing.

The Project

The project was split in three phases:
1) In the first phase, we worked in a sub-set of the project in which the program used the RadixHashJoin methodology to join 2 static tables that were used as relations. In the end, the result of the join was printed in stdout.

2) In the second phase, we expanded the program to read dynamic data (tables) from multiple files and map them in memory. After the tables get loaded, the program reads queries from the stdin, parses them, executes them and prints the results to the stdout.

3) In the third phase, we optimized the application by adding multithreading and join ordering. We implemented statistic gathering for each table that is being mapped, as well as statistic gathering for each query that is to be executed. We use these statistics to find the best join order, thus reducing the amount of intermediate table results that are created while joining the tables.

Additional/Bonus optimizations in the third phase

In the third phase of the project, we added additional/bonus optimizations to reduce the query execution time.
The optimizations were the following:
1) Multi-threaded SUM-calculation.
2) Blocking the execution of individual queries which are guaranteed to produce zero results. During the statistics gathering, when a filter of a query is guaranteed to produce zero results, all other predicates will produce zero results too, because the query is a connected graph.
3) Remove duplicate joins inside the queries.

Program's flow of execution

When the program runs, it does the following:
1. Sets the I/O Streams.
2. Initializes the Threadpool.
3. Loads the tables from the files into the main memory.
4. Creates the Relations-array (used for hosting the columns of the tables for joining them).
5. Creates the sums-structure (used for hosting the results from each query).
6. Receives and processes the queries.
7. Destroys the Threadpool and de-allocates memory.

Explanation of main program-phases.

Table Loading

In order to load the tables, the program first receives the names of the tables from stdin and then maps each table to the main memory and calculates the initial statistics.
These are the statistics that are calculated for the numeric-data of each column of each table:
1) The lower (min) value.
2) The upper (max) value.
3) The count of all the values.
4) The count of the distinct values.

We use these statistics to reorder the joins of the queries to achieve faster join-execution.

Query Processing

Parsing

Every query being read from the stdin gets parsed and its members (**FROM** - **WHERE** - **SELECT**) are saved in a query-structure.
The query members are:
1. The "**RelationIDs**" which are the names of the tables which participate in the query along with their aliases.
2. The "**Predicates**" which contain the filters and the joins, based on which the results will be calculated.
3. The "**Selections**" which contain the table and the column numbers that the SUMs will be calculated on.

Statistics & Best Tree Join-order

After parsing a query, we gather statistics for the predicates in order to predict the best order for the joins to be executed in.
In order to find the best join-order, we first gather the statistics for the filters and then we gather statistics for every join depending on the join type (self join, radixhashjoin, auto correlation, etc). Firstly, the BestTree function creates a adjacency matrix from the joins of the query, where every element is 1 or 0 depending on if it's adjacent with another or not. Then, it needs to find best sets that contain 2 Relations (1 join), 3 Relations (2 joins), etc... Bitwise operations are used to judge between different sets(for example 0011 is the set (0,1) and 0101 is the set (0,2)). The sets of 2 are created from every individual join. When a set is found, then the statistics are gathered for that set and the cost is calculated. After the sets of 2, come the sets of 3 where the same idea applies (same for sets of 4, etc). One set is better than another if they are the same number, for example [(0,1), 2] and [(1,2), 0] are the same number 0111 but the order of the joins is different, meaning the cost of joins is different and the set that produced the smallest number of intermediate results is the better one.

After finding the best set/best order of joins, the program reorders the joins in the query_info accordingly and it passes it to the function that is going to execute the query.

Execution and results

After re-ordering the joins (if needed), we execute the query. The filters are executed first and after their execution one or more (if there are more than one filters) intermediate tables get created to store the results (row ids) produced by the filters.
Afterwards, we execute the joins one-by-one (their order will be the best since it will be changed after the **Best-Tree** optimization).

There are ***3*** types of joins:
1) If none of the relations that are being joined exist in an intermediate table, join them and save the results in a new intermediate table.
2) If only one of the relations exists in an intermediate table, get the rowIDs from the intermediate table, get the real data from original relation, join the 2 relations and save the new row IDs in the already existing intermediate table.
3) If both relations exist in an intermediate table, there are 2 options. One, if they are in the same intermediate table, you have to do a "filter", column to column comparison and keep only the rows that the columns are equal. And two, if they are in different intermediate tables, you have to join the 2 relations and then "combine" the 2 intermediate tables into 1.

In the end, we will have the produced results in the final intermediate table (only one because every query is a connected graph). Those results will then get SUM-med depending on the query-selections and those SUMs will be printed in stdout.

Experiments we did.... (H1, H2, MULTI-THREADING, JOINED_ROW-IDS_NUM)

H1_PARAM: After some experiments, we found out that by assigning it to 3 we get an improved execution time.

H2_PARAM: We chose the value 251 as the h2 value, even though it doesn't make a huge different compared to 151 for example.

Multithreading: For the "parallel" relation histogram creation, relation partition and join of relations, we use number of threads equal to the number of buckets, which means 8 (2 ^ H1_PARAM). But for the parallel SUM calculation, after some experiments, we found out that splitting the sums at 32 or 64 parts is a better number than just splitting it at 8 parts (number of threads). The time of the SUM execution was improved significantly from ~0.55 to ~0.30 (45%).

JOINED_ROW-IDS_NUM: The size of every result was changed from 1MB to 128KB after adding multithreading because each thread had its own results, were as in the past all the results of the join were written in one result struct.