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

Duy-Hung Phan
EURECOM
phan@eurecom.fr

Pietro Michiardi
EURECOM
michiard@eurecom.fr

***Abstract*—Data summarization is essential for users to interact with data. Current state of the art algorithms to optimize its most general form, the multiple Group By queries, have limitations in scalability. In this paper, we propose a novel algorithm, Top-Down Splitting, that scales to hundreds or even thousands of attributes and queries, and that quickly and efficiently produces optimized query execution plans. We analyze the complexity of our algorithm, and evaluate, empirically, its scalability and effectiveness through an experimental campaign. Results show that our algorithm is remarkably faster than alternatives in prior works, while generally producing better solutions. Ultimately, our algorithm reduces up to 34% the query execution time, when compared to un-optimized plans.**

I. Introduction

Data is one of the most valuable assets to a company as it is transformed to become decisional information. Typically, in large organizations, users share the same platform to process data by issuing queries to the data management system, whether it is a relational database, a traditional data warehouse or a modern big-data system such as Google BigTable [1] and MapReduce [2]. Regardless of the underlying technology to store and process data, typical users or data analytic applications issue many queries and expect results as fast as possible. As a consequence, query optimization is vitally important.

In this paper, we focus on optimizing a predominant operation in databases: data summarization. Users that interact with data (especially "*big data*") constantly feel the needs of computing aggregates to extract insights and obtain value from their data assets. Of course, humans can not be expected to parse through Gigabytes or Terabytes of data. In fact, typically, users interact with data through *data summaries*. A summary is obtained by grouping data on various combinations of dimensions (*e.g.*, by location and/or time), and by computing aggregates of those data (*e.g.*, count, sum, mean, *etc.*) on such combinations. These summaries are then used as input data for all kinds of purposes such as joining with other data, visualization on dashboards, business intelligence decisions, data analysis, anomaly detection, *etc.* From this perspective, we consider data summarization as a crucial task that is performed extremely frequently. The workload and query templates of industrial benchmarks for databases justify this point. For instance, 20 of 22 queries in TPC-H [3] and 80 out of 99 queries in TPC-DS [4] are data summarization queries. In addition, a significant portion of queries in TPC-H and TPC-DS access the same data, which means some data is *"hotter"* than others. A cross-industry study in [16] actually confirms this observation. All together, these bestow a great chance for mulitple Group By query optimization to achieve better performance.

The family of data summarization queries consists of four operators: *Group By*, *Rollup*, *Cube* and *Grouping Sets*. A Group By operator finds all records having identical values *w.r.t.* a set of attributes, and computes aggregates over those records. For instance, consider a table *CarSale(CS)* with two attributes *model(M)*, and *package(P)*, the query: *SELECT M, Count(*) FROM CarSale GROUP BY (M)* counts the volume of car sales for each model.

The Group By operator is the building block of data summarization, as all other operators are its generalizations. A Cube operator (introduced by Gray *et al.* [5]) computes Group Bys corresponding to all possible combinations of a list of attributes. Thus, a Cube query such as *Select M, P, Count(*) From CS Group By Cube(M, P)* can be rewritten into *four* Group By queries:

```
Q1: Select M,P,Count(*) From CS Group By(M, P)
Q2: Select M, Count(*) From CS Group By(M)
Q3: Select P, Count(*) From CS Group By(P)
Q4: Select Count(*) From CS Group By(*)
```

The Group By (*) in *Q4* denotes an ***all*** Group By, (or sometimes called empty Group By). A Rollup operator [5] considers a list of attributes as different levels of one dimension, and it computes aggregates along this dimension upward level by level. Thus a Rollup query like *Select M, P, Count(*) From CS Group By Rollup(M, P)* computes volume sale for Group Bys (M, P), (M), (*) (or *Q1*, *Q2*, *Q4*) in the above example. The Rollup and Cube operators allow users to compactly describe a large number of combinations of Group Bys. Numerous solutions for generating the whole space of data Cube and Rollup have been proposed [6]–[10]. However, in the era of *"big data"*, datasets with hundreds or thousands of attributes are very common (*e.g.* data in biomedical, physics, astronomy, *etc.*). Due to the large number of attributes, generating the whole space of data Cube and Rollup is inefficient. Also, very often users are not interested in the set of all possible Group Bys, but only a certain subset. The Grouping Sets operator facilitates this preference by allowing users to specify the exact set of desired Group Bys. In short, Cube, Rollup and Grouping Sets are convenient ways to declare multiple Group By queries.

Example 1: Consider a scenario in medical research, in which there are records of patients with different diseases. There are many columns (attributes) associated with each patient such as *age*, *gender*, *city*, *job*, *etc.* A typical data analytic

task is to measure correlations between the diseases of patients and one of the available attributes. For instance, heart attack is often found in elderly people rather than teenagers. This can be validated by obtaining a data distribution over two-column Group By *(disease, age)* and by comparing the frequency of heart attack of elderly ages ($age \geq 50$) versus teenagers age ($12 \leq age \leq 20$). Typically, for newly developed diseases, a data analyst would look into many possible correlations between these diseases and available attributes. A Grouping Sets query allows her to specify different Group Bys like *(disease, age)*, *(disease, gender)*, *(disease, job)*, *etc.*

In this paper, we tackle the most general problem in optimizing data summarization: how to efficiently compute a set of multiple Group By queries. This problem is known to be NP-complete ([7], [11]), and all state of the art algorithms ([7], [11]–[13]) use heuristic approaches to approximate the optimal solution. However, none of prior works scales well with large number of attributes, *and/or* large number of queries. Therefore, in this paper, we present a novel algorithm that:

- Scales well with both large numbers of attributes and numbers of Group By queries. In our experiment, the latency introduced by our query optimization algorithm is several orders of magnitude smaller than that of prior works. As the optimization latency is an overhead that we should minimize, our approach is truly desirable.
- Empirically performs better than state of the art algorithms: in many cases our algorithm finds a better execution plan, in many other cases it finds a comparable execution plan, and in only a few cases it slightly trails behind.

In the rest of the paper, we formally describe the problem in Section II. We then discuss the related work and their limitations in Section III to motivate the need for a new algorithm. The details of our solution with a complexity analysis are presented in Section IV . We continue with our experimental evaluation in Section V. A discussion about our algorithm and its extension is in Section VI. Finally we conclude and discuss our future work in Section VII.

II. Problem Statement

There are two types of query optimization: single-query optimization and multi-query optimization. As its name suggest, single-query optimization optimizes a single query by deciding, for example, which algorithm to run, configuration to use and optimized values for parameters. An example is the work in [14]: when users issue a Rollup query to compute aggregates over *day*, *month* and *year*, the optimization engine automatically picks the most suitable state of the art algorithms [15] and set the appropriate parameter to obtain the lowest query response time.

On the other hand, multi-query optimization optimizes the execution of a set of multiple queries. In large organizations, there are many users who share the same data management platform, resulting in a high probability of systems having concurrent queries to be processed. A cross industry study [16] shows that not all data is equal: in fact, some input data is *"hotter"* (*i.e.* get accessed more frequently) than others. Thus, there are high chances of users accessing these "hot" files concurrently. This is also verified by in industrial benchmarks (TPC-H and TPC-DS) in which their queries frequently access the same data. The combined outcome is that optimizing multiple queries over the *same input data* can be significantly beneficial.

The problem we address in this paper, the multiple Group By query optimization (MGB-QO), can come from both scenarios. From the single-query optimization perspective, any Cube, Rollup or Grouping Sets query is equal to multiple Group Bys. From the multi-query optimization perspective, the fact that many users issue one Group By over the same data means multiple Group Bys and it requires optimization. More formally, we consider an *offline* version of the problem:

- For a time window ω, without loss of generality, we assume the system receives data summarization queries over the same input data that contains one of the following operators:
  - Group By
  - Rollup
  - Cube
  - Grouping Sets
- These queries correspond to n Group By queries $\{Q_1, Q_2, ..., Q_n\}$

In reality, hardly any query arrives at our system at the exact same time. The time window ω can be interpreted as a period of time in which queries arrive and are treated as concurrent queries. The value of ω can either be predetermined or dynamically adjusted to suit the system workload and scheduler, which lead to the *online* version of this problem. However, the online problem is not addressed in this paper: it remains part of our future work.

A. Definitions

We assume that the input data is a table T with m attributes (columns). Let $S = \{s_1, s_2, ..., s_n\}$ be the set of groupings that have to be computed from n queries $\{Q_1, Q_2, ..., Q_n\}$, where s_i is a subset of attributes of T. Each query Q_i is a Group By query:

Q_i: Select s_i, Count(*) From T Group By s_i

To simplify the problem, we assume that all queries perform the same aggregate measure (function) (*e.g.* Count(*)). Later in Section VI-B, we discuss the solution to adapt to different aggregate measures.

a) Search DAG: Let $Att = \{a_1, ...a_m\} = \bigcup_{i=1}^{\infty} s_i$ be the set of all attributes that appear in n Group By queries. We construct a *directed acyclic search graph* $G = (V, E)$ defined as follows. A node in G represents a grouping (or a Group By query). V is the set of all possible combinations of groupings constructed from Att plus a special node: the root node T. The root node is essentially the input data itself.

An edge $e = (u, v) \in E$ from node u to node v indicates that grouping v can be computed directly from grouping u. For instance, an edge $AB \rightarrow A$ means that grouping A can be computed from grouping AB. There are two costs associated with an edge e between two nodes: a *sort cost* $c_{sort}(e)$ and a *scan cost* $c_{scan}(e)$. If grouping AB is sorted in order of (A, B), computing grouping A would require no additional sort, but only a scan over the grouping AB. We denote this

cost by $c_{scan}(e)$. However if grouping AB is not sorted, or sorted in order of (B, A), computing grouping A would require a global sort on the attribute A, incurring a sort cost $c_{sort}(e)$. The costs are of course different in two cases. We note the only exception: the root node. If input data is not sorted, then all outgoing edges from the root node have only one sort cost.

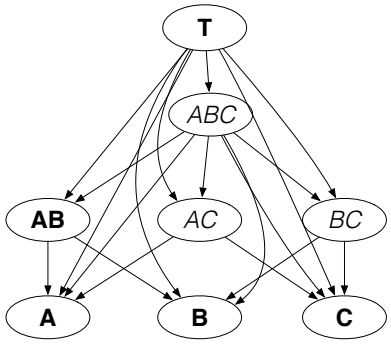

Fig. 1: An example of a search DAG

We call G a search DAG. Next, we show an example with four queries. In this example, we have an input table $T(A, B, C)$ and four Group By queries:

```
Q1: Select A, Count(*) From T Group By(A)
Q2: Select B, Count(*) From T Group By(B)
Q3: Select C, Count(*) From T Group By(C)
Q4: Select A,B,Count(*) From T Group By(A,B)
```

From the above definitions, we have:

- $S = \{A, B, C, AB\}$.
- $Att = \{A, B, C\}$.
- $V = \{T, *, A, B, C, AB, AC, BC, ABC\}$.

It is easy to see that $S \subseteq V$. We call S the *terminal* (or *mandatory*) nodes: all of these nodes have to be computed and materialized as these are outputs of our Group By queries $\{Q_1, Q_2, Q_3, Q_4\}$. Other nodes in $V \setminus S$ are *additional* nodes which may be computed if it helps to speed up the execution of computing S. In this example, even though grouping ABC is not required, computing it allows $S = \{A, B, C, AB\}$ to be directly computed from ABC rather than the input table T. If the size of ABC is much smaller than T, the execution time of S is indeed reduced. Because V contains all possible combinations of groupings constructed from Att, we are sure that all possible groupings that help reduce the total execution cost are inspected. We also prune the space of V to exclude nodes that have no outgoing edges to at least one of the terminal nodes, *i.e.* these nodes certainly cannot be used to compute S. The final search DAG for the above example is shown in Figure 1.

Intuitively if a grouping is used to compute two or more groupings, we want to store it in memory or disk to serve later rather than recompute it.

b) Problem Statement: In data management systems, the problem of multiple Group By query optimization is processed through both logical and physical optimization. In logical optimization, we set to find an optimal *solution tree* $G' = (V', E')$. The solution tree G' is a directed subtree from G, rooted at T, that covers all terminal nodes $s_i \in S$. It can be seen as a logical plan for computing multiple Group By queries efficiently. This is the main objective of this paper.

The physical optimization, as its name suggests, takes care of all physical details to execute the multiple Group By queries and return actual aggregates. Understandably, different data management systems have different architectures to organize their data layout, disk access, indexes, *etc.* Thus, naturally each system may have is own technique to implement the physical multiple Group By operator. For reference purpose, some example techniques are PipeSort, PipeHash [7], Partition-Cube, Memory-Cube [10] or newer technique for multiprocessors in [17] for databases, or In-Reducer Grouping [15], [18] for MapReduce and its extensions. We note that the physical optimization is not our paper's target: our solution is not affected by any particular technique. Also, regardless of the physical techniques, the grouping order is guided by the solution tree G' obtained from our logical optimization.

More formally, in the optimized solution tree G' we have:

1) $S \subseteq V' \subseteq V$.
2) $E' \subset E$ and for any edge $e(u, v) \in E'$, there is only *one* type of cost associated to edge e:
$$c(e) = \begin{cases} c_{sort}(e) \\ c_{scan}(e) \end{cases}$$
3) From any node $u \in V'$, there is at most *one* outgoing scan edge.

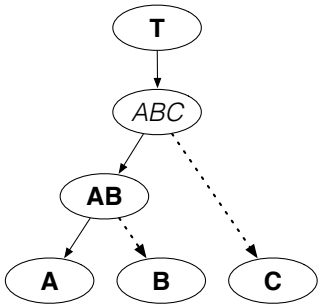

Fig. 2: An example of a solution tree

An optimal solution tree is the solution tree with the minimal total execution cost $C(E') = \sum_{e \in E'} c(e)$. Figure 2 shows an optimal solution tree for the above example. The dotted lines represent sort edges, and the solid lines show the scan edges. The bold nodes are the required grouping (*i.e.* terminal nodes). The italic node (ABC) is the additional node whose computation helps to reduce the total execution cost of G'. Additional groupings BC and AC are not computed as doing so does not bring down the cost of G'.

Finding the optimal solution tree for multiple Group By queries is an NP-complete problem[11], [19]. State of the art algorithms use heuristic approaches to approximate the solution. In the next Section, we discuss in more detail why none of those algorithms scale well with large number of attributes, and/or large number of Group By queries. This motivates us to find a more scalable approach.

B. Cost model

Our primary goal is to find the solution tree G' with a small total execution cost. The total execution cost is the sum of the cost from all edges in G'. Therefore, we need a cost model that assigns the scan and sort costs to all edges in our search graph. However, our work does not depend on a specific cost model as its main purpose is to quantify the execution time of computing a node (a Group By) from another node. Any appropriate cost model for various systems like parallel databases, MapReduce systems, *etc.* can be plugged into our algorithm.

III. Related Work

Optimizing data summarization in traditional databases has been one of the main tasks in database research. The multiple Group By query problem are studied through the lenses of the most general operator in data summarization: Grouping Sets. The Grouping Sets is syntactically an easy way to specify different Group By queries at the same time, therefore all of Group By, Rollup, Cube queries can be translated directly into a Grouping Sets query.

To optimize Grouping Sets queries, the common approach is to define a directed acyclic graph (DAG) of nodes, where each node represents a single Group By appearing in the Grouping Sets query [7], [11]–[13], [19]. An edge from node *u* to node *v* indicates that grouping *v* can be computed from grouping *u*. For example: group BC can be computed from BCD.

There are two major differences among various works to compute Grouping Sets. The first difference is the *cost model*: how to quantify a cost (expressed as a weight of an edge) to compute a group *v* from a group *u*. PipeSort [19] sets the weight of an edge (u, v) to be proportional to the cost of re-sorting *u* to the most convenient order to compute *v*. For example, to compute BC, the main cost would be to resort ABC to BCA to compute (B, C). This is a sort cost. However, if grouping ABC is already in the sorting order of (B, C, A), the cost to compute BC would be mainly scan (hence scan cost). In contrast, [12] and [11] simplify the cost model by having only one weight for each edge (u, v), regardless of how physically v is computed: the weight of an edge (u, v) is equal to the cardinality of group *u*.

The other difference is, given a DAG of Group By nodes and weighted edges with appropriate costs, how to construct an optimal execution plan that covers all required Group Bys with the minimum total cost. This problem is proven to be NP-complete ([11], [19]), thus approximations through heuristic approaches are studied.

The work in [12] gives a simple greedy approach to address the problem. It considers Group By nodes in descending order of cardinality. Each Group By is connected to one of its super nodes. Super nodes of v is any node u such that (u, v) exists. If there are super nodes that can compute this Group By without incurring a sorting cost, it chooses the one with the least cost. This Group By becomes a scan child of its parent node. If all super nodes already have a scan child, it chooses the super nodes with the least sort cost. This approach is called *Smallest Parent*. It is simple and fast, however it does not consider any *additional* node that can help reducing the total cost. In the rest of this section, we consider algorithms that include also additional nodes.

In [7], the authors transform the problem into a *Minimal Steiner Tree* (MST) on directed graph problem. Because the cost of an edge depends on the sorting order of the parent node, a Group By node is transformed into multiple nodes: each corresponds to a sorting order that can be generated (using permutation) from the original Group By node. Then the approach in [7] adds cost to all pairs of nodes, and uses some established approximation of MST to retrieve the optimized solution. The main drawback of this approach is that, the transformed DAG contains a huge number of added nodes and edges (because of permutation), and even a good approximation of MST problem cannot produce solutions in feasible time, as any good MST approximation is at least $\mathcal{O}(|V|^2)$ where $|V|$ is the number of nodes. For example, to compute Cube with 8 attributes, the transformed DAG consists of $109,601$ nodes and $718,178,136$ edges, w.r.t. 256 nodes and $6,561$ edges of the original DAG.

In another work [13], the authors present a greedy approach on the search DAG of Grouping Sets. Given a partial solution tree T (which initially includes only the root node), the idea is to consider all other nodes to find one that can be added to T with the most benefit. When the node x is added to T, it is first assumed to be computed from the input data set, and this incurs a sort cost. Then the algorithm in [13] tries to minimize this cost, by finding the best parent node from which x can be computed. Once x's parent is chosen, this approach finds all the nodes that are beneficial if computed from x rather than its current parent. This benefit is then subtracted by the cost of adding x to yield the total benefit of adding x to T (which the benefit value can be positive or negative). This process is repeated until it cannot find any node that brings positive benefit to add to T. The complexity of this approach is $\mathcal{O}(|V||T|^2)$ where $|V| = 2^m$, m is the number of attributes and $|T|$ is the size of the solution tree, which is typically larger than the number of terminal nodes ($|T| \geq n$). Note that while $|V|$ is much smaller than the number of nodes in [7] because of no added permutation, it is still problematic if m is large.

While the approach in [13] is more practical than the approach in [7], it cannot scale well with a search DAG of a much larger number m of attributes, in which the full space of additional nodes and edges can not be efficiently enumerated. To address this problem, [11] proposes a bottom-up approach. It first constructs a naïve plan in which all mandatory nodes are computed from the input data set. Then from all children nodes of the input data set, it considers all pairs of nodes (x, y) that can be merged. For each pair, it computes the cost of a new plan obtained by merging this pair of nodes. After that, it pick the pair, say (v_1, v_2), that has the lowest cost and replace the original plan with this new plan. In this new plan, the node $v_1 \cup v_2$ is included to the solution tree. In other words, an additional node is only considered and added if and only if it is the parent of at least two different nodes. This eliminates the task of scanning all nodes in the search DAG, making this algorithm a major improvement over previously described algorithms. At some point in time, if all the possible pairs result in a worse cost then the current plan, the algorithm stops. This algorithm calls $\mathcal{O}(n^3)$ times the procedure of merging two

nodes where n is the number of terminal nodes. The merging procedure has the complexity of $\mathcal{O}(n)$. Overall, the complexity of this algorithm is $\mathcal{O}(n^4)$.

The advantage of the algorithm described in [11] is that, it scales irrespectively to the space of $|V|$ but only to n, the number of Group By queries. If n is small, it scales better than [7], [13]. However, for dense multiple Group By queries, *i.e.* large n and small m (*e.g.* computing Cube of 10 attributes results in $n = 1024$), this algorithm scale worse than [7], [13]. This motivates us for a more scalable and efficient algorithm to approximate the solution tree.

IV. The Top-Down Splitting Algorithm

In this Section, we propose a heuristic algorithm called Top-Down Splitting to find a solution tree in the multiple Group By query optimization discussed in Section II. Our algorithm scales well with both large numbers of attributes and large number of Group By queries. Compared to state of the art algorithms, our algorithm runs remarkably faster without sacrificing the effectiveness. In Section IV-A, we present our algorithm in detail with its complexity evaluation in Section IV-B. Finally, we discuss the choice of appropriate values for an algorithm-wise parameter, as it affects directly the running time of our algorithm.

A. Top-Down Splitting algorithm

Our algorithm consists of two steps. The first step is to build a preliminary solution tree that consists of only terminal nodes and the root node. Taking this preliminary solution tree as its input, the second step aims to repeatedly optimize the solution tree by adding new nodes to reduce the total execution cost. While the second step sounds similar to [13], we do not consider the whole space of additional nodes. Instead, we consider only additional nodes that can evenly split a node's children into k subsets. Here k is an algorithm-wise parameter set by users. By trying to split a node's children into k subsets, we apply a structure to our solution tree: we transform the preliminary tree into a k-way tree (*i.e.* at most k fan-out). Observing the solution trees obtained from state of the art algorithms, we see that most of the times they have a relatively low fan-out k.

a) Constructing the preliminary solution tree: This step returns a solution tree including only terminal nodes (and of course, the root node). Later, we further optimize this solution tree. The details of this step are shown in Algorithm 1.

We sort the terminal nodes in descending order of their cardinality. As we traverse through terminal nodes in descending order, we add them to the preliminary solution tree G': we find their parent node in G' with the smallest sort cost (line 5). Obviously, nodes with smaller cardinality cannot be parents of a higher cardinality node. Thus we assure that all possible parent nodes are examined. Up to this point, we have considered only the sort cost. When all terminal nodes are added, we update the scan/sort connection between a node u and its children. Essentially, the $fix\_scan(u)$ procedure finds a child node of u that brings the biggest cost reduction when its edge is turned from *sort* to *scan* mode. The output of Algorithm 1 is a solution tree G' which is not yet optimized.

Algorithm 1 Step 1: Constructing preliminary solution

```
1:  function BUILD_PRELIMINARY_SOLUTION
2:      G' ← T
3:      sort S in descending order of cardinality
4:      for v ∈ S do
5:          u_min = arg min_u c_sort(u, v) | u ∈ G'
6:          G' ← G' ∪ v: add v to G'
7:          E' ← E' ∪ e_sort(u_min, v)
8:      end for
9:      for u ∈ G' do
10:         fix_scan(u)
11:     end for
12:     return G'
13: end function
```

Algorithm 2 Step 2: Optimizing G'

```
1:  procedure TOPDOWN_SPLIT(u, k)
2:      repeat
3:          b ← partition_children(u, k)
4:      until (b == false)
5:      Children = {v_1...v_q} | (u, v_i) ∈ E'
6:      for v ∈ Children do
7:          topdown_split(v, k)
8:      end for
9:  end procedure
```

b) Optimizing the solution tree: In this step, we call $topdown\_split(T, k)$, with T is the root node, to further optimize the preliminary solution tree obtained in Algorithm 1. The procedure $topdown\_split(u, k)$ (Algorithm 2) repeatedly calls $partition\_children(u, k)$ (Algorithm 3) that splits the children of node u into *at most* k subsets. The function $partition\_children(u, k)$ returns ***true*** if it can find a way to optimize u, *i.e.* split children of node u into smaller subsets and reduce the total cost. Otherwise, it returns ***false*** to indicate that children of node u cannot be further optimized. We then recursively apply this splitting procedure to each child node of u. Since the flow of our algorithm is to start partitioning from the root down to the leaf nodes, we call it the *Top-Down Splitting* algorithm.

The function $partition\_children(u, k)$ (Algorithm 3) tries to split the children of u into *at most* k subsets. Each of these k subsets is represented by an additional node that is the union of all nodes in that subset. 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. Observing the solution tree obtained from state of the art algorithms, we see that in many situation, the optimal splitting strategy may not be exactly k, but a value k' ($1 \leq k' \leq k$). By trying every possible split k' from 1 to k, we compute the new total execution cost with new additional nodes, and retain the best partition scheme, *i.e.* the one with the lowest total cost. Then, we update the solution graph accordingly: removing edges from u to children, adding new nodes and edges from u to new nodes, and from new nodes to children of u.

The $divide\_subsets(u, k')$ (Algorithm 4) is called to divide all children of u into k' subsets and return k' new

Algorithm 3 Find the best strategy to partition children of a node u to at most k subsets

function PARTITION_CHILDREN(u, k)
  $CN = \{v_1 ... v_q\} | (u, v_i) \in E'$
  if $q \leq 1$ **then** ▷ q: number of child nodes
    return false
  end if
  $C_{min} = cost(G')$
  $SS \leftarrow \emptyset$
  if $k > q$ **then**
    $k = q$ ▷ constraint: $k \leq q$
  end if
  for $k' = 1 \rightarrow k$ **do**
    $A = divide\_subsets(u, k')$
    compute the new cost C'
    if $C' < C_{min}$ **then**
      $C_{min} \leftarrow C'$ ▷ remember the lowest cost
      $SS \leftarrow A$ ▷ remember new addition nodes
    end if
  end for
  if $SS \neq \emptyset$ **then**
    Update G' according to SS
    return true
  else
    return false
  end if
end function

Algorithm 4 Dividing children into k' subsets

function DIVIDE_SUBSETS(u, k')
  $CN = \{v_1 ... v_q\} | (u, v_i) \in E'$
  sort CN by the descending order of cardinality
  $C_{min} = cost(G')$
  for $i = 1 \rightarrow k'$ **do**
    $SS_i \leftarrow \emptyset$ ▷ initialize subsets i^{th}
  end for
  for $v \in CN$ **do**
    $i_{min} = \arg\min_i attach(v, SS_i) | i \in 1, ...k'$
    $SS_{i_{min}} \leftarrow SS_{i_{min}} \cup v$
  end for
  return $SS = \{SS_i\} \forall 1 \leq i \leq k'$
end function

additional nodes. At first, we sort the children nodes (CN) in descending order of their cardinality. As we traverse through these children nodes, we add each child node into a subset that yields the smallest cost. The cost of adding a child node v into a subset SS_i is:

$$attach(v, SS_i) = \big[c_{sort}(u, SS_i \cup v) + c_{sort}(SS_i \cup v, v) - c_{sort}(u, v)\big]$$

Here SS_i denotes the additional node representing the i^{th} subset ($i \leq k'$). If a node v is attached to a subset SS_i, the new additional node is updated: $SS_i \leftarrow SS_i \cup v$.

Now that we have described our two steps, our algorithm is described in Algorithm 5.

Algorithm 5 Top-Down Splitting algorithm

$G' =$ *build_preliminary_solution()*
$topdown\_split(G'.getRoot(), k)$

B. Complexity of our algorithm

In this Section, we evaluate the complexity of our algorithm in the best case and the worst case scenarios. The average case complexity depends on uncontrolled factors such as: input data distribution, relationship among multiple Group Bys, specific cost models, *etc.* We cannot compute the average complexity without making assumptions on such factors. Therefore this remains part of our future work. Empirically, we observe that in our experiments the average case leans towards the best case with just a few exceptions that are closer to the worst case.

a) The worst case scenario: As the first step and the second step of our algorithm are consecutive, the overall complexity is the maximum complexity of two steps. It is easy to see that the complexity of Algorithm 1 is $\mathcal{O}(n^2)$ where $n = |S|$ is the number of Group By queries.

For the second step, we first analyze the complexity of Algorithm 3: it calls $\mathcal{O}(k)$ times the $divide\_subsets$ function and it computes $\mathcal{O}(k)$ times the cost of the modified solution tree. The complexity of the $divide\_subsets$ function (*i.e.* Algorithm 4) is $\mathcal{O}(\max(k^2, kq))$. As we cannot divide q children nodes into more than q subsets, $k \leq q$. Therefore the complexity of Algorithm 4 is $\mathcal{O}(kq)$. It is not difficult to see that q is bounded by n, *i.e.* $q \leq n$. The case of $q = n$ happens when all mandatory nodes connect to the root node. Therefore the worst case complexity of Algorithm 4 is $\mathcal{O}(kn)$

Since Algorithm 3 limits itself in only modifying node u and its children, we can compute the new cost by accounting only altered nodes and edges. There are at most k new additional nodes, and there are q children nodes of node u, so computing each time a new cost of the solution tree is in $\mathcal{O}(k+q)$ time. As $k \leq q \leq n$, the complexity of computing a new cost is $\mathcal{O}(n)$, which is smaller than $\mathcal{O}(kn)$ of Algorithm 4. As such, the worst case complexity of Algorithm 3 is $\mathcal{O}(k^2 n)$

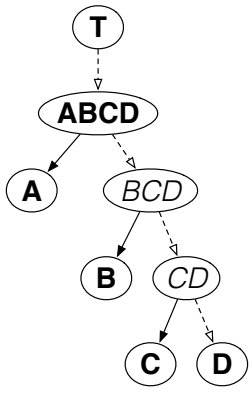

Fig. 3: An example of worst case scenario with $k = 2$

The complexity of Algorithm 2 depends on how many times $partition\_children$ is called. Let $|V'|$ be the number of nodes in the final solution tree. Clearly $topdown\_split$

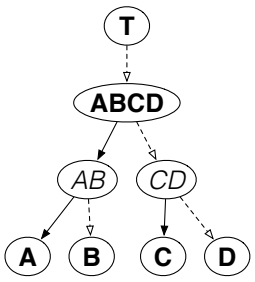

Fig. 4: An example of best case scenario with $k = 2$

is called at most $|V'|$ times, and each time it calls $partition\_children$ at least once. In order for $topdown\_split$ to terminate, $partition\_children$ has to return ***false***, and it does so in $\mathcal{O}(|V'|)$ time.

Now, for each time $topdown\_split$ is called, $partition\_children$ is called more than once if and only if it returns ***true***, which means at least an additional node is added to V'. When an additional node is added, it puts together a new subset, which consists of *at least* 2 children nodes or more. In other words, if an additional node is formed, in the worst case it applies a binary structure to the solution tree that has maximum n leaves nodes. A property of binary trees states that $|V'| \leq 2n - 1$, which means there are no more than $n - 1$ additional nodes in the final solution tree. As a consequence, in the worst case, $partition\_children$ returns ***true*** in essentially $\mathcal{O}(n)$ time. Since $|V'| \leq 2n - 1$, it also returns ***false*** in $\mathcal{O}(|V'|) \equiv \mathcal{O}(n)$ time.

The worst-case complexity of Algorithm 2 (*i.e.* our second step) is $\mathcal{O}(k^2n^2)$. As $\mathcal{O}(k^2n^2)$ is higher than $\mathcal{O}(n^2)$ of the first step, the worst case complexity of our algorithm is $\mathcal{O}(k^2n^2)$. Figure 3 shows an example of the optimized solution tree obtained in the worst case scenario.

b) The best case scenario: In the best case scenario, we obtain a *balanced* k-way solution tree. Figure 4 shows an example of such a balanced solution tree. In this scenario, Algorithm 2 calls $partition\_children$ to return ***true*** in $\mathcal{O}(log_k n)$ times instead of $\mathcal{O}(n)$ times like the worst case scenario. Therefore, the best case complexity is $\mathcal{O}(k^2 n log_k n)$.

C. Choosing appropriate values of k

Our algorithm depends on an algorithm-wise parameter: k representing the fan-out of the solution tree. We observe that for solution trees obtained from state of the arts algorithm, the value of k is rather small. For example, let us consider a primary study case that motivates the work in [11]: a Grouping Sets query to compute all single-column Group By in a table with m attributes (*i.e.* the number of Group By is equal to m). In this example, small values of k such as $2 \leq k \leq 4$ are sufficient to find an optimized solution tree. In our experiments in Section V, high values of k do not result in a lower cost solution tree. We note that our observation is in line to what observed in [11].

For any node u, let q_u be the number of its children. Clearly we cannot force to split u's children into more than q_u subsets, *i.e.* $k \leq q_u$. We denote $k\_max_u = q_u$. Thus, any value of k higher than q_u is wasteful, and our algorithm does not consider such values (line 9 in Algorithm 3).

On the other hand, for some node u, we cannot split its children into less than a certain number of subsets. Let us consider an example in which we want to partition 5 children nodes of $u = ABCDE$: $ABCD, ABCE, ABDE, ACDE$ and $BCDE$. Clearly splitting these children nodes into any number of subsets smaller than 5 is not possible, as merging any pairs of nodes results in the parent node u itself. In this example, 5 is the minimum number of subsets for node u. Values of k smaller than 5 result in no possible splits. We call this the lower bound of k. To find an exact lower bound of k in a specific node u is not a trivial task. For instance, let us replace 5 children nodes of $u = ABCDE$ with: A, B, C, D, E. In this situation, the lower bound of k for node u is 2. We denote $k\_min_u = 2$

As k is an algorithm wise parameter, we have the following: $k\_min_u \leq k, \forall u \in T$. Obviously, we can set $k = max(k\_min_u)$. However, doing this is not always beneficial. Let us continue with our example, as Figure 3 and Figure 4 suggests, for node $ABCD$ and its children (A, B, C, D), $k = 2$ is sufficient to obtain an optimized solution tree; in other words, for this node, $k = 5$ is wasteful.

Algorithm 6 Adaptively dividing children

```
1:  function DIVIDE_SUBSETS(u, k')
2:      CN = {v_1...v_q}|(u, v_i) ∈ E'
3:      sort CN by the descending order of cardinality
4:      C_min = cost(G')
5:      p = k'                          ▷ p: the current number of subsets
6:      for i = 1 → p do
7:          SS_i ← ∅                    ▷ initialize subsets i^th
8:      end for
9:      for v ∈ CN do
10:         i_min = arg min_i attach(v, SS_i)|(SS_i ∪ v) ≠ u
11:         if i_min ≠ null then
12:             SS_{i_min} ← SS_{i_min} ∪ v
13:         else
14:             p ← p + 1               ▷ increase number of subsets
15:             SS_p ← v                ▷ add v to the new subset
16:         end if
17:     end for
18:     return SS
19: end function
```

In the general case, if we partition children nodes of u into a predetermined number of subsets k, *i)* for some node u it could be impossible to partition in such a way; *ii)* for some node u' it may be wasteful. Again, our observation is that most nodes have a very low fan-outs. Nodes with high upper bound of k are relatively scarce. So our strategy is to attempt partitioning children nodes of u into small numbers of subsets (*i.e.*, k is small). Whenever such a split is unachievable, we dynamically increase our number of subsets until the partition is possible. We modify Algorithm 4 to reflect the new strategy (Algorithm 6). The gist of this algorithm is that, we can attach a child node v of u to a subset SS_i if and only if $(SS_i \cup v) \neq u$. When there is no such SS_i, we add a *new* subset (*i.e.* at this node, we increase the number of subsets by 1).

V. Experiments and Evaluation

An optimization algorithm for the multiple Group By query problem can be evaluated from three different aspects:

- *Optimization latency*: the time (in second) that an algorithm takes to return the optimized solution tree. It is also the optimizing overhead. The lower the optimization latency, the better. This is an important metric to assess the scalability of an algorithm.
- *Solution cost*: given a cost model and a solution tree, it is the total of scan and sort cost associated to edges of the tree. A lower cost means a better tree. This metric assesses the effectiveness of an algorithm.
- *Runtime* of the solution tree: the execution time (in second) to compute n Group By queries using the optimized execution plan.

In this Section, we empirically evaluate the performance of our algorithm compared to other state of the art algorithms. The experimental results of this Section can be summarized as follows:

- The optimization latency of our algorithm is up to several orders of magnitude smaller than other algorithms when scaling to *both* large number of attributes *and* large number of Group By queries. In our experiments, the empirical results suggest that, on average, our algorithm leans towards the best case scenario more than to the worst case scenario (analyzed in Section IV-B).
- We do not sacrifice the effectiveness of finding an optimized solution cost for low latency. In fact, compared to other algorithms, in many cases our algorithm finds better solutions, in many other cases it finds comparable ones, and in only a few cases it slightly trails behind.
- Using PipeSort as the physical implementation to compute multiple Group By queries, we show that our algorithm can reduce the execution runtime significantly (up to 34%) compared to the naïve solution tree, in which all Group Bys are computed from the input data.

A. Experiment Setup

The experiments are run on a machine with 8GB RAM. To evaluate the latency and the solution cost of various algorithms, we synthetically generate *query templates*. Each query template consists of *i)* a list of Group By queries; *ii)* cardinalities of nodes. The cardinalities of nodes can be obtained from available datasets using the techniques described in [20], [21], or can be randomly generated (with an uniform distribution, or a power law distribution to represent skewed data). In some situations (*e.g.* large number of attributes), we cannot effectively generate all node cardinalities. In this situation, we take the product of cardinality of each attribute in a node to be the cardinality of that node.

To evaluate the improvement that an optimized solution tree brings compared to a naïve one, we issue a query that contains all two-attribute Group Bys from the *lineitem* table of TPC-H [3]. This table contains 10 million records with 16 attributes. For each algorithm, we report its optimization latency as well as the query runtime obtained when we execute the PipeSort operator guided by its solution tree.

The state of the art algorithms that we compare are the ones presented in [13] and in [11]. We omit the algorithm presented in [7] because it is shown to be inferior to the algorithm in [13]. For convenience, we name our algorithm *Top-Down Splitting* or *TDS*, the one in [11] *Bottom-Up Merge* (*BUM*), and finally the one in [13] *Lattice Partial Cube* or *LPC*. For the case of our algorithm, as k can have multiple values, we evaluate both cases: small value $k = 3$ and large value $k = n$.

B. Cost model in our experiments

As discussed in Section II-B, we need a cost model to assign scan and sort cost to edges of the search graph. In our experiments, we use a simple cost model as a representative in evaluating all different algorithms to find G'. Again, we stress that any appropriate cost model can be used. We define the two costs of an edge as follows:

- Scan cost: $c_{scan}(u \rightarrow v) = |u|$ where $|u|$ is the size (cardinality) of node u
- Sort cost: $c_{sort}(u \rightarrow v) = |u| * log_2|u|$

We assume that the cardinality of any node u is readily available: estimating $|u|$ is not the focus of our work, and we rely on works of cardinality estimation such as [20], [21]. Clearly, a bad cardinality estimation worsen the quality of a solution tree, but all algorithms suffer from the same issue.

C. Scaling with the number of attributes

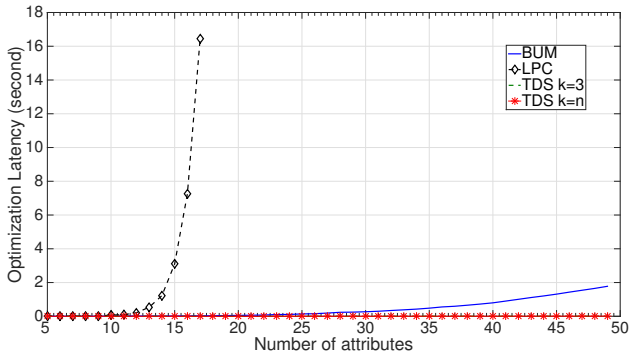

Fig. 5: Single-attribute Group By - Optimization latency.

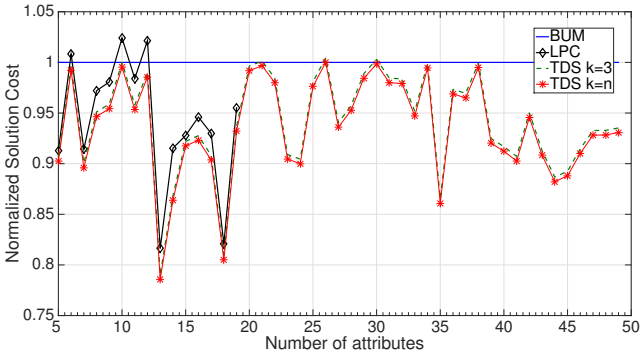

Fig. 6: Single-attribute Group By - Normalized solution cost.

In this experiment, we generate the query templates as follows: *i)* each query template consists of all single-column (or single-attribute) Group By be generated from a table T; *ii)* the number of attributes, m, in table T is from 5 to 49; *iii)* for each number of attributes, we randomly generate 100 different sets of grouping cardinalities, in which 50 sets have uniform distribution, and 50 sets have power law distribution with $\alpha = 2.5$. For each number of attributes and algorithm, we compute the average of the solution cost of the optimized solution tree; as well as the average of its optimization latency.

The results for optimization latency are plotted in Figure 5. The latency of the Lattice Partial Cube algorithm exponentially increases with the number of attributes. This is in line with its complexity of $\mathcal{O}(2^m|T|^2)$ where m is the number of attributes. For the sake of readability, we omit the latency of LPC for large number of attributes. In this experiment, with its complexity of $\mathcal{O}(n^4)$ the Bottom-Up Merge latency scales better than LPC. Nevertheless, as the line of BUM starts to take off at the end, we expect that for large n (*e.g.* $n \geq 100$), BUM has a rather high optimization latency. Our algorithm achieves the best scalability: it takes less than 0.01 second to optimize $n = 49$ queries for both cases of k: $k = 3$ and $k = n$. Understandably, the latency of TDS with $k = n$ is higher than TDS with $k = 3$. However, as both cases have very small latencies, this is indistinguishable in Figure 5.

We note that the solution cost depends on the cost model, and our cost model (see Section V-B) depends on grouping cardinalities, which are different in every query template we generate. Thus, we normalize every cost to a fraction of the solution cost obtained by a baseline algorithm (here we choose BUM) so that it is easier to compare the solution costs obtained by different algorithms. The normalized solution costs are shown in Figure 6. Due to the large amount of time for LPC to complete with large number of attributes m, we skip running LPC for $m \geq 20$. For most number of attributes, on average, our algorithm finds better solution trees than BUM, sometimes its cost is up to 20% smaller. Only in few query templates, our solution tree's cost is a little higher (within 1.5%) than BUM. Compared to LPC, TDS produces comparable execution plans. We notice several spikes of TDS and LPC. The reason is because BUM merges 2 Group Bys at a time and tends to produce uneven subsets, especially when the number of queries is an odd number. However, for some queries, BUM merging results in even subsets. In this case, its solution trees are close to TDS and LPC - thus the spikes.

Table I shows the average of solution costs obtained by each algorithm for every query template, normalized to fractional costs of BUM. Altogether, our algorithm is a little better than LPC. Also as expected, the execution cost acquired by TDS $k = 3$ is a little higher than for $k = n$, however by not much (less than 0.5%). In summary, our results indicate that for a comparable, and often lower cost than that of prior works, our approach yields substantial savings in optimization latency and scalability.

Algorithm	BUM	LPC	TDS $k = 3$	TDS $k = n$
Normalized Cost	1	0.9419	0.9406	0.9361

TABLE I: Average solution cost - Single-attribute queries

D. Scaling with the number of queries

In this experiment, we assess the scalability of various algorithms with respect to the number of queries. To fulfill such a goal, we limit the number of attributes to be very small ($3 \leq m \leq 9$), and generate the query templates for a Cube query, *i.e.* all possible combination of Group By queries. The node cardinalities are generated similarly to Section V-C.

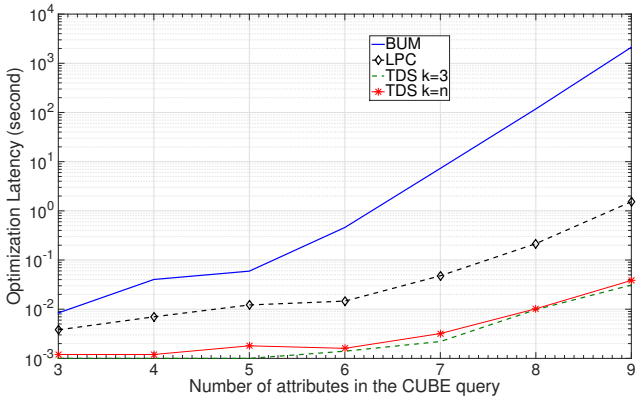

Fig. 7: Cube queries - Optimization latency

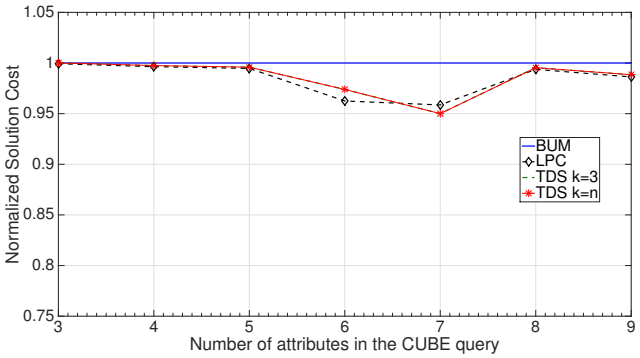

Fig. 8: Cube queries - Normalized solution cost

The optimization latency for this experiment is in Figure 7. To emphasize the difference in latency between two cases TDS $k = 3$ and TDS $k = n$, we select the *log* scale for the y-axis. From Figure 7, we see that the latency of TDS $k = 3$ is slightly lower than TDS $k = n$. Nevertheless, in both cases our algorithm still scales remarkably better than other algorithms. With $m = 9$, there are $2^9 = 512$ number of queries: it takes our algorithm less than 0.1 second to complete. As we mentioned in Section III, in the case of dense multiple Group By queries, *i.e.* large n and small m, the BUM algorithm actually scales worse than the LPC algorithm (of which the complexity in the case of a Cube query of becomes $\mathcal{O}(n^3)$).

Figure 8 shows the solution cost of different algorithms in a Cube query. Despite having the highest latency and thus more time to generate optimized plans, the BUM algorithm does not produce the best solution tree (*i.e.* lowest execution cost). The reason is that BUM starts the optimization process from a naïve solution tree where all nodes are computed directly from

the input data. For each step, BUM considers all possible pairs to merge and it selects the one with the lowest cost. As BUM is a gradient search approach, for large number of queries, there are too many paths that can lead to local optimum. In contrast to BUM, LPC and TDS start the optimization process from a viable solution tree T, which *i)* has much lower cost compared to the naïve solution tree; *ii)* has far less cases (*e.g.* paths) to consider. In the case of a Cube query, the initial solution tree T in LPC and TDS is closely similar to the final solution tree, with only some minor modifications. This helps both algorithms to achieve much lower latency. Between our algorithm and LPC, generally the solution tree obtained by LPC is slightly better than TDS (both cases). However, the differences are within 1.5%, which is acceptable if we want to trade effectiveness in finding solution tree for better scalability. For example, when $m = 9$, our algorithm runs in less than 0.05 second, while LPC runs in 1.5 second. Between two cases of TDS, we actually find very similar solution trees since they both start from similar preliminary trees.

E. Scaling with both number of queries and attributes

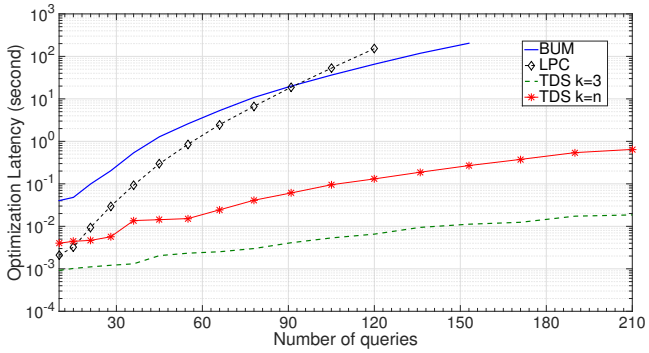

Fig. 9: Two-attribute Group By - Optimization latency

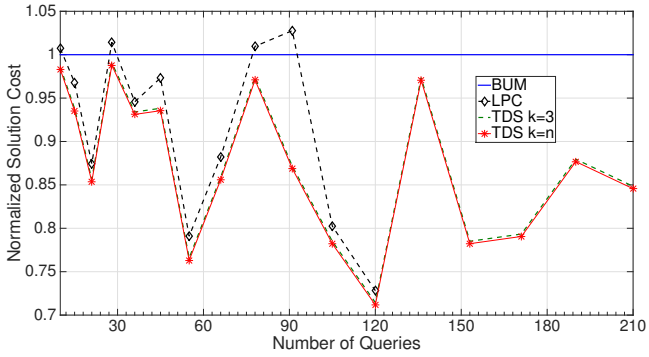

Fig. 10: Two-attribute Group By - Normalized solution cost

In this experiment, we compare optimization algorithms by scaling both the aforementioned factors at the same time: number of attributes and number of queries. To achieve such a goal, we design the query templates to include all two-attribute Group By queries from a table T. We set the number of attributes m from 5 to 21, and this makes the number of queries, which is $\binom{m}{2}$, increase as well. The grouping cardinalities are generated similarly to Section V-C.

The optimization latency shown in Figure 9 exhibits the same traits that we observe in Sections V-C and V-D. For low number of attributes, LPC has lower latencies compared to BUM. However, the space of additional nodes scales exponentially with the number of queries, so starting from $m = 15$ ($n = 91$), optimization latency of BUM gets smaller than that of LPC. Unsurprisingly, our algorithm has distinctly low overhead. In fact, the experimental results give us strong confidence that our algorithm is ready to scale up to hundreds or even thousands of attributes and queries. An end-to-end evaluation of our optimization techniques handled with the physical implementation of Group By operators is part of our on-going work, which will ultimately validate the scalability and efficiency of our approach.

Figure 10 shows the solution cost of different algorithms. Again, we normalize it to a fraction of the BUM total cost. In some cases, we have spikes where BUM merging results in even subsets, which is also the goal of TDS. In most cases, TDS actually finds smaller solution costs than that of BUM. Even in minor cases where TDS trails behind BUM, the difference is less than 3%, which is acceptable considering such a low optimization latency it brings. Table II presents the total average of execution cost obtained from each algorithm. An interesting observation here is that, despite having as much as 5 times the latency of TDS $k = 3$ (see Figure 9), the solution tree returned by TDS $k = n$ actually has less than 1% smaller total cost on average. In general, our solution trees have lower execution cost than LPC, but not by much. For some cases, both LPC and our algorithm find significantly smaller solution trees than BUM.

Algorithm	BUM	LPC	TDS $k = 3$	TDS $k = n$
Fractional Cost	1	0.9186	0.8760	0.8732

TABLE II: Average solution cost - Two attribute queries

F. The impact of cardinality skew

As we mention in Section V-C, the node cardinalities are randomly generated with two different distributions: uniform and power law with $\alpha = 2.5$. Overall, the skew introduced by the power law distribution does not affect the latency of our algorithm: on average, queries generated from both distributions have roughly the same runtime[1]. In spite of that, the total cost obtained from skewed cardinalities is generally higher than the solution cost from uniform cardinalities. On average, it is 6.8% higher, with some particular cases that are up to 18% higher. Our explanation is that, for uniform cardinalities, it is easier to *evenly* partition children nodes into subsets, while skewed cardinalities tend to return a very big subset and many small subsets. As a very big subset is used to compute other nodes, most likely it increases the solution cost by a large margin.

G. Quality of solution trees

A solution tree is a logical plan to direct the physical operator to execute multiple Group By operator. A solution

[1]Uniform distribution has a slightly higher running time of 1.4%

tree G'_1 is of higher quality than a solution tree G'_2 if the runtime to execute G'_1 is smaller than the runtime to execute G'_2. In this experiment, we implement the PipeSort algorithm as the physical operator to execute the solution trees returned from TDS, LPC and BUM. We also execute a naïve solution tree to prepare a baseline for comparison.

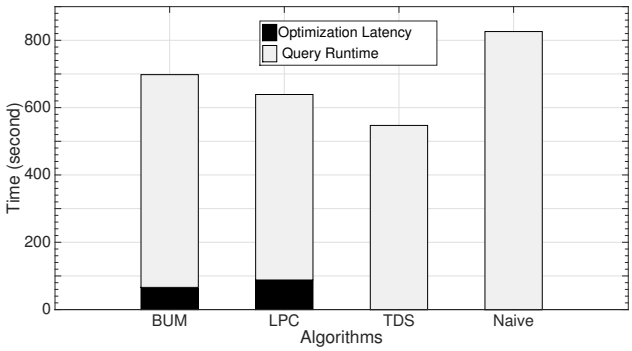

Fig. 11: The optimization latency and query runtime.

Algorithm	BUM	LPC	TDS	Naïve
Optimization Latency (s)	66	87	0.17	0
Query Runtime (s)	632	551	547	826
Total Runtime (s)	698	638	547.17	826
Improvement (%)	15.49	22.76	33.75	0

TABLE III: The optimization latency and query runtime

The dataset we use in this experiment is the *lineitem* table from the industrial benchmark TPC-H [3]. It contains 10 million records with 16 attributes. Our query consists of all two-attribute Group Bys from *lineitem*. For each algorithm, we report, in Figure 11, its optimization latency as well as its query runtime. Detail numbers found in Table III indicate that multiple Group By query optimization techniques actually reduce the query runtime over a naïve solution. We observe that, in this workload, the optimization latency of TDS is almost 0% of the total runtime. This is in contrast to that of BUM (9.45%) and LPC (13.63%). With BUM and LPC, since the optimization latency contributes a non-negligible part to the total runtime, instead of 23.48% and 33.29% improvement respectively, they improve only 15.49% and 22.76%. Also, we note that the solution trees of LPC and our algorithm, TDS, are identical. This lead to the virtually same query runtime. Nevertheless, overall our algorithm provides greater performance boost because of its close-to-zero latency.

VI. Discussions and Extensions

In this Section, first we discuss about the intuition of our algorithm, and attempt to explain why its solution cost may be better than other algorithms in many cases. Then we discuss about solutions to extend our algorithm to handle different aggregate functions.

A. Intuition and Discussion

In the multiple Group By query optimization problem, to design a scalable algorithm, the first building block to consider is how to explore the potential additional groupings (*i.e.* nodes) to include in a solution. As the space of additional groupings can be very large, they cannot be effectively generated. Therefore, to scale to a large number of queries and attributes, we cannot explore all possible additional groupings. A more scalable approach is to consider merging terminal nodes to form new groupings, as the subset of these additional groupings is much likely considerably smaller than their full space. Both our algorithm and BUM in [11] use this approach.

The difference between TDS and BUM is the process to construct new groupings. At each step, BUM only considers merging two groupings into a new one. Meanwhile, TDS evaluates splitting all children nodes of a grouping into at most k subsets, each with a new grouping. The implication of these steps on the algorithm's complexity is already discussed in Section III and IV-B. Here, we intuitively discuss why, in general, we believe that TDS can produce better solution trees than BUM. The main reason is because TDS makes a more *"global"* decision than BUM at each step of their process. When considering partitioning children nodes, TDS uses available information at the moment: *i)* cardinalities of all nodes; *ii)* associated costs to pair of nodes; *iii)* multiple ways to split. In addition, while it is making a decision of putting together a new grouping, TDS inspects the connection of this newly formed grouping with respect to all other groupings available. As a top-down approach, when TDS triggers a splitting decision in a high-level grouping (*e.g.* the root node), it dramatically decreases the total execution cost. Even though the subtree optimization might be far from optimal, early decisions are more important.

In contrast, the merging process in BUM solely depends on two individual nodes. With so little information at hand, BUM tends to trigger groupings that decrease the solution cost by a relatively small margin (because BUM is a bottom-up approach). In addition, since the initial solution tree is a naïve solution, there are so many pairs of nodes such that inspecting the potential merging of all pairs leads to a local optimum.

B. Different Aggregate Functions

The multiple Group By optimization problem is based on the premise that a Group By can be computed from the results of another Group By, instead of the input data. To assure this property, the aggregate measure (function) must be either *distributive*, or *algebraic* [5]: fortunately, almost all common aggregate functions are so. Let us consider an input data T which is split into p chunks C_i. A function F is:

- *Distributive* if there is a function G such that: $F(T) = G(F(C_1), F(C_2), ..., F(C_p))$. Usually, for many distributive functions like Min, Max, Sum, *etc.*, $G = F$. For $G \neq F$, an example is $Count$ which is also distributive with $G = sum()$.
- *Algebraic* if there are functions G and H such that: $F(T) = H(G(C_1), G(C_2), ..., G(C_p))$. Examples are $Average, MinN, MaxN, Standard\_deviation$. In function $Average$, for instance, G is to collect the sum of elements and the number of elements, while H adds up these two components and divides the global sum by the total number of elements from all the chunks to obtain final results.

Thus far, we have assumed that all Group By queries compute the same aggregate function ($Count(*)$ in our example). Typically this is the case in single-query optimization when a user issues a Grouping Sets query. Nonetheless in multi-query optimization, more often the aggregate functions are different. An easy way to adapt our solution to different aggregate functions is to separate Group By queries into groups of the same aggregate function. However, this approach may decrease the opportunity to share pre-computed Group Bys, and it may end up computing a large portion of Group Bys from the input data (or from a much larger Group Bys). For instance, let us consider the following queries:

```
Q1: Select A, Count(*) From T Group By(A)
Q2: Select B, Sum(v) From T Group By(B)
```

Here v is an integer value in table T. Using the aforementioned approach we end up with both Group Bys A and B computed from the input data T.

Another approach is to apply our optimization to the set of all Group Bys queries. For a Group By, not only its aggregates are computed, but also are all those of their successors. To continue our example, our algorithm suggests computing grouping AB from T, then A and B from AB. When computing AB from T, we evaluate and store both aggregates, $Count(*)$ and $Sum(v)$. At this moment, Group By $Q1$ is obtained from grouping AB with aggregate $Count(*)$, while we use AB with aggregate $Sum(v)$ to calculate Group By $Q2$. The downside of this approach is to incur the cost of storing potentially numerous intermediate aggregates.

For systems in which the storage cost is relatively expensive compared to reading/sorting a large amount of data, the first approach may be preferred. For other systems, in which the sorting cost is relatively expensive (*e.g.* it requires a global shuffle of data over network), the second approach may be a more viable option. Facilitating users in making the right choice of approaches is the challenge that we will tackle in our future work.

VII. Conclusion and Future Work

Data summarization is a crucial task to understand and interact with data. This is exacerbated by the increasingly large amount of data that is collected nowadays. Such data is often multi-dimensional, characterized by a very large number of attributes. This calls for the design of new algorithms to optimize the execution of data summarization queries.

In this work, we presented our method to address the general problem of optimizing multiple Group By queries, thus filling the gap left by current proposals that cannot scale in the number of concurrent queries or the number of attributes each query can handle. We have shown, both theoretically and experimentally, that our algorithm incurs in extremely small latencies, compared to alternative algorithms, when producing optimized query plans. This means that, in practice, our algorithm can be applied at the scale that modern data processing tasks require, dealing with data of hundreds of attributes and executing thousands of queries. In addition, our experimental evaluation illustrated the effectiveness of our algorithm to find optimized solution trees. In fact, in many cases, our algorithm outperformed others in terms of producing optimized solutions, while being remarkably faster. Finally, we discussed about the intuition behind our algorithm and the possible approaches to extend it to handle general, heterogeneous queries in terms of diversity of aggregate functions.

We conclude by noting that our algorithm can be easily integrated to current optimization engines of relational databases or traditional data warehouses. Instead, using our algorithm to optimize query execution on recent systems such as Hadoop, Spark and their respective high-level, declarative interfaces, requires the development of an appropriate cost model as well as an optimization engine to transform original query plans into optimized ones.

Our future work aims at *i)* further studying our algorithm with respect to the analysis of its computational complexity; *ii)* evaluating our algorithm in an end-to-end system implementation; *iii)* integrating our proposed algorithm into a multi-query optimization engine for Spark, that we have designed and built such that it can accept arbitrary optimization modules, including one for multiple Group By queries.

References

[1] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM TCS*, 2008.

[2] J. Dean and S. Ghemawat, "MapReduce : Simplified Data Processing on Large Clusters," in *ACM OSDI*, 2004.

[3] "Tpc-h, decision support benchmark." [Online]. Available: http://www.tpc.org/tpch/

[4] "Tpc-ds, new decision support benchmark standard." [Online]. Available: http://www.tpc.org/tpcds/

[5] J. Gray *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *DMKD*, 1997.

[6] V. Harinarayan *et al.*, "Implementing data cubes efficiently," in *Proc. ACM SIGMOD*, 1996.

[7] S. Sarawagi *et al.*, "On computing the data cube," IBM Almaden Research Center, Tech. Rep., 1996.

[8] R. T. Ng *et al.*, "Iceberg-cube computation with PC clusters," *ACM SIGMOD Record*, 2001.

[9] K. Beyer and R. Ramakrishnan, "Bottom-up computation of sparse and iceberg cubes," in *ACM SIGMOD*, 1999.

[10] K. A. Ross and D. Srivastava, "Fast computation of sparse datacubes," in *PVLDB*, 1997.

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

[12] S. Ballamkonda *et al.*, "Evaluation of Grouping Sets by reduction to group-by clause, with or without a rollup operator, using temporary tables," Patent US Patent 6,775,681, 2004.

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

[14] D.-H. Phan *et al.*, "Efficient and self-balanced rollup aggregates for large-scale data summarization," in *BigData Congress*, 2015.

[15] H. D. Phan *et al.*, "On the design space of MapReduce ROLLUP aggregates," in *BeyondMR*, 2014.

[16] Y. Chen *et al.*, "Interactive analytical processing in big data systems: A cross-industry study of MapReduce workloads," in *PVLDB*, 2012.

[17] J. Cieslewicz and K. A. Ross, "Adaptive aggregation on chip multiprocessors," in *PVLDB*, 2007.

[18] A. Baer *et al.*, "Two parallel approaches to network data analysis," in *LADIS*, 2011.

[19] S. Agarwal *et al.*, "On the computation of multidimensional aggregates," in *Proc. VLDB*, 1996.

[20] P. J. Haas *et al.*, "Sampling-based estimation of the number of distinct values of an attribute," in *PVLDB*, 1995.

[21] S. Chaudhuri *et al.*, "Effective use of block-level sampling in statistics estimation," in *Proc. ACM SIGMOD*, 2004.