# Selecting Subexpressions to Materialize at Datacenter Scale

Alekh Jindal  Konstantinos Karanasos  Sriram Rao  Hiren Patel
Microsoft
{aljindal,kokarana,sriramra,hirenp}@microsoft.com

## ABSTRACT

We observe significant overlaps in the computations performed by user jobs in modern shared analytics clusters. Naïvely computing the same subexpressions multiple times results in wasting cluster resources and longer execution times. Given that these shared cluster workloads consist of tens of thousands of jobs, identifying overlapping computations across jobs is of great interest to both cluster operators and users. Nevertheless, existing approaches support orders of magnitude smaller workloads or employ heuristics with limited effectiveness.

In this paper, we focus on the problem of subexpression selection for large workloads, i.e., selecting common parts of job plans and materializing them to speed-up the evaluation of subsequent jobs. We provide an ILP-based formulation of our problem and map it to a bipartite graph labeling problem. Then, we introduce BIGSUBS, a vertex-centric graph algorithm to iteratively choose in parallel which subexpressions to materialize and which subexpressions to use for evaluating each job. We provide a distributed implementation of our approach using our internal SQL-like execution framework, SCOPE, and assess its effectiveness over production workloads. BIGSUBS supports workloads with tens of thousands of jobs, yielding savings of up to 40% in machine-hours. We are currently integrating our techniques with the SCOPE runtime in our production clusters.

**PVLDB Reference Format:**
Alekh Jindal, Konstantinos Karanasos, Sriram Rao, Hiren Patel. Selecting Subexpressions to Materialize at Datacenter Scale. *PVLDB*, 11(7): 800-812, 2018.
DOI: https://doi.org/10.14778/3192965.3192971

## 1. INTRODUCTION

Shared analytics clusters have become the de facto way for large organizations to analyze and gain insights over their data. At Microsoft, we operate multiple such shared datacenter-scale clusters, in a setting that is similar to other Fortune-500 companies [48, 5, 45, 3]. Each cluster is comprised of tens of thousands of machines, storing exabytes of data, and supports thousands of users, collectively running hundreds of thousands of daily batch jobs [41]. Figures 2(a) and 2(b) summarize the number of jobs and users, respectively, on a typical day at a few analytics clusters at Microsoft. Such clusters run business critical workloads and are expensive to run and operate. Therefore, operators and users alike are consistently exploring new opportunities to speed-up job performance.

*Proceedings of the VLDB Endowment,* Vol. 11, No. 7
Copyright 2018 VLDB Endowment 2150-8097/18/03.
DOI: https://doi.org/10.14778/3192965.3192971

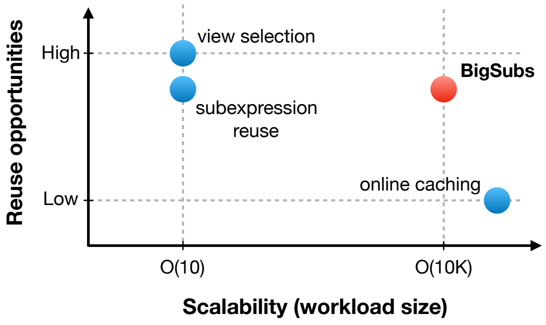
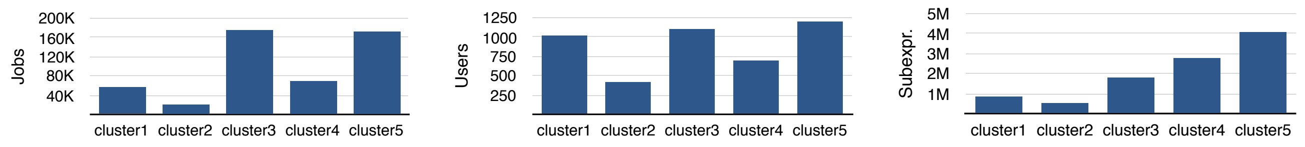


**Figure 1: Approaches for computation reuse.**

The shared nature of these clusters and the lack of coordination between users that often belong to different parts of the organization, lead to unnecessary overlaps across user computations, which in turn cost companies millions of dollars. An analysis of our cluster workloads reveals that over 45% of the daily jobs submitted by approximately 65% of the users have commonalities, resulting in millions of subexpression overlaps, as reported in Fig. 2(c). Therefore, our target is to *identify and reuse such subexpressions*. As we show, this can lead to saving up to 40% machine-hours in our clusters (see §5, Table 1).

The computation reuse problem has been extensively studied by the database community under different formulations, such as view selection [35, 46, 20, 19], multi-query optimization [43, 42, 38, 49], and subexpression reuse [53, 12, 8, 32, 39]. These approaches have a similar high-level goal: given a set of queries, find commonalities across them in order to minimize a cost function (e.g., query evaluation, storage, and subexpression maintenance costs), under a set of constraints (e.g., space budget). Unfortunately, they have all targeted workloads with tens [46, 53, 12, 8, 42] or few hundreds of queries [4, 16]. As explained above, this is *three orders of magnitude smaller* than the size of the daily workloads in our clusters, making such techniques non-applicable in our setting.

To overcome the scalability issue, more recent approaches tailored towards large-scale clusters (e.g., Nectar [18]), have considered computation reuse from an online caching perspective, rather than considering the whole workload upfront. Such systems observe the queries as they arrive in the system and choose which subexpressions to materialize for future reuse, based on some heuristics. While these approaches are not constrained by the workload size, they rely on local optimization decisions. Consequently,



(a) Total Jobs. (b) Total Users. (c) Overlapping Subexpressions.

**Figure 2: Workload summary from a typical day in some of Microsoft's analytics clusters.**

the resulting gains can be far from an optimal configuration. In fact, we show that such approaches can achieve three orders of magnitude less savings than global optimizations (see §5, Fig. 10).

The need for global optimization decisions is further motivated by the observation that business-critical jobs in analytics clusters are typically *recurrent*. Instances of the same job are issued periodically over new batches of data, e.g., hourly, daily, or weekly [7, 14, 28]. In fact, over 60% of the jobs in our clusters are recurrent, with the majority of them being submitted daily [28].

To deal with the unprecedented scale of our setting, we explore a novel, highly scalable approach to the well-studied problem of computation reuse. Fig. 1 positions our algorithm, BIGSUBS, within the broader literature on computation reuse. We focus on *subexpression selection*, a specialization of the view selection problem that considers subtrees of logical query plans as view candidates. We follow a holistic approach that periodically considers the entire workload with tens of thousands of queries, and selects the most promising subexpressions to materialize for improving the evaluation of subsequent queries.

In particular, we map subexpression selection to a bipartite graph labeling problem. The graph's vertices represent the queries and the candidate subexpressions, while the edges connect each query with its subexpressions. Then we split the graph labeling into two sub-problems: (i) labeling the subexpression vertices, which dictates the subexpressions that will be materialized, and (ii) labeling the edges, which determines the materialized subexpressions that will be used to evaluate each query. To scale to the aforementioned workload sizes, we follow a vertex-centric graph processing model [34] that iteratively performs the above labeling steps in parallel until they converge. For the vertex labeling part, we follow a probabilistic approach, while for the edge labeling part we solve local per-query ILPs.

**Contributions.** In summary, we make the following contributions:

- We provide an ILP-based formulation of our subexpression selection problem. Our formulation is easily extensible to capture several variations of the subexpression selection problem (§2).
- We map subexpression selection to a bipartite graph labeling problem, and present our BIGSUBS algorithm that performs the labeling in parallel using an iterative approach. To the best of our knowledge, this is the first parallel subexpression selection algorithm (§3).
- We express BIGSUBS using vertex-centric programming and provide an implementation in SCOPE [52], our internal SQL-like distributed execution engine with support for iterations (§3).
- We describe a set of optimizations to reduce the space of possible solutions that are explored by BIGSUBS, without compromising the quality of the returned solution (§4).
- The results of our experimental evaluation on production workloads show that BIGSUBS: (i) supports workloads with tens of thousands of jobs, (ii) outperforms heuristics-based approaches by up to three orders of magnitude, and (iii) can save 10–40% machine-hours on our clusters (§5).

BIGSUBS sits at the core of our CLOUDVIEWS system [26], which we are deploying in our SCOPE clusters to perform the end-to-end process of subexpression selection, materialization, and reuse.

## 2. PROBLEM FORMULATION

In this section, we define (§2.1) and formulate (§2.2) the problem of subexpression selection, i.e., selecting common subexpressions to materialize and reuse across queries.

Fig. 3 illustrates the key considerations in our problem with four queries ($Q_1$–$Q_4$) and two candidate subexpressions ($S_1$ and $S_2$).

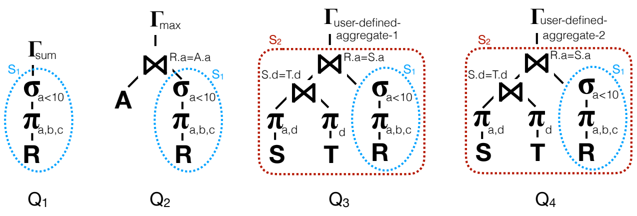


**Figure 3: Illustrating the subexpression selection problem.**

$S_1$ appears in more queries than $S_2$, but might be less useful as it captures smaller computation than $S_2$. On the other hand, $S_2$ might be much larger than $S_1$ in terms of byte size and thus consume more storage budget. Finally, if $S_2$ is materialized, then $S_1$ becomes redundant for evaluating $Q_3$ and $Q_4$,[1] as it is a subtree of $S_2$.

As also mentioned in §1, we consider workloads with recurring queries of fixed periodicity (mostly daily or weekly [28]) and having computation overlap across queries. Moreover, we consider an append-only storage system with queries operating on new batches of data arriving between each recurring instance. Given that a significant part of our query workload is relatively static, our goal is to periodically (offline) select the subexpressions to materialize. Each time one of these subexpressions references new data, it gets (re-)materialized using fresh data and is used across all queries that use the same subexpression over the same data.

### 2.1 Problem Statement

Let $Q = \{q_1, q_2, \ldots, q_n\}$ be a set of queries over a set of data sources $\mathcal{D}$, and $C_{\mathcal{D}}(q_i)$ be the cost of evaluating query $q_i$ over $\mathcal{D}$. Let $\mathcal{S} = \{s_1, s_2, \ldots, s_m\}$ be a set of materialized (stored) subexpressions and $C_{\mathcal{S}}(q_i)$ be the cost of evaluating $q_i$ over $\mathcal{D} \cup \mathcal{S}$. The cost $C_{\mathcal{S}}(Q)$ of evaluating the whole query set over $\mathcal{D} \cup \mathcal{S}$ is equal to the sum of evaluation costs of every query in $Q$, i.e., $C_{\mathcal{S}}(Q) = \sum_{i=1}^{n} C_{\mathcal{S}}(q_i)$.[2]

**Subexpression selection.** Our goal is to select the subexpression set $\mathcal{S}$ from the set $\mathbb{S}$ of all possible candidate subexpressions for $Q$ that minimizes the cost of evaluating $Q$ under a constraint set $\mathbb{C}$:

$$\underset{\mathcal{S} \subseteq \mathbb{S}}{\operatorname{argmin}}\, C_{\mathcal{S}}(Q), \text{ under } \mathbb{C} \qquad (1)$$

Below, we first provide the definition of a candidate subexpression, and then discuss the candidate enumeration and selection.

[1] $S_1$ may still be materialized and used for evaluating $Q_1$, $Q_2$.

[2] Typically, each query $q_i$ is associated with a weight $w_i$ that denotes its relative importance to the rest of the queries, such as the frequency with which it gets executed [20, 18]. To keep the formulation simple, and without loss of generality, we assume that $w_i = 1$ for all queries.

DEFINITION 1 (CANDIDATE SUBEXPRESSION). *Let $q$ be a query and $t$ be the tree corresponding to a logical plan of $q$. Any subtree $t'$ of $t$ is a* candidate subexpression *for $q$.*

**Candidate subexpression enumeration.** The candidate enumeration method determines the set of candidate subexpressions $\mathbb{S}$, out of the set of all possible subexpressions of the queries in $Q$ that will be considered for selection. A *complete* subexpression enumeration strategy considers all possible subexpressions in $Q$. In our approach, for each query we use the logical plan that is output by our optimizer [52] and restrict our enumeration to the plan's subexpressions. Although this choice compromises optimality, it was made purposefully to: (i) be less intrusive to the optimizer (less changes in this complex system component mean easier adoption of our approach both in our production setting and by external systems); (ii) reduce the search space, given our workload sizes; (iii) reuse existing plan signatures [4, 53] to efficiently decide subexpression equivalence; and (iv) use precise statistics from previous query runs instead of solely relying on the optimizer's estimates.

**Utility of subexpressions.** The utility of a subexpression $s$ over a query $q$ is the reduction in the evaluation cost of $q$ by using $s$. This is equivalent to the difference between the cost $C_{\mathcal{D}}(s_j)$ of evaluating $s_j$ over the base data and the cost $C_{acc}(s_j)$ of accessing $s_j$ after it has been materialized. We make the assumption that $C_{acc}(s_j)$ is equal to scanning $s_j$. More sophisticated cost models that take into account pipelining of operators or physical attributes (e.g., ordering, partitioning) could be used instead.

DEFINITION 2 (UTILITY OF SUBEXPRESSION). *Let $q_i$ be a query and $s_j$ one of its candidate subexpressions. We define the utility $u_{ij}$ of $s_j$ for $q_i$ to be:*

$$u_{ij} = C_{\mathcal{D}}(s_j) - C_{acc}(s_j) \tag{2}$$

For a given set of subexpressions $S$, there might be multiple rewritings $R_k$ of $q$ using different subsets of $S$. We define the utility of $S$ for $q$ to be the maximum evaluation cost reduction that can be achieved by one of the rewritings. Thus, we will also need to consider different rewritings in order to select the subexpressions to materialize.

DEFINITION 3 (UTILITY OF SUBEXPRESSION SET). *Let $q_i$ be a query and $S$ be a set of candidate subexpressions. Let $R_i^{max}$ be the rewriting that leads to the highest cost reduction for $q_i$. We define the utility $U_{\mathcal{S}}(q_i)$ of $S$ for $q_i$ to be:*

$$U_{\mathcal{S}}(q_i) = \sum_{s_j \in R_i^{max}} u_{ij} \tag{3}$$

The utility $U_{\mathcal{S}}(Q)$ of $\mathcal{S}$ for the query set $Q$ is the sum of utilities of $\mathcal{S}$ for each query in $Q$, i.e., $U_{\mathcal{S}}(Q) = \sum_{i=1}^{n} U_{\mathcal{S}}(q_i)$.

**Subexpression interactions.** To avoid using redundant subexpressions for evaluating a query (e.g., $S_1$ and $S_2$ for $Q_3$ in Fig. 3), we introduce the notion of interacting subexpressions.

DEFINITION 4 (INTERACTING SUBEXPRESSIONS). *Two candidate subexpressions $s_1$, $s_2$ for query $q$ are* interacting, *if the tree corresponding to the logical plan of one is a subtree of the other.*

To capture such interactions between candidate subexpressions, we define the *interaction matrix* $X$ to be a symmetric $m \times m$ matrix, where $m$ is the number of all possible candidate subexpressions for the query set. Element $x_{jk}$ has value 1 if the $j^{th}$ subexpression interacts with the $k^{th}$ one, otherwise it is 0.[3]

[3]A subexpression $s_c$ might appear multiple times in a query. In that case, we mark it as interacting with another subexpression $s'_c$, if at least one of the occurrences of $s_c$ interacts with $s'_c$ in a query.

**Subexpression cost and constraints.** The cost of a subexpression is a linear combination of several components, such as the storage footprint required for its materialization and its maintenance cost. As described in the beginning of the section, we discard updates in our append-only system and therefore consider storage footprint as the only cost component. However, our cost model can easily be extended to consider other components. For a subexpression set $\mathcal{S}$, we define $b_{\mathcal{S}} = [b_1, b_2, \ldots, b_m]$ as the column vector representing the cost of each of the subexpressions in $\mathcal{S}$. We define the combined cost $B_{\mathcal{S}}$ of $\mathcal{S}$ as the sum of the costs of its subexpressions. Then, our constraint set $\mathbb{C}$ includes a budget $B_{max}$ that $B_{\mathcal{S}}$ has to respect.

**Utility-based subexpression selection.** Having defined the utility and cost of a subexpression set, we can rewrite our problem formulation of Eq. 1 to an equivalent one that selects the subexpression set $\mathcal{S}$ with the maximum utility for the query set:

$$\underset{\mathcal{S} \subseteq \mathbb{S}}{\text{argmax}} \sum_{i=1}^{n} U_{\mathcal{S}}(q_i), \text{ with } B_{\mathcal{S}} \leq B_{max} \tag{4}$$

**Query rewriting using subexpressions.** Once we have selected and materialized our subexpressions, we provide our optimizer with the set of subexpressions that should be used for each query $q_i$. The query optimizer performs the actual rewriting of the query, adding additional operations, such as data ordering or partitioning, if needed [52].

## 2.2 Subexpression Selection as an ILP

We now describe the ILP formulation of our problem. Let $z_j$ be a $0-1$ variable indicating whether the $j^{th}$ subexpression is selected or not. Considering a budget $B_{max}$ for the total cost of the selected subexpressions and taking into account Eq. 3, we can rewrite our problem of Eq. 4 as follows:

$$\begin{aligned} \text{maximize} & \sum_{i=1}^{n} \sum_{s_j \in R_i^{max}} u_{ij} \cdot z_j \\ \text{s.t.} & \sum_{j=1}^{m} b_j \cdot z_j \leq B_{max} \end{aligned}$$

Note that $R_i^{max}$ is computed based on: (i) the subexpression utilities $u_{ij}$, (ii) the subexpression selection variables $z_j$, as a subexpression that is not selected cannot be used for rewriting, and (iii) the subexpression interactions $x_{jk}$, as the selected subexpressions should not be redundant. This makes the above formulation non-linear. To solve this problem, we introduce the binary variables $y_{ij}$ to denote whether the subexpression $s_j$ is selected by query $q_i$ for rewriting. This reduces the above formulation to the following linear one:

$$\begin{aligned} \text{maximize} & \sum_{i=1}^{n} \sum_{j=1}^{m} u_{ij} \cdot y_{ij} \\ \text{s.t.} & \sum_{j=1}^{m} b_j \cdot z_j \leq B_{max} \\ & y_{ik} + \frac{1}{m} \sum_{\substack{j=1 \\ j \neq k}}^{m} y_{ij} \cdot x_{jk} \leq 1 \quad \forall i \in [1,n], k \in [1,m] \\ & y_{ij} \leq z_j \quad \forall i \in [1,n], j \in [1,m] \end{aligned}$$

The second constraint above uses the interaction matrix $X$ (see §2) to ensure that we do not use subexpressions that are covered by others in the same query. The left-hand side of this constraint ensures that if the $k^{th}$ view is selected ($y_{ik} = 1$), none of its interacting views is selected ($\sum_{\substack{j=1 \\ j \neq k}}^{m} y_{ij} \cdot x_{jk} = 0$). Otherwise, if the $k^{th}$

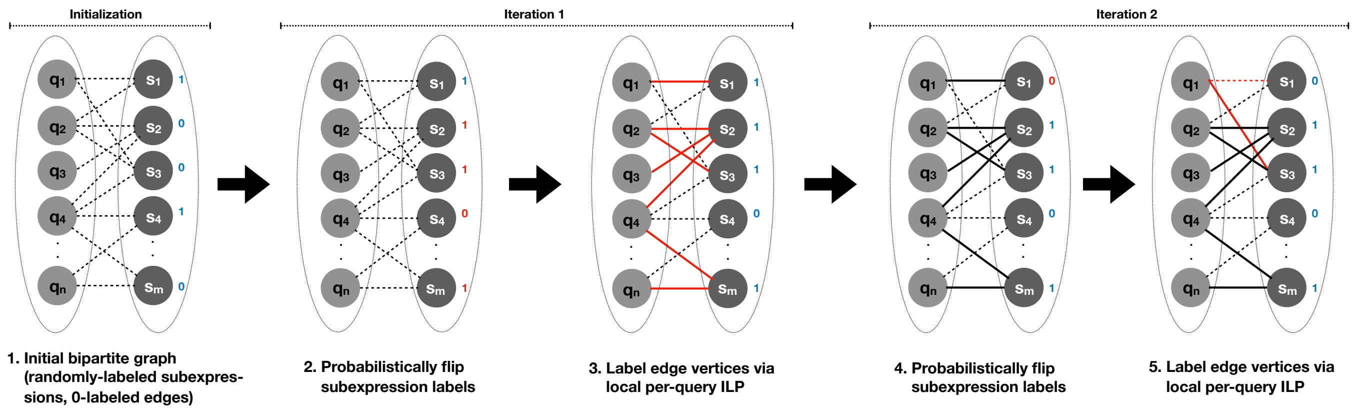

Initialization
Iteration 1
Iteration 2
1. Initial bipartite graph (randomly-labeled subexpressions, 0-labeled edges)
2. Probabilistically flip subexpression labels
3. Label edge vertices via local per-query ILP
4. Probabilistically flip subexpression labels
5. Label edge vertices via local per-query ILP

**Figure 4: Illustrating first two iterations of subexpression selection via bipartite graph labeling. We assume each subexpression has storage footprint $b_j$=1 and the total budget is $B_{max}$=3. Subexpression labels are shown next to the vertices. For the edges, we use solid lines when label is 1 and dashed ones otherwise. At each iteration, we mark with red the labels whose value changed.**

view is not selected ($y_{ik} = 0$), then one or more of the interacting views could be selected.[4] We apply this constraint to every view, i.e., $k \in [1, m]$. The last constraint ensures that a subexpression is used for query rewriting only when it is materialized.

The above ILP becomes intractable for very large workloads, where $m$ and $n$ are in the order of a million. The large number of integer variables is a problem even for state-of-the-art solvers, as we show in §5. Therefore, below we present an approximation scheme to scale subexpression selection to larger workloads.

## 3. SCALING SUBEXPRESSION SELECTION

In this section, we present our solution to the subexpression selection problem that scales to tens of thousands of queries. We map our problem to a bipartite graph labeling problem with constraints (§3.1), describe our approximate BIGSUBS algorithm for solving it efficiently (§3.2), and discuss how we leverage existing distributed execution platforms to execute BIGSUBS in parallel (§3.3).

### 3.1 Bipartite Graph Labeling Problem

Our ILP problem formulation (§2.2) can be split into multiple smaller ILPs by mapping it to a labeling problem over a bipartite graph. Let $Q$ be a set of $n$ queries and $\mathbb{S}$ be the set of $m$ candidate subexpressions for the queries in $Q$. Consider a bipartite graph $G = (V_Q, V_{\mathbb{S}}, E)$, where each vertex $v_{q_i} \in V_Q$ corresponds to query $q_i \in Q$, and each vertex $v_{s_j} \in \mathbb{S}$ corresponds to subexpression $s_j \in \mathbb{S}$. There is an edge $e_{ij} \in E$ between $v_{q_i}$ and $v_{s_j}$, if $s_j$ is a subexpression of $q_i$. Each edge $e_{ij}$ is also associated with $u_{ij}$ representing the utility of $s_j$ for $q_i$. As in §2.1, we use $x_{jk}$ to capture subexpression interactions, as well as $b_j$ and $B_{max}$ for the storage footprint of $s_j$ and the total storage capacity, respectively.

Our subexpression selection problem can now be expressed as the following graph labeling problem:
(i) Assign a $0{-}1$ label $z_j$ to each vertex $v_{s_j} \in V_{\mathbb{S}}$, subject to the following constraint:

$$\sum_{j=1}^{m} b_j \cdot z_j \leq B_{max} \tag{5}$$

(ii) assign a $0{-}1$ label $y_{ij}$ to each edge $e_{ij} \in E$, subject to the following optimization:

[4]We normalize $\sum_{\substack{j=1\\ j\neq k}}^{m} y_{ij} \cdot x_{jk}$ by $m$ to make sure that the resulting quantity is between 0 and 1.

$$\text{maximize} \quad \sum_{j=1}^{m} u_{ij} \cdot y_{ij} \tag{6}$$

$$\text{s.t.} \quad y_{ik} + \frac{1}{m}\sum_{\substack{j=1\\ j\neq k}}^{m} y_{ij} \cdot x_{jk} \leq 1 \qquad \forall k \in [1, m] \tag{7}$$

$$y_{ij} \leq z_j \qquad \forall j \in [1, m] \tag{8}$$

As in our ILP formulation, the $z_j$ labels denote whether a subexpression $s_j$ will be materialized, and the $y_{ij}$ labels denote whether $s_j$ will be used when evaluating $q_i$. Eq. 5 guarantees that the chosen $z_j$ values will respect the storage capacity, while Eq. 6–8 find the rewritings with the highest utility for each query given some $z_j$ values. The $z_j$ values that maximize the total utility for all queries is equivalent to the solution of the global ILP of §2.2.

For example, consider the leftmost bipartite graph in Fig. 4. The vertices represent $n$ queries and $m$ subexpressions, and edges connect queries with their subexpressions, e.g., $s_1$ and $s_3$ are subexpressions of $q_1$. For simplicity, we assume that each subexpression has a storage footprint of 1, and the budget for materialization is 3. The rightmost graph in Fig. 4 represents a solution to the graph labeling problem. The subexpression label is shown next to the respective vertex. For the edge labels, we use solid lines to denote 1-labeled edges and dashed lines for 0-labeled ones. In this instance, subexpressions $s_1$, $s_2$, $s_m$ are chosen to be materialized. Query $q_1$ will be evaluated using $s_1$, $q_2$ using $s_2$ and $s_3$, etc.

In contrast to our ILP formulation, by using the above graph labeling formulation, we can isolate the problems of finding which subexpressions to materialize ($z_j$) and which subexpressions to use when evaluating each query ($y_{ij}$). This approach has two major benefits: (i) instead of solving a single giant ILP, we solve a number of much smaller ILPs, which makes them computationally feasible, and (ii) by splitting the problem to several sub-problems that can be executed independently, we can easily parallelize them.

### 3.2 The BIGSUBS Algorithm

We now present an approximate algorithm, BIGSUBS, to solve the bipartite graph labeling problem efficiently. BIGSUBS probabilistically sets subexpression vertex labels, attempting to fulfill the global capacity constraint (Eq. 5), and solves the smaller ILPs (Eq. 6–8) given a set of subexpression labels.

#### 3.2.1 Overview

Our BIGSUBS algorithm follows an iterative approach. Each iteration includes two steps: (i) assign labels to the subexpression

**Algorithm 1:** BIGSUBS

```
Input  : $G$=($V_Q$, $V_{\mathbb{S}}$, $E$), subexpression utilities $u_{ij}$, interactions
         $x_{jk}$, storage footprints $b_j$, storage budget $B_{max}$, iterations $k$
Output: subexpressions to materialize $z_j$,
         subexpressions to use when evaluating each query $y_{ij}$
1  Initialize($z_j$)                    // initialize $V_{\mathbb{S}}$ labels
2  $y_{ij}$ = 0, $\forall e_{ij} \in E$                  // initialize $E$ labels
3  $B_{cur} = \sum_j b_j \cdot z_j$                   // current budget used
4  $U_{max} = \sum_{i,j} u_{ij}$    // upper bound for total utility
5  $U^j_{max} = \sum_i u_{ij}$      // upper bound for utility of $s_j$
6  iter = 0
7  updated = 0
8  while updated or iter < $k$ do
       // Subexpression vertex labeling
9      foreach $v_j \in V_{\mathbb{S}}$ do
10         $p_{flip}$ = FlipP($b_j$,$B_{cur}$,$U_{cur}$,$U^j_{cur}$)
11         $z^{new}_j$ = DoFlip($p_{flip}$)
12         if $z^{new}_j$ != $z_j$ then
13             updated = 1
14             if $z^{new}_j$ == 0 then
15                 $B_{cur}$ −= $b_j$
16             else
17                 $B_{cur}$ += $b_j$
18             $z_j = z^{new}_j$
       // Edge labeling
19     $U_{cur}$ = 0                     // current total utility
20     $U^j_{cur}$ = 0              // current utility of each $s_j$
21     foreach $v_i \in V_Q$ do
22         $M_i = \{j : e_{ij} \in E \ \& \ j : u_{ij} > 0\}$
23         $M'_i = \{j : e_{ij} \in E \ \& \ z_j > 0\}$
24         $y_{ij}$ = LocalILP($u_{ij}$,$x_{jk}$)      // solve local ILP
25         $U_{cur}$ += $\sum_{j \in M_i \cap M'_i} u_{ij} \cdot y_{ij}$
26         foreach $j : e_{ij} \in E$ do
27             $U^j_{cur}$ += $u_{ij} \cdot y_{ij}$
28     iter++
29 return $z_j, y_{ij}$
```

vertices, and (ii) given the subexpression vertices determined at the first step, assign labels to the edges. This two-step process is repeated until there is no change to the vertex and edge labels or until we reach a predefined number of iterations.

BIGSUBS is outlined in Algorithm 1. It takes as input the bipartite graph $G$ encoding the $n$ queries and their $m$ candidate subexpressions, along with the subexpression utilities $u_{ij}$, interactions $x_{jk}$ and storage footprints $b_j$, and the maximum allowed number of iterations $k$. It outputs the subexpressions to be materialized ($z_j$) and the subexpressions to be used when evaluating each query ($y_{ij}$). Here we present the basic version of the algorithm. We discuss various optimizations in §4.

We first randomly assign labels to the subexpression vertices and 0-labels to the edges (lines 1–2). We also initialize various auxiliary variables (lines 3–7), which we discuss later. Then the iterative part of the algorithm is performed. For the subexpression vertex labeling part, we use a probabilistic approach, which allows us to perform decisions without central coordination (lines 9–18). Based on the current utility and used budget, we calculate the probability of changing the label of a vertex. Similar probabilistic techniques have been used in other graph labeling problems [36]. The edge labeling part is done by solving local ILP problems (lines 21–27).

Fig. 4 illustrates the first two iterations of the BIGSUBS. Upon label initialization (Step 1), in the first iteration, given we are below the capacity budget $B_{max}$=3, we flip the labels of $s_2$,$s_3$,$s_4$, and $s_m$ (Step 2). Given the new labels, we choose rewritings for each of the queries to maximize each query's utility (Step 3), while respecting subexpression interactions (e.g., $s_1$ with $s_3$ for $q_1$). In the second iteration, given we are above budget, we choose to not materialize $s_1$ (Step 4). Accordingly, we change the labels of the edges adjacent to $q_1$ to use $s_3$ instead of $s_1$ (Step 5).

Below we provide details on the vertex (§3.2.2) and edge labeling (§3.2.3) steps, and present an analysis of BIGSUBS (§3.2.4).

### 3.2.2 *Labeling Subexpression Vertices*

Our goal is to assign 1-labels to the subexpressions that will lead to the highest utility, while abiding by the budget constraint of Eq. 5. To perform distributed decisions in the presence of this global aggregation constraint, we assign labels probabilistically. We iterate over all subexpressions, and for each subexpression $s_j$ we calculate the probability $p^j_{flip}$ of changing the current $s_j$ label (lines 10–11 in Algorithm 1), taking into account both the budget $B_{cur}$ used by the currently selected subexpressions and the contribution $U^j_{cur}$ of $s_j$ to the current total utility $U_{cur}$ of the selected subexpressions. In particular, we use the following formulas for the probabilities:

$$p^j_{capacity} = \begin{cases} 1 - B_{cur}/B_{max} & \text{if } B_{cur} < B_{max} \\ 1 - B_{max}/B_{cur} & \text{otherwise} \end{cases}$$

$$p^j_{utility} = \begin{cases} 1 - U^j_{cur}/U_{cur} & \text{if } z_j = 1 \\ \frac{U^j_{max}/b_j}{U_{max}/B_{max}} & \text{if } iter \leq p \text{ or } B_{cur} \leq B_{max} - b_j \\ 0 & \text{otherwise} \end{cases}$$

$$p^j_{flip} = p^j_{capacity} \cdot p^j_{utility}$$

The intuition in computing $p^j_{flip}$ is twofold: (i) the further we are from the storage budget, the higher the probability of flipping labels should be, as captured by the $p_{capacity}$ component; (ii) the higher the current utility of an 1-labeled subexpression (i.e., a subexpression that is currently selected for materialization), the lower the chances it will be unselected (first case of $p_{utility}$)[5], while the higher the potential benefit of a 0-labeled one, the higher the chances it will be selected (second case of $p_{utility}$). We define the potential benefit $U^j_{max}$ of a 0-labeled subexpression as the total utility if it were to be used by all possible queries. Similarly, $U_{max}$ is the sum of $U^j_{max}$ over all subexpressions.

Note that with $p^j_{flip}$ we want to strike a balance between avoiding local optima and converging to a solution reasonably fast. To improve convergence speed, after $p$% of the maximum allowed number of iterations $k$, we only turn a label of a subexpression from 0 to 1 if the maximum budget is respected even after we add the budget of this subexpression.[6] We call these *stricter iterations*.

After performing the label flipping, we also update the counter for the currently used capacity $B_{cur}$ (lines 12–17).

### 3.2.3 *Labeling Edges*

Given the subexpressions chosen in the vertex-labeling step, we then determine which subexpressions will be used to evaluate each query by labeling the edges that are adjacent to that query, following the optimization goal in Eq. 6 and respecting the constraints in Eq. 7–8. To simplify this optimization, we make the observation that a subexpression $s_j$ cannot be used to evaluate a query $q_i$ (i.e., $y_{ij} = 0$) when it has no utility for $q_i$ ($u_{ij} = 0$) or when it is not materialized ($z_j = 0$). Accordingly, we define the sets $M_i = \{j : u_{ij} > 0\}$ and $M'_i = \{j : z_j > 0\}$. Using these sets, we can significantly reduce the dimensions of $y_{ij}$ and the number of constraints we need to consider in our optimization, as follows:

[5]Note that we can apply logarithmic or exponential normalization to make the distribution more uniform in the [0,1] interval.

[6]In practice, we got good results by setting $p = 80\%$, but lower values can be used if convergence speed is more important than solution quality.

$$\text{maximize} \sum_{j \in M_i \cap M'_i} u_{ij} \cdot y_{ij} \quad (9)$$

$$\text{s.t. } y_{ik} + \frac{1}{m} \sum_{\substack{j \in M_i \cap M'_i \\ j \neq k}} y_{ij} \cdot x_{jk} \leq 1 \quad \forall k \in M_i \cap M'_i \quad (10)$$

$M_i$ is known statically for each query vertex in the graph, while $M'_i$ is determined by the subexpressions that were selected in the subexpression vertex-labeling step of this iteration. We let each query vertex $v_{q_i}$ to keep track of $u_{ij}$ and $x_{jk}$ for $j \in M_i \cap M'_i$. Note that in this case $u_{ij}$ is a $1 \times |M_i \cap M'_i|$ vector and $x_{jk}$ is a $|M_i \cap M'_i| \times |M_i \cap M'_i|$ matrix, both significantly smaller than before. As a result, the above ILP problem can be computed locally at each query vertex in the graph to determine the labels of its adjacent edges (line 24 in Algorithm 1).

### 3.2.4 *Analysis*

We now focus on the complexity, correctness, and convergence of BIGSUBS.

**Complexity.** Instead of solving a single global ILP, with BIGSUBS we are solving $n$ ILPs, where the size of each problem is the effective degree $d_i$ of query vertices, i.e., $d_i = |M_i \cap M'_i|, \forall i \in n$. Additionally, we label each of the $m$ subexpression vertices. Thus, the worst case complexity of BIGSUBS is given as $k \cdot (m + n \cdot 2^{max(d_i)})$, where $k$ is the number of iterations. Note that $d_i$ is bounded by either the largest query size or the maximum number of subexpressions allowed to be used in a given query. In that case, the algorithm scales linearly with the sum of the number of queries and the number of subexpressions, i.e., $(m + n)$.

**Correctness.** Recall that our ILP formulation in §2.2 has three constraints for exact optimal solution. The first constraint limits the total cost to $B_{max}$. Since BIGSUBS is an approximate algorithm with each subexpression vertex making independent decisions on whether to flip their labels, this constraint is not strictly observed. Hence, we need to study whether the final cost comes close to $B_{max}$. Since the first $p\%$ iterations are designed to explore all interesting subexpressions, we consider the remaining $(1 - \frac{p}{100}) \cdot k$ stricter iterations. For these iterations, a flip from 0 to 1 is only possible if the total cost is within $B_{max}$ (the second case of $p^j_{utility}$). So we focus on the case when the cost is overshooting the budget and a flip from 1 to 0 is supposed to happen. The corresponding probability is given as: $p^j_{flip} = (1 - \frac{U^j_{cur}}{U_{cur}})(1 - \frac{B_{max}}{B_{cur}})$. Thus, the total cost in iteration $t + 1$ is given as: $B^{t+1}_{cur} = B^t_{cur} - \sum^m_{j=1} p^j_{flip} \cdot b_j$. Assuming that there is a set of very low utility subexpressions, i.e., $\frac{U^j_{cur}}{U_{cur}} \approx 0$, that should be flipping their labels from 1 to 0, we can approximate the cost in iteration $t + 1$ as: $B^{t+1}_{cur} = B^t_{cur} - (1 - \frac{B_{max}}{B^t_{cur}}) \cdot B_{total}$, where $B_{total}$ is the total cost of those low utility expressions.[7] Figure 5 shows that the cost error, i.e., $\frac{(B_{cur} - B_{max})}{B_{max}} \cdot 100$, approaches 0 as the number of iterations increases. We set $B_{total} = B^0_{cur} - B_{max}$, i.e., the total weight of low utility subexpressions is equal to the cost overshoot.

The second constraint in our ILP formulation of Section 2.2 applies restrictions to subexpression interaction for each query, thereby making it readily partitionable across query vertices (as done in Equation 10 of our local ILP). The final constraint in our global ILP, i.e., $y_{ij} \leq z_j$, is enforced by construction of $M'$ since each of the smaller ILPs have all $z_j$ as ones.

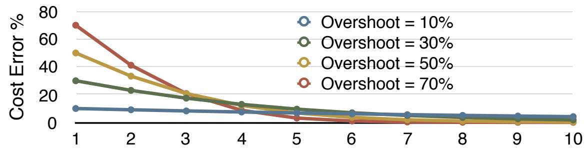


**Figure 5: The cost budget error over last $(1 - \frac{p}{100}) \cdot k$ iterations.**

**Convergence.** As above, we focus on the last $(1 - \frac{p}{100}) \cdot k$ stricter iterations. Observe that for a given set of subexpression vertex labels, the local ILPs will always find a fixed solution, and hence $U_{cur}$ will converge. Even when the subexpression vertex labels flip from 0 to 1, the flipping stops once the cost budget is reached (the second case of $p^j_{utility}$), and so $U_{cur}$ will again converge. Thus, we consider the remaining case of subexpression vertex labels flipping from 1 to 0, the probability of which is given as: $p^j_{flip} = (1 - \frac{U^j_{cur}}{U_{cur}})(1 - B_{ratio})$, where $B_{ratio}$ is either $\frac{B_{cur}}{B_{max}}$ or $\frac{B_{max}}{B_{cur}}$ depending on whether $B_{cur} < B_{max}$. Assuming that every subexpression thats flips its label to 0 was used in at least one of the queries (otherwise, it does not impact $U_{cur}$ anyway), the new total utility in iteration $t + 1$ is given as: $U^{t+1}_{cur} = U^t_{cur} - \sum^m_{j=1} p^j_{flip} \cdot U^j_{cur}$. Expanding the terms gives us: $U^{t+1}_{cur} = B_{ratio} \cdot U^t_{cur} + \frac{1 - B_{ratio}}{U^t_{cur}} \cdot \sum^m_{j=1} U^j_{cur} \cdot U^j_{cur}$. Using Cauchy-Schwarz inequality, we get: $(B_{ratio} + \frac{1 - B_{ratio}}{m}) \cdot U^t_{cur} \leq U^{t+1}_{cur} \leq U^t_{cur}$. We saw in the correctness discussion above that $B_{cur}$ converges to $B_{max}$, i.e., $B_{ratio}$ converges to 1, and as a result $U^{t+1}_{cur}$ converges as well.

Note that our probabilistic graph labeling algorithm does not guarantee halting at the optimal solution, and might instead end up in a local optimum. This is similar to iterative hill-climbing algorithms, as also noted in [36].

## 3.3 Distributed Execution

In this section, we describe how we scale-out the execution of BIGSUBS by using a distributed execution framework. As described in Algorithm 1, in each iteration of our BIGSUBS algorithm, we first examine each subexpression vertex to determine its label, and then examine each query vertex to determine the labels of its adjacent edges. Given that all label assignments are performed locally to the graph's vertices, our algorithm fits well with the vertex-centric graph processing model as popularized by systems such as Giraph [1] or GraphLab [17]. Following this model, each vertex assigns a label to itself or to its adjacent edges, based on whether it is a subexpression or a query vertex, respectively, and then sends messages containing the newly assigned labels to all its neighbors. Thus, any off-the-shelf vertex-centric graph system could run our algorithm in a distributed fashion.

Similarly, instead of using a dedicated graph engine, we can execute BIGSUBS using a standard SQL engine, as shown in Vertexica [24, 25, 27] or Grail [13]. This is particularly useful in our setting, since big part of the preprocessing, i.e., extracting queries and generating subexpressions, is already done via SCOPE, which is a SQL-like processing engine. Therefore, to avoid stitching together multiple engines and moving data across them, we opted for executing BIGSUBS on SCOPE.

The implementation of BIGSUBS using the SCOPE query language [52] is shown in Listing 1. Note that we generate and store all candidate subexpressions upfront. Then, we start by creating the three tables $V_Q(v_i, u_i, x_i, y_i)$, $V_{\mathbb{S}}(v_j, b_j, z_j)$, and $E(v_i, v_j)$, and also set the values of $U_{max}$ and $B_{max}$ (lines 1–11). Thereafter, in

[7]Expecting low utility subexpressions to flip is the stricter case; otherwise, $B_{total}$ is simply a fraction of the total cost of all subexpressions and our proof still holds.

```
1  Vq=REDUCE (SSTREAM @repeatedSubexpressions)
2      ON JobGUID
3      USING InitJobVertices();
4  Vs=SELECT HashTag AS vj, Size AS bj, InitZj() AS zj
5      FROM (SSTREAM @repeatedSubexpresssions)
6      GROUP BY HashTag,Size;
7  E=SELECT JobGUID AS vi, HashTag AS vj
8      FROM (SSTREAM @repeatedSubexpressions);
9  Umax=SELECT SUM(InclusiveTime) AS umax
10         FROM (SSTREAM @repeatedSubexpressions);
11 Bmax=SELECT SUM(vj) AS bmax FROM Vs;
12
13 LOOP(i,@maxIter)
14 {
15   Vs=REDUCE(
16       SELECT * FROM Vq
17       INNER JOIN E ON Vq.vi==E.vi
18       INNER JOIN Vs ON E.vj==Vs.vj
19       CROSS JOIN Cu
20       CROSS JOIN Cb
21       CROSS JOIN Umax
22       CROSS JOIN Bmax)
23     ON vj
24     USING ProbabilisticLabels(@B,@i,@maxIter);
25
26   U_cur=SELECT SUM(Product(uij,yij)) AS u_cur FROM Vq;
27   B_cur=SELECT SUM(bj*zj) AS b_cur FROM Vs;
28
29   Vq_update=REDUCE(
30               SELECT vi,uij,xij,vj FROM Vq
31               INNER JOIN E ON Vq.vi==E.vi
32               INNER JOIN Vs ON E.vj==Vs.vj
33               WHERE zj>0)
34             ON vi
35             USING SolveLocalILP();
36
37   Vq=SELECT Vq.vi, Vq.uij, Vq.xij,
38        (Vq_update.uij==null?Vq.yij:Vq_update.yij) AS yi
39       FROM Vq LEFT JOIN Vq_update
40       ON Vq.vi==Vq_update.vi;
41 }
42 Vs_selected = SELECT * FROM Vs WHERE zj>0;
43 OUTPUT Vs_selected TO @selectedSubexpressions
```

**Listing 1: Distributed graph labeling using SCOPE.**

each iteration we join the three tables ($V_Q$, $V_{\mathbb{S}}$, $E$) and aggregate alternatively on $V_{\mathbb{S}}.v_j$ and $V_Q.v_i$, respectively, to assign the vertex and edge labels (lines 13–41). We update the subexpression vertices with new labels (lines 15–24) and use the new $V_{\mathbb{S}}$ to assign the edge labels. We left-join $V_Q$ with the updated query vertices ($V_Q\_update$) in order to get the new $V_Q$, which will be used in the next iteration (lines 29–40). Finally, we return all $V_{\mathbb{S}}$ elements with $z_j > 0$, i.e., those that are chosen for materialization (lines 42–43).

## 4. OPTIMIZATIONS

In this section, we describe a number of optimizations to improve the performance of the local per-query ILP problems that we need to solve at each iteration of our graph labeling algorithm. We focus on optimizing the local ILPs, since labeling subexpression vertices is done in constant time, while the local ILPs could be arbitrarily large due to possible skew in the size of the queries in the workload, thereby causing an entire iteration to block.

### 4.1 Branch-and-bound

To solve the local ILPs discussed in §3.2.3 (line 24 of Algorithm 1), a naïve approach would be to consider all combinations (i.e., the power set) of candidate subexpressions and pick the one with the maximum utility (objective function of Eq. 9) that does not have interacting subexpressions (constraint of Eq. 10). Another approach would be to use a generic ILP solver, which is not tailored towards our specific use case either (see also §5.2). Instead, we solve the local ILPs using a branch-and-bound technique.

The idea is that once we find interacting subexpressions, we no longer need to consider any superset of these subexpressions, as those will also violate the interaction constraints. Fig. 6 illustrates our technique through a toy example including three subexpressions $s_1, s_2, s_3$, with $s_1$ interacting with $s_2$. When we visit the set $\{s_1, s_2\}$, we prune the search space below it, and thus $\{s_1, s_2, s_3\}$ is never explored. This can lead to significant pruning, especially in the presence of many interacting subexpressions.

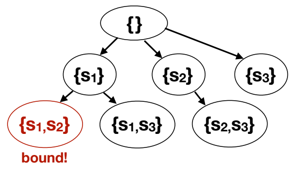


**Figure 6: Illustrating branch-and-bound for three subexpressions $\{s_1, s_2, s_3\}$, with $\{s_1, s_2\}$ interacting.**

Algorithm 2 shows the pseudocode for solving the ILP using our branch-and-bound technique. It takes as input the vector with the utility $u_{ij}$ of each candidate subexpression $s_j$ for query $q_i$ and the interaction matrix $x_{jk}$ between subexpressions. Note that at this point the number of subexpressions is at most $m$; in practice it is much smaller after the dimensionality reduction we discussed in §3.2.3, since here we consider only subexpressions that have already been selected for materialization. We recursively explore different evaluation vectors (i.e., vectors that denote whether a subexpression will be used in evaluating $q_i$), and return the one that yields the maximum utility. During exploration, we first check whether the current vector has interacting subexpressions and bound the exploration if this is the case (lines 6–7 in Algorithm 2). Each time we reach a complete evaluation vector (i.e., one in which we have given values to all subexpression elements), we check its utility using Eq. 6 and update the best vector found so far if we exceeded its utility (lines 8–11). Otherwise, the algorithm branches with and without setting the $j^{th}$ subexpression element to 1 (lines 12–15).

### 4.2 Exploration Strategies

We now consider different exploration strategies to achieve better pruning. The branch-and-bound algorithm discussed so far (Algorithm 2) explores the subexpression space in a bottom-up fashion, i.e., it incrementally adds more subexpressions until an interaction is found. We could alternatively explore the search space in a top-down fashion, i.e., select all subexpressions and incrementally drop one-by-one until we reach a non-interacting set. Fig. 7 shows a top-down exploration for our running example. Here we start with all three subexpressions selected, and by removing $s_1$ or $s_2$ we reach non-interacting subexpression sets. When there are few interactions between the subexpressions, this technique can lead to solutions faster. Moreover, given that our utility function is monotonically decreasing when we remove subexpressions, we can further bound the exploration when we find a set whose utility is below the best utility so far.

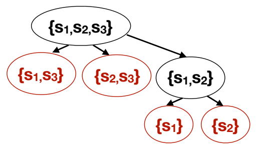


**Figure 7: Top-down exploration for our running example.**

As already discussed, bottom-up exploration tends to be beneficial in case of highly interactive candidate subexpressions, while top-down exploration is more beneficial in the presence of few interactions. To this end, we could dynamically pick the exploration

**Algorithm 2:** `LocalILP`

```
Input  : $u_{ij}$: $1 \times m$ vector of utilities for query $q_i$
         $x_{jk}$: $m \times m$ subexpression interaction matrix
Output: $y_{ij}$: subexpressions to use for evaluating $q_i$ (eval. vector)
// Initialize to 0s
1  $y_{ij}^{cur}$ = AllZeros()          // current eval. vector
2  $y_{ij}^{best}$ = AllZeros()          // best eval. vector
3  BranchAndBound($y_{ij}^{cur}$, 0)
4  return $y_{ij}^{best}$

5  Function BranchAndBound($y_i^{cur}$, $j_s$)
6      if IsInteracting($y_{ij}^{cur}$, $x_{jk}$) then
7          return          // bound if interaction found
8      if $j_s == m$ then
9          if Utility($y_{ij}^{cur}$) > Utility($y_{ij}^{best}$) then
10             $y_{ij}^{best}$ = $y_{ij}^{cur}$          // update best
11         return
12     $y_{ij}^{cur}[j_s]$ = 1     // attempt to use $j_s$-th subexpr.
13     BranchAndBound($y_{ij}^{cur}$, $j_s$+1)
14     $y_{ij}^{cur}[j_s]$ = 0                 // backtrack
15     BranchAndBound($y_i^{cur}$, $j_s$+1)
```

direction based on the frequency of non-zero elements in the interaction matrix $X$. Specifically, for an $m \times m$ interaction matrix, we define the *interaction fraction* as *NonZeros*$(X)/(m(m-1))$. When the interaction fraction is greater than a threshold, we pick the top-down exploration, otherwise we follow the bottom-up one.

Taking the dynamic exploration a step further, we observe that at each branching step of our branch-and-bound, the interaction fraction for the remaining subexpressions varies. Therefore, instead of choosing upfront, we can adaptively decide to use a bottom-up or a top-down approach at each step of the algorithm.

## 4.3 Skipping Trivial States

Finally, we present some additional techniques to avoid trivial states of the exploration and thus reach a high-utility solution faster.

Notice that subexpressions that do not interact with others will definitely be part of the best solution since our utility increases monotonically as more subexpressions are added. Selecting such subexpressions upfront, instead of starting with the all-zero $y_{ij}$ vector, can significantly reduce our search space. Fig. 8 shows that this technique reduces the search space by 3 for our running example.

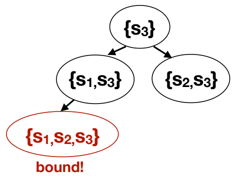


**Figure 8: Better initialization given that $s_3$ does not interact with any other subexpression.**

Similar to the better initialization, while exploring the solution search space, we could find subexpressions that do not interact with any of the *remaining* elements. Thus, in each iteration, we check whether there exist such remaining subexpressions, and if so, we directly set the corresponding $y_{ij}$ elements to 1. In our running example, consider two new elements, $s_4$ and $s_5$, which interact only with $s_1$. As shown in Fig. 9, we can immediately include the new elements to the $\{s_2, s_3\}$ set, as this will lead to no interactions.

Finally, we can also skip altogether the ILPs for queries whose subexpressions' labels have not changed since the last vertex labeling iteration of Algorithm 1. This would require keeping track of the previous subexpression vertex labels.

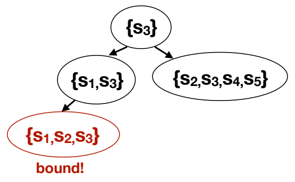


**Figure 9: Dynamically adding new subexpressions $s_4$ and $s_5$ (which interact only with $s_1$) to the $\{s_2, s_3\}$ set.**

## 5. EVALUATION

In this section, we present an evaluation of our subexpression selection algorithm over both production and synthetic workloads. The production workloads used in this paper are derived from SCOPE [52] jobs that are predominantly run in our clusters. Briefly, SCOPE exposes a SQL-like query language in which users write scripts that are then compiled to Dryad DAGs [51] and subsequently executed on the shared cluster.

We split our evaluation in two main parts. First, we assess the benefit of BIGSUBS on production workloads at Microsoft's analytics clusters, both in terms of supported workload sizes and improvement over heuristic-based approaches (§5.1). Second, we perform a detailed analysis of our algorithm, including comparison with optimal ILP solutions, scalability on single machine, convergence, and pruning effectiveness of our optimizations (§5.2).

The main takeaways from our evaluation are the following:

- BIGSUBS handles large production workloads comprising tens of thousands of jobs, significantly outperforms heuristics-based approaches by up to three orders of magnitude, and can potentially save 10–40% machine-hours[8] (Table 1 and Fig. 10).
- In contrast to an implementation of our global ILP formulation of §2.2 using a state-of-the-art commercial optimizer, which cannot support more than $2K$ subexpressions on a desktop machine (Fig. 11), our approach scales up to $50K$ subexpressions on the same single machine (Fig. 12).
- It converges to a solution that respects the space budget (Fig. 13), while our pruning optimizations can effectively reduce the search space by more than $90\%$ in most cases (Fig. 14).

## 5.1 Impact on Production Workloads

To study the impact of our approach on real use cases, we use BIGSUBS to select subexpressions to materialize in one of our production clusters. Our goal is to be able to (i) perform subexpression selection over our large production workloads (§5.1.2), and (ii) select subexpressions with higher utility than existing subexpression selection heuristics (§5.1.3). Below we first describe our workloads and the tuning of our algorithm, and then present our results.

### 5.1.1 Setup

**Workloads.** In our analytics clusters, we maintain a job repository that stores various metadata and statistics about all SCOPE jobs that get executed. This metadata is collected at different levels of granularity, including cluster, business unit[9], job, operator, and task levels. For the purposes of our experiments, we used the job repository to extract the following sets of workloads for one day's worth of SCOPE jobs (Table 1 shows the detailed description):

[8]Each job consists of tasks that get executed in the cluster in parallel. We define as *machine-hours* of a job, the sum of execution times of all its tasks.
[9]Our clusters are split in business units to be shared across different parts of the company for administrative purposes.

Table 1: **Distributed execution over large production workloads. Workload details are in the upper half of the table; results of our algorithm's execution in the lower half.**

| | Workload | Workload1 | Workload2 |
|---|---|---|---|
| Production SCOPE Workloads | Overlapping Users | 9 | 748 |
| | Overlapping Jobs | 4,315 | 42,182 |
| | Total Subexpr. | 293,467 | 1,812,854 |
| | Unique Subexpr. | 63,315 | 446,954 |
| | Unique Subexpr. Size | 19.6PB | 26.7PB |
| Subexpression Selection Algorithm | Compute Nodes | 500 | 500 |
| | Iterations | 10 | 10 |
| | Storage Cap | 1PB | 1PB |
| | Runtime | ~1 hr. | ~3 hrs. |
| | Selected Subexpr. | 62% | 44% |
| | Impacted Jobs | 90% | 79% |
| | Impacted Users | 100% | 94% |
| | Machine-time Savings | 10.4% | 42.3% |

- *Workload1* includes all jobs of a business unit for that day. It contains $4.3K$ overlapping jobs and $63K$ unique subexpressions.
- *Workload2* includes all jobs in one of our clusters for the same day. It contains $42K$ overlapping jobs (more than hundreds of thousands of total jobs) and $447K$ unique subexpressions.

**Candidate subexpression enumeration.** For each job, we traverse its logical query tree, as it is output by the SCOPE optimizer, and emit all its subtrees as candidate subexpressions. We filter out subexpressions that appear only once, since they do not present any computation reuse opportunity.

**Subexpression utility.** The utility of a subexpression is computed based on Eq. 2. Instead of relying on the optimizer's cost estimates, which can be significantly inaccurate (due to missing or inaccurate statistics, UDFs, error propagation, etc.), we rely on the observed machine-hours that are stored in our job repository during the previous execution of the job. Whenever such statistics are not available, we fall back to the optimizer's estimates.

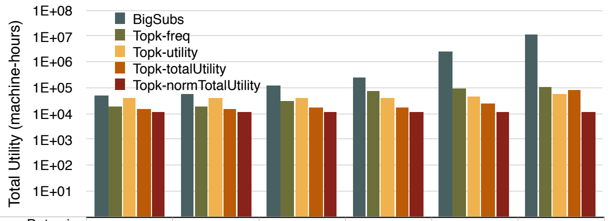
**Subexpression interactions.** Two subexpressions are interacting if the logical tree of the one is a subtree of the other (see Def. 4). Interactions are detected during subexpression enumeration and are stored in the interaction matrix $X$.

**Subexpression costs.** Similar to subexpression utility, we use the statistics stored in the job repository to collect accurate values for the cost (i.e., the storage footprint) of a subexpression.

**Evaluation methodology.** In evaluating our algorithm, note that it is not feasible to run our production workloads multiple times over the same data. Therefore, to quantify the machine-hour savings when a set of subexpressions is used to evaluate a query, we sum the utilities of these subexpressions for the given query, using information from the job repository and Eq. 2, as described above.

### 5.1.2 *Support for Large Production Workloads*

Table 1 shows the results of running the distributed SCOPE implementation of our subexpression selection algorithm (Listing 1) over the two production workloads. For materializing subexpressions, we budgeted a maximum storage capacity of $1PB$ while the total data processed by the jobs is in the orders of EB (exabytes). Despite using only a fraction of total capacity, we are able to materialize 62% and 44% of the subexpressions for *Workload1* and *Workload2*, respectively. These subexpressions are in turn able to impact 79–90% of the jobs and 90–100% of the users. The algorithm runs in less than an hour for *Workload1* and close to three hours for *Workload2*. These execution times are reasonable given that our subexpression selection is an offline process that is executed every few days or even weeks.



| Heuristic \ Data size | 10GB | 100GB | 1TB | 10TB | 100TB | 1PB |
|---|---|---|---|---|---|---|
| Topk-freq | 2.69 | 2.95 | 4.22 | 3.47 | 29.57 | 112.92 |
| Topk-utility | 1.30 | 1.43 | 3.19 | 6.46 | 60.72 | 207.95 |
| Topk-totalUtility | 3.38 | 3.72 | 7.52 | 15.30 | 108.50 | 144.38 |
| Topk-normTotalUtility | 4.42 | 4.86 | 10.88 | 21.46 | 221.51 | 939.43 |

**Figure 10: Total utility (machine-hour savings) on production *Workload1* using different selection methods and cost budgets. Since the graph shows utility (i.e., savings), higher is better. The table shows the improvement factor of BigSubs when compared to the other schemes (higher factor is better).**

Most importantly, our subexpression selection algorithm yields machine-hour savings of 10.4% and 42.3% for the two workloads, respectively. Such improvements translate to millions of dollars worth of operational costs being saved in Microsoft production clusters. Interestingly, the improvement when considering the jobs of the whole cluster is significantly higher compared to considering only a single business unit. This validates our observation at §1 that the shared nature of production clusters leads to significant redundant computations across users, who have no way of identifying such overlaps. It also highlights the importance of an algorithm that can support workloads that span the whole cluster's load, comprising several tens of thousands of jobs, compared to the 10s or 100s supported by traditional subexpression selection methods.

### 5.1.3 *Comparison with Heuristics-based Approaches*

A number of heuristics have been proposed in prior works for identifying interesting subexpressions. Nectar [18], for instance, selects subexpressions by looking at their total runtime to size ratio. To examine the quality of our BigSubs algorithm compared to such approaches in terms of total utility (measured in machine-hours) achieved, we consider the following top-k heuristics:

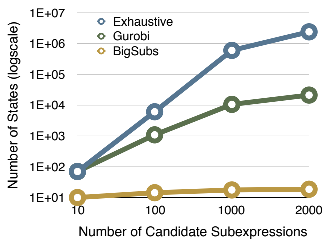
**(1) Topk-freq.** Rank subexpressions by the number of times they appear in different queries.

**(2) Topk-utility.** Compute the utility of each subexpression for each query, and rank subexpressions by their maximum utility for a single query.

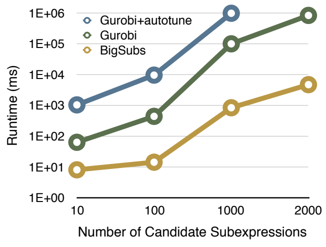
**(3) Topk-totalUtility.** Similar to Topk-utility, but we compute for each subexpression the sum of utilities it achieves for all queries.

**(4) Topk-normTotalUtility.** Rank subexpressions by their total utility divided by their storage footprint in bytes. This is similar to the approach followed by Nectar [18].

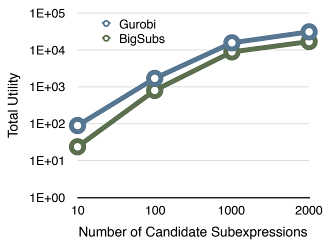
For each of these heuristics, we select subexpressions in their ranked order until we exceed the storage budget. Fig. 10 shows the total utility (machine-hours) of BigSubs compared to the heuristics on *Workload1* with varying storage budget, i.e., for different cost-benefit trade-offs. In general, by taking a global view of the workload and the subexpression interactions, BigSubs performs uniformly better across the range of storage budgets. In particular, it results in 30% to 400% better utility for a $10GB$ budget. The improvement increases to more than two orders of magnitude for larger storage budgets, with a maximum of 940× over *Topk-normTotalUtility* for a $1PB$ budget.



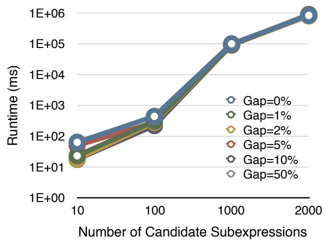
(a) Number of States.



(b) Algorithm runtime.



(c) Total utility.



(d) Relaxed optimal.

**Figure 11: Comparison of BIGSUBS with variants of the OPTIMAL algorithm (using the Gurobi optimizer) on a single machine.**

**Table 2: Parameters of our workload generator.**

| Parameter | Meaning | Default |
|---|---|---|
| NumberOfSubexpr | Number of candidate subexpr. | 25 |
| SubexprToJobRatio | Average number of candidate subexpr. per job | 0.6 |
| SubexprToCapRatio | Cost constraint as a proportion of #subexpr. | 0.8 |
| RelevantPerJob | Number of possible subexpr. rewrites per job | 10 |
| InteractionProb | Chance of interaction between two subexpr. | 0.2 |
| Iterations | #iterations in bipartite graph labeling | 20 |
| InitZjProb | Probability to initialize subexpr. vertices to 1 | 0 |
| AdaptiveThreshold | Switch point between bottom-up and top-down | 0.2 |
| StricterFlipping | % of iterations before stricter flipping (§3.2.2) | 80% |

None of the heuristics manages to successfully capture the constraints of subexpression selection. By comparing *Topk-freq* with *Topk-utility*, we observe a trade-off between subexpression frequency and utility: leaf-level subexpressions are the most frequent but have smaller utility; larger subexpressions have higher utility but smaller frequency. For smaller budgets, it pays off to select complex subexpressions of higher utility (*Topk-utility*) that are often smaller in size. With sufficiently large budgets though, ranking based on frequency (*Topk-freq*) gives better results. On the other hand, *Topk-totalUtility* and *Topk-normTotalUtility* combine subexpression utility and frequency across queries, but, interestingly, they perform worse than the two simpler heuristics. By selecting high-utility subexpressions that appear in multiple queries, but without taking into account interactions, they also select closely-related subexpressions appearing in the same queries, which are redundant and thus do not contribute to the total utility. Therefore, attempting to make decisions across multiple queries without accounting for interactions can be detrimental to the solution quality.

## 5.2 Detailed Analysis

To better understand the behavior of BIGSUBS and to fairly compare it with the ILP formulation of §2.2 (termed OPTIMAL hereafter) that is not parallelizable, we implemented a version of BIGSUBS in $C\#$ that runs the same iterative Algorithm 1 on a single machine. We use this version to drive a set of micro-experiments and compare BIGSUBS with OPTIMAL (§5.2.1), study BIGSUBS' scalability (§5.2.2) and convergence (§5.2.3), and assess the effectiveness of our pruning optimizations (§5.2.4).

For the purpose of these experiments, we built a synthetic workload generator, whose parameters and their default values are summarized in Table 2. Given the desired number of subexpressions (*NumberOfSubexpr*), the generator determines the number of queries using the *SubexprToJobRatio*. It randomly sets the cost of each subexpression to a value from 1 to 10. Then it generates the utility matrix $U$ (see §2.1) by setting each element $u_{ij}$ to a random value from 1 to 10 with probability *RelevantPerJob*/$\#jobs$; otherwise the entry is 0. Moreover, it generates the subexpression interaction matrix $X$ by setting each element $x_{ij}$ to 1 with probability *InteractionProb*.[10]

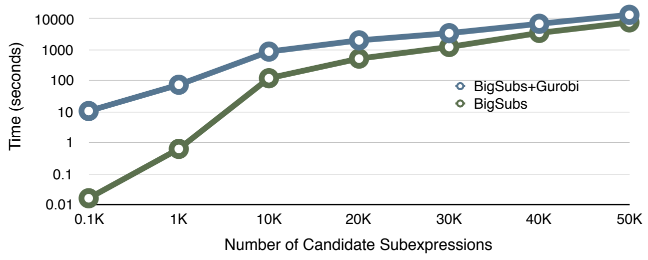


**Figure 12: Scalability of BIGSUBS on a single machine.**

We use a desktop machine running Windows 10 Enterprise Edition, having 32 GB of memory and two 2.4 GHz processors with a total of 12 cores and 24 hardware threads. Unless otherwise specified, we report the average over three runs in our experiments.

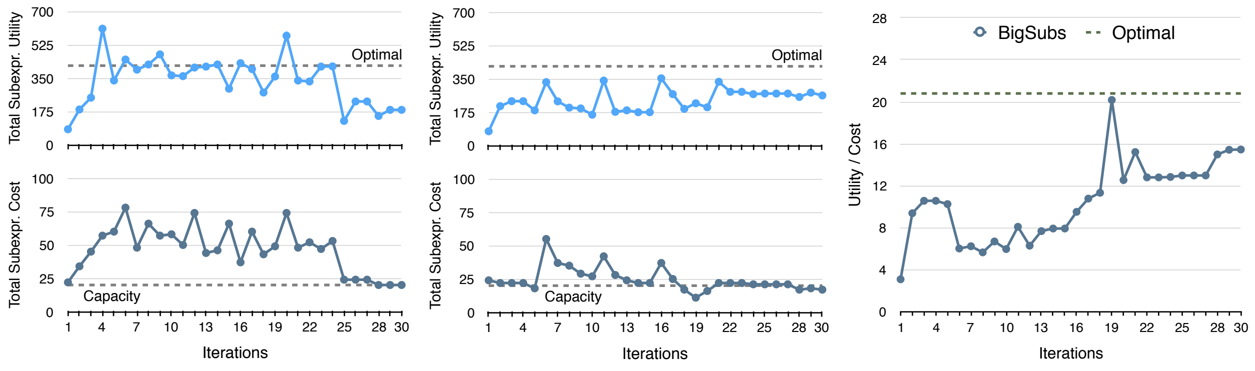
### 5.2.1 *Comparison with Optimal*

To compare with the OPTIMAL, we implemented the global ILP formulation of §2.2 using Gurobi (version 7.5.2), one of the most widely-used commercial optimizers [2]. Fig. 11 reports our findings for workloads of up to 2000 subexpressions. Beyond that size, Gurobi ran out of memory on our desktop with 32 GB of memory.

Fig. 11(a) illustrates the number of possible solutions (states) considered by BIGSUBS and Gurobi, as well as the states that an exhaustive approach would consider (computed by the formula $2^{mn}$). Gurobi performs a presolve phase that prunes a large number of states, when compared to exhaustive. However, given that it solves a global ILP problem, it considers up to three orders of magnitude more states than BIGSUBS. On the other hand, BIGSUBS, by solving smaller ILP problems and applying domain-specific optimizations, avoids blowing up the search space. Similarly, Fig. 11(b) shows that the actual runtime of Gurobi is two orders of magnitude longer than that of BIGSUBS. For example, for 2000 subexpressions, Gurobi takes 14 minutes, while BIGSUBS takes just 4.7 seconds. We also experimented with Gurobi's utility that automatically tunes the parameter values for a given input ("Gurobi+autotune"), but the resulting end-to-end time of parameter tuning and optimization ended up being even longer. Note, however, that the utility of Gurobi's output (Fig. 11(c)) is, as expected, higher than BIGSUBS' (by up to $2\times$), so for small input sizes (up to 2000 subexpressions, which is not practical in our setting), Gurobi could be used instead.

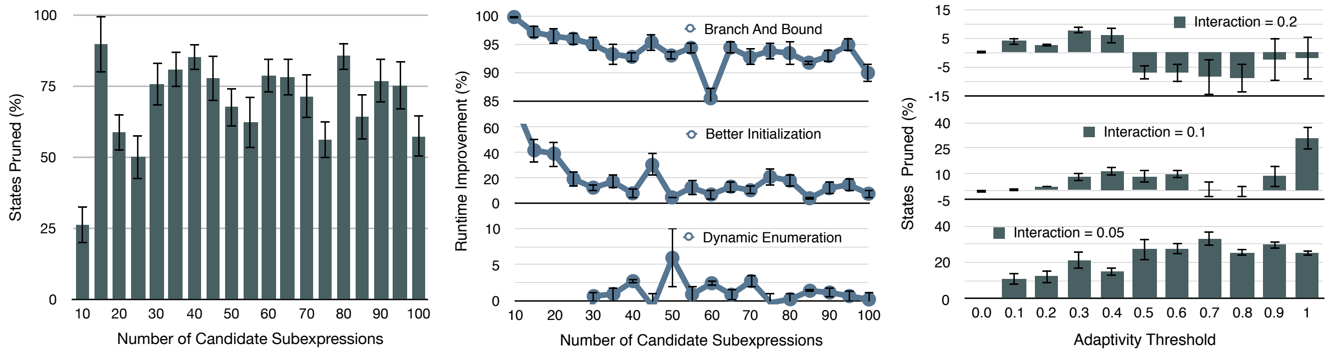
Finally, we tried to relax Gurobi's optimality by increasing the optimality "gap tolerance" from 0% (optimal) to 50%, which allows the solver to terminate before it reaches the optimal solution. As Fig. 11(d) shows, this did not significantly improve Gurobi's

[10]Note that our generator purposely does not specify the direction of interaction between two subexpressions (i.e., containment), as this is not required in our formulation of §2.2. For interacting subexpressions $s_1$ and $s_2$, the workload is valid both when the utility of $s_1$ is higher than that of $s_2$ ($s_1$ contains $s_2$) and when it is lower ($s_1$ is contained in $s_2$).



(a) Flipping frequency=1. (b) Flipping frequency=5. (c) Utility/cost ratio.

**Figure 13: Convergence of BIGSUBS.**



(a) Effectiveness of brand-and-bound exploration technique. (b) Improvement in runtime by successively adding pruning optimizations. (c) Adaptive enumeration in BIGSUBS.

**Figure 14: Pruning effectiveness of BIGSUBS.**

runtimes for our problem, although we observed that it did compromise the achieved total utility.

### 5.2.2 *Scalability*

To assess BIGSUBS' scalability, in Fig. 12 we report our algorithm's runtime for workloads with an increasing number of subexpressions. For the largest workload that we tried (with 50,000 subexpressions), our algorithm is tractable with 128 minutes of runtime on a single desktop machine.

As an alternative approach, we tried using Gurobi to solve the local ILP problems at each of the job vertices, instead of our branch-and-bound algorithm. Interestingly, as Fig. 12 shows, this variant ("BIGSUBS+Gurobi") comes quite close to standard BIGSUBS for large workloads. However, it is still 75% slower than standard BIGSUBS for 50,000 subexpressions, which shows the importance of our local optimizations, as we further discuss in §5.2.4. Observe that this variant returns solutions of the same total utility as BIGSUBS, as they both find the optimal solution to each local ILP.

### 5.2.3 *Convergence*

Recall that we expect BIGSUBS to converge to a solution within the cost (capacity) bound. To evaluate its convergence, we use a workload with 25 subexpressions and the default parameters of Table 2. As described in §3.2.2, we apply stricter flipping in the last 20% of the iterations, which was empirically chosen as the default setting. Fig. 13(a) shows that the total cost converges to the cost budget and correspondingly the total utility reaches a fixed value. The "Optimal" line in the figure is computed by solving the global ILP with Gurobi. To further observe the impact of stricter flipping, we consider a variant of our algorithm that always applies stricter flipping with the exception of one in every five iterations (we use a frequency of five, as this gave us best results in practice). Fig. 13(b) shows the utility and cost achieved. Thanks to the stricter flipping, this variant avoids changing vertex labels too often, the total cost remains closer to the cost budget, and the achieved total utility is higher. For this variant with stricter flipping, Fig. 13(c) depicts the ratio of utility with cost, indicating that our algorithm finds a better utility per cost unit as the number of iterations increases. Note that higher frequencies do not lead to better utility, since the exploration tends to get stuck to local optima.

### 5.2.4 *Pruning Effectiveness*

Finally, we assess the effectiveness of our pruning optimizations (discussed in §4) over the non-optimized version of BIGSUBS.

Fig. 14(a) shows the reduction in states achieved by the branch-and-bound enumeration (see §4.1) compared to the non-optimized version by detecting interactions between subexpressions fast. Although the achieved reduction varies depending on the exact workload that was generated each time, the reduction in states is always significant, ranging from 26% to 90% of the states.[11]

Fig. 14(b) shows the actual improvement in runtime by successively enabling the pruning techniques of §4. Branch-and-bound reduces the runtime by over 90%, better initialization further reduces the runtime by up to 15%, and dynamic enumeration achieves an additional reduction of up to 10%. Observe that dynamic enumeration's improvement might be slightly negative,

[11]For each workload we report the average and standard error (error bars) over five runs. Some variation across runs for the same workload is to be expected given the probabilistic nature of our approach.

when it mistakenly picks top-down enumeration, although the default bottom-up was the better choice. In practice, we observed only a few such cases. We ran more experiments (not shown here) and observed similar pruning effectiveness while varying other parameters in our workload generator, e.g., subexpressions per job, subexpression interaction, cost budget, initialization probability.

Fig. 14(c) shows the additional pruning yielded by adaptive enumeration (see §4.2), i.e., by dynamically picking the enumeration direction at each branching point of Algorithm 2, based on a threshold. Fig. 14(c) shows results for varying adaptivity thresholds and subexpression interaction probabilities. Adaptive enumeration is more effective for less interacting workloads, with a state reduction of up to 32%. In these cases, higher thresholds that avoid changing direction too often tend to work better. For more interacting workloads, as discussed in §4.2, the default bottom-up enumeration is expected to work better, and thus adaptive enumeration is less effective and should be avoided or used after careful tuning.

## 6. RELATED WORK

**View selection.** View selection is closely related with the problem of subexpression selection that we consider in this work in that it picks a set of views to materialize for a given set of queries, in order to minimize a cost function (e.g., query evaluation, view maintenance) under some constraints (e.g., space budget) [35]. Several early techniques were proposed in the context of data warehouses, including AND/OR graphs [20, 19], modeling the problem as a state optimization [46], and lattices to represent data cube operations [22]. Kodiak [33] recently used materialized views to improve query latency in analytics clusters. All these efforts target workloads with only a few tens of queries.

**Subexpression selection.** Views are more generic than subexpressions, as they can consider computation that does not appear in the logical query plan. This increases the space of possible solutions, and complicates query containment and answering queries using views [21]. Since scalability is a major concern in our setting, in the current version of BIGSUBS, we focus only on subexpressions. Subexpression selection has also been considered in SQL Server [53]. Other related works have looked at common subexpressions within the same job script [44]. However, none of these approaches has considered workloads of size similar to ours.

**Multi-query optimization (MQO).** MQO [43] is similar to view selection with the difference that views are typically only transiently materialized for the execution of a given query set. Heuristic approaches have incorporated MQO in a Volcano-style optimizer, using AND/OR DAGs, but without considering a space budget [42]. A recent work devised an approximation algorithm that runs in time quadratic to the number of common subexpressions and provides theoretical guarantees on the quality of the solution obtained [30]. Other approximate solutions have used genetic algorithms by treating a vector of query plans as chromosomes [6]. Such approximation schemes could potentially be leveraged by BIGSUBS, but necessary modifications have to be made to ensure scalability and support for our problem's constraints. However, unlike our approach, these schemes are not easily parallelizable.

Multi-query optimization has also been studied in the context of map-reduce [38, 49]. These works consider sharing opportunities only for map and reduce operators, thus their applicability is limited. PigReuse [8] addresses MQO for Pig scripts, using an AND/OR graph representation and an ILP-based solution. All these works consider workloads with a few tens of queries.

**Reusing intermediate results.** Researchers have further looked at recycling intermediate results that already get materialized [23, 37]. This problem has received particular attention in big data platforms, since (i) there is a lot of recurring computation, (ii) optimization time is relatively short compared to the job execution time, (iii) performance and resource benefits can be significant. ReStore [12] considers the caching of map-reduce job outputs, given a space budget. Others have studied history-aware query optimization with materialized intermediate views [39] and fair cache allocation among multiple cloud tenants [31]. Nectar [18] caches all sub-computations (intermediate results) based on a cache insertion policy that incorporates cache lookup frequency and previous runtimes for computing an intermediate result. In contrast to subexpressions, intermediate results are typically the output from the end of a pipelined stage, which consists of multiple operators.

**Shared workload optimization.** Many works have considered building a common query plan for a set of queries to share operators [54, 40] A global optimization approach to find the best overall shared plan is presented in [15]. Shared optimization has been suggested in the context of publish/subscribe systems too [10, 9, 11]. Although these systems also deal with a large number of queries, the considered queries tend to be simple and belong to a few unique templates (e.g., queries that differ only in their constants). Delta [29] deals with more complex queries, but focuses on creating a network of views used to evaluate other views. Finally, scan sharing has been explored in big data systems, such as MapReduce [38], Hive [47] and Pig [50]. Unlike such approaches, we opted to keep each job separate: operator sharing in pay-as-you-go job services makes billing and accounting tedious, while it introduces artificial dependencies between jobs, which become even worse in the case of failures.

**Large-scale graph analytics.** The vertex-centric programming model has become popular for performing analytics over large graphs in parallel [34, 1, 17]. Although the implementation of our graph labeling algorithm is based on our internal SQL-like SCOPE engine, our ideas are applicable to any of the above systems. We expect the runtime of our algorithm to be even better on these systems, given their native support for iterative graph computations.

Finally, our approach draws ideas from existing graph algorithms, and in particular from Spinner [36] for the probabilistic flipping part of our algorithm.

## 7. CONCLUSION

Large organizations depend on shared analytics clusters to process and analyze vast amounts of data every day. The shared nature of these clusters leads to substantial computation overlaps across users and their jobs, which in turn wastes computational resources. Existing methods for identifying common computations either do not scale to the massive workloads that are typical in these clusters or make local optimization decisions that lead to limited savings. In this paper, we introduced a novel algorithm, called BIGSUBS, to select subexpressions to materialize over very large workloads. Our key idea is to split the overall problem into two subproblems, which we solve separately in an iterative manner. We further showed several optimizations to significantly prune the solution space we explore. We then described a distributed implementation of BIGSUBS using SCOPE. Our evaluation results on production workloads show that BIGSUBS: (i) supports workloads with tens of thousands of jobs, (ii) identifies subexpression that can result in potential machine-hour savings of up to 40% in our clusters, and (iii) outperforms heuristics-based approaches by three orders of magnitude. BIGSUBS is part of our CLOUDVIEWS system for subexpression reuse, which we are currently integrating with the SCOPE runtime in our production clusters.

## 8. REFERENCES

[1] Apache Giraph Project. http://giraph.apache.org/, 2018.

[2] Gurobi optimizer. http://www.gurobi.com, 2018.

[3] Presto. http://prestodb.io, 2018.

[4] S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated selection of materialized views and indexes in SQL databases. In *VLDB*, pages 496–505, 2000.

[5] Amazon Athena. https://aws.amazon.com/athena/.

[6] M. A. Bayir, I. H. Toroslu, and A. Cosar. Genetic Algorithm for the Multiple-Query Optimization Problem. *IEEE Trans. Systems, Man, and Cybernetics, Part C*, 37(1):147–153, 2007.

[7] N. Bruno, S. Agarwal, S. Kandula, B. Shi, M.-C. Wu, and J. Zhou. Recurring job optimization in SCOPE. In *SIGMOD*, pages 805–806, 2012.

[8] J. Camacho-Rodríguez, D. Colazzo, M. Herschel, I. Manolescu, and S. R. Chowdhury. Reuse-based optimization for pig latin. In *CIKM*, pages 2215–2220, 2016.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, F. Reiss, and M. A. Shah. TelegraphCQ: Continuous dataflow processing. In *SIGMOD*, page 668, 2003.

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, 2000.

[11] Y. Diao, P. M. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.

[12] I. Elghandour and A. Aboulnaga. Restore: Reusing results of mapreduce jobs. *PVLDB*, 5(6):586–597, 2012.

[13] J. Fan, A. G. S. Raj, and J. M. Patel. The case against specialized graph analytics engines. In *CIDR*, 2015.

[14] A. D. Ferguson, P. Bodik, S. Kandula, E. Boutin, and R. Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *EuroSys*, pages 99–112, 2012.

[15] G. Giannikis, D. Makreshanski, G. Alonso, and D. Kossmann. Shared workload optimization. *PVLDB*, 7(6):429–440, 2014.

[16] F. Goasdoué, K. Karanasos, J. Leblay, and I. Manolescu. View selection in semantic web databases. *PVLDB*, 5(2):97–108, 2011.

[17] GraphLab Create. https://turi.com/.

[18] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In *OSDI*, pages 75–88, 2010.

[19] H. Gupta. Selection of Views to Materialize in a Data Warehouse. In *ICDT*, pages 98–112, 1997.

[20] H. Gupta and I. S. Mumick. Selection of views to materialize in a data warehouse. *IEEE Trans. Knowl. Data Eng.*, 17(1):24–43, 2005.

[21] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[22] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *ACM SIGMOD*, pages 205–216, 1996.

[23] M. Ivanova, M. L. Kersten, N. J. Nes, and R. Goncalves. An architecture for recycling intermediates in a column-store. In *SIGMOD*, pages 309–320, 2009.

[24] A. Jindal and S. Madden. GraphiQL: A Graph Intuitive Query Language for Relational Databases. In *IEEE BigData*, pages 441–450, 2014.

[25] A. Jindal, S. Madden, M. Castellanos, and M. Hsu. Graph Analytics using Vertica Relational Database. In *IEEE BigData*, pages 1191–1200, 2015.

[26] A. Jindal, S. Qiao, H. Patel, Z. Yin, J. Di, M. Bag, M. Friedman, Y. Lin, K. Karanasos, and S. Rao. Computation reuse in analytics job service at Microsoft. In *SIGMOD*, 2018.

[27] A. Jindal, P. Rawlani, E. Wu, S. Madden, A. Deshpande, and M. Stonebraker. VERTEXICA: your relational friend for graph analytics! *PVLDB*, 7(13):1669–1672, 2014.

[28] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. n. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated SLOs for enterprise clusters. In *OSDI*, pages 117–134, 2016.

[29] K. Karanasos, A. Katsifodimos, and I. Manolescu. Delta: Scalable data dissemination under capacity constraints. *PVLDB*, 7(4):217–228, 2013.

[30] T. Kathuria and S. Sudarshan. Efficient and Provable Multi-Query Optimization. In *PODS*, pages 53–67, 2017.

[31] M. Kunjir, B. Fain, K. Munagala, and S. Babu. ROBUS: Fair Cache Allocation for Data-parallel Workloads. In *SIGMOD*, pages 219–234, 2017.

[32] J. LeFevre, J. Sankaranarayanan, H. Hacigümüs, J. Tatemura, N. Polyzotis, and M. J. Carey. Opportunistic physical design for big data analytics. In *SIGMOD*, pages 851–862, 2014.

[33] S. Liu, B. Song, S. Gangam, L. Lo, and K. Elmeleegy. Kodiak: Leveraging materialized views for very low-latency analytics over high-dimensional web-scale data. *PVLDB*, 9(13):1269–1280, 2016.

[34] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: A System for Large-scale Graph Processing. In *SIGMOD*, pages 135–146, 2010.

[35] I. Mami and Z. Bellahsene. A survey of view selection methods. *SIGMOD Record*, 41(1):20–29, 2012.

[36] C. Martella, D. Logothetis, A. Loukas, and G. Siganos. Spinner: Scalable Graph Partitioning in the Cloud. In *ICDE*, pages 1083–1094, 2017.

[37] F. Nagel, P. A. Boncz, and S. Viglas. Recycling in pipelined query evaluation. In *ICDE*, pages 338–349, 2013.

[38] T. Nykiel, M. Potamias, C. Mishra, G. Kollios, and N. Koudas. MRShare: Sharing across multiple queries in MapReduce. *PVLDB*, 3(1):494–505, 2010.

[39] L. L. Perez and C. M. Jermaine. History-aware query optimization with materialized intermediate views. In *ICDE*, pages 520–531, 2014.

[40] L. Qiao, V. Raman, F. Reiss, P. J. Haas, and G. M. Lohman. Main-memory scan sharing for multi-core cpus. *PVLDB*, 1(1):610–621, 2008.

[41] R. Ramakrishnan, B. Sridharan, J. R. Douceur, P. Kasturi, B. Krishnamachari-Sampath, K. Krishnamoorthy, P. Li, M. Manu, S. Michaylov, R. Ramos, N. Sharman, Z. Xu, Y. Barakat, C. Douglas, R. Draves, S. S. Naidu, S. Shastry, A. Sikaria, S. Sun, and R. Venkatesan. Azure data lake store: A hyperscale distributed file service for big data analytics. In *SIGMOD*, pages 51–63, 2017.

[42] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD*, pages 249–260, 2000.

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

[44] Y. N. Silva, P. Larson, and J. Zhou. Exploiting common subexpressions for cloud query processing. In *ICDE*, pages 1337–1348, 2012.

[45] The Apache Software Foundation. Powered by Apache Hadoop. http://wiki.apache.org/hadoop/PoweredBy, 2018.

[46] D. Theodoratos and T. K. Sellis. Data warehouse configuration. In *VLDB*, pages 126–135, 1997.

[47] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. Hive: a warehousing solution over a map-reduce framework. *PVLDB*, 2(2), 2009.

[48] A. Verma, L. Pedrosa, M. R. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *EuroSys*, pages 18:1–18:17, 2015.

[49] G. Wang and C. Chan. Multi-query optimization in MapReduce framework. *PVLDB*, 7(3):145–156, 2013.

[50] X. Wang, C. Olston, A. D. Sarma, and R. Burns. CoScan: Cooperative Scan Sharing in the Cloud. In *SOCC*, pages 11:1–11:12, 2011.

[51] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *OSDI*, 2008.

[52] J. Zhou, N. Bruno, M.-C. Wu, P.-Å. Larson, R. Chaiken, and D. Shakib. SCOPE: parallel databases meet MapReduce. *VLDB J.*, 21(5):611–636, 2012.

[53] J. Zhou, P. Larson, J. C. Freytag, and W. Lehner. Efficient exploitation of similar subexpressions for query processing. In *ACM SIGMOD*, pages 533–544, 2007.

[54] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative scans: Dynamic bandwidth sharing in a DBMS. In *VLDB*, pages 723–734, 2007.