Efficient and Provable Multi-Query Optimization

Tarun Kathuria
Microsoft Research
tarunkathuria@gmail.com

S. Sudarshan
Indian Institute of Technology Bombay
sudarsha@cse.iitb.ac.in

ABSTRACT

Complex queries for massive data analysis jobs have become increasingly commonplace. Many such queries contain common subexpressions, either within a single query or among multiple queries submitted as a batch. Conventional query optimizers do not exploit these subexpressions and produce sub-optimal plans. The problem of multi-query optimization (MQO) is to generate an optimal *combined* evaluation plan by computing common subexpressions once and reusing them. Exhaustive algorithms for MQO explore an $\mathcal{O}(n^n)$ search space. Thus, this problem has primarily been tackled using various heuristic algorithms, without providing any theoretical guarantees on the quality of their solution.

In this paper, instead of the conventional cost minimization problem, we treat the problem as maximizing a linear transformation of the cost function. We propose a greedy algorithm for this transformed formulation of the problem, which under weak, intuitive assumptions, provides an approximation factor guarantee for this formulation. We go on to show that this factor is optimal, unless $\mathsf{P} = \mathsf{NP}$. Another noteworthy point about our algorithm is that it can be easily incorporated into existing transformation-based optimizers. We finally propose optimizations which can be used to improve the efficiency of our algorithm.

Keywords

Approximation algorithms; hardness of approximation; Multi-query optimization

1. INTRODUCTION

Modern data analytics platforms frequently have to run scripts that contain a large number of complex queries. Often, these queries contain common subexpressions due to the nature of the analysis performed. These subexpressions may occur within a single complex query which i) contains multiple correlated nested subqueries or ii) if the database contains many materialized views which are referenced multiple times in the query. A more interesting case where common subexpressions arise is when a batch of related queries are being executed together.

Conventional query optimizers are not suited for such scenarios since they do not exploit these subexpressions and instead produce locally optimal plans for each query. These plans can be globally sub-optimal since they do not make use of the shared subexpressions while generating the plans. The goal of multi-query optimization (MQO) is to generate query plans where these subexpressions are executed once and their results used by multiple consumers. The best plan is selected in a completely cost-based manner.

We now present an example to illustrate the MQO problem and how locally optimal plans may be globally sub-optimal for multiple queries in the presence of common subexpressions.

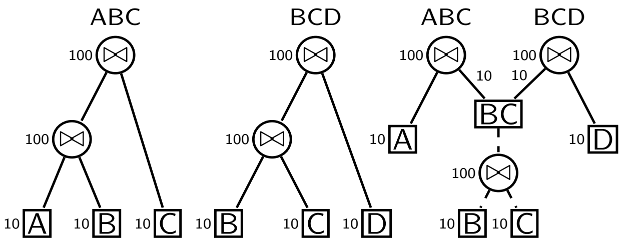
Example 1. (Example 1.1 in [25]) Consider a batch consisting of two queries ($A \bowtie B \bowtie C$) and ($B \bowtie C \bowtie D$) whose locally optimal plans (i.e., individual best plans) are ($A \bowtie B$) $\bowtie C$ and ($B \bowtie C$) $\bowtie D$ respectively. The individual best plans for the two queries do not have any common subexpressions. However, consider a locally sub-optimal plan for the first query $A \bowtie (B \bowtie C)$. It is clear that $(B \bowtie C)$ is a common subexpression and can be computed once and used by both queries.

Consider the following instantiation of the various costs for the two queries shown in Figure 1. Suppose the base relations A, B, C and D each have a scan cost of 10 units. Each of the joins have a cost of 100 units, giving a total evaluation cost of 460 units for the locally optimal plans shown in Figure 1a. On the other hand, in the plan shown in Figure 1b, the common subexpression $(B \bowtie C)$ is first computed and materialized on the disk at a cost of 10. Then, it is scanned twice - the first time to join with A in order to compute the first query, and the second time to join it with D in order to compute the second - at a cost of 10 per scan. Each of these joins have a cost of 100 units. Thus, the total cost of this consolidated plan is 370 units, which is lesser than the cost of the locally optimal plan in Figure 1a.

It should be noted that blindly sharing a subexpression may not always lead to a globally optimal strategy. For example, there may be cases where the cost of joining the subexpression $(B \bowtie C)$ with A is very large compared to the cost of the plan $(A \bowtie B) \bowtie C$; in such cases it may make no sense to reuse $(B \bowtie C)$ even if it were available. □

The benefits of a good algorithm for MQO are not just restricted to multiple queries in a batch but can also be used to find better plans for a single complex query. Consider an example of a large query consisting of multiple subqueries

PODS'17, May 14-19, 2017, Chicago, Illinois, USA
© 2017 ACM. ISBN 978-1-4503-4198-1/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3034786.3034792



a) Traditional execution
No sharing
Total cost=460

b) Execution with
BC shared
Total cost=370

Figure 1: MQO example (from [25]) illustrating benefit of sharing subexpressions

with a common subexpression between two subqueries. Traditional query optimizers do not consider such sharing, but multi-query optimization techniques have been developed to find the best plans taking such sharing into account, such as [26, 27, 25, 32, 28] . While the early work on multi-query optimization, e.g. [26, 27], focused on queries with only selections and joins, later work, e.g. [25, 32, 28], which are based on the Volcano/Cascades query optimization framework [9, 10] use an AND-OR DAG representation of the query plan space to handle arbitrary queries.

Dynamic-programming techniques for join order enumeration, as well as transformation-rule based optimization techniques based on the Volcano/Cascades framework with optimizations described in [21], run in time $\mathcal{O}(3^n)$ for a query that computes the join of n relations, when all join orders are considered.

However, when these techniques are extended to handle multi-query optimization, they need to consider all subexpressions that are potentially shared by multiple query plans, or multiple parts of the same query plan. The number of such common subexpressions can be $\mathcal{O}(2^n)$ when we consider join queries involving n relations. An optimal plan for the set of queries may materialize and share up to n of these common subexpressions. A naive exhaustive algorithms for MQO would consider all such subsets of cardinality n, leading to a very high cost. The best known exhaustive algorithm takes $\mathcal{O}(n^n)$ time [31], which is infeasible for even moderate numbers of relations. Thus, work in this area relies on heuristics to restrict the space of alternatives considered [25, 28, 32]. While such algorithms seems to work well in practice, to the best of our knowledge there has been no work that provides theoretical guarantees on the quality of solution obtained by such heuristics.

Thus, an open question is

Can we devise an algorithm which runs in time polynomial in the number of shared nodes (common subexpressions) which provides us with theoretical guarantees on the quality of the solution obtained as compared to the optimal? If so, what is the best possible polynomial-time approximation algorithm?

As a first step towards answering this question, we propose a reformulation of the MQO problem, the motivation for which is stated next.

The canonical multi-query optimization problem is concerned with minimizing cost of the query plan for a set of queries by choosing a set of nodes to materialize (say M) and then finding the optimal plan exploiting nodes in M. Another way to look at this problem is to maximize the "materialization-benefit" we get by materializing M *with respect to* a naive execution plan which is locally optimal and does not exploit any common subexpressions. More formally, this corresponds to maximizing the difference of the cost of the best plan in which the set of materialized nodes is M from the latter. As this is just a linear transformation of the cost function, it is clear that the maximizer of the materialization-benefit will be the minimizer of the cost.

Roy et al. [25] assume a property on the cost function that they call the "monotonicity heuristic". This essentially corresponds to assuming the supermodularity of the cost function defined on the set of nodes to be materialized. In [25], this assumption is used to speed up their greedy algorithm via a heap-based argument which exploits the supermodularity. This is similar to the LazyGreedy algorithm described in [16] for speeding up monotone submodular function maximization subject to cardinality constraints via the well-known greedy algorithm, which is also used by [25]. On the queries used in their experiments, it was observed that the plan obtained with or without assuming supermodularity led to the same plan. This seems to imply that the supermodularity assumption may be a reasonable one and may hold in practice.

1.1 Our contribution

The contributions of this paper are as follows

- Motivated by [25], we proceed with the "monotonicity heuristic" assumption (which implies the submodularity of the materialization benefit function). *Under this assumption*, we propose an approximation algorithm for the *underlying problem* of unconstrained, normalized submodular maximization (UNSM). Note that we allow the submodular function to take negative values, which has not been considered previously and poses a significant challenge[1]. Our algorithm runs in time $\mathcal{O}(u^2)$, where u is the size of the universe. In the MQO setting, where u is the number of shared nodes, this translates to a $\mathcal{O}(2^{2n})$ time algorithm instead of the exhaustive $\mathcal{O}(n^n)$ algorithm.

- We then present a hardness of approximation proof for the UNSM problem, which matches that obtained by our algorithm, assuming $\mathsf{P} \neq \mathsf{NP}$.

- We present optimizations to our algorithm to improve the running time of the algorithm, without sacrificing any theoretical guarantees.

- We also consider a special case of the problem of submodular maximization under cardinality constraints.
  - A natural extension to our greedy algorithm for this problem is presented. We further propose a pruning strategy to reduce the search space before running our greedy algorithm, by exploiting this cardinality constraint.
  - While, at this point, we do not formally prove any theoretical guarantees on the approximation factor for this constrained problem, we show that the answer obtained by our greedy algorithm is the same when run with or without this pruning.

[1] Inapproximability results when the submodular function may be unnormalized are well known [8].

- We compare our algorithm against the Greedy algorithm and stand-alone Volcano (without MQO) on queries from the TPCD benchmark and show significant benefits.

It is important to note that our approximation guarantees are for the benefit-maximization problem, under the submodularity assumption, and do not imply a multiplicative factor approximation to the cost minimization problem. However, results in our experimental section shows that our proposed algorithm performs as well as or better than the Greedy heuristic of [25].

Our techniques for the problem of multi-query optimization are presented in the context of query optimizers based on the Volcano/Cascades framework [10, 9]. This framework for optimizing queries uses transformation rules which makes it inherently extensible, and has been implemented in several widely-used commercial database systems such as Microsoft SQL Server. It should be noted, however, that our algorithm is agnostic to the query optimization framework, and can be easily extended to other frameworks as well.

Organization. In Section 2, we present a detailed overview of multi-query optimization in the context of the Volcano framework which was presented in [25] along with how submodular maximization arises in this context. Section 3 presents our greedy algorithm for unconstrained, normalized submodular maximization with the proof of its approximation factor guarantee. In Section 4, we prove the hardness of approximation of the unconstrained, normalized submodular maximization which rules out better approximation factors than the one attained by our algorithm, under the assumption of $\mathsf{P} \neq \mathsf{NP}$. Section 5 presents ways to speed up our algorithm. We present experimental results on benchmark queries in Section 6. Related work in the areas of MQO and submodular maximization is presented in Section 7. We conclude and discuss directions for future work in Section 8.

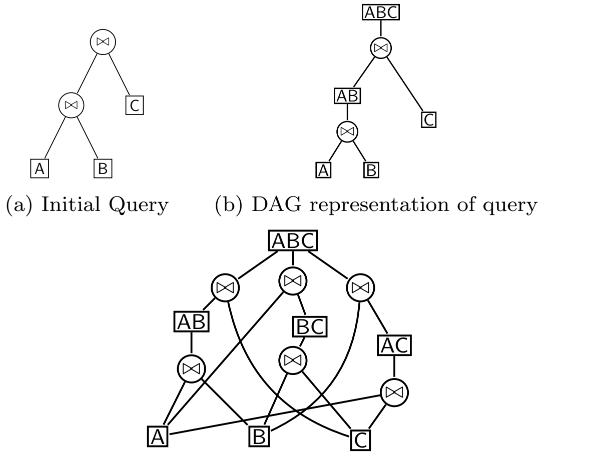
2. PRELIMINARIES

This section presents some relevant background in (Multi)-Query Optimization in the Volcano framework followed by some preliminaries of submodular maximization and finally ends with how submodular maximization arises in MQO. Readers well-versed in MQO techniques in Volcano may skip to the Section 2.3 directly.

2.1 Query Optimization in Volcano

The Volcano/Cascades query optimization framework [10, 9] is based on a system of equivalence rules, which specify that the result of a particular transformation of a query tree is the same as the result of the original query tree. The key aspect of this framework is the efficient implementation of the transformation rule-based approach.

The Volcano framework uses the AND-OR DAG representation [10, 23] for compactly representing the given query and its alternative query plans. An AND-OR DAG is a directed acyclic graph whose nodes can be divided into AND-nodes and OR-nodes; the AND-nodes have only OR-nodes as children and the OR-nodes have only AND-nodes as children. An AND-node corresponds to an algebraic operator, such as the join operator ($\bowtie$) or a select operator (σ). It represents the expression defined by the operator and its inputs. An OR-node represents a set of logical expressions that generate the same result set; the set of such expressions is defined by the children AND nodes of the OR node, and their inputs. Hereafter, we refer to the OR-nodes and AND-nodes as equivalence nodes and operator nodes respectively.



(a) Initial Query

(b) DAG representation of query

(c) Expanded LQDAG after transformation (Commutativity not shown explicitly)

Figure 2: Initial Query and LQDAG Representation

The given query tree is initially represented in the AND-OR DAG formulation. For example, the query tree of Figure 2a is initially represented in the AND-OR DAG formulation, as shown in Figure 2b. Equivalence nodes are shown as boxes, while operator nodes are shown in circles.

The initial AND-OR DAG is then expanded by applying all possible logical transformations on every node of the initial DAG created from the given query. Suppose the only possible transformations are join associativity and commutativity. Then the plans $A \bowtie (B \bowtie C)$ and $(A \bowtie C) \bowtie B$, as well as several plans equivalent to these, modulo commutativity, can be obtained by transformations on the initial AND-OR DAG of Figure 2b. These are represented in the DAG shown in Figure 2c. The AND-OR DAG representation after applying all the logical tranformations is called the (expanded) Logical Query DAG (or LQDAG).

Each operator node can have different physical implementations; for example, a join operator can be implemented as a hash join, a nested loop join or as a merge join. Once the LQDAG has been generated, physical implementation rules are applied on the logical operators to generate the physical AND-OR DAG, which is called the Physical Query DAG or PQDAG for short.

Properties of the results of an expression, such as sort order, that do not form part of the logical data model are called physical properties [10]. The importance of exploiting physical properties such as sort order and partitioning of result sets is well known in traditional query optimization. The DAG is actually built and stored using a "memo" structure, a concise data structure used in the Volcano/Cascades framework to represent the entire space of equivalent query evaluation plans succintly. The AND-OR DAG representation considered for MQO actually works on the PQDAG but we present our algorithms to work at the LQDAG level for brevity.

2.2 Multi-Query Optimization in Volcano

This subsection primarily focuses on the techniques presented in [25] for MQO in the Volcano framework. In order

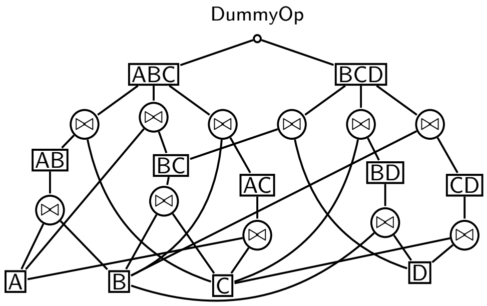


Figure 3: Combined LQDAG for queries in Example 1

to extend the Volcano AND-OR DAG generation for MQO on a batch of queries to be jointly optimized, the queries are represented together in a single DAG, sharing subexpressions. The DAG is converted to a rooted DAG by adding a dummy operation node, which does nothing, but has the root equivalence nodes of all the queries as its inputs.

The two main challenges for a multi-query optimizer are :

1. Recognizing possibilities of shared computation by identifying common subexpressions.
2. Finding a globally optimal evaluation plan exploiting the common subexpressions identified.

Roy et al. [25] present an efficient hashing-based algorithm that identifies the set of all common subexpressions, including subqueries that are syntactically different but semantically equivalent, in a single bottom-up traversal of the LQDAG by using the "memo" structure; for details see [25]. This is similar to the "expression fingerprinting" used to identify the common subexpressions in [28]. The combined LQDAG for the queries of Example 1 is shown in Figure 3. This step takes exponential time as the size of the DAGs may itself be exponential and is unavoidable, even in single-query optimization.

Similar to the single query optimization done by Volcano, in a single-pass, one can annotate each node in the DAG with its estimated cost. Note that the cost estimator functions are taken as input to the optimizer, i.e., the optimizer algorithm is agnostic to the cost estimates. Indeed, this is one of the reasons why the Volcano query optimizer framework is widely used. It is important to note that in the single query optimization as well as the multi query optimization setting, one assumes that the cost estimates provided to us are correct for any guarantees to hold. Thus, we also work under the assumption that the cost estimates are correct. After the common subexpressions are identified and the cost of each node computed, the next task is to find the best consolidated plan for the queries exploiting the subexpressions.

In this paper, we are primarily concerned with the optimization philosophy adopted by the Greedy algorithm in [25] which is presented next. For a set of equivalence nodes S, let $bestCost(Q, S)$ (for brevity, $bc(S)$) denote the cost of the optimal plan for Q given that nodes in S are to be materialized (this includes the cost of computing and materializing nodes in S). Here Q is the combined query DAG with the dummy root operator node with inputs being the DAGs of $Q_1, \ldots, Q_k$, as described above. The $bc(S)$ function, of course, depends on the cost estimates and is treated as a black-box for the MQO algorithms. Given a set of nodes S to be materialized, [25] present an efficient scheme to find the best plan and the best cost, $bc(S)$ (this includes the cost of materializing S, which may be done in multiple ways and is figured out by the optimizer in [25] as well).

Now, we just need to identify the subset S of nodes in the AND-OR DAG for which $bestCost(Q, S)$ is minimum. However, an exhaustive algorithm which enumerates all possible subsets S will take time exponential in the size of the AND-OR DAG, which itself may be exponential in the number of relations. In [25], an intuitive greedy algorithm is proposed, which iteratively picks which node to materialize. At each iteration, the node x that gives the maximum reduction in the cost, if materialized, is chosen to be added to the current set of materialized nodes X. While this greedy algorithm is shown to work well in practice, [25] does not provide any theoretical guarantees on the quality of solution obtained via this algorithm. The algorithm is presented below for completeness.

Algorithm 1 Greedy Algorithm of [25]

```
$X = \emptyset$
$Y =$ Set of shareable equivalence nodes in the DAG
while $Y \neq \emptyset$ do
    Pick $x \in Y$ which minimizes $bc(X \cup \{x\})$
    if $bc(X) > bc(X \cup \{x\})$ then
        $X = X \cup \{x\}, Y = Y \setminus \{x\}$
    else
        $Y = \emptyset$
    end if
end while
return $X$
```

As noted in [25], the nodes materialized in the globally optimal plan are just a subset of the ones that are shared in some plan for the query. It is, thus, sufficient to search only over the set of *shareable* equivalence nodes, instead of searching over the entire set of equivalence nodes in the DAG.

Clearly, some assumptions on the cost function have to be made in order to give theoretical guarantees for any algorithm. Furthermore, it is desirable to make assumptions which may hold in practice. Roy et al. [25] make an additional assumption which they call the "monotonicity heuristic".
Define $benefit(x, X)$ as $bc(X) - bc(X \cup \{x\})$. The assumption is that

$$\forall\ Y \subseteq X,\ \forall\ x \notin X,\ benefit(x, X) \leq benefit(x, Y).$$

They [25] make this assumption in order to improve the running time of their greedy algorithm via a heap-based argument which corresponds to the LazyGreedy algorithm [16] for faster monotone, submodular maximization. Their experiments, however, show that the plans obtained with and without the assumption had exactly the same cost. While the assumption may not always hold, their experiments seem to indicate that the assumption may be a reasonable one, in practice. Thus, in this paper, we work under this assumption to devise an algorithm *with theoretical guarantees* on its performance for maximizing the "materialization benefit".

2.3 Submodular Maximization

Let U be a universe of $n = |U|$ elements, let $f : 2^U \to \mathbb{R}$ be a function. For simplicity, we use the notation $f'(u, S)$ to denote the incremental value in f of adding u to S, i.e., $f'(u, S) = f(S \cup \{u\}) - f(S)$.

Definition 1. *(Submodular Functions)*
A function $f : 2^U \to \mathbb{R}$ is called submodular if

$$\forall\ A \subseteq B \subseteq U,\ \forall\ u \in U \setminus B, \textit{we have } f'(u, A) \geq f'(u, B).$$

Definition 2. *(Supermodular Functions)*
A function $f : 2^U \to \mathbb{R}$ is called supermodular if

$$\forall\ A \subseteq B \subseteq U,\ \forall\ u \in U \setminus B, \textit{we have } f'(u, A) \leq f'(u, B).$$

Definition 3. *(Additive Functions)*
A function $c : 2^U \to \mathbb{R}$ is called additive if it is of the form $c(S) = \sum_{e \in S} c(\{e\})$.

Definition 4. *(Monotone Functions)*
A function $f : 2^U \to \mathbb{R}$ is said to be monotone if

$$\forall A \subseteq B \subseteq U, \textit{we have } f(A) \leq f(B).$$

Definition 5. *(Normalized Functions)*
A function $f : 2^U \to \mathbb{R}$ is called normalized if $f(\emptyset) = 0$.

Given a normalized submodular function $f : 2^U \to \mathbb{R}$, the unconstrained, normalized submodular maximization (UNSM) problem is to find a set $S \subseteq U$ which maximizes the value of f, i.e., $\arg\max_{S \subseteq U} f(S)$.

Since submodular maximization problems are in general NP-hard and can only be approximated, a simple additive scaling of the function by a large constant to make the function non-negative and running an algorithm like [2] suffers in the approximation factor and moreover does not guarantee a multiplicative approximation.

It is well-known that any non-monotone submodular function f, with the constraint that $f(\emptyset) = 0$, can be written as the difference of a non-negative monotone submodular function f_M and an additive "cost" function c. However, multiple such decompositions are possible and as we will show, there is one particular decomposition (the decomposition in Proposition 1) which will give us the best approximation ratio and a matching hardness of approximation.

Proposition 1. *Any normalized, non-monotone (which may take negative values) submodular function f can be decomposed as*

$$f(S) = f_M(S) - c(S) \quad , \forall\ S \subseteq U$$

where f_M is a monotone submodular function and c is an additive cost function. In particular, one possible decomposition is

$$f^*_M(S) = f(S) + \sum_{e \in S} (f(U \setminus \{e\}) - f(U))$$
$$c^*(S) = \sum_{e \in S} (f(U \setminus \{e\}) - f(U))$$

Proof. The proof is provided in Appendix A. □

Since our approximation ratio depends on the decomposition and owing to the importance of the decomposition in Proposition 1, we refer to it as f^*_M and c^*.

2.4 Multi-Query Optimization and UNSM

We now describe the changes to the MQO formulation of [25] and show the role submodularity plays in the same. As defined above, $bestCost(Q, S)$ includes the cost of computing and materializing the set of PQDAG nodes to be materialized S. Consider a scenario where S was already materialized and we just have to find the optimal plan which *may or may not* use the materialized nodes in S. However, no further nodes may be chosen to be materialized. The cost of the optimal plan can be thought of as the *best use cost* and the function is thus called $bestUseCost(Q, S)$. This function is monotonically decreasing since as more nodes are materialized, we will exploit the additional nodes only if they lead to a reduction in cost. Of course, the cost of materializing S needs to be taken into account and we call that function $c(S)$. Clearly, $bestCost(Q, S) = bestUseCost(Q, S) + c(S)$. For brevity, we refer to $bestUseCost(Q, S)$ as $buc(S)$.

The MQO problem can be thought of as maximizing the "materialization-benefit" ($mb(S)$ for brevity) we get in the plan cost by exploiting common subexpressions over a naive execution plan which is just locally optimal and does not exploit subexpressions. Clearly the cost of the latter is $bc(\emptyset) = buc(\emptyset)$. Mathematically, $mb(S)$ is defined as

$$\begin{aligned} mb(S) &= bc(\emptyset) - bc(S) \\ &= buc(\emptyset) - (buc(S) + c(S)) \\ &= (buc(\emptyset) - buc(S)) - c(S) \end{aligned}$$

The function in parenthesis in the last line is a monotonically increasing function since $buc(S)$ is a monotonically decreasing function. Also, if the set of materialized nodes S are "far apart" in the PQDAG, the cost of computing and materializing a node $e \in S$ can be thought of as being independent of the other nodes in S. This motivates us to assume that the c function is additive. Of course, this assumption need not be true. For example, if two of the equivalence nodes in S are just below each other, we can significantly benefit by computing the "lower" node and then just reading it from disk to compute the "upper" node. As proved in Proposition 1, under the assumption of submodularity, mb can always be decomposed into a difference of monotone, submodular function and an additive function[2]. Observe that

$$\forall X,\ \forall x \notin X, benefit(x, X) = -bc'(x, X)$$

Thus, the "monotonicity heuristic" assumption is essentially that the *bestCost* function is supermodular. This implies that mb is submodular. Note that mb is normalized. Thus, the problem is essentially the UNSM problem with mb as the submodular function. The reason why materialization benefit for a particular set of nodes may be negative is due to the fact that there may be certain nodes which may have very high materialization cost but may not have high benefit.

3. THE MARGINAL GREEDY ALGORITHM

In this section, we propose a greedy algorithm for the UNSM problem for which we prove an approximation guarantee. A proof of a matching hardness of approximation, under the assumption of P ≠ NP is presented in the next section.

Given a decomposition of a non-monotone, normalized submodular function f, let the monotone submodular and

[2] The decomposition in Proposition 1 does not actually correspond to the cost of materializing nodes but parallels are drawn for intuition

additive functions be denoted by f_M and c. Thus, the problem we want to solve is as follows

$$\max_{S \subseteq U} f(S) = \max_{S \subseteq U} f_M(S) - c(S)$$

The MarginalGreedy algorithm (Algorithm 2) has been proposed before [30], albeit for non-negative, *monotone* submodular maximization under knapsack constraints. At each iteration, the algorithm greedily selects the element with the highest use-benefit to cost ratio from those elements which satisfy a knapsack constraint. In our case, however, there is no knapsack constraint and instead we add elements as long as it leads to an increase in the value of f. We emphasize that the problem in our case is considerably different than this problem and highlight the differences in subsection 3.1.

Algorithm 2 MarginalGreedy Algorithm

```
$X = \emptyset$
$Y =$ Set of shareable equivalence nodes in the DAG
while $Y \neq \emptyset$ do
    Pick $x \in Y$ which maximizes $r(x, X) = \frac{f'_M(x,X)}{c(\{x\})}$
    if $r(x, X) > 1$ then
        $X = X \cup \{x\}, Y = Y \setminus \{x\}$
    else
        $Y = \emptyset$
    end if
end while
return $X$
```

The MarginalGreedy algorithm also finally adds all elements with negative c values. This was also done in Sviridenko's case [30] as one can only increase the value of the function without increasing the budget. This is fine for us as well and can only raise the value of the function f. This is because f_M is monotone so including more elements only raises its value and we are subtracting off some negative c values which can only raise the value of f. If the decomposition used is the one given in Proposition 1, we can compute the term in the summation for each element once and store it. This can be done in just $n + 1$ $bc(S)$ invocations (for the sets U and for $U \setminus \{e_i\}$ $\forall e_i \in V$). We note that while Algorithm 2 is presented referencing shared nodes in the DAG, the algorithm works for any instance of UNSM with an arbitrary universe of elements U.

3.1 Approximation Factor of Marginal Greedy

Let Θ be an optimal solution. Let X_i denote the set of nodes selected by Algorithm 2 just after the i^{th} iteration. Define $\Delta_{f_M}(E, S) = f_M(S \cup E) - f_M(S)$, where E and S are subsets of U.

We state the main theorem of this section which mentions the approximation guarantee Algorithm 2 provides. The approximation factor is not a constant and instead depends on the value of the f and c functions at optimal.

Theorem 1. *The answer obtained by the MarginalGreedy algorithm (X) satisfies the following inequality*

$$f(X) \geq \left[1 - \frac{c(\Theta)}{f(\Theta)} \ln(1 + \frac{f(\Theta)}{c(\Theta)})\right] f(\Theta).$$

We prove the theorem after presenting a lemma and its corollary which are central to the proof. At a high level, the lemma states that upto a certain point in the execution of the algorithm, there exists an element that can be picked and has a marginal-benefit to cost ratio which is at least the marginal-benefit to cost ratio we would get if we picked all remaining elements in the optimal solution.

Lemma 1. *At any iteration $i+1 < n$ in the execution of the MarginalGreedy algorithm, if $f_M(X_i) < f(\Theta)$, then there exists some element $e \in \Theta \setminus X_i$ that satisfies*

$$\frac{\Delta_{f_M}(\{e\}, X_i)}{c(\{e\})} \geq \frac{\Delta_{f_M}(\Theta, X_i)}{c(\Theta)}.$$

Proof. Firstly, note that if

$$f_M(X_i) < f(\Theta) = f_M(\Theta) - c(\Theta) \leq f_M(\Theta),$$

then $\Theta \backslash X_i \neq \emptyset$. This is because f_M is monotonically increasing. Also, note that if S is fixed, $\Delta_{f_M}(E, S)$ is a submodular function in E, due to submodularity of f_M.

We consider two cases. Since the f_M function is monotonically increasing, the numerators on both sides of the inequality are non-negative.

Case 1. $\Delta_{f_M}(\Theta, X_i) = 0$
In this case, the RHS of the inequality is 0. Since the f_M function is monotonically increasing, $\forall e' \in \Theta \setminus X_i$, we have

$$\frac{\Delta_{f_M}(e', X_i)}{c(\{e'\})} \geq \frac{\Delta_{f_M}(\Theta, X_i)}{c(\Theta)}.$$

Since $\Theta \setminus X_i \neq \emptyset$, any element $e' \in \Theta \setminus X_i$ satisfies the required inequality.

Case 2. $\Delta_{f_M}(\Theta, X_i) > 0$
We first show that there exists some element $e \in \Theta$ for which the inequality holds. Assume the contradiction, i.e.,

$$\forall e \in \Theta, \frac{\Delta_{f_M}(\{e\}, X_i)}{c(e)} < \frac{\Delta_{f_M}(\Theta, X_i)}{c(\Theta)}.$$

$$\therefore c(e)(\Delta_{f_M}(\Theta, X_i)) > c(\Theta)(\Delta_{f_M}(\{e\}, X_i)).$$

Summing up over all $e \in \Theta$, we get

$$\sum_{e \in \Theta} c(e)(\Delta_{f_M}(\Theta, X_i)) > \sum_{e \in \Theta} c(\Theta)(\Delta_{f_M}(\{e\}, X_i))$$
$$\implies (\Delta_{f_M}(\Theta, X_i)) \sum_{e \in \Theta} c(e) > c(\Theta) \sum_{e \in \Theta} (\Delta_{f_M}(\{e\}, X_i))$$
$$\implies (\Delta_{f_M}(\Theta, X_i)) c(\Theta) > c(\Theta) \sum_{e \in \Theta} (\Delta_{f_M}(\{e\}, X_i))$$
$$\implies \Delta_{f_M}(\Theta, X_i) > \sum_{e \in \Theta} (\Delta_{f_M}(\{e\}, X_i)).$$

Since X_i is fixed, $\Delta_{f_M}(E, X_i)$ is a submodular function in E. Thus, we have

$$\Delta_{f_M}(\Theta, X_i) \leq \sum_{e \in \Theta} (\Delta_{f_M}(\{e\}, X_i)).$$

This leads to a contradiction. Thus, there exists some element $e' \in \Theta$ for which the required inequality holds.

Now, observe that the RHS of the required inequality in this case is strictly positive and $\forall e \in X_i$, the LHS of the inequality is 0. Hence, $e' \notin X_i$ and we are done. □

Corollary 1. *When the conditions of Lemma 1 hold,*

$$\frac{\Delta_{f_M}(\{e\}, X_i) - c(\{e\})}{\Delta_{f_M}(\{e\}, X_i)} \geq \frac{\Delta_{f_M}(\Theta, X_i) - c(\Theta)}{\Delta_{f_M}(\Theta, X_i)}.$$

PROOF. From Lemma 1, we have

$$\frac{\Delta_{f_M}(\{e\},X_i)}{c(\{e\})} \geq \frac{\Delta_{f_M}(\Theta,X_i)}{c(\Theta)}.$$

Since f_M is monotonically increasing, it implies

$$\frac{\Delta_{f_M}(\{e\},X_i)-c(\{e\})}{\Delta_{f_M}(\{e\},X_i)} \geq \frac{\Delta_{f_M}(\Theta,X_i)-c(\Theta)}{\Delta_{f_M}(\Theta,X_i)},$$

and we are done. □

PROOF. (of Theorem 1) Say the MarginalGreedy algorithm runs for $l \leq n$ iterations. Define $\alpha(X_i)$ to be the rate of increase of f with respect to f_M just after the i^{th} iteration (and thus the current chosen set of elements is X_i). Further, let $e \in U \setminus X_i$ be the next element that will be chosen by the MarginalGreedy algorithm. Note that e is actually a function of X_i and, thus, once X_i is fixed, so is e. Mathematically,

$$\alpha(X_i) = \frac{f(X_i \cup \{e\}) - f(X_i)}{\delta(f_M(X_i))}$$

where $\delta(f_M(X_i)) = f_M(X_i \cup \{e\}) - f_M(X_i)$.

Let $j \leq l$ be the maximal index such that $f_M(X_j) < f(\Theta)$. The rate of increase at iteration i of the algorithm is at least as large as choosing the element from $\Theta \setminus X_i$ with the rate presented in LHS of Corollary 1.

The corollary also implies that while $f_M(X_i) < f(\Theta)$, the greedy algorithm has an element that it can pick. This implies that $j < l$. Thus, we have

$$f(X_l) = \sum_{i=0}^{l-1} \alpha(X_i)\delta(f_M(X_i)).$$

Using Corollary 1,

$$\begin{aligned} f(X_l) &\geq \sum_{i=0}^{l-1}\left(\frac{f_M(\Theta)-f_M(X_i)-c(\Theta)}{f_M(\Theta)-f_M(X_i)}\right)\delta(f_M(X_i)) \\ &\geq \sum_{i=0}^{l-1}\left(1-\frac{c(\Theta)}{f_M(\Theta)-f_M(X_i)}\right)\delta(f_M(X_i)). \end{aligned}$$

Since the term in the parenthesis in the last line is a decreasing function of $f_M(X_i)$, we get

$$\begin{aligned} f(X_l) \geq & \int_0^{f_M(X_l)} \left(1-\frac{c(\Theta)}{f_M(\Theta)-u}\right)du \\ \geq & \int_0^{f(\Theta)} \left(1-\frac{c(\Theta)}{f_M(\Theta)-u}\right)du \\ = & \Big[u + c(\Theta)\ln(f_M(\Theta)-u)\Big]_0^{f(\Theta)} \\ = & f(\Theta) + c(\Theta)\ln\left(\frac{f_M(\Theta)-f(\Theta)}{f_M(\Theta)}\right) \\ = & f(\Theta) + c(\Theta)\ln\left(\frac{c(\Theta)}{f(\Theta)+c(\Theta)}\right) \\ = & f(\Theta) - c(\Theta)\ln\left(\frac{c(\Theta)+f(\Theta)}{c(\Theta)})\right) \\ = & f(\Theta) - c(\Theta)\ln\left(1+\frac{f(\Theta)}{c(\Theta)})\right) \\ = & \left[1-\frac{c(\Theta)}{f(\Theta)}\ln\left(1+\frac{f(\Theta)}{c(\Theta)}\right)\right]f(\Theta). \end{aligned}$$

This concludes our proof and gives us our required approximation factor of $\left[1-\frac{c(\Theta)}{f(\Theta)}\ln\left(1+\frac{f(\Theta)}{c(\Theta)}\right)\right]$. □

Since the approximation ratio depends on the decomposition (specifically the function c), it is natural to ask whether different decompositions can lead to different solutions and approximation ratios. This is indeed the case; given a decomposition f_M and c, we can add a positive linear function $d(S) = \sum_{i \in S} d_i$ to both f_M and c, we still have a valid decomposition and the approximation factor has become smaller. This is because $f(\Theta)$ is fixed but $c(\Theta)$ becomes larger and clearly, the ratio is a decreasing function of c. Since this is the case, one may ask what is the "best" decomposition for this problem? We now show that the decomposition in Proposition 1, f_M^* and c^*, is indeed the best decomposition. This is done by first improving the ratio for an arbitrary decomposition and then showing that the improvement procedure for f_M^* and c^* does not lead to any improvement. In fact, in the next section, we will show a hardness of approximation which matches the ratio provided by this decomposition.

First we show how to obtain from an arbitrary decomposition f_M and c, another decomposition $\widetilde{f}_M$ and $\widetilde{c}$ such that the ratio improves. This happens if we can subtract a linear term from f_M and c while preserving monotonicity of f_M based on the above argument. And then we show that for f_M^* and c^*, this improvement procedure returns f_M^* and c^*

Proposition 2. *Given an arbitrary decomposition f_M and c of a normalized submodular function f, i.e., $f(S) = f_M(S) - c(S) \ \forall \ S \subseteq V$ with monotone f_M and consider another decomposition*

$$\widetilde{f}_M(S) = f_M(S) - \sum_{i \in S}\big(f_M(U) - f_M(U \setminus i)\big)$$

$$\widetilde{c}(S) = c(S) - \sum_{i \in S}\big(f_M(U) - f_M(U \setminus i)\big)$$

Then, $\widetilde{f}_M$ is monotone. Furthermore, for the decomposition in Proposition 1, f_M^* and c^*, $\widetilde{f}_M^* = f_M^*$ and $\widetilde{c}^* = c^*$.

PROOF. The proof is provided in Appendix A. □

We now remark on certain aspects of the algorithm and its analysis. Since the algorithm is inspired by [30], one may ask whether running that algorithm for multiple values of the budget in the knapsack constraint leads to the same answer. Indeed, this is the case with the budget being the value of $c(\Theta)$. However, since we do not apriori know $c(\Theta)$, we would have to potentially try out a large number of budget values which is not feasible. Furthermore, our analysis of the approximation ratio crucially uses the fact that we are actually running the algorithm on this decomposition of f in order to maximize f itself, and not maximizing a monotone submodular function subject to knapsack constraints.

4. INAPPROXIMABILITY OF UNSM

In this section, we prove a hardness of approximation result for the UNSM problem, when the size of the universe is part of the input, which matches the approximation factor given by the MarginalGreedy algorithm in Theorem 1 when the decomposition used is f_M^* and c^* as defined in Proposition 1.

Theorem 2. *For any $\varepsilon > 0$, it is NP-hard to approximate the unconstrained, normalized submodular maximization problem to a factor of at least*

$$\left(1 - \frac{\ln(1+\gamma)}{\gamma} + \varepsilon\right).$$

Here, $\gamma = \frac{f(\Theta)}{c^*(\Theta)}$ and Θ is an optimal solution to the UNSM problem.

This approximation factor depends on the value at optimal (which may go to 0), implying that a constant factor approximation to the UNSM problem is unlikely.

Before proving Theorem 2, we first present a separation result of the Max Coverage problem which is central to the proof of Theorem 2.

4.1 Inapproximability of Max Coverage

An instance $\mathcal{I} = (X, \mathcal{S})$ of the Set Cover problem is defined as follows: we are given the ground set $X = \{e_1, e_2, \ldots, e_n\}$ and $\mathcal{S} = \{S_1, S_2, \ldots, S_m\} \subseteq 2^X$. The goal is to choose the minimum number of sets $\mathcal{O} \subseteq \mathcal{S}$ such that $\bigcup_{S_i \in \mathcal{O}} S_i = X$. Feige [7] showed that for any $\varepsilon > 0$, there is no $(1-\varepsilon)\ln n$-approximation polynomial time algorithm for this problem unless $\mathsf{NP} \subseteq \mathsf{DTIME}(n^{O(\log\log n)})$. The hardness was later proved under the weaker assumption of $\mathsf{P} \neq \mathsf{NP}$ by [18, 5].

A problem closely related to the Set Cover problem is the Max Coverage problem. An instance of the Max Coverage problem consists of an instance $\mathcal{I} = (X, \mathcal{S}, l)$ where X is the ground set, $\mathcal{S}$ is a collection of subsets of X, and $l \leq m$ is an integer specifying the budget. The goal is to select l sets $S_{i_1}, S_{i_2} \ldots, S_{i_l}$ and cover as many elements of the ground set as possible. Feige [7] shows that it is NP-hard to approximate this problem to a factor better than $1 - 1/e$. Krishnaswamy and Sviridenko [14] prove the following separation result (which is an extension of the Max Coverage hardness stated above) which is of interest to us.

Theorem 3. *(Theorem 2.2 in [14]) Suppose there exists a polynomial algorithm, which for some constants $B \geq 1$ and $0 < \varepsilon < e^{-B}$ has the following property : Given any instance $(X, \mathcal{S}, l)$ of Max Coverage with optimal value equal to $|X|$ (i.e., there exist l sets that cover the ground set X completely), the algorithm picks a collection of βl sets for some $\beta \in [0, B]$ which can cover $(1 - e^{-\beta} + \varepsilon)n$ elements. Then $\mathsf{P} = \mathsf{NP}$. Note that we allow the algorithm to pick different values of β for different instances of the problem.*

Theorem 2.2 in [14] is actually stated under the stronger assumption of $\mathsf{NP} \not\subseteq \mathsf{DTIME}(n^{O(\log\log n)})$. Their reduction relies on the hardness of Set Cover which, at the time of that paper, was known only under this stronger assumption. Leveraging the set cover hardness result by [18, 5] under the weaker assumption of $\mathsf{P} \neq \mathsf{NP}$, we arrive at Theorem 3 without any changes to the proof provided in [14].

Note that the coverage function $f(\mathcal{A}) = \left|\bigcup_{S \in \mathcal{A}} S\right|$ is a monotone, submodular function. The proof of Theorem 2 proceeds by considering a special case of UNSM where for a Max Coverage instance, $f_M(\mathcal{A})$ is taken to be a scaling of the coverage function and the additive cost function $c(\mathcal{A})$ is a scaling of the cardinality of the chosen set of subsets $\mathcal{A}$. We call this the Profitted Max Coverage problem.

Problem 1. *(The Profitted Max Coverage problem) An instance of this problem consists of an instance $\mathcal{I} = (X, \mathcal{S}, l)$ like the Max Coverage problem. Consider γ to be a constant for this problem whose value will be revealed later.*

Let $f_M(\mathcal{A}) = \frac{(\gamma+1)}{\gamma} \frac{\left|\bigcup\limits_{S \in \mathcal{A}} S\right|}{n}$ and $c(\mathcal{A}) = \frac{1}{\gamma}\frac{|\mathcal{A}|}{l}$. The goal is to maximize

$$\begin{aligned} f(\mathcal{A}) &= f_M(\mathcal{A}) - c(\mathcal{A}) \\ &= \frac{(\gamma+1)}{\gamma} \frac{\left|\bigcup_{S \in \mathcal{A}} S\right|}{n} - \frac{1}{\gamma}\frac{|\mathcal{A}|}{l} \end{aligned}$$

Proof. (of Theorem 2) We want to show that if there exists a polynomial time algorithm which approximates the Profitted Max Coverage problem to a ratio better than

$$1 - \frac{\ln(\gamma+1)}{\gamma} + \varepsilon\frac{(\gamma+1)}{\gamma},$$

then $\mathsf{P} = \mathsf{NP}$.

We consider a hard instance $\mathcal{I} = (X, \mathcal{S}, l)$ of the Max Coverage problem such that the optimal value is n (i.e., there exist l sets to cover the entire ground set X). Now, let functions f, f_M and c be defined as in Problem 1.

[*Completeness*] Let us take a collection of l sets $\mathcal{G} = \{S_{i_1}, S_{i_2}, \ldots, S_{i_l}\}$ that cover the ground set X (such a collection exists because $\mathcal{I}$ is a Max Coverage instance with optimal value n). The optimal value of the corresponding Profitted Max Coverage instance occurs when exactly the sets in $\mathcal{G}$ are chosen.

$$\begin{aligned} f(\mathcal{G}) &= \frac{(\gamma+1)}{\gamma}\frac{n}{n} - \frac{1}{\gamma}\frac{l}{l} \\ &= \frac{(\gamma+1)}{\gamma} - \frac{1}{\gamma} \\ &= 1. \end{aligned}$$

Observe that $\frac{f(\mathcal{G})}{c(\mathcal{G})} = \gamma$.

[*Soundness*] It is easy to see that we will never choose more than $(\gamma+1)l$ sets as the function f will take negative values in those cases.

For any set, say $\mathcal{F}$, of βl (where $\beta \in [0, \gamma+1]$) subsets from $\mathcal{S}$ which cover at most $(1 - e^{-\beta} + \varepsilon)n$ elements, the value of the Profitted Max Coverage instance in this case is at most:

$$\begin{aligned} f(\mathcal{F}) &\leq \frac{(\gamma+1)}{\gamma}\frac{(1-e^{-\beta}+\varepsilon)n}{n} - \frac{1}{\gamma}\frac{\beta l}{l} \\ &= \frac{(\gamma+1)}{\gamma}(1-e^{-\beta}+\varepsilon) - \frac{1}{\gamma}\beta \\ &= \frac{(\gamma+1)(1-e^{-\beta}+\varepsilon) - \beta}{\gamma}. \end{aligned}$$

Differentiating the expression in the last line w.r.t β and setting the derivative to 0, we get

$$\begin{aligned} &\frac{\gamma+1}{\gamma}(e^{-\beta}) - \frac{1}{\gamma} = 0 \\ \implies &e^{\beta} = (\gamma+1) \\ \implies &\beta = \ln(\gamma+1) \leq (\gamma+1). \end{aligned}$$

Thus, the value $f(\mathcal{F})$ is always less than the value attained for that value of β and is

$$f(\mathcal{F}) \leq 1 - \frac{\ln(\gamma+1)}{\gamma} + \varepsilon\frac{(\gamma+1)}{\gamma}.$$

Now, if there exists a polynomial time algorithm (say Alg) which solves the Profitted Max Coverage problem to a factor better than $1 - \frac{\ln(\gamma+1)}{\gamma} + \varepsilon\frac{(\gamma+1)}{\gamma}$, then on any input instance of the Max Coverage problem such that the optimal value is n, Alg will output a set $\mathcal{F}$ such that $f(\mathcal{F}) > 1 - \frac{\ln(\gamma+1)}{\gamma} + \varepsilon\frac{(\gamma+1)}{\gamma}$ (since the optimal value is 1). Thus, $\mathcal{F}$ covers strictly more than $(1 - e^{-\beta} + \varepsilon)n$ elements with $\beta = \frac{|\mathcal{F}|}{l}$ (by contrapositivity). By Theorem 3, we have $\mathsf{P} = \mathsf{NP}$.

The above argument establishes the hardness for $\gamma = \frac{f(\Theta)}{c(\Theta)}$ for the function c defined in Problem 1. Since the factor depends only on $c(\Theta)$, if we can show that $c(\Theta) = c^*(\Theta)$ for these hard instances, we would be done. This can be shown by considering the expression for $c^*(\Theta)$ in this case :

$$\begin{aligned}
c^*(\Theta) &= \sum_{i \in \Theta} \left(f(U \setminus \{i\}) - f(U)\right) \\
&= \sum_{i \in \Theta} \left(f_M(U \setminus \{i\}) - f_M(U) - c(U \setminus \{i\}) + c(U)\right) \\
&= \sum_{i \in \Theta} \left(c(U) - c(U \setminus \{i\})\right) + \sum_{i \in \Theta} \left(f_M(U \setminus \{i\}) - f_M(U)\right) \\
&= c(\Theta) + \sum_{i \in \Theta} \left(f_M(U \setminus \{i\}) - f_M(U)\right) \\
&= c(\Theta) + \frac{(\gamma+1)}{\gamma \cdot n} \sum_{i \in \Theta} \left[\left|\bigcup_{S \in U \setminus \{i\}} S\right| - \left|\bigcup_{S \in U} S\right|\right]
\end{aligned}$$

Note that all the hard instances of SetCover and Max Coverage are derived from the construction of [15]. All such instances are such that each element has multiple subsets which may cover it (intuitively if there is only one subset which covers a particular element in any hard instance, then we will pick it and get a smaller, easier instance of the problem). Since the union of all subsets of the given instance is n and so is the union of all but one of the available subsets in the hard instance, each term in the above summation is 0. This implies that $c^*(\Theta) = c(\Theta)$ and we are done. □

5. SPEEDING UP THE MARGINAL GREEDY

In the worst case, the MarginalGreedy algorithm runs in $\mathcal{O}(n^2 \cdot \mathrm{EO})$ time, where n is the number of shareable nodes and EO is the time to evaluate $bc(S)$, i.e., the time to optimize the batch of queries given the set of nodes S, to be materialized. This makes the algorithm expensive since n itself may be exponential in the worst case. Thus, we would like to reduce the time taken by the algorithm without sacrificing on the theoretical guarantees on the quality of the solution proved in Section 3. In this section, we present some optimizations to our algorithm to improve its running time.

5.1 Basic Optimizations

We first note that two optimizations presented in [25] can be used for our algorithm as well. Their first observation is about searching only over all the shareable nodes. As noted above, this can be directly used by us since our algorithm just presents a different heuristic for choosing which nodes to materialize. Their second optimization presents a way to incrementally update the *bestCost* function for various sets that exploits the result of earlier cost computations to incrementally compute the new plan. Since the *mb* function is just a linear transformation of the *bestCost* function and our greedy algorithm (at least when the decomposition presented in the proof of Proposition 1 is used) is also concerned with just successive differences in the values of the *bestCost* function, their optimization can also be used to speed up our algorithm; for details see [25].

Another optimization (not in [25]) that can be made is based on a simple observation of the greedy algorithm and by exploiting submodularity. In the i^{th} iteration, the MarginalGreedy algorithm needs to compute the maximum benefit to cost ratio $\frac{f'_M(e, X_{i-1})}{c(\{e\})}$. Thus, if while scanning elements to compute the maximum, we encounter an element that has the marginal-benefit to cost ratio less than 1, we can remove it from the set Y of elements to be searched over as it will never be picked by the MarginalGreedy algorithm in the future iterations either. This is because f_M is also submodular and the size of X_i always increases as i increases so the value of the marginal-benefit to cost ratio only decreases as the algorithm proceeds and will never become greater than 1. A similar optimization for the simple greedy algorithm used for monotone, submodular maximization under cardinality constraints is also possible.

5.2 The Lazy Marginal Greedy algorithm

The third optimization in [25] essentially leverages supermodularity to improve the running time of the greedy algorithm. The argument is similar to that used by [16] for the LazyGreedy algorithm. We observe that a similar argument as the ones presented in these two papers may be used for the MarginalGreedy algorithm and is presented next.

As noted previously, in each iteration i, the MarginalGreedy algorithm must identify the element e with the maximum marginal-benefit to cost ratio $\frac{f'_M(e, X_{i-1})}{c(\{e\})}$. For each element e, the denominator is fixed and the marginal benefits are monotonically nonincreasing during the iterations of the algorithm, i.e., $f'_M(e, X_i) \geq f'_M(e, X_j)$ whenever $i \leq j$. Thus, instead of recomputing $\frac{f'_M(e, X_{i-1})}{c(\{e\})}$ for each element $e \in V$, which requires $\mathcal{O}(n)$ computations of f, the LazyMarginalGreedy algorithm maintains a list of upper bounds $u(e)$ (initialized to a large value) on the marginal-benefit to cost ratio sorted in decreasing order (using a heap).

In each iteration, the algorithm extracts the element with largest $u(e)$ from the ordered list of remaining elements and add its to the current solution. If, after this update, $u(e) \geq u(e')\ \forall e' \neq e$, then submodularity guarantees that $\frac{f'_M(e, X_{i-1})}{c(\{e\})} \geq \frac{f'_M(e', X_{i-1})}{c(\{e\})}\ \forall e' \neq e$, and therefore the algorithm has identified the element with the largest marginal benefit to cost ratio without computing the ratio for a potentially large number of elements e'.

5.3 Universe Reduction under size constraints

We may sometimes want to consider a cardinality constraint (say k) on the number of nodes to be materialized. This may arise due to storage constraints which only allow materialization of a few subexpressions. We adapt our greedy algorithm for this constraint by simply stopping after k elements are picked.

While, at this point, we do not show any theoretical approximation guarantees for this problem, there is a way to leverage this cardinality constraint to prune out certain elements from the ground set U. This preprocessing step may be used to reduce the size of the set of PQDAG nodes U on which the algorithm will be run.

We show that the algorithm run on this reduced set is the same as that obtained when the algorithm runs on the full set. This check is useful only when there is a cardinality constraint of $k < n$, as we will show.

Theorem 4. *Let $U = \{e_1, \ldots, e_n\}$ be the set of all shareable PQDAG nodes ordered as*

$$\frac{f'_M(e_1, U \setminus \{e_1\})}{c(\{e_1\})} \geq \cdots \geq \frac{f'_M(e_n, U \setminus \{e_n\})}{c(\{e_n\})}.$$

Furthermore, let

$$U' = \{e \in U \big| \frac{f_M(e)}{c(\{e\})} \geq \frac{f'_M(e_k, U \setminus \{e_k\})}{c(\{e_k\})}\} \text{ for } k < n.$$

The output of the MarginalGreedy algorithm (with cardinality constraint of k) when it runs on U is the same as the output when it runs on U'.

Proof. The proof is provided in Appendix A. □

It is important to note that this strategy may not always lead to a reduction in the ground set but it may lead to pruning in certain cases.

Note that this pruning procedure can be modified to work for the simple greedy algorithm for monotone, submodular maximization under cardinality constraints. The proof is also along similar lines as those stated above.

6. EXPERIMENTAL SECTION

We now describe our experimental setup and findings. We worked with the original C++ code of Pyro which implemented the Greedy algorithm [25]. We extended it by implementing the Marginal Greedy algorithm. All the optimizations discussed in Section 5 are implemented with the exception of the one discussed in subsection 5.3 as we are mainly interested in the best plan without imposing any cardinality constraints.

The optimizer rule set consists of select push down, join commutativity and associativity (to generate bushy join trees), and select and aggregate subsumption. The physical operators included sort-based aggregation, merge join, nested loop join, indexed selection and relation scan. The implementation includes handling physical properties (sort order and presence of indices) on base and intermediate relations, unification and subsumption during DAG generation (see [25] for details).

The block size was taken as 4KB and our cost functions assume 6MB is available to each operator during execution (we also conducted experiments with memory sizes of 128MB). Standard techniques were used for estimating costs, using statistics about relations. The cost estimates are of the standard resource consumption estimates (see Appendix C of Roy's thesis [24] for details) which contain an I/O component and a CPU component, with seek time as 10 msec, transfer time of 2 msec/block for read and 4 msec/block for write, and CPU cost of 0.2 msec/block of data processed.

We assume that intermediate results are pipelined to the next input, using an iterator model as in Volcano; they are saved to disk only if the result is to be materialized for sharing. The materialization cost is the cost of writing out the results sequentially. The tests were performed on a 2.4 GHz Intel i7 processor laptop with 8GB memory running Linux. We compare Marginal Greedy with Greedy and stand-alone Volcano (no MQO). The optimization time of our Marginal Greedy algorithm was very close to that of the Greedy algorithm in [25]. The optimization times are measured as CPU time.

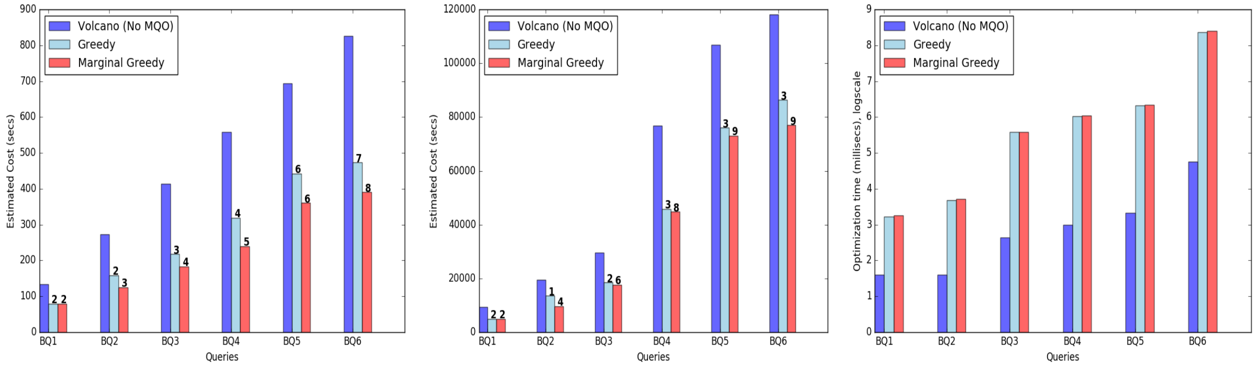
6.1 Experiment 1 (Batched TPCD Queries)

The workload for the first experiment models a system where several TPCD queries are executed as a batch. The workload consists of subsequences of the queries Q3, Q5, Q7, Q8, Q9 and Q10. Each query was repeated twice with different selection constants. Composite query BQi consists of the first i of the above queries, and we used composite queries BQ1 to BQ6 in our experiments. The TPCD database is used at a scale of 1 (1 GB total size), with a clustered index on the primary keys for all the base relations. We also ran the queries in this experiment and the next at a scale of 100 (total size 100GB).

Note that although a query is repeated with two different values for a selection constant, we found that the selection operator generally lands up at the bottom of the best Volcano plan tree, and the two best plan trees may not have common subexpressions.

The results on the two workloads (1GB and 100 GB total sizes) are shown in Figure 4. The number on top of the bars for Greedy and Marginal Greedy denotes the number of materialized nodes. Greedy does substantially better than Volcano (without MQO) by upto 57%. Marginal Greedy always does as well as or better than Greedy. In fact, the results are the same only for BQ1 where both chose to materialize the two nodes which lead to benefit. For all other queries in the experiment with 1GB Total Size, the improvement of Marginal Greedy is always between 12% and 25%. This is primarily due to the number of materialized nodes by Marginal Greedy being more than that by Greedy. BQ5 is especially interesting in Figure 4a as the number of materialized nodes is the same yet there is almost a 20% improvement over Greedy. In fact, for queries from BQ4 to BQ6, the intersection in the materialized nodes by the two algorithms had an overlap of 1 or 2 only.

In the experiment with 100GB Total Size (Figure 4b), as mentioned, the nodes chosen to be materialized for BQ1 are the same for both algorithms. For the rest of the queries, the number of materialized nodes is much larger than in the 1GB size dataset. While the relative gains in this dataset might seem comparable or slightly lesser than those observed in the smaller dataset, the actual gains in these cases are substantial due to large costs coming from these large data sizes. In these queries, there were 1 or 2 nodes which had substantially more benefit and got picked by both Greedy and Marginal Greedy. While Greedy picked a few more nodes which seemed benefical initially, Marginal Greedy picked many more nodes, each of which had moderate benefit but lead to an overall decrease in the cost. This behaviour was particularly observed in BQ5 and BQ6 and we conjecture for larger sets of queries on larger data sets, this behavior may be more pronounced.



(a) 1GB Total Size (b) 100GB Total Size (c) Optimization Time (logscale)

Figure 4: Results for batched TPCD queries (Experiment 1)

The optimization times for the queries are shown in Figure 4c. Since the values for Greedy and Marginal Greedy were very close to each other, we present the results in logscale. As can be seen, the optimization times are very close to each other. We stress that while the execution cost of a query depends on the size of the underlying data, the cost of optimization does not.

6.2 Experiment 2 (Stand-Alone TPCD Queries)

Roy et al. [25] also had an experiment consisting of four individual queries based on TPCD using the same data sizes (1GB and 100GB) and the same indices. These queries had common subexpressions within themselves and benefitted from MQO to optimize just those queries individually. However, in each of these queries only one node was beneficial and hence, both algorithms found that node and resulted in the same answer. We present the results here for completeness. We explain these queries themselves and the actual results are presented in Figure 5 in Appendix B.

TPCD query Q2 has a large nested query, and repeated invocations of the nested query in a correlated evaluation could benefit from reusing some of the intermediate results. Greedy and Marginal Greedy gave a plan with an estimated cost of 79 seconds for the smaller data set and 1929 seconds for the larger one. Decorrelation is an alternative to correlated evaluation and Q2-D is a (manually) decorrelated version of Q2 (due to decorrelation, Q2-D is actually a batch of queries). Multi-query optimization also gives substantial gains on the decorrelated query Q2-D, results in a plan of estimated cost 46 and 2059 for the two data sizes respectively, by both algorithms. We next considered the TPCD queries Q11 and Q15, both of which have common subexpressions, and hence make a case for multi-query optimization. For Q11, both the greedy algorithms lead to a plan of approximately half the cost as that returned by Volcano. The improvements for Q15 are similar but more pronounced for the smaller data set.

The conclusion based on the experiments seems to be that when there are multiple possible nodes that can be materialized, Greedy chooses the nodes which result in considerable improvements early on but Marginal Greedy is more global and chooses to materialize more nodes which might have moderate benefit individually but can result in overall benefits.

7. RELATED WORK

We now present the related work in the areas of multi-query optimization and submodular maximization.

7.1 Multi-Query Optimization

The MQO problem has received significant attention in the past [26, 20, 27, 22, 29]. Initial work [26, 20, 22, 27] proposed solutions that were not fully integrated with the query optimizer and were primarily exhaustive.

Subramanian and Venkataraman [29] consider sharing only among the best plans of the query; this approach can be implemented as an efficient, post-optimization phase in existing systems, but can be highly suboptimal.

To choose the set of nodes to be materialized, Roy et al. [25] use a greedy algorithm discussed in detail in Section 2. Dalvi et al. [4] explores the possibility of sharing intermediate results by pipelining, avoiding unnecessary materializiations. Thomas et al. [31] consider the MQO problem in Volcano taking scheduling and caching into account. They present an exhaustive algorithm which takes $\mathcal{O}(n^n)$ time, which is clearly infeasible.

Zhou et al. [32] propose a framework to use common subexpressions for MQO and materialized view selection in a query optimizer based on the Cascades framework [9]. The focus however is on "covering" subexpressions at the LQDAG level and they do not take into account competing physical properties like sort orders and partitioning properties from different consumers.

Silva et al. [28] consider physical properties in a cost-based fashion. However, their solution is also based on heuristics which materializes *every* common subexpression at the LQDAG level. The best physical property for each subexpression is chosen and all consumers are forced to use the same physical property, which can be sub-optimal. Even with this heuristic, their approach can be very expensive when there are many potential physical properties for each subexpression.

7.2 Submodular Maximization

Submodular maximization has received a significant amount of attention in optimization [3, 19, 2] with wide applicability in machine learning, computer vision and information retrieval [11, 12, 1, 13]. In this problem, we are given a submodular function f and a universe U, with the goal of selecting a subset $S \subseteq U$ which maximizes $f(S)$.

Typically, S must satisfy additional feasibility constraints such as cardinality, knapsack or matroid constraints.

This problem is NP-hard even for the simplest problems which involve only *cardinality constraints* and monotone functions. Nemhauser et al. [19] show that a simple greedy algorithm gives a $(1 - 1/e)$ approximation for monotone submodular maximization under cardinality constraints. They further show that it is NP-hard to obtain a better approximation guarantee. Sviridenko [30] presents a modified greedy algorithm for monotone submodular function maximization under knapsack constraints and is the main motivation for our algorithm.

Buchbinder et al. [2] gave a $1/2$-approximation algorithm for unconstrained non-monotone submodular maximization, for which there is a matching hardness result. However, all these results assume non-negativity of the function f. Mittal and Shulz [17] show that a constant factor approximation for non-negative supermodular minimization is NP-hard. Inapproximability of non-monotone submodular maximization (with possibly negative values) is also well known. More specfically, it is NP-hard to even decide whether the optimum is non-negative or not for a general non-monotone submodular function (which may take negative values). We were able to sidestep this hardness as we already knew that the optimum is greater than or equal to zero due to the normalized assumption. To the best of our knowledge, ours is the first work which, under the assumption of $f(\emptyset) = 0$, provides an approximation algorithm with a matching hardness of approximation result for unconstrained non-monotone submodular maximization when the function may take negative values. Since the hardness of approximation factor depends on the optimal (and may go to 0), this rules out constant factor approximations for the problem even in the restricted setting of $f(\emptyset) = 0$.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have presented a reformulation of the well-studied MQO problem. Under the assumption of supermodularity of the *bestCost* function, we propose a greedy algorithm for the maximization problem and provide an approximation factor guarantee for our algorithm. We then showed that obtaining a better approximation factor than the one attained by our greedy algorithm is NP-hard. Such a theoretical guarantee on the quality of any heuristic has not been presented before. Since the underlying problem solved in this paper is the unconstrained, normalized submodular maximization problem, with possibly negative values, we believe our results can be useful beyond just MQO.

One area of future work is the problem of *non-negative*, non-monotone submodular maximization problem under cardinality constraints and more generally, matroid constraints. This is an open problem and even the most recent work [6] has a considerable gap in the approximation ratio and the hardness of approximation known. We would like to see if ideas in this paper like the "best decomposition" can be used to devise algorithms with better guarantees for that problem.

9. ACKNOWLEDGMENTS

We would like to thank Sebastien Tavenas, Amit Deshpande, Deeparnab Chakrabarty and Ravishankar Krishnaswamy for comments on the paper.

10. REFERENCES

[1] Y. Boykov and M. Jolly. Interactive graph cuts for optimal boundary and region segmentation of objects in N-D images. In *ICCV*, pages 105–112, 2001.

[2] N. Buchbinder, M. Feldman, J. Naor, and R. Schwartz. A tight linear time (1/2)-approximation for unconstrained submodular maximization. In *53rd Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 649–658, 2012.

[3] G. Călinescu, C. Chekuri, M. Pál, and J. Vondrák. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM J. Comput.*, 40(6):1740–1766, 2011.

[4] N. N. Dalvi, S. K. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. *J. Comput. Syst. Sci.*, 66(4):728–762, 2003.

[5] I. Dinur and D. Steurer. Analytical approach to parallel repetition. In *Symposium on Theory of Computing, STOC*, pages 624–633, 2014.

[6] A. Ene and H. L. Nguyen. Constrained submodular maximization: Beyond 1/e. In *IEEE 57th Annual Symposium on Foundations of Computer Science, FOCS*, pages 248–257, 2016.

[7] U. Feige. A threshold of ln n for approximating set cover. *J. ACM*, 45(4):634–652, 1998.

[8] U. Feige, V. S. Mirrokni, and J. Vondrák. Maximizing non-monotone submodular functions. *SIAM J. Comput.*, 40(4):1133–1153, 2011.

[9] G. Graefe. The Cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[10] G. Graefe and W. J. McKenna. The Volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 209–218, 1993.

[11] S. Jegelka and J. A. Bilmes. Submodularity beyond submodular energies: Coupling edges in graph cuts. In *The 24th IEEE Conference on Computer Vision and Pattern Recognition, CVPR*, pages 1897–1904, 2011.

[12] D. Kempe, J. M. Kleinberg, and É. Tardos. Maximizing the spread of influence through a social network. In *Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 137–146, 2003.

[13] P. Kohli, M. P. Kumar, and P. H. S. Torr. P3 & beyond: Solving energies with higher order cliques. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007.

[14] R. Krishnaswamy and M. Sviridenko. Inapproximability of the multi-level uncapacitated facility location problem. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*, pages 718–734, 2012.

[15] C. Lund and M. Yannakakis. On the hardness of approximating minimization problems. *J. ACM*, 41(5):960–981, 1994.

[16] M. Minoux. Accelerated greedy algorithms for maximizing submodular set functions. *Optimization Techniques*, pages 234–243, 1977.

[17] S. Mittal and A. S. Schulz. An FPTAS for optimizing a class of low-rank functions over a polytope. *Math. Program.*, 141(1-2):103–120, 2013.

[18] D. Moshkovitz. The projection games conjecture and the NP-hardness of ln n-approximating set-cover. *Theory of Computing*, 11:221–235, 2015.

[19] G. L. Nemhauser, L. A. Wolsey, and M. L. Fisher. An analysis of approximations for maximizing submodular set functions - I. *Math. Program.*, 14(1):265–294, 1978.

[20] J. Park and A. Segev. Using common subexpressions to optimize multiple queries. In *Proceedings of the Fourth International Conference on Data Engineering (ICDE)*, pages 311–319, 1988.

[21] A. Pellenkoft, C. A. Galindo-Legaria, and M. L. Kersten. The complexity of transformation-based join enumeration. In *VLDB*, pages 306–315, 1997.

[22] A. Rosenthal and U. S. Chakravarthy. Anatomy of a mudular multiple query optimizer. In *Fourteenth International Conference on Very Large Data Bases (VLDB)*, pages 230–239, 1988.

[23] N. Roussopoulos. View indexing in relational databases. *ACM Trans. Database Syst.*, 7(2):258–290, 1982.

[24] P. Roy. *Multi Query Optimization and Applications.* Ph.d. thesis, Indian Institute of Technology, Bombay, 2001.

[25] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 249–260, 2000.

[26] T. K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, 1988.

[27] K. Shim, T. K. Sellis, and D. S. Nau. Improvements on a heuristic algorithm for multiple-query optimization. *Data Knowl. Eng.*, 12(2):197–222, 1994.

[28] Y. N. Silva, P. Larson, and J. Zhou. Exploiting common subexpressions for cloud query processing. In *IEEE 28th International Conference on Data Engineering (ICDE)*, pages 1337–1348, 2012.

[29] S. N. Subramanian and S. Venkataraman. Cost-based optimization of decision support queries using transient views. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data*, pages 319–330, 1998.

[30] M. Sviridenko. A note on maximizing a submodular set function subject to a knapsack constraint. *Oper. Res. Lett.*, 32(1):41–43, 2004.

[31] D. Thomas, A. A. Diwan, and S. Sudarshan. Scheduling and caching in multiquery optimization. In *Proceedings of the 13th International Conference on Management of Data (COMAD)*, pages 150–153, 2006.

[32] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 533–544, 2007.

APPENDIX

A. ADDITIONAL PROOFS

We now present the missing proofs. We reproduce the theorem statements for convenience.

Proposition 1. (in the main paper) Any normalized, non-monotone (which may take negative values) submodular function f can be decomposed as

$$f(S) = f_M(S) - c(S) \quad ,\forall\ S \subseteq U$$

where f_M is a monotone submodular function and c is an additive cost function. In particular, one possible decomposition is

$$f^*_M(S) = f(S) + \sum_{e \in S} (f(U \setminus \{e\}) - f(U))$$

$$c^*(S) = \sum_{e \in S} (f(U \setminus \{e\}) - f(U))$$

Proof. It is easy to see that c is additive and that

$$\forall\ S \subseteq X,\ \text{we have } f(S) = f_M(S) - c(S)$$

Since c is additive and f is submodular, f_M is also submodular since for arbitrary $S_1 \subset S_2 \subset U$ and an arbitrary $e \in U \setminus S_2$,

$$\begin{aligned}
& f_M(S_1 \cup \{e\}) - f_M(S_1) \\
&= f(S_1 \cup \{e\}) - c(S_1 \cup \{e\}) - f(S_1) + c(S_1) \\
&= f(S_1 \cup \{e\}) - f(S_1) - c(\{e\}) \quad \text{(by linearity of } c) \\
&\geq f(S_2 \cup \{e\}) - f(S_2) - c(\{e\}) \quad \text{(by submodularity of } f) \\
&= f(S_2 \cup \{e\}) - f(S_2) - c(S_2 \cup \{e\}) + c(S_2) \quad \text{(by linearity)} \\
&= f_M(S_2 \cup \{e\}) - f_M(S_2).
\end{aligned}$$

Now we just have to show that f_M is monotone. Consider an arbitrary $S \subset U$ and an arbitrary $e \in U \setminus S$. Let us consider the expression

$$\begin{aligned}
& f_M(S \cup \{e\}) - f_M(S) \\
= \ & f(S \cup \{e\}) - f(S) + (f(U \setminus \{e\}) - f(U)) \\
= \ & (f(S \cup \{e\}) - f(S)) - (f(U) - f(U \setminus \{e\})) \\
\geq \ & 0
\end{aligned}$$

The inequality in the last line follows from the fact that $S \subseteq U \setminus \{e\}$ and the submodularity of f. The terms in the summation can be suitably scaled to ensure that c is zero only at $\emptyset$ and positive everywhere else. □

Proposition 2. (in the main paper) Given an arbitrary decomposition f_M and c of a normalized submodular function f, i.e., $f(S) = f_M(S) - c(S)\ \forall\ S \subseteq V$ with monotone f_M and consider another decomposition

$$\widetilde{f}_M(S) = f_M(S) - \sum_{i \in S} \big(f_M(U) - f_M(U \setminus i)\big)$$

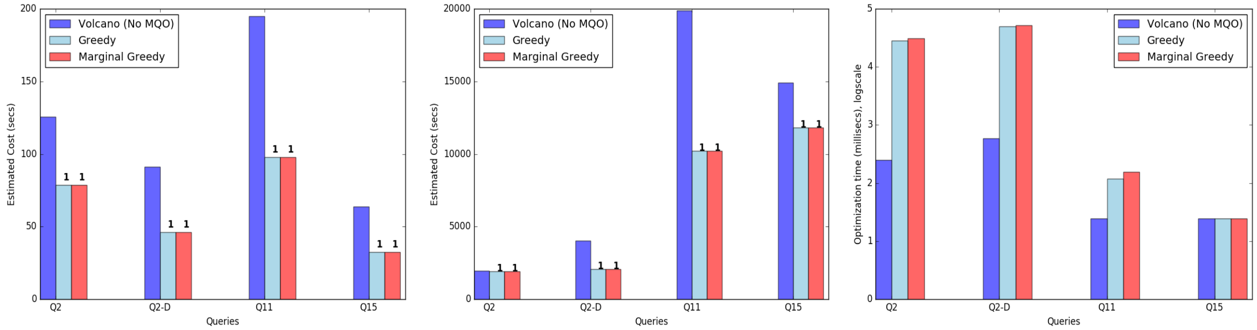
$$\widetilde{c}(S) = c(S) - \sum_{i \in S} \big(f_M(U) - f_M(U \setminus i)\big)$$

Then, $\widetilde{f}_M$ is monotone. Furthermore, for the decomposition in Proposition 1, f^*_M and c^*, $\widetilde{f}^*_M = f^*_M$ and $\widetilde{c}^* = c^*$.

Proof. To show monotonicity of $\widetilde{f}_M$, it is enough to show $\forall j \in U,\ \forall S \subseteq U \setminus \{j\}, \widetilde{f}_M(S \cup \{j\}) - \widetilde{f}_M(S) \geq 0$.

$$\begin{aligned}
& \widetilde{f}_M(S \cup \{j\}) - \widetilde{f}_M(S) \\
&= f_M(S \cup \{j\}) - f_M(S) - \big(f_M(U) - f_M(U \setminus \{j\})\big) \\
&\geq 0 \quad \text{(by submodularity of } f_M)
\end{aligned}$$

For the second part, we just expand the expressions to get the desired result.



(a) 1GB Total Size (b) 100GB Total Size (c) Optimization Times (logscale)

Figure 5: Results for stand-alone TPCD queries (Experiment 2)

$$\begin{aligned}\widetilde{c}^*(S) &= c^*(S) - \sum_{i\in S} \left(f^*_M(U) - f^*_M(U\setminus\{i\})\right)\\ &= c^*(S) - \sum_{i\in S} f(U) - f(U\setminus\{i\}) + \big(f(U\setminus\{i\}) - f(U)\big)\\ &= c^*(S)\end{aligned}$$

The computation for $\widetilde{f}^*_M(S)$ proceeds similarly. □

Theorem 4. (in the main paper) Let the set of all shareable PQDAG nodes $U = \{e_1, \ldots, e_n\}$ be ordered as

$$\frac{f'_M(e_1, U\setminus\{e_1\})}{c(\{e_1\})} \geq \ldots \geq \frac{f'_M(e_n, U\setminus\{e_n\})}{c(\{e_n\})}.$$

Furthermore, let

$$U' = \{e \in U \mid \frac{f_M(e)}{c(\{e\})} \geq \frac{f'_M(e_k, U\setminus\{e_k\})}{c(\{e_k\})}\} \text{ for } k < n.$$

The output of the MarginalGreedy algorithm (with cardinality constraint of k) when it runs on U is the same as the output when it runs on U'.

Proof. Without loss of generality, assume that the algorithm, when run on U', terminates after the full k steps. Let the sequence of chosen elements, in order of inclusion, be $\{s_1, s_2, \ldots, s_k\}$ and for all $i \in [k]$, let $X_i = \{s_1, s_2, \ldots, s_i\}$, as before. Clearly, $\emptyset = X_0 \subset X_1 \subset X_2 \subset \ldots \subset X_k$.

Case 1. $k = n$
This is a simple case in which all elements are chosen and, thus, U' should be equal to U which is shown as follows $\forall e \in U$, we have

$$\begin{aligned}\frac{f_M(\{e\})}{c(\{e\})} &= \frac{f'_M(e, \emptyset)}{c(\{e\})}\\ &\geq \frac{f'_M(e, U\setminus\{e\})}{c(\{e\})} \qquad \text{(by submodularity)}\\ &\geq \frac{f'_M(e_k, U\setminus\{e_k\})}{c(\{e_k\})}.\end{aligned}$$

Hence, all elements of U are going to be in U' since they all satisfy the condition to be in U'. In this case, the check is clearly wasteful since the ground set has no reduction and a lot of functional calls are made. In the MQO context, this corresponds to invoking a lot of $bestCost(Q, S)$ calls, each of which are moderately expensive. Thus, in this case, the preprocessing step should just check if $k = n$ and if so, directly pass the full ground set to the greedy algorithm.

Case 2. $k < n$ & $X_k = \{e_1, e_2, \ldots, e_k\}$.
In this case, the theorem follows trivially since U' will contain all elements in X_k, along with some other elements.

Case 3. $k < n$ & $X_k \neq \{e_1, e_2, \ldots, e_k\}$.
We first make a claim which we will prove later.

Claim 1. *For all $i \in \{1, 2, \ldots, k\}$, we have*

$$\frac{f'_M(s_i, X_{i-1})}{c(\{s_i\})} \geq \frac{f'_M(e_i, U\setminus\{e_i\})}{c(\{e_i\})}.$$

The claim is used to show that elements in $U \setminus U'$ will never be picked by the MarginalGreedy algorithm. Intuitively, for any element $e \notin U'$, the $\frac{f'_M(e, X_i)}{c(\{e\})}$ ratio of picking it is largest in the first iteration (by submodularity) and that itself is less than the element with the smallest ratio of the elements selected by the greedy algorithm. So, it is guaranteed that the greedy algorithm does not pick any element which is not in U'. This is easy to see and is as follows
For all $e \in U \setminus U'$, we have

$$\frac{f'_M(e, \emptyset)}{c(\{e\})} = \frac{f_M(e)}{c(\{e\})} < \frac{f'_M(e_k, U\setminus\{e_k\})}{c(\{e_k\})}.$$

By Claim 1,

$$\begin{aligned}&\frac{f'_M(s_k, X_{k-1})}{c(\{s_k\})} \geq \frac{f'_M(e_k, U\setminus\{e_k\})}{c(\{e_k\})}\\ \implies &\frac{f'_M(s_k, X_{k-1})}{c(\{s_k\})} > \frac{f'_M(e, \emptyset)}{c(\{e\})}.\end{aligned}$$

We now present the proof of Claim 1.

Proof. (of Claim 1) The case of $e_i \notin X_{i-1}$ is trivial due to submodularity and the greedy algorithm.

Thus, we just have to prove for the case when $e_i \in X_{i-1}$ Since $|X_{i-1}| = i - 1$, X_{i-1} cannot include all elements from the set $\{e_1, e_2, \ldots, e_i\}$. Thus, there exists some element, say, $e_z \in e_1, e_2, \ldots, e_i$ such that $e_z \notin X_{i-1}$.

$$\begin{aligned}\text{Thus, we have } &\frac{f'_M(s_i, X_{i-1})}{c(\{s_i\})}\\ &= \max_{e\in U\setminus X_{i-1}} \frac{f'_M(e, X_{i-1})}{c(\{e\})}\\ &\geq \frac{f'_M(e_z, X_{i-1})}{c(\{e_z\})}\\ &\geq \frac{f'_M(e_z, U\setminus\{e_z\})}{c(\{e_z\})} \text{ (by submodularity)}\end{aligned}$$

$$\geq \frac{f'_M(e_i, U \setminus \{e_i\})}{c(\{e_i\})}$$

□

This concludes our proof.

B. RESULTS OF EXPERIMENT 2

In this section, we present the results of Experiment 2 (Stand-alone TPCD). The results are shown in Figure 5.