A Novel, Low-Latency Algorithm for Multiple Group-By Query Optimization

Report on the implementation of the algorithms from the 2016 ICDE Conference paper (Phan, Michiardi)

Daniel Kane
daniel.kane7@gmail.com

Ethan Letourneau
eml14g@my.fsu.edu

I. PROBLEM & CONTRIBUTIONS

Phan and Michiardi work in the area of optimization of data summarization with their 2016 ICDE paper [1], for which they claim the Group By operation is the fundamental building block. The problem addressed and solved by the paper involves efficient solving of a set of multiple Group By queries – namely, it is pointed out that no prior work on the subject is able to effectively scale well with both a large number of queries and very high dimensional data, i.e. tables with large numbers of attributes. The algorithm proposed, the Top-Down Splitting (TDS) algorithm, can scale well with both considerations, finding equally or more optimal solutions than competing works, and executes in a faster time than those competitors, some by several orders of magnitude.

Our contributions to this problem are in the form of an implementation. Using the C++ programming language, we've implemented the algorithms of the paper and meshed them together to create the TDS algorithm. We've additionally built programs to simulate the template Group By query provided by the paper naïvely – that is, without any optimization, meaning the set of Group By queries fed to the program are drawing directly from the base table each time – as well as to utilize the execution plans built by the TDS algorithm. We have simulated partial experimentation detailed in the paper, that being the execution of all possible two-element Group By queries on the table of the dataset (being the "lineitem" table from the TPC-H benchmark) for both the TDS-implementing and naïve programs on several magnitudes of sizes of the table. We detail the results of the experiments later in Section IV – in short, our implementation worked as intended, and saw a 44.28% average decrease in runtime from the naïve approach to the optimized approach we implemented.

II. RELATED WORKS

Several papers predating ours have tackled efficiency in data summarization, including many that specifically deal with multiple Group By queries as ours does. The Bottom-Up Merge (BUM) algorithm introduced in [2] is one of the primary motivators for the TDS algorithm. Like the authors of our paper, the authors of [2] acknowledge the computational hardness of the problem and employ approximation techniques to execute multiple Group By queries efficiently. It is similar to the TDS in finding an execution plan in the form of a directed acyclic graph, wherein the edges represent the cost of computing the child grouping from the parent grouping. The solution boasts a 4.5 times speedup over a naïve (no optimization) approach but does *not* scale with a large number of queries. Figure 1 on the next page shows the execution time for several approaches including BUM with increasing numbers of queries, for which BUM seems to take a near-linear growth, several orders of magnitude more than either implementation of TDS.

There is also a difference in cost definition between works – [2] identifies two different cost models, the first being cardinality (the cost of an edge e = (u,v) being the cardinality of u) and the second being the cost of an execution plan as the summation of the costs of the queries in it based on the estimation from the DBMS's query optimizer. Our paper's "scan cost" is similar to the cardinality cost, but we have no DBMS-based cost plan. Additionally, [2] only uses one of these costs at a time in the execution plans, whereas the TDS uses both sort and scan costs in tandem.

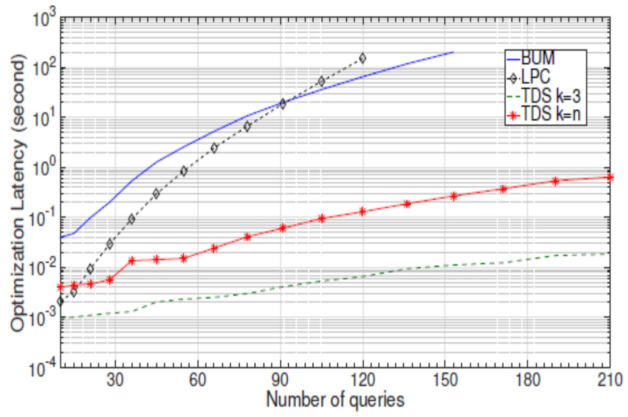

Figure 1: Time to compute optimal solution for growing number of queries

Another work semi-indirectly similar to our paper is [3], the paper that introduces the Lattice Partial Cube (LPC) algorithm. The authors of our paper spell out that both Cube and Rollup operations are different ways to express different types of multiple Group By queries, so the optimization of either can relate to the optimization of the latter. [3] asserts that constructing the full cube in such queries can require large computation costs, and since some data might not be required in a cube, they address the need for efficient computation of partial data cubes. LPC creates a schedule tree in a top-down method for cube construction, similar to TDS. Accordingly, it is noted that the solution trees obtained from TDS in our paper and from the LPC method are nearly identical. TDS champions a superior runtime regardless because, unlike LPC, it does not consider the entire space of additional nodes – only those that can be split evenly by k, a parameter set by the user. Thus, in our implementation approach, there will be at most k fan-out. The authors of our paper note the correlation for using k, stating their observations: "state of the art [efficient] algorithms…have a relatively low fan-out k."

For a growing number of queries, the LPC algorithm shows a linear growth in execution time as shown by Figure 9 in [1], similar to that of BUM. The more staggering shortfall in performance is with a growing number of attributes – it is shown that LPC scales exponentially in time taken to calculate the execution plan with this on single-attribute Group By queries, while the TDS algorithm is shown to remain close to zero shown in Figure 2 below.

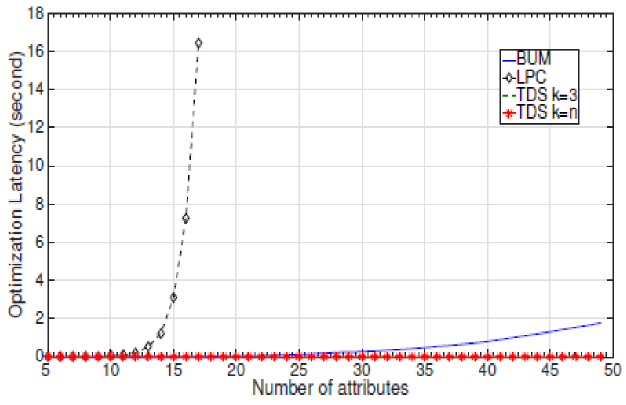

Figure 2: Time to compute optimal solution for growing number of attributes

BUM and LPC are the two most similar works that can be compared to that of our authors, and thus they were the two algorithms that the authors tested TDS against. However, in the implementation we've done, we do not test against these, as implementations that mirror ours for TDS are not openly available for either algorithm. So, as stated before, we test against only the naïve approach, and on the metric of total runtime.

III. IMPLEMENTATION

A. Algorithm Description

Before the running of TDS, a directed acyclic graph (DAG) that is to be passed into the algorithm is constructed; it is denoted as G=(V,E), where every v ∈ V represents a grouping (Group By query) and each e = (u,v) ∈ E denotes that the grouping v can be computed from the grouping u. Each edge additionally carries a cost for the computation, either being a scanning or sorting cost. The authors defined the scanning cost for any e = (u,v) would simply be the cardinality of u (|u|), while the sorting cost would be equal to $|u|*\log_2(|u|)$. We went along with this cost model in our implementation; however, any appropriate cost model can be plugged into TDS. The initial DAG described is to have only the root node (the data table) and the "terminal nodes", the groupings that are requested by the multiple Group By queries. We build this by first sorting the groupings by their cardinalities, and then adding them one-by-one to the DAG, making them children nodes of the node in the DAG yielding the least cost. The Group By queries that were used by the authors (and us) were all of the following template:

SELECT s_i, COUNT(*) FROM T GROUP BY(s_i)

In this, T is the table we query from, and s_i is one or more attributes in the table.

The aforementioned initial DAG, after construction, is then iterated through to perform the *fix_scan* function on each node, which looks at all of the outgoing edges of the node, and switches the cost associated with at most one of them that brings the most cost reduction to scanning cost (all of the initial costs are set to sorting). After this, the initial "naïve" execution plan is completed. This DAG is then fed into the actual Top-Down Splitting Algorithm.

Top-Down Split takes in the root node of the preliminary DAG we constructed and some number k to be discussed later. All this part does is repeatedly call another function, called Partition Children, on the node passed in, until the function returns false (returns true as long as it can further optimize the children of the node, split them into more subsets). Once returning false, TDS is recursively called on all children of the node.

Partition Children attempts to split the passed-in node's children nodes into at most k subsets, represented by a new node that is the union of all attributes in the subset. From the authors, "the intuition is that, instead of computing children nodes directly from u, we try to compute them from one of these k additional nodes and check if this reduces the total execution cost". It tries each possible split, from 1 to k subsets, and picks the one of least cost that is less than the cost of the tree as it already is. The actual dividing of subsets is done by yet another function, Divide Subsets.

Divide Subsets divides the children nodes of the passed-in node into k subsets – first sorting them on cardinality and then adding each node into a subset that yields the least cost. We attempt to split them into every possible combination returning at most k subsets. The authors present a second version of this algorithm wherein k is flexible, and more than k subsets can be returned if adding a node to any of the possible subsets would make it equal to the passed-in node, it is put in its own new empty subset. We do not bother with the initial version of this function and only implemented this version, as the authors proved it to be more effective.

B. Implementation Description

We implemented the entirety of this algorithm in the C++ programming language, meaning that we did not extend a DBMS with it. To do that, we would have had to override a query optimizer of some sort, and at our level of knowledge of DBMS's we could not do so proficiently. As such, we built the following:

- A program to simulate the template Group By query, taking in a table name and vector of attributes.
- An implementation of a naïve solution to multiple Group By queries, wherein each query is executed one after the other (naïve.cpp).
- An implementation that takes in a query execution plan tree (which we build from the TDS algorithm)
- Functions to implement each of the functions described in the TDS algorithm

The C++ programs manipulate text files that simulate relational database tables, wherein each line in a file is a single tuple, each attribute separated by a "|". The tables were generated from the TPC-H benchmark's "lineitem" table, the files for which contain 100 rows of the table each in the above-mentioned format – we concatenated them together into files of various sizes.

We built the program to simulate the template grouping based on the notion that all the template does is find unique groupings of the provided attributes and count how many of each of them there are. So, we made a C++ Struct to have a vector of strings for the grouping and an integer to count the number of instances. We used a vector of these Structs and ran through each line of the table – if a grouping not in the vector was encountered, it was added, and the count was set to 1. If it was already in the result set, the count was incremented. The naïve solution then just successively ran this on each grouping that was requested, with no intermediate tables constructed. We made this naïve solution in order to have a case to compare the TDS-acquired results to in terms of runtime, to see if we were able to improve upon it with our implementation.

In both the preliminary tree construction and the TDS algorithm, we are tasked with sorting groupings on their cardinalities. Of course, to get an exact cardinality for groupings we would have to actually compute them, which would defeat the purpose of the algorithm. The authors state in the original paper that their algorithms assume the cardinality is readily available, and that they utilized outside methods to estimate the cardinalities of groupings. So, we built our own simple cardinality estimator function – it essentially picks a random row from 1 to 20,000, and then looks at the following 1,000 rows, alternatingly as to look at only 500 rows. We insert each rows' grouped attributes into a C++ Set, which does not allow duplicate entries, and then divide the size of the set by 500. This gives us a number from 0 to 1, that when multiplied by the size of the table, can give us a rough estimate of the number of rows that would be in that grouping based on the sample data. When comparing the estimated cardinalities to the real cardinalities of groupings, we found that this uncomplicated estimator sufficed in returning values close enough to the real cardinalities, and erroneous sorting was unlikely to happen.

Additional to the cardinality estimator, we have utilized other helper functions to run the algorithms of the paper more smoothly. Housed in the "utils.cpp" file we have the aforementioned estimator, along with functions to translate column names to their indices and vice-versa (for file reading), a function to tokenize a string on delimiter "|", which is used to parse the table files and was modified from a function found in [4]. Also housed in this file is the structure we use to represent the DAG's, being a "Node" struct containing a string vector to represent the columns to be grouped, and a vector of Node objects representing the children nodes of the node. This Node structure is used throughout the implementations of the algorithms to act as the tree structures. We also made small helper functions for calculating sort and scan costs from a node to another node. Additionally here we have a function to compute the cost of an entire tree, "get_cost", used in the Partition Children algorithm.

The implementations of the five algorithms of the paper are hardly changed from their descriptions in the original paper (which have been explained in the previous section), simply translated to real code. The Top-Down Splitting function is nearly line-by-line identical to the pseudocode from the original paper. Each of the other algorithms utilize some helper functions (previously mentioned or too trivial to mention) that encompass the general goals of what the original algorithms intend. Worth mentioning is that, as stated before, we implemented the "k = n" version of Divide Subsets, so while the initial call of TDS has the value of 3 passed in for *k*, it does not make a difference in the dividing of subsets later on.

IV. EVALUATION & RESULTS

As mentioned previously, the dataset for the experimentation is in line with what the authors of the original paper used, being the TPC-H benchmark's largest table, "lineitem". The experimentation from the authors on the metric of total execution time was done by leveraging all possible two-attribute Group By queries upon the table – with 16 attributes in the table, this comes out to 119 queries, but we excluded attributes that, when paired with other attributes in a group by, commonly make the cardinality equal to or near that of the original table from this due to it being the primary key of the table (writing Group By queries with such would yield groupings with cardinalities equal to that of the original table, and due to the nature of the implementation, this would yield extremely lengthy runtimes). So, we use only the attributes that come out to relatively low cardinalities by themselves, those being the attributes on (with indices being 0 through 15) indices 3, 4, 6, 7, 8, 9, 13, and 14 and leverage all possible two-attribute queries on those, coming out to 28 queries. Once we have all of these, we randomly pick from the set to execute, and run an increasing number of them together. For our experiments, we run 2, 4, 6, 8, 10, 20, and 25 of them at a time on both the naïve execution and using the TDS to compare the runtimes. We have the pool of groups stored in the "groups.txt" file and used a random number generator to pick the groupings for each experiment (also recorded in said file).

The TPC-H benchmark has a database generation tool, "dbgen", which allows for the generation of randomized rows for any of the tables in the benchmark. So, we generated a random set of 184,000 rows for testing purposes to ensure the algorithms of the paper were working correctly and generated a second table of 2 million rows for experimentation purposes – the random sets of two-attribute Group By's were run on this table. The experimentation was done on the personal computer of Mr. Letourneau, which runs on an Intel Core i7-8700k processor @ 3.7 GHz (with a 10% factory overclocking), 16GB 3200 MHz physical RAM, with all files stored on a 512 GB Intel M.2 Solid State Drive.

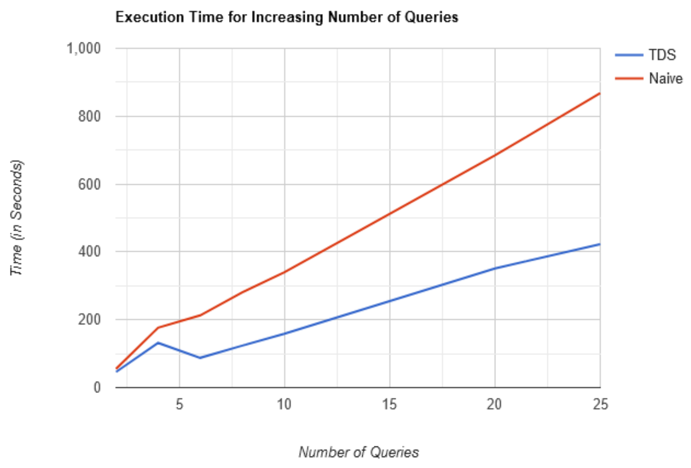

Figure 3 - Execution time for increasing number of Group By queries, our implementation vs. the naive solution.

As shown in Figure 3, the execution time for an increasing number of Group By queries computed by our naïve approach had a linear growth, whereas the growth of the optimal solution following our TDS-computed execution plans grew much slower. As the number of queries grew, the opportunities for more optimization over simply executing each query from the original table in order grew as well, resulting in the decreased time of execution. The exact execution times of each run are found in the "results_summary.txt" file in our repository (original reportings of these times are found at the bottom of each naïve run output file, and at the bottom of each of the TDS run's final grouping output file). In summary, the average decrease in execution time between the naïve and TDS solutions was 44.26%. The largest set of multiple Group By queries, the 25-query set, took 868.145 seconds using our naïve solution, and only

422.771 seconds, a bit less than half the time, using the TDS-guided optimized solution.

V. CONCLUSIONS

With an increase (on average) of 44.26% in efficiency when following our TDS execution plan, we exceeded the goal of meeting the optimization the authors reported over the queries being executed without TDS. We doubt the possibility that we implemented the algorithms in a superior fashion to that of the authors – we speculate the cause of this difference could be due to the method of implementation: whereas the authors of the original paper used a DBMS with an actual querying language and extended its query optimizer, we created an implementation of the template query and simply used C++ I/O on plain text files, where there was absolutely no optimization on the naïve approach. Other variables likely contributing to this include the difference in the experimental data set (us using only low-cardinality columns to prevent unreasonably long runtimes, which in turn meant less possible groupings to draw from) and less impactfully likely differences in the hardware the experiments were conducted on. Nonetheless, we did achieve better runtimes using our approach, and therefore consider the implementation a success.

In studying and implementing this paper, as well as in reading the papers detailed in the related works, we learned a great deal about data summarization and the process of optimizing queries in general using algorithms and execution plans. Further work we could do pertaining to this paper starts with a difference in implementation. Due to our lack of experience in working with altering the query optimizers of DBMS's and the time limit of this semester to implement the original paper, we were guided to use the simple implementations as described in this document – future work could be in learning how to implement the algorithms as the authors did.

VI. WORKLOAD & SOURCE CODE

The work of this implementation was done in its entirety by Daniel Kane and Ethan Letourneau. As far as the milestones for the project are concerned, Mr. Letourneau authored the Project Proposal and this Final Report, while Mr. Kane authored the Literature Survey and the Status Report. We both worked together on the conceptualization for how to implement the project in C++, with help along the way from Dr. Zhao in ideas on how to start and what was required.

Mr. Kane implemented the building of the preliminary tree and the Partition Children function in the TDS algorithm. Mr. Letourneau implemented the naïve solution, the optimal solution, and the remaining TDS functions (being Top-Down Split and Divide Subsets). Mr. Letourneau generated the testing table of 184,000 lines as well as the experimental table of 2 million rows. The source code for the project can be found at the GitHub repository listed at [5] – this contains all of the files (source code, makefile, experimental results, and so on) except the 2 million row table that we conducted the experimentation on, due to GitHub capping individual file sizes at 25 MB; the 2 million row table file will be given to the instructor via email or USB. The README file in the repository gives instructions on how to compile and run our programs, and how to replicate the experiments.

VII. REFERENCES

[1] D. Phan, P. Michiardi, "A Novel, Low-latency Algorithm for Multiple Group-By Query Optimization", in Proc. IEEE 32nd International Conference on Data Engineering (ICDE), 2016

[2] Z. Chen et al., "Efficient computation of multiple group-by queries," in Proc. ACM SIGMOD 2005, 2005.

[3] F. Dehne et al., "Computing partial data cubes," in Proc. HICSS, 2004.

[4] "Split Strings in C++ Using a Delimiter" (https://www.techiedelight.com/split-string-cpp-using-delimiter/)

[5] "COP5725 Implementation Project" (https://github.com/ethan-letrno/cop5725-implementation-project)