Optimization of Continuous Queries with Shared Expensive Filters [*]

Kamesh Munagala
Duke University
kamesh@cs.duke.edu

Utkarsh Srivastava
Stanford University
usriv@cs.stanford.edu

Jennifer Widom
Stanford University
widom@cs.stanford.edu

ABSTRACT

We consider the problem of optimizing and executing multiple continuous queries, where each query is a conjunction of filters and each filter may occur in multiple queries. When filters are expensive, significant performance gains are achieved by sharing filter evaluations across queries. A shared execution strategy in our scenario can either be *fixed*, in which filters are evaluated in the same predetermined order for all input, or *adaptive*, in which the next filter to be evaluated is chosen at runtime based on the results of the filters evaluated so far. We show that as filter costs increase, the best adaptive strategy is superior to any fixed strategy, despite the overhead of adaptivity. We show that it is NP-hard to find the optimal adaptive strategy, even if we are willing to approximate within any factor smaller than logarithmic in the number of queries. We present a greedy adaptive execution strategy and show that it approximates the best adaptive strategy to within a factor polylogarithmic in the number of queries and filters. We also show how the execution overhead of adaptive strategies can be reduced by appropriate precomputation. Finally, we present a thorough experimental evaluation demonstrating the effectiveness of our techniques.

1. INTRODUCTION

We consider the problem of optimizing a collection of *continuous queries* [3], where each query is a conjunction of filters on the incoming data stream. We focus on scenarios that exhibit *sharing*, meaning the same filter may occur in multiple queries. The goal of our optimization is to minimize the overall cost of evaluating the filters by sharing filter evaluations across multiple queries. Note that our problem is different from that addressed in *publish-subscribe* systems [21, 23], which generally focus on special techniques for indexing a large number of queries (or subscriptions) to quickly identify which subscriptions match the incoming data. The differences are discussed in more detail in Section 1.1.

As an example of our scenario, suppose several information analysts are all monitoring an incoming stream S of images. Each analyst identifies the images of interest to him by specifying a collection of filters $F_1, \dots, F_k$ that the image must satisfy. Thus, each analyst poses a continuous query of the form:

SELECT * FROM S WHERE $F_1 \wedge \dots \wedge F_k$

In this scenario, filters will most likely be shared, as more than one analyst may be looking for certain characteristics. For example, one analyst might be interested in outdoor images (filter F_1) with at least five people in it (filter F_2), while another analyst might be interested in outdoor images (filter F_1 again) with at least one person clad in black (filter F_3).

Filters that detect patterns within an image are often expensive to evaluate, thereby motivating the need to share filter processing across queries to increase throughput. (For example, the filter in the OpenCV library [19] to detect number of faces, when running on a 1.8 GHz machine, takes an average of 0.5 seconds on an 800×600 image.) Numerous other applications such as network monitoring, video surveillance, monitoring of voice calls, and intrusion detection, also exhibit shared expensive predicates and have high throughput requirements (e.g., a video feed needs to be processed in real time).

A naïve execution strategy for such scenarios is to evaluate each query independently on the incoming stream. However, since filters are shared, this approach can perform a significant amount of redundant work. A better alternative is to choose a *shared execution strategy* for a given collection of queries. In a shared strategy, filters are evaluated on each data item in some order dictated by the strategy, until all of the queries are resolved. (Since each query is a conjunction of filters, a query is resolved as soon as one of the query filters evaluates to false, or when all of the query filters evaluate to true.) In this way, filter evaluations are shared across all queries. In this paper, we address the problem of finding the optimal shared execution strategy for any given collection of queries that share expensive filters.

Finding the optimal shared execution strategy poses the following major challenges:

1. **Filter placement**. The decision whether a filter should be evaluated earlier or later in a shared strategy should be made by taking all of the following factors into account:
   - **Cost:** Filters with low cost should preferably be evaluated early, since they might resolve queries at lower cost.
   - **Selectivity:** The average fraction of incoming data items that satisfy a filter is referred to as the *selectivity* of that filter. Filters with lower selectivity should preferably be evaluated early, since they are more likely to resolve queries by evaluating to false.
   - **Participation:** The number of queries that contain a given filter is referred to as the *participation* of that filter. Filters with higher participation should preferably be evaluated early, since they can decide the results of a larger number of queries.

   For a given filter, these three factors may give contradictory suggestions for its placement, so we must devise a placement method that takes all the factors into account.

[*]This work was supported by the National Science Foundation under grant IIS-0324431, by a Stanford Graduate Fellowship from Sequoia Capital, and by a fellowship from Microsoft Corporation.

2. **Execution Overhead**. Executing a shared strategy incurs some amount of overhead, e.g., keeping track of which filters have been evaluated, and which queries have been resolved. This overhead is in addition to the cost of the filters evaluated by the strategy. Thus the overall choice of the optimal strategy must take into account the expected total cost of filters evaluated by a strategy as well as its execution overhead.

To address the above challenges, we first consider the space of possible shared execution strategies and outline two broad categories of strategies: *fixed* and *adaptive*. In a fixed strategy, the order in which filters are evaluated is predetermined and is the same for each data item on the stream. In an adaptive strategy, at any stage the next filter to be evaluated is chosen based on the results of the filters evaluated so far. Adaptive strategies (sometimes also called *conditional plans* [10]) have a higher execution overhead since at each step they incur the cost of choosing the next filter to evaluate. However, in terms of the expected total cost of filters evaluated, the best adaptive strategy is often superior to any fixed strategy. Specifically, we show problem instances where any fixed strategy has a cost $\Omega(\mu)$ times the cost of the best adaptive strategy, where μ is the maximum number of queries in which a filter is present (measuring the *extent of sharing* of filters across queries). Thus, as the extent of sharing and filter costs increase, the higher execution overhead of adaptive strategies is compensated for by the savings obtained in filter evaluation cost, thus motivating the need to design and analyze adaptive strategies.

We then consider the optimization problem of finding the least-cost adaptive strategy, where the cost of a strategy is the expected total cost of filters evaluated by it. A key idea behind our results is to model this optimization problem as a probabilistic version of the well-known *set cover* problem [12]. Based on the similarity to set cover, we first show a lower bound: it is NP-hard to find an adaptive strategy whose cost approximates the optimal cost to within any factor smaller than $\ln m$, where m is the number of queries. We then give a greedy adaptive strategy, and we show that its cost approximates the optimal cost to within a factor $O(\log^2 m \log n)$, where m is the number of queries as before, and n is the number of filters. Our experiments indicate that the greedy strategy performs very well in practice.

Finally, we consider the problem of reducing the execution overhead for adaptive strategies. One method is to precompute and store which filter is to be evaluated next for each possible combination of outcomes of the filters evaluated so far. (Essentially, we would be materializing the "decision tree" corresponding to the adaptive strategy.) However, in general, this approach requires space exponential in the number of filters. We give an algorithm that takes into account the amount of space available to store the decision tree and decides which parts of the decision tree should be precomputed so that the execution overhead of the strategy is minimized.

1.1 Related Work

There has been work pertaining to shared query execution in both the data streams context [3] as well as the classical DBMS context. For data streams, most of the work on shared query execution is for *publish-subscribe* systems, e.g., [7, 21, 23]. In publish-subscribe systems, the problem setting is essentially the same as ours: Each user specifies (or subscribes to) the data of interest by specifying conditions on the incoming data stream, and the goal is to dispatch each incoming data item to all matching subscriptions. Publish-subscribe systems generally focus on scenarios where the number of subscriptions is very large (of the order of millions), and they use special disk-based indexing mechanisms to efficiently locate the matching subscriptions for a given data item. Unlike our work, their focus is not on deciding the order of filter evaluation, since filters are considered cheap. Another important distinction is that publish-subscribe systems often exploit the internal structure of filters to achieve sharing [13]. Such techniques to exploit sharing do not extend to the "black-box" user-defined filters we consider, or to other general filters such as those implemented by a table lookup or a semijoin. In the continuous query context, exploiting sharing for continuous sliding-window aggregates, but not for expensive filters, is considered in [1]. Expensive filters are considered in [4], but only for a single query.

In conventional relational DBMSs, shared query execution has been considered under *multi-query optimization* [8]. This work focusses mostly on exploiting common join subexpressions in queries, and does not consider expensive filters. The techniques developed in this paper can also be applied in the classical relational setting for optimizing multiple queries with expensive filters. Optimization of a single query with expensive filters has been considered in [6, 15]. In the context of active databases, the Ariel system [14] has been designed for efficiently evaluating a collection of rules or triggers using *discrimination networks*. However, just as in publish-subscribe systems, sharing is achieved by exploiting the internal structure of the trigger conditions. Furthermore, optimization in Ariel is mainly through randomization and hill-climbing with no provable guarantees of optimality.

Note that our notion of an adaptive execution strategy is not related to adaptive plans in *Eddies* [2], or even Eddies with content-based routing [5]. In Eddies, different plans may be chosen for different tuples in order to adapt to changes in operator costs and selectivities over time. In our case, we assume that operator costs and selectivities do not change over time. Rather, different plans may be chosen for different tuples since the next operator (filter) to be evaluated is based on the results of the filters evaluated so far. There has been work on sharing in the Eddies context [16], but this work does not provide provable performance guarantees.

We note that our problem is similar in spirit to [11], where adaptive querying of information sources is considered, each of which answers a subset of queries with some probability, time delay and cost. The significant difference with our work is that they assume each information source (predicate in our case) satisfies each query with some probability which is independent of its satisfying another query; in our setting, the filter satisfies all queries it belongs to with the same probability in a perfectly correlated fashion. This makes our problem harder to approximate and we need different solution strategies and analysis techniques.

As far as we are aware, we are the first to consider the shared queries problem presented in this paper.

1.2 Summary of Contributions

- We formally define the problem of optimizing a collection of queries with conjunctions of shared filters (Section 2).
- We explore the space of shared execution strategies and show that when filters are expensive, the optimal strategy is adaptive (Section 3).
- We show the hardness of finding (even approximating) the optimal adaptive strategy, and we give a greedy adaptive strategy that approximates the optimal strategy to within a factor $O(\log^2 m \log n)$, where m is the number of queries and n is the number of filters (Section 4).
- We show how the execution overhead of an adaptive strategy can be reduced by appropriate precomputation (Section 5).

```
Algorithm ExecuteStrategy($\mathcal{P}$)
$\mathcal{P}$: Shared execution strategy for the set of queries $\mathcal{Q}$
 1. for each data item $s$ on stream $S$
 2.     for each query $Q \in \mathcal{Q}$          /* Initialization */
 3.         status($Q$) $\leftarrow$ unresolved, numFiltersLeft($Q$) $\leftarrow |Q|$
 4.     while (status($Q$) $=$ unresolved for some $Q$)
 5.         choose next filter $F$ to be evaluated according to $\mathcal{P}$
 6.         evaluate $F$ on $s$
 7.         if ($F$ evaluates to false)
 8.             for each $Q$ where $F \in Q$ status($Q$) $\leftarrow$ false
 9.         else
10.             for each $Q$ where $F \in Q$ numFiltersLeft($Q$)--
11.             if (numFiltersLeft($Q$) $= 0$) then status($Q$) $\leftarrow$ true
```

Figure 1: Execution Algorithm for a Shared Execution Strategy

- We give a thorough experimental evaluation showing that our techniques lead to a significant improvement in performance over more naïve techniques (Section 6).

2. PRELIMINARIES

Consider an incoming stream S of data items. Let there be a set of m queries $\mathcal{Q} = \{Q_1, \ldots, Q_m\}$ posed on stream S. Each query Q_i is a conjunction of filters on the items of S:

$$Q_i : \text{ SELECT * FROM } S \text{ WHERE } F_i^1 \wedge \ldots \wedge F_i^k \qquad (1)$$

For the rest of the paper, we denote the query Q_i as the set of filters $\{F_i^1, \ldots, F_i^k\}$, omitting the implicit conjunction. Thus, $F \in Q$ denotes that filter F occurs in query Q. Filters may be shared among queries. For example, $F_i^a = F_j^b$ denotes that the ath filter in query Q_i is the same as the bth filter in query Q_j. Let there be n distinct filters over all the queries in $\mathcal{Q}$, denoted by the set $\mathcal{F} = \{F_1, \ldots, F_n\}$. For a given collection of queries and filters, we also define the following variables:

μ	maximum number of queries a filter is present in
κ	maximum number of filters present in a query
λ	number of (F_i, Q_j) pairs where F_i is present in Q_j

Table 1: Variables in a Given Problem Instance

Note that μ measures the extent of sharing of filters between queries. Also note that $\lambda \leq \min(m\kappa, n\mu)$.

A shared execution strategy $\mathcal{P}$ for the queries in $\mathcal{Q}$ gives an order in which the filters in $\mathcal{F}$ should be evaluated on the items of stream S. Formally, a shared execution strategy in its most general form is defined as follows:

DEFINITION 2.1. (SHARED EXECUTION STRATEGY). *A shared execution strategy $\mathcal{P}$ for the set of queries $\mathcal{Q}$ is a function that takes as input the set of filters evaluated so far and their results, and decides the next filter to be evaluated.* □

Strategy $\mathcal{P}$ in its general form is said to be *adaptive*. In the special case when $\mathcal{P}$ always evaluates filters in the same order for each data item, regardless of the results of the filters evaluated so far, $\mathcal{P}$ is said to be *fixed*. This classification is elaborated upon in Section 3.

The execution algorithm for a shared strategy $\mathcal{P}$ is as follows (see Figure 1). For each incoming item s on stream S, we first initialize the status of every query in $\mathcal{Q}$ to unresolved (Lines 2-3). At any stage, the next filter F to be evaluated is chosen according to strategy $\mathcal{P}$, and F is evaluated on item s (Lines 5-6). If F evaluates to false, then all queries that contain F are resolved to false (Line 8). Otherwise, if F evaluates to true, F is removed from all the queries it is part of. Any query Q that now becomes empty (i.e., F was the only remaining filter in Q), is resolved to true (Lines 10-11). This process is continued until all queries in $\mathcal{Q}$ are resolved. We assume the set of queries $\mathcal{Q}$ is indexed on filters, so that for any filter F, the queries that contain F can be determined efficiently (so that Lines 8 and 11 can be executed efficiently).

Next we describe our cost model for shared execution strategies (Section 2.1), then formally define the problem of optimization of queries with shared expensive filters (Section 2.2).

2.1 Cost Model

In order to compare different shared execution strategies and to choose the best one among them, we need to associate a cost expression with every strategy. The execution cost of a shared strategy consists of two major components:

1. **Cost of filters evaluated**. This is the total cost incurred in Line 6 of Algorithm *ExecuteStrategy* in Figure 1.

2. **Execution overhead**. This is the cost incurred in the rest of the algorithm in Figure 1, i.e., excluding Line 6. Execution overhead consists of two parts—bookkeeping cost (such as keeping track of the number of filters remaining in each query), and the cost of adaptivity, i.e., the cost incurred by $\mathcal{P}$ in deciding the next filter to be evaluated (Line 5). The cost of adaptivity depends on the specific plan $\mathcal{P}$, but the bookkeeping cost is independent of $\mathcal{P}$ and (for any tuple) is at most $O(1)$ per filter present in a query, making it $O(\lambda)$ overall.

We are focussing on applications with expensive filters (e.g., detecting patterns in images, and others mentioned in the introduction), and thus the first component of execution cost typically dominates the second. In this paper, we primarily focus on filter evaluation cost and we address the problem of designing execution strategies that minimize the expected total cost of filters evaluated. In practice, it is also important to keep the execution overhead low, and we give techniques for doing so in Section 5.

To arrive at an expression for the expected total cost of filters evaluated by a strategy, as in much of previous work [4, 6, 15], we assume that for each filter $F_i \in \mathcal{F}$, the following two quantities are known:

- **Cost**: The average per-item processing time (or intuitively, the cost) of filter F_i is denoted by c_i.

- **Selectivity**: The average fraction of data items that satisfy filter F_i is referred to as the *selectivity* of filter F_i, and is denoted by s_i. The selectivity s_i can also be interpreted as the probability $\Pr[F_i = \text{ true}]$, where the probability is taken over the distribution of input data items.

For simplicity of presentation and analysis, we assume *independent filters* in the remaining discussion, i.e., the selectivity of a filter does not depend on the filters already evaluated. Note that our algorithms are more general, and work even for *correlated filters* (as demonstrated by our experiments in Section 6.2.5); we mention the simple modifications needed wherever appropriate.

Consider a shared execution strategy $\mathcal{P}$. Let e_i be the probability that $\mathcal{P}$ will evaluate filter F_i on an incoming item s. In general,

e_i may be less than 1 because all the queries might get resolved before F_i is evaluated. Then the expected cost of filters evaluated by strategy $\mathcal{P}$ is given by:

$$\text{cost}(\mathcal{P}) = \sum_{i=1}^{n} e_i \cdot c_i \qquad (2)$$

The exact expression for e_i depends on the specific type of strategy $\mathcal{P}$ and is given in Section 3. e_i can be written in terms of selectivities of filters as shown by the following example.

EXAMPLE 2.2. *Let there be two queries $Q_1 = \{F_1, F_2\}$ and $Q_2 = \{F_2, F_3\}$. Let $\mathcal{P}$ be a fixed execution strategy that always evaluates the filters in the order F_1, F_3, F_2. According to strategy $\mathcal{P}$, filters F_1 and F_3 will always need to be evaluated. Thus $e_1 = e_3 = 1$. However, filter F_2 will need to be evaluated only if at least one of F_1 or F_3 evaluates to true. Since filters are independent, the probability that F_2 will need to be evaluated (i.e., $\Pr[F_1 = \text{true} \vee F_3 = \text{true}]$ is $e_3 = s_1 + s_3 - s_1 s_3$. Thus the cost of strategy $\mathcal{P}$ according to* (2) *is:*

$$\text{cost}(\mathcal{P}) = c_1 + c_3 + (s_1 + s_3 - s_1 s_3)c_2 \qquad \square$$

2.2 Problem Statement

The problem of optimizing queries with shared expensive filters can be defined as follows:

DEFINITION 2.3. (OPTIMIZATION OF QUERIES WITH SHARED EXPENSIVE FILTERS). *Given a set of queries $\mathcal{Q}$ of the form* (1), *find a shared execution strategy $\mathcal{P}$ (Definition 2.1) such that cost($\mathcal{P}$) given by* (2) *is minimized.* $\square$

The problem of finding the best adaptive strategy is NP-hard (Theorem 4.2). We therefore focus on designing approximation algorithms. We give the standard definition of approximation ratio that is used to measure the quality of an approximation algorithm.

DEFINITION 2.4 (APPROXIMATION RATIO). *An algorithm A has an approximation ratio k (or is a k-approximation) if for all possible instances of the problem, A is guaranteed to result in a solution whose cost is at most k times the cost of the optimal solution.* $\square$

3. SHARED EXECUTION STRATEGIES

Recall Definition 2.1 of a shared execution strategy. In general, a strategy $\mathcal{P}$ decides the next filter to be evaluated based on the results of the filters evaluated so far. Such a strategy is referred to as an *adaptive* strategy. However, a simple special case is when $\mathcal{P}$ evaluates filters in a fixed order, independent of the results of the filters evaluated so far. In this case, $\mathcal{P}$ is referred to as a *fixed* strategy. We study these two types of strategies in the following subsections.

3.1 Fixed Strategies

DEFINITION 3.1. (FIXED STRATEGY). *A fixed strategy is a shared execution strategy that evaluates the filters in $\mathcal{F}$ on any data item in a fixed order (say $F_1, \ldots, F_n$ without loss of generality). No redundant work is done: if the evaluation of $F_1, \ldots, F_{i-1}$ resolves all queries that contain F_i, the evaluation of F_i is skipped.* $\square$

We first show how to calculate the cost of a fixed strategy $\mathcal{P}$. For this, we need to find the probability e_i that $\mathcal{P}$ evaluates F_i (recall (2)). F_i will be evaluated only if the evaluation of $F_1, \ldots, F_{i-1}$ is not sufficient to resolve all queries that F_i is part of. Unfortunately, there is no simple closed form for the probability e_i. However, e_i is simple to calculate as follows. Consider each possible result v of the evaluation of $F_1, \ldots, F_{i-1}$ (there are 2^{i-1} possible values for v). Calculate the probability $p(v)$ of v by using independence of filters and that $\Pr[F_k = \text{true}] = s_k$ for any k. Let $b(v)$ be the indicator variable denoting whether any query containing F_i remains unresolved if the result of evaluating $F_1, \ldots, F_{i-1}$ is v. Then e_i is given by

$$e_i = \sum_{v} p(v) \cdot b(v) \qquad (3)$$

EXAMPLE 3.2. *We redo Example 2.2 using* (3). *Recall the fixed strategy $\mathcal{P} = F_1, F_3, F_2$. Since F_1 is the first filter in $\mathcal{P}$, $e_1 = 1$. Next, since F_3 needs to be evaluated regardless of the result of evaluating F_1, we have $e_3 = 1$. To calculate e_2, we consider all possible results of evaluating F_1, F_3.*

v	$p(v)$	$b(v)$
$F_1 =$ *false*, $F_3 =$ *false*	$(1 - s_1)(1 - s_3)$	*0*
$F_1 =$ *false*, $F_3 =$ *true*	$(1 - s_1)s_3$	*1*
$F_1 =$ *true*, $F_3 =$ *false*	$s_1(1 - s_3)$	*1*
$F_1 =$ *true*, $F_3 =$ *true*	$s_1 s_3$	*1*

The last 3 rows in the table above have $b(v) = 1$ since at least one of Q_1 and Q_2 (both of which contain F_2) remain unresolved. Thus $e_2 = (1 - s_1)s_3 + s_1(1 - s_3) + s_1 s_3 = s_1 + s_3 - s_1 s_3$ as in Example 2.2. $\square$

Fixed strategies are of interest because it is known from previous work that for the special case of a single query (i.e., $m = 1$), the optimal strategy is a fixed one, and is given by the following theorem[1].

THEOREM 3.3. *[15] For a single query that is a conjunction of filters $F_1, \ldots, F_n$, it is optimal to evaluate the filters in increasing order of rank, where rank$(F_i) = c_i/(1 - s_i)$.* $\square$

Now consider the strategy *FixedGreedy* that is a naïve extension of the above theorem to multiple queries. *FixedGreedy* processes the queries sequentially in arbitrary order. For each query, the filters are evaluated in the order given by the single-query greedy algorithm (Theorem 3.3). We now show a performance bound for *FixedGreedy*.

THEOREM 3.4. FixedGreedy *is a μ-approximation (recall definition of μ from Table 1).*

PROOF. Suppose the optimal strategy interleaves evaluation of filters from different queries. However, for any single query, the expected cost of the filters evaluated by the optimal strategy for resolving that query is at least the cost spent by *FixedGreedy* (since *FixedGreedy* is optimal for any single query). Since each filter appears in at most μ queries, the cost of *FixedGreedy* can be at most μ times that of the optimal strategy. $\square$

In Theorem 3.7, we will show that this approximation ratio of $O(\mu)$ is the best possible for *any* fixed strategy. Theorem 3.4 shows that the performance of *FixedGreedy* deteriorates as the extent of sharing, μ increases. Intuitively, *FixedGreedy* performs badly in the case of multiple queries because the placement of filters in the strategy does not take into account the *participation* of each filter, i.e., the number of queries in which the filter participates. In fact,

[1]The problem is NP-hard when filter selectivities are correlated; however, the same greedy ordering modified to use conditional selectivities yields an approximation ratio of 4.

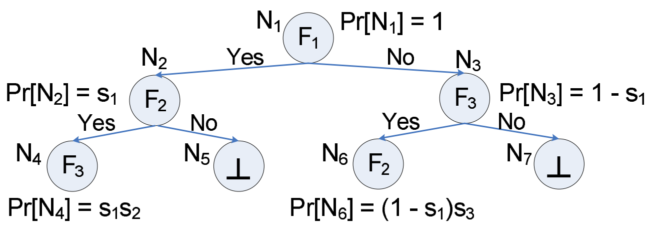

Figure 2: Decision Tree for Example 3.5

our main adaptive strategy presented in Section 4 can be viewed as an extension of Theorem 3.3 to take participation into account.

Fixed strategies have a low execution overhead: they only incur the usual bookkeeping cost of $O(\lambda)$ for each data item (recall Section 2.1). To avoid redundant work, when a filter F is chosen to be evaluated, we iterate through all the queries in which F participates. If all these queries are resolved, evaluation of F is skipped. The cost of these iterations is also of the same order as the bookkeeping cost (at most $O(\lambda)$ for each data item). Apart from this, the decision step in Line 5 of Figure 1 is very cheap, since filters are evaluated in a fixed order. However, in terms of the expected total filter evaluation cost (given by (2)), fixed strategies are often inferior to the more general adaptive strategies, as shown next.

3.2 Adaptive Strategies

Recall Definition 2.1 that defines general adaptive strategies. An adaptive strategy can be represented conveniently as a *decision tree* in which each node corresponds to a filter and has two children referred to as the `Yes` child and the `No` child respectively.

To build the decision tree for a given adaptive strategy $\mathcal{P}$, note that the first filter to be evaluated by $\mathcal{P}$ must be a fixed filter F (since there are no previous filters on which F can depend). Make F the root of the decision tree. Add as the `Yes` child of F the filter that $\mathcal{P}$ evaluates next if F evaluates to true. If the evaluation of F to true resolves all queries and there are no more filters to be evaluated, the `Yes` child of F is a special leaf node marked $\perp$. The `No` child of F is constructed similarly based on what happens if F evaluates to false. The subtrees of the `Yes` and `No` children are then built recursively. Note that multiple nodes of the decision tree may correspond to the same filter.

EXAMPLE 3.5. *We continue with our running example (Example 2.2). Recall the queries $Q_1 = \{F_1, F_2\}$ and $Q_2 = \{F_2, F_3\}$. Consider the adaptive strategy $\mathcal{P}$ that first evaluates F_1. If F_1 evaluates to true, $\mathcal{P}$ evaluates F_2, otherwise it evaluates F_3. The third remaining filter is then evaluated if needed. The decision tree corresponding to $\mathcal{P}$ is shown in Figure 2, with nodes named $N_1, \dots, N_7$.*

The execution of $\mathcal{P}$ for each data item can be viewed as starting at the root of the corresponding decision tree and traversing to one of the leaves based on filter results. We assume that no two nodes on the path from the root to a leaf can correspond to the same filter (since that would constitute redundant filter evaluation). Let $\Pr[N]$ denote the probability that a node N is visited during execution of $\mathcal{P}$. Let N_{yes} and N_{no} denote the children of N. If the filter at N is F_i, we have (by using $\Pr[F_i = \text{true}] = s_i$ and independence of filters):

$$\begin{aligned} \Pr[N_{yes}] &= \Pr[N] \cdot s_i \\ \Pr[N_{no}] &= \Pr[N] \cdot (1 - s_i) \end{aligned} \quad (4)$$

Thus, given that Pr[root]=1 (since the root is always visited), $\Pr[N]$ can be calculated for each node N. For Example 3.5, $\Pr[N]$ for each node N is annotated in Figure 2.

As in the case of fixed strategies, to calculate the cost of an adaptive strategy $\mathcal{P}$, we must calculate the probability e_i that $\mathcal{P}$ evaluates filter F_i (recall (2)). If filter F_i occurs at nodes $N_i^1, \dots, N_i^k$ of the decision tree, then e_i is given by

$$e_i = \sum_{j=1}^{k} \Pr[N_i^j] \quad (5)$$

EXAMPLE 3.6. *We continue with our running example (Example 2.2) and the adaptive strategy $\mathcal{P}$ shown in Figure 2. Since F_1 occurs at the root, $e_1 = 1$. F_2 occurs at nodes N_2 and N_5. From* (5), *$e_2 = \Pr[N_2] + \Pr[N_5] = s_1 + (1 - s_1)s_3$. Similarly, $e_3 = 1 - s_1 + s_1 s_2$. From* (2)*:*

$$cost(\mathcal{P}) = c_1 + \big(s_1 + (1 - s_1)s_3\big) \cdot c_2 + \big(1 - s_1 + s_1 s_2\big) \cdot c_3 \quad \square$$

We now motivate the need for designing adaptive strategies by showing that they are vastly superior in performance to any fixed strategy when the sharing parameter μ is large.

THEOREM 3.7. *There exist problem instances where the cost of any fixed strategy is $\Omega(\mu)$ times the cost of the optimal adaptive strategy.* $\square$

The proof of the above theorem appears in Appendix A. The bound given by the above theorem is tight since we have already seen (in Theorem 3.4) that the fixed strategy *FixedGreedy* is a μ-approximation.

Adaptive strategies have higher execution overhead than fixed strategies since they incur the cost of deciding the next filter to be evaluated at each step. One way to avoid this cost is to store the entire decision tree corresponding to the strategy in memory. However, storing the entire decision tree is infeasible in general since its size can be exponential in the number of filters.

In the next section, we focus on filter evaluation cost and consider the problem of finding the adaptive strategy that minimizes the expected total cost of filters evaluated for a given set of queries. Then in Section 5, we show how the execution overhead of an adaptive strategy can be reduced by appropriate precomputation.

4. FINDING THE OPTIMAL STRATEGY

In this section, we consider the problem of finding the optimal strategy for a given set of queries (Definition 2.3). We first show the similarity of our problem to the well-known set cover problem (Section 4.1). Based on this similarity, we show the hardness of finding (or even approximating) the optimal strategy in the general case. We then identify special cases where this hardness does not hold (Section 4.2). We then give a lower bound on the cost of the optimal strategy (Section 4.3), followed by our general greedy adaptive strategy in Section 4.4.

4.1 Hardness and Set Cover

DEFINITION 4.1 (SET COVER PROBLEM [12]). *Given a collection $\mathcal{S}$ of k sets $S_1, \dots, S_k$, choose a minimum collection $\mathcal{C}$ of sets from $\mathcal{S}$ that covers the universal set, i.e., $\bigcup_{S_i \in \mathcal{C}} S_i = \bigcup_{i=1}^{k} S_i$.*

For our problem, let the universal set be the set of all queries $\mathcal{Q}$, and let each filter F_i be a set $S_i = \{Q \in \mathcal{Q} | F_i \in Q\}$. Thus, a filter covers all the queries that it can potentially resolve, and the aim is to pick the least-cost collection of filters that resolve (or cover) all the queries; hence the similarity to set cover. However, our problem departs from classical set cover in that the notion of a filter resolving (or covering) a query is probabilistic. Thus, when a filter F_i is picked, it resolves the set of queries S_i only with probability $1 - s_i$, i.e., when it evaluates to false. Otherwise, with probability s_i, i.e., when it evaluates to true, it resolves only those queries in S_i in which it was the solitary filter. Thus our problem can be viewed as a new probabilistic version of set cover that, to the best of our knowledge, has not been considered before.

We now show the hardness of finding the optimal adaptive strategy, and then identify certain special cases where this hardness does not hold. The following hardness result for our optimization problem follows easily from a reduction from set cover (recall that m is the number of queries), and the proof is omitted.

THEOREM 4.2. *No polynomial-time algorithm for finding an adaptive strategy can have an approximation ratio $o(\ln m)$, unless P=NP.*

4.2 Limited Phase Adaptivity

Before considering full-blown adaptive strategies, we show in this section that when κ (the maximum number of filters in a query) is small, there exist strategies that make only a small number of adaptive decisions, and yet have a good approximation ratio. Such strategies that make a small number of adaptive decisions are interesting since they are closer to fixed strategies in having a low execution overhead.

An adaptive strategy that makes k adaptive decisions evaluates its filters in $k + 1$ phases, where the filters in each phase are fixed according to the results of filter evaluations in the previous phases. The next theorem (whose proof will be presented in an extended version of the paper) follows by observing the similarity of the problem to *Vertex Cover* for $\kappa = 2$ and to *κ-Hypergraph Vertex Cover* for general κ.

THEOREM 4.3.
1. *When $\kappa = 2$, there exists a polynomial-time algorithm to find a strategy that is a 2-approximation, and that makes no adaptive decisions, i.e., is fixed.*
2. *For general κ, there exists a polynomial-time algorithm to find a strategy that is an $O(\kappa^2)$-approximation, and that makes $\kappa - 2$ adaptive decisions.*

The above theorem adequately demonstrates the power of adaptivity for our problem: For $\kappa = 3$, using Theorem 3.7, any fixed strategy is an $\Omega(\mu)$-approximation. However, by Theorem 4.3, for $\kappa = 3$, we can find an adaptive strategy that is a 9-approximation and makes only one adaptive decision. Thus, allowing for adaptive strategies with just one intermediate decision can yield a dramatic performance benefit of $\Theta(\mu)$ for such instances.

We now build towards designing a general adaptive strategy. We start by presenting an efficiently computable lower bound on the cost of the optimal adaptive strategy.

4.3 Lower Bound on Cost

In this section, we prove a lower bound on the cost of the optimal strategy. This lower bound is used to bound the approximation ratio of our algorithm. It is also used in our experiments (Section 6) as a benchmark to compare our algorithm against.

For every query $Q_i \in \mathcal{Q}$, let r_i denote the probability that Q_i resolves to false. Then,

$$r_i = 1 - \prod_{j|F_j \in Q_i} s_j \tag{6}$$

Consider the following linear program with variables $e_1, \ldots, e_n$.

$$\text{Minimize} \quad \sum_{j=1}^{n} c_j \cdot e_j \qquad \text{subject to:} \tag{7}$$

$$\begin{array}{lrcl} \forall \text{ queries } Q_i \in \mathcal{Q} & \sum_{j \mid F_j \in Q_i} (1 - s_j)e_j & \geq & r_i \\ \forall \text{ filters } F_j \in \mathcal{F} & e_j & \in & [0, 1] \end{array}$$

THEOREM 4.4. *The cost of the optimal strategy is lower bounded by the optimum value of the linear program in* (7).

PROOF. Consider any adaptive strategy $\mathcal{P}$. Let e_j denote the probability that $\mathcal{P}$ evaluates filter F_j. By the union bound in probability theory, $\Pr[Q_i \text{ resolves to false}] = r_i \leq \sum_{j \mid F_j \in Q_i} \Pr[F_j$ is evaluated $\wedge\ F_j =$ false$]$. Since F_j evaluating to false is independent of F_j being evaluated, we have:

$$r_i \leq \sum_{j \mid F_j \in Q_i} (1 - s_j)e_j$$

From (2), the cost of $\mathcal{P}$ is $\sum_{j=1}^{n} c_j \cdot e_j$. Thus, the optimum value of the linear program in (7) is a lower bound on the cost of any adaptive strategy. □

The above linear program essentially approximates the probability of filter evaluations for any decision tree by using linear constraints that are derived by applying union bounds. This makes the lower bound efficiently computable. However, we have not encoded the constraint that the values of e_j must be obtainable from some decision tree by (5), which seems hard to do through a linear program. Thus, the above lower bound may be loose. However, for our problem, it turns out to be good enough for designing approximation schemes. (In our experiments, we found that this lower bound was only about a factor of 2 lower than the actual optimal cost).

This general technique of lower bounding the value of an adaptive strategy using linear constraints is also used in the context of stochastic scheduling [9, 20] to develop non-adaptive strategies that are constant factor approximations to the best adaptive strategy. For our problem however, we use the linear program to design approximate adaptive strategies.

4.4 Greedy Adaptive Strategy

In this section, we describe our general greedy adaptive execution strategy and provide theoretical guarantees regarding its cost. Recall the similarity of our problem to set cover (Section 4.1). The following greedy algorithm is the best known polynomial-time algorithm for set cover: Start by picking the set that covers the maximum number of elements. Then at each stage pick the set that covers the maximum number of uncovered elements, and continue until all elements are covered.

Let us try using this algorithm to find an adaptive strategy for our problem, which is a probabilistic version of set cover. Our aim is to resolve all queries. Thus, at any stage, we should pick the filter that is expected to resolve the maximum number of unresolved queries per unit cost. This algorithm, *Greedy*, is shown in Figure 3 (written assuming it will be invoked repeatedly by Line 5 of algorithm *ExecuteStrategy* in Figure 1).

If there exists an unresolved query Q that has only a single unevaluated filter F, then evaluation of F is necessary to resolve Q. Hence we first evaluate any such filter F (Lines 1-2 of Figure 3). Suppose filter F_i occurs in p_i unresolved queries. Then with probability $1 - s_i$, F_i resolves p_i queries (when it evaluates to false), otherwise with probability s_i it does not resolve any queries (when

```
Algorithm *Greedy*
1. if ($\exists$ unresolved query $Q$ with exactly 1 unevaluated filter)
2.    pick the single filter in $Q$ to be evaluated next
3. else
4.    $p_i \leftarrow$ number of unresolved queries $F_i$ is part of
5.    pRank($F_i$) $\leftarrow \frac{c_i}{p_i(1-s_i)}$
6.    pick unevaluated filter with min pRank to be evaluated next
```

Figure 3: Greedy Adaptive Strategy

it evaluates to true, since there are no queries with a single filter remaining). Thus the expected number of queries resolved by F_i is $p_i(1 - s_i)$. Analogous to Theorem 3.3, we define the pRank of F_i as the ratio of c_i to $p_i(1 - s_i)$, and then pick the filter with the minimum pRank (Lines 4-6 of Figure 3). Thus, *Greedy* can also be seen as an extension of the technique in Theorem 3.3 to take *participation* of a filter into account, i.e., the number of queries in which a filter occurs.

If the filter selectivities are *correlated*, we need a simple modification to the definition of pRank. Let s_i denote the conditional selectivity of F_i given the results of all filters evaluated so far. Then pRank$(F_i) \leftarrow \frac{c_i}{p_i(1-s_i)}$, where p_i denotes the number of unresolved queries F_i participates in.

The algorithm *Greedy* turns out to be hard to analyze since the number of adaptive decisions made is large. Instead, we have analyzed a natural variant of *Greedy* (called *GreedyVariant*) that makes use of the bound in Section 4.3 more directly and hence is easier to analyze. *GreedyVariant* evaluates chunks of filters non-adaptively before making the next adaptive decision (in contrast to *Greedy* that evaluates one filter before making the next adaptive decision). *GreedyVariant* is given in Appendix B (Figure 13). We have obtained the following performance bound for *GreedyVariant* (the proof is provided in the extended technical report [18]).

THEOREM 4.5. *Algorithm GreedyVariant (Figure 13) has an approximation ratio of* $O(\log^2 m \log n)$. $\square$

Note that although the above theorem assumes independent filters, we have performed experiments for both independent and correlated filters. In our experiments, *GreedyVariant* always performs worse than *Greedy* (although only slightly so), which gives strong evidence that *Greedy* itself has similar theoretical guarantees. In practice, *Greedy* performs very well and produces the optimal solution on most instances. We now consider the execution overhead of *Greedy* at each invocation.

THEOREM 4.6. *The per-tuple execution overhead of algorithm* Greedy *is* $O(\lambda \log n)$ *for independent filters and* $(\lambda + n^2 t) \log n)$ *for correlated filters, where* t *is the time for computing a conditional selectivity value. Recall that* λ *is the total number of* (F_i, Q_j) *pairs where* F_i *is present in* Q_j.

PROOF. We maintain a priority queue of filters sorted by pRank. Each time a filter is evaluated, consider the queries that this filter participates in. If these queries are resolved, they are deleted. The total work involved in deleting resolved queries is $O(\lambda)$. The pRanks of all filters that participate in these queries needs to be updated. For a given query, this operation happens once when the query is satisfied. Therefore, the total number of times the pRank of filters is updated is $O(\lambda)$. Each of these updates involves manipulating the heap, which takes $O(\log n)$ time.

If filter selectivities are correlated, each time a filter is evaluated, the pRanks of all other filters needs to be updated since their conditional selectivity changes. The number of such updates is $O(n^2)$. The total amount of work is now $O((\lambda + n^2 t) \log n)$. We assume that computing the new selectivity takes t time; in practice, for simple types of correlation, this computation can be implemented in $O(1)$ time using table lookup. $\square$

Note that the execution overhead of *Greedy* is only $O(\log n)$ times worse than the fixed bookkeeping overhead. This however can be significant, especially for cheap filters. We now show how the execution overhead of *Greedy* (or any adaptive strategy in general) can be reduced by precomputation.

5. REDUCING EXECUTION OVERHEAD

One way to reduce the execution overhead of an adaptive strategy $\mathcal{P}$, given arbitrary amounts of memory, is to precompute and store the entire decision tree corresponding to $\mathcal{P}$. Deciding the next filter to be evaluated is then a simple lookup into the stored decision tree. However, in general the size of the decision tree may be exponential in the number of filters, making it infeasible to store the entire decision tree unless the total number of filters $|\mathcal{F}|$ is small. In practice, we may have sufficient memory to precompute and store only some M nodes of the decision tree, with the rest of the filter selections occurring dynamically. Our goal is to decide which M nodes from the decision tree to precompute and store, such that the expected execution overhead incurred is minimized.

Consider the execution overhead incurred when $\mathcal{P}$ is executed with M stored nodes. Recall from Section 3.2 that the execution of an adaptive strategy for each data item can be viewed as starting at the root of the decision tree and traversing down to some leaf node. For any node N_i, let $o[N_i]$ denote the overhead incurred in computing node N_i, and let $\Pr[N_i]$ be the probability of visiting node N_i (from (4)). If N_i is stored, its contribution to the execution overhead is 0 since only a simple lookup is needed whenever N_i is visited. If N_i is not stored, the expected contribution of N_i to the overall execution overhead is given by:

$$O[N_i] = \Pr[N_i] \cdot o[N_i] \tag{8}$$

Clearly, to minimize the expected overhead given space for storing only M nodes, we must store those nodes that have the M highest values for $O[N_i]$.

For this paper (in our experiments), we assume $o[N_i] = c$ for every node N_i where c is some constant. However, our algorithm only requires that $o[N_i]$ decreases as we go down the decision tree, i.e., if node N_i is an ancestor of node N_j, then $o[N_i] \geq o[N_j]$. This assumption is reasonable because the overhead of deciding the next filter often depends on the number of unevaluated filters (as in *Greedy*) or the number of unresolved queries, both of which decrease as we go down the decision tree. Note that the probability of visiting nodes also decreases as we go down the tree (from (4)).

Algorithm *Precompute* (Figure 4) decides the best M nodes to be stored without computing the whole decision tree. Since $o[N_i]$ and $\Pr[N_i]$ both decrease as we go down the decision tree, $O[N_i]$ also decreases on going down the decision tree. Hence any node in the subtree of node N_i need not be considered for storing unless N_i has already been stored. Thus, algorithm *Precompute* starts by considering only the root for storing, and then every time a node N_i is chosen for storing, its children are added to the set of nodes being considered for storing. This process continue until M nodes have been chosen, and requires only $O(M)$ node computations. This reasoning yields the following theorem.

```
Algorithm Precompute$(M, \mathcal{P})$
$M$: number of decision-tree nodes that can be stored
$\mathcal{P}$: Adaptive strategy $\mathcal{P}$
 1. candidates $\leftarrow$ {root of decision tree corresponding to $\mathcal{P}$}
 2. while (number of stored nodes $< M$)
 3.     store node $N_i \in$ candidates with maximum $O[N_i]$ (from (8))
 4.     candidates $\leftarrow$ candidates $- \{N_i\} \cup$ children of $N_i$
```

Figure 4: Algorithm for Precomputing an Adaptive strategy

THEOREM 5.1. *Algorithm Precompute decides the M nodes to store such that the execution overhead is minimized.* □

6. EXPERIMENTS

We now present an experimental evaluation of our techniques. We carried out two broad categories of experiments—one measuring the expected total cost of filters evaluated by a strategy, and the other measuring the execution overhead incurred by a strategy. For our cost experiments, we compared our shared execution strategy *Greedy* (Figure 3) against the following two naïve execution strategies[2]:

1. *Naïve*: This strategy models what a system that optimizes queries one at a time would do when multiple queries of the form (1) are registered. Each query executes in isolation from other queries and evaluates its filters in rank order for every data item (Theorem 3.3). Filter evaluations are not shared across queries.

2. *Shared*: This strategy models a first cut towards sharing. Each query evaluates its filters in rank order as in *Naïve*. If a query Q needs to evaluate a filter F that has already been evaluated by some other query, Q reuses the previous result. Each query evaluates one filter at a time with round robin execution among the queries. Additionally, when a filter F evaluates to false, all queries containing F are marked as resolved and do not execute any further for that data item.

For our execution overhead experiments, we included another execution strategy called *Cache* in our comparison. *Cache* is *Greedy* augmented with a fixed-size cache to precompute and store some nodes of the corresponding decision tree according to Algorithm *Precompute* in Figure 4.

To capture a wide range of scenarios, we experimented with synthetic as well as real problem instances. The main findings from our experiments are:

1. For the problem instances we generated, the expected total cost of filters evaluated by *Greedy* is about twice the optimal cost, whereas for *Shared* it can be up to 8 times the optimal cost, and for *Naïve* up to 12 times the optimal cost.

2. As expected, *Greedy* has a higher execution overhead than *Shared*. However, the execution overhead can be substantially reduced (comparable to *Shared*) by using *Cache*.

We first describe how we generated problem instances for our experiments (Section 6.1). Due to space constraints, we have been able to include only a subset of our experimental results. We report on our cost experiments in Section 6.2 and our overhead experiments in Section 6.3.

[2]We do not report cost numbers for *GreedyVariant* as they were very close to those for *Greedy*.

6.1 Problem Instance Generation

We built a *problem instance generator* with the following input parameters:

n, m	Number of filters and queries respectively
$\tilde{\mathbf{c}}, \tilde{\mathbf{s}}, \tilde{\mathbf{p}}$	Range of filter costs, selectivities, and participation respectively

Table 2: Input Parameters to Problem Instance Generator

The generator first creates a set $\mathcal{F}$ of n filters and a set $\mathcal{Q}$ of m empty queries. For each filter in $\mathcal{F}$, the cost, selectivity, and participation (the number of queries it is part of) are chosen uniformly at random from the ranges $\tilde{\mathbf{c}}$ and $\tilde{\mathbf{s}}$ and $\tilde{\mathbf{p}}$ respectively. Let $\bar{p}$ be the midpoint of the range $\tilde{\mathbf{p}}$. Then on average each filter occurs in $\bar{p}$ queries.

Each query $Q_i \in \mathcal{Q}$ is assigned a weight w_i chosen uniformly at random from $[0, 1]$. Each filter $F_j \in \mathcal{F}$ (having participation p_j) is then added to a randomly chosen set of p_j queries from $\mathcal{Q}$ with query Q_i being chosen with probability proportional to w_i. Thus, at the end of the generation process, the expected number of filters in a query is proportional to its weight. Assigning weights to queries avoids always generating problem instances in which all queries have roughly equal number of filters. The average number of filters per query (across all queries) is $n\bar{p}/m$.

For our synthetic problem scenarios, filter evaluation is simulated by randomly deciding whether a filter evaluates to true or false, with $\Pr[F_j \text{ evaluates to true}] = s_j$, independently for every filter (except for Section 6.2.5 where we experiment with correlated filters).

6.2 Cost Experiments

In Sections 6.2.1 to 6.2.4, we experiment with synthetic problem instances, varying the input parameters to our problem instance generator, and studying their effect on the expected total cost of filters evaluated by the execution strategies *Greedy*, *Shared*, and *Naïve*. Then in Section 6.2.6, we report our results on a real problem instance.

For each generated problem instance I, and for each strategy $\mathcal{P}$ being compared, we first measured the average cost of $\mathcal{P}$ on I by simulating the execution of $\mathcal{P}$ on 1000 input tuples. We found that 1000 tuples were enough to measure the average cost accurately since even on increasing the number of tuples, the measured average cost did not undergo any considerable change (typically within 2%). We then computed a lower bound on the cost of the optimal strategy on I by using the linear program in Section 4.3. Finally, we computed the ratio of the average cost of $\mathcal{P}$ on I to this lower bound (referred to as the *approximation ratio* of $\mathcal{P}$ on I). Note that in reality this ratio is only an upper bound on the actual approximation ratio and not the exact approximation ratio, since it is obtained by using a lower bound on the optimal cost and not the exact optimal cost (which seems hard to find without an exhaustive search).

To compare various strategies for a particular setting of parameters, we randomly generated 100 problem instances with the same parameter settings using our problem instance generator (Section 6.1). For strategy $\mathcal{P}$, we then use the worst (or maximum) approximation ratio of $\mathcal{P}$ on any of these 100 instances as a metric for comparing $\mathcal{P}$ with other strategies. We use the maximum and not the average approximation ratio as a metric since for an execution strategy to be good, it is not enough if it performs well on average across all problem instances, but it should perform well on any given problem instance. Note that our use of the maximum approximation ratio as

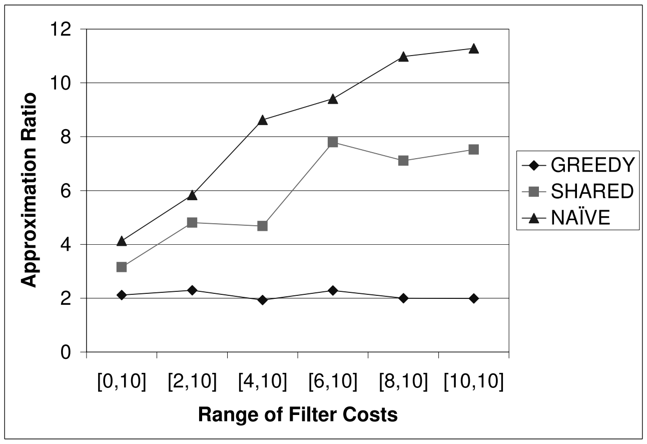

Figure 5: Effect of Filter Costs

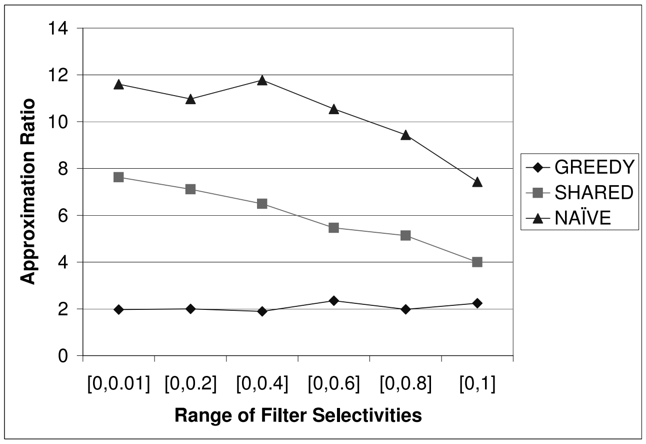

Figure 6: Effect of Filter Selectivities

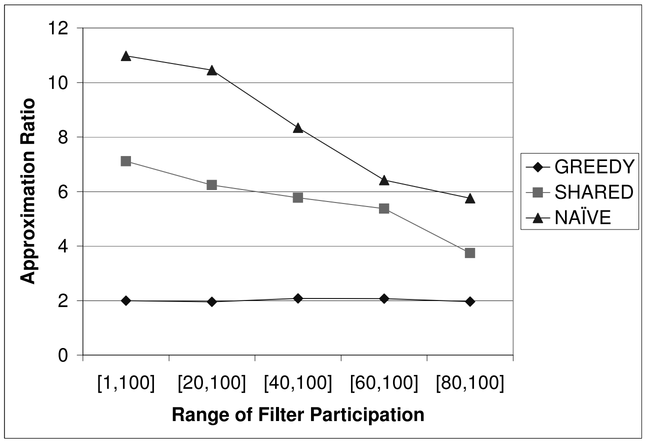

Figure 7: Effect of Filter Participation

a metric for comparison does not imply that we are focussing on the worst-case performance of an execution strategy as opposed to its average-case performance. For any given problem instance, we still consider the average cost of filters evaluated by a strategy, and not the worst-case cost.

We now present the results of our cost experiments. Unless varied for a specific experiment, the parameters for the problem instance generator were fixed at $n = 150$, $m = 100$, $\tilde{\mathbf{c}} = [8, 10]$, $\tilde{\mathbf{s}} = [0, 0.2]$, and $\tilde{\mathbf{p}} = [1, 100]$.

6.2.1 *Effect of Filter Costs*

Figure 5 shows the worst approximation ratio of the various execution strategies as the range of filter costs $\tilde{\mathbf{c}}$ is varied from $[0, 10]$ to $[10, 10]$. *Greedy* consistently has an approximation ratio of about 2, independent of $\tilde{\mathbf{c}}$, whereas the approximation ratio of both *Naïve* and *Shared* generally worsens as the range of filter costs narrows: When the range of filter costs is wider, there are some cheap filters that might resolve queries at a low cost even for the naïve algorithms. However, as all filters become expensive, the naïve algorithms incur a high cost that is reflected in a higher approximation ratio.

6.2.2 *Effect of Filter Selectivities*

Figure 5 shows the worst approximation ratio of the various execution strategies as the range of filter selectivities $\tilde{\mathbf{s}}$ is varied from $[0, 0.01]$ to $[0, 1]$. Both *Naïve* and *Shared* perform very badly when all filters have extremely low selectivity. However, as the range $\tilde{\mathbf{c}}$ widens to $[0, 1]$, their performance improves: When all filters have low selectivity, significant gains can be achieved by choosing the right set of filters to evaluate first (since they can resolve most of the queries). As we include filters with higher selectivity, queries cannot be resolved early and the choice of the order of evaluation does not matter as much (e.g., in the extreme case when all filters have selectivity 1, any strategy must evaluate all filters). Our strategy *Greedy* is able to perform well (with an approximation ratio of about 2) independent of the range $\tilde{\mathbf{s}}$.

6.2.3 *Effect of Filter Participation*

Figure 7 shows the worst approximation ratio of the various execution strategies as the range of filter participation $\tilde{\mathbf{p}}$ is varied from $[1, 100]$ to $[80, 100]$. As in previous experiments, *Greedy* consistently has an approximation ratio of about 2 independent of $\tilde{\mathbf{p}}$. However, *Naïve* and *Shared* perform very badly when $\tilde{\mathbf{p}}$ is wide, and their performance improves only when $\tilde{\mathbf{p}}$ becomes narrow: When $\tilde{\mathbf{p}}$ is narrow, all filters have similar participation, and the order of filters according to pRank as used by *Greedy* (Line 5 of Figure 3) is not too different from their order according to rank as used by *Naïve* and *Shared* (Theorem 3.3).

6.2.4 *Effect of Number of Filters*

In this experiment, our goal was to study the effect of the ratio of number of filters to number of queries (n/m) on the performance of various execution strategies. For this purpose, we varied n from 20 to 220 while m was held constant at 100. Figure 8 shows the worst approximation ratio of the various execution strategies. *Greedy* again has a consistent approximation ratio of about 2. However, the performance of both *Naïve* and *Shared* first worsens, and then improves as the number of filters is increased: When the number of filters is low, there is less room for mistakes in choosing which filters to evaluate first, and when the number of filters is high, there is a higher probability of having relatively cheap filters or low selectivity filters that can resolve queries early at a low cost. Thus, an intermediate number of filters is the worst case for *Naïve* and *Shared*.

6.2.5 *Correlated Filters*

In this final cost experiment on synthetic problem instances, our goal was to determine how our approach performs with correlated filters. To capture correlations among filters, we use the relatively simple model introduced in [4]. In this model, the n filters are divided into $\lceil n/k \rceil$ groups containing k filters each, where k is called the correlation index. If two filters belong to different groups, they

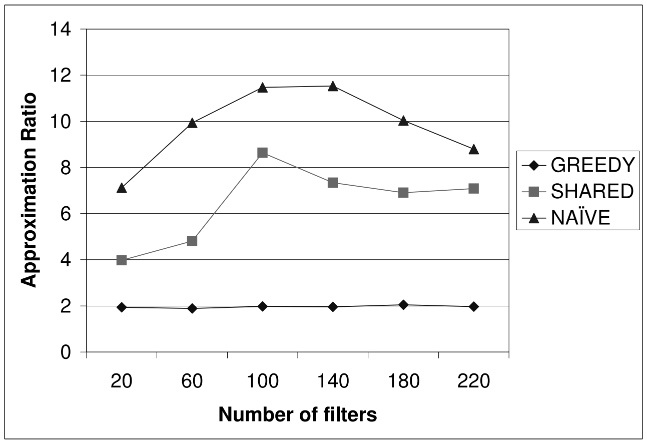

Figure 8: Effect of Number of Filters

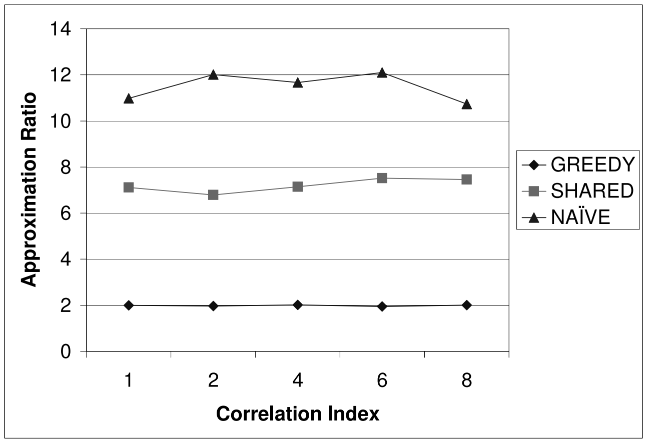

Figure 9: Effect of Correlation Among Filters

are independent, otherwise they are positively correlated, agreeing on 80% of input tuples. Thus $k = 1$ implies independence, while $k = n$ implies all filters are correlated with each other. On fixing k and an unconditional selectivity for each group, the conditional selectivities of the filters is implied by the 80% rule.

Figure 9 shows the worst approximation ratio of the various execution strategies as the correlation index is varied from 1 to 8. We also proportionally varied the total number of filters from 150 to 1200 so that the number of independent filter groups remained constant. Note that the versions of *Shared* and *Naïve* that we used for this experiment used conditional rather than unconditional selectivities to order filters (see footnote 2 in Section 3.1). The performance of all three strategies can be seen to be essentially independent of the correlation index: By using conditional selectivities instead of unconditional selectivities, all the three strategies are guarded equally against correlation, and since the number of independent filter groups remains constant, the performance does not change significantly.

6.2.6 *Real Problem Instance*

In this experiment, we applied filters on a stream of images and constructed queries using these filters. The filters we used were relatively simple but still expensive: each filter determined whether the amount of match in the image for a given color was more than a threshold or not (e.g. a large amount of green can be an indicator of vegetation). In reality, such filters may be more complicated [19] (e.g., they may perform texture matching to detect vegetation).

We used a collection of about 580 images of 800×600 resolution. We set up four filters $F_1, \ldots, F_4$ looking for different colors. Each filter had roughly the same cost. The selectivity of $F_1, \ldots, F_3$ on our image collection was about 1 in 20 while that of F_4 was about 3 in 20. We constructed three queries $\{F_1, F_4\}$, $\{F_2, F_4\}$, and $\{F_3, F_4\}$, and executed them over our collection. The filter processing time for various execution strategies was as follows (Table 3):

	Greedy	*Shared*	*Naïve*
Total time (msec)	3477.79	7070.48	7283.88
Average per image (msec)	6.01	12.21	12.58

Table 3: Performance over a Stream of Images

Even for this extremely simple scenario, *Greedy* yields about a factor of two improvement over *Shared* or *Naïve*: Both *Shared* and *Naïve* evaluate $F_1, \ldots, F_3$ before F_4 since $F_1, \ldots, F_3$ have lower rank, but *Greedy* recognizes that F_4 has a higher participation and evaluates it first. Due to the small number of filters and queries, the execution overhead in this experiment was practically zero.

6.3 Execution Overhead Experiments

In our execution overhead experiments, we varied the parameters to our problem instance generator and studied their effect on the average per-tuple execution overhead incurred by the strategies *Greedy*, *Cache*, and *Shared*. We also studied the effect of varying the number of nodes stored by *Cache*. We did not include *Naïve* in our overhead comparisons since *Naïve* requires no bookkeeping and hence incurs negligible execution overhead.

The average per-tuple execution overhead for a strategy $\mathcal{P}$ was measured by simulating the execution of $\mathcal{P}$ on 1000 input tuples. Similar to our cost experiments, for a particular setting of parameters, we randomly generated 100 problem instances and then used the maximum average overhead incurred by $\mathcal{P}$ on any of these instances as a metric for comparing $\mathcal{P}$ against other strategies.

Unless varied for a specific experiment, the parameters were fixed at $n = 300$, $m = 200$, $\tilde{\mathbf{c}} = [0, 10]$, $\tilde{\mathbf{s}} = [0, 1]$, and $\tilde{\mathbf{p}} = [0, 200]$. The number of nodes stored by *Cache* was fixed at 1000. The execution overhead incurred by a strategy is clearly independent of filter costs, hence we omit the experiments where $\tilde{\mathbf{c}}$ is varied. We also omit the experiments where $\tilde{\mathbf{s}}$ is varied, as they do not reveal any interesting trends except that as we include filters with higher selectivity, the execution overhead of all the three strategies uniformly increases (since more filters need to be evaluated to resolve queries).

6.3.1 *Effect of Filter Participation*

Figure 10 shows the execution overhead of the various strategies as the range of filter participation $\tilde{\mathbf{p}}$ is varied from $[0, 200]$ to $[160, 200]$. *Shared* maintains a very low execution overhead even for high values of participation. However, the overhead of *Greedy* increases with increasing participation because then every query has a higher number of filters and every time a query is resolved, we have to iterate through all the filters to maintain their pRanks. But the execution overhead of *Cache* remains low even with increasing participation since the above computation by *Greedy* is avoided as long as the next filter to be evaluated is given by a node that is cached.

We omit the results of the experiment in which n was varied, since the results were very similar to Figure 10: As the number of filters increases, the net effect on overhead is only due to an increase in the average number of filters per query—the same effect as that obtained by increasing the participation of every filter.

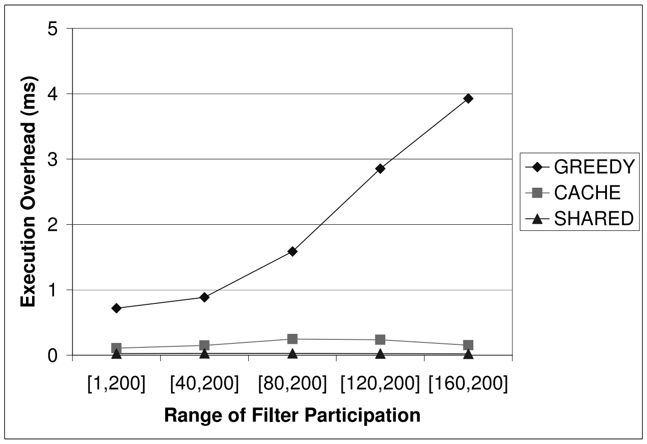

Figure 10: Effect of Filter Participation

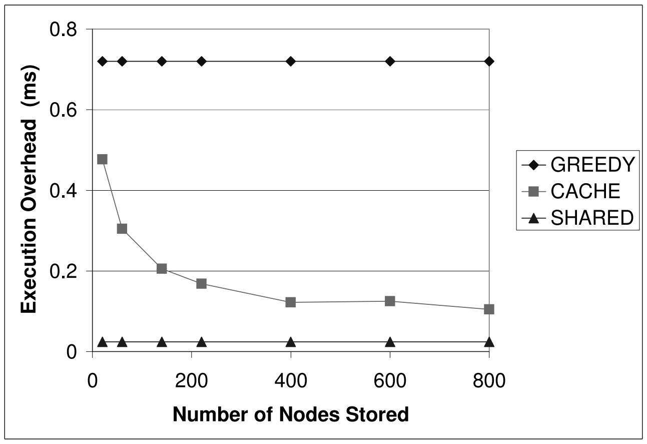

Figure 11: Effect of Number of Nodes Stored

6.3.2 *Effect of Number of Nodes Stored*

Figure 11 shows how the execution overhead of *Cache* varies as the number of nodes of the decision tree stored by *Cache* is varied from 20 to 800. The constant execution overheads of *Greedy* and *Shared* for this experiment are also shown in Figure 11 for comparison. Much of the benefit of precomputation can be realized even with a small number of nodes being stored (as low as 400): There are only a few nodes in the decision tree that have a high probability of being visited, and precomputing them is enough to save much of the overhead incurred.

7. CONCLUSIONS

We considered the problem of optimizing a collection of queries where each query is a conjunction of possibly expensive filters, and filters may be shared across queries. We explored the space of shared execution strategies and showed that for multiple queries, a fixed execution strategy (of the type commonly used for a single query) may be suboptimal in terms of the expected total cost of filters evaluated. Instead, the optimal strategy for multiple queries is adaptive, i.e., the next filter to be evaluated is decided based on the results of the filters evaluated so far. We proved the hardness of approximating the optimal adaptive strategy to any factor smaller than logarithmic in the number of queries. We gave a simple greedy adaptive execution strategy that approximates the optimal strategy to within a factor polylogarithmic in the number of queries and filters. We also gave a simple method to reduce the execution overhead of any adaptive strategy.

An interesting direction for future work is to consider the same problem of optimizing multiple queries with shared expensive filters, but in a distributed rather a centralized setting. In a distributed setting many more factors need to be taken into account, e.g., the costs of filters at different nodes, the computational capacities of nodes, and the communication costs between the nodes, thus leading to a challenging optimization problem.

Acknowledgements

We are grateful to Serge Plotkin for suggesting the problem, and to Shivnath Babu, and Arvind Arasu for helpful initial discussions.

8. REFERENCES

[1] A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, pages 336–347, 2004.

[2] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[4] S. Babu et al. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, 2004.

[5] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing: Different plans for different data. In *Proc. of the 2005 Intl. Conf. on Very Large Data Bases*, pages 757–768, 2005.

[6] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. on Database Systems*, 24(2):177–228, 1999.

[7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[8] N. Dalvi, S. Sanghai, P. Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Proc. of the 2001 ACM Symp. on Principles of Database Systems*, 2001.

[9] B. Dean, M. Goemans, and J. Vondrák. Approximating the stochastic knapsack problem: The benefit of adaptivity. In *Proc. of the 2004 Annual IEEE Symp. on Foundations of Computer Science*, 2004.

[10] A. Deshpande, C. Guestrin, S. Madden, J. M. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proc. 2004 Intl. Conf. Very Large Data Bases*, 2004.

[11] O. Etzioni et al. Efficient information gathering on the internet. In *Proc. of the 1996 Annual Symp. on Foundations of Computer Science*, pages 234–243, 1996.

[12] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[13] E. Hanson. The Interval Skip List: A data structure for finding all intervals that overlap a point. In *WADS*, pages 153–164, 1991.

[14] E. Hanson. Rule condition testing and action execution in Ariel. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–58, 1992.

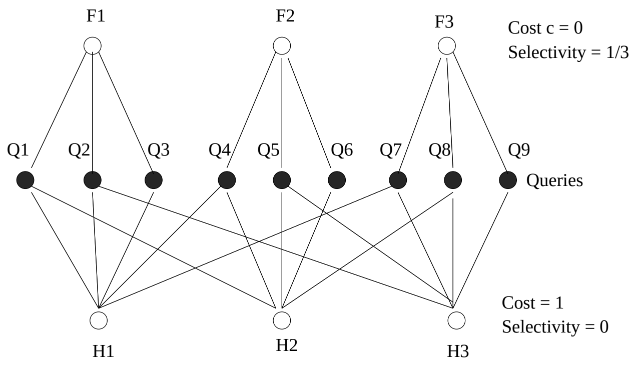

Figure 12: Construction for Theorem 3.7 when $m = 9$ and $n = 3$

[15] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 267–276, 1993.

[16] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[17] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[18] K. Munagala, U. Srivastava, and J. Widom. Optimization of continuous queries with shared expensive filters. Technical report, Stanford University, Nov. 2005. Available at http://dbpubs.stanford.edu/pub/2005-36.

[19] Open source computer vision library. http://sourceforge.net/projects/opencvlibrary/.

[20] M. Skutella and M. Uetz. Scheduling precedence-constrained jobs with stochastic processing times on parallel machines. In *Proc. of the 2001 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 589–590, 2001.

[21] R. Strom et al. Gryphon: An information flow based approach to message brokering. In *Intl. Symp. on Software Reliability Engineering*, 1998.

[22] V. Vazirani. *Approximation Algorithms*. Springer, 2001.

[23] T. Yan and H. Garcia-Molina. The SIFT information dissemination system. *ACM Trans. on Database Systems*, 24(4):529–565, 1999.

APPENDIX

A. PROOF OF THEOREM 3.6

We construct a problem instance as follows. Let $n = \sqrt{m}$. We have n filters $F_1, F_2, \ldots, F_n$ with cost $c = 0$ and selectivity $s = 1/n$. There are another n filters $H_1, \ldots, H_n$ of cost $c = 1$ and selectivity $s = 0$. There are m queries $Q_1, Q_2, \ldots, Q_m$ which are divided into n disjoint groups $G_1, G_2, \ldots, G_n$. Filters F_i and H_i are present in all queries in group G_i. In addition, we add each filter H_i to one query from each of the remaining groups. This addition is done such that any query in group G_j has at most one filter H_i ($i \neq j$) mapped to it. Note that each query in group G_i has at most 3 filters—F_i, H_i and one H_j for some $j \neq i$. Thus, $\kappa = 3$. Also, $\mu = \Theta(n)$. The construction is illustrated in Figure 12 for the case $m = 9$, and $n = 3$.

Without loss of generality, any strategy first evaluates all the filters $F_1, \ldots, F_n$ since these filters have zero cost. Since the selectivity of these filters is $1/n$, in expectation one group of queries remains unresolved at the end of this evaluation. The best adaptive strategy then evaluates the filters H_i corresponding to the unresolved groups G_i, spending unit cost for each unresolved group. Since the expected number of unresolved groups is 1, the expected cost of the best adaptive strategy is $O(1)$.

Now consider any fixed strategy. It has to choose an ordering of $H_1, \ldots, H_n$ in advance, and at best can choose a random ordering since the instance is symmetric on the indices of the filters. Fix the event that exactly one group of queries G_{j^*} is unresolved after evaluating $F_1, \ldots, F_n$. This event happens with probability $1/e$. The filter H_{j^*} appears at location $n/2$ in the ordering of $H_1, \ldots, H_n$ in expectation. All filters H_i before H_{j^*} in the ordering need to be evaluated since they are present in some query in the unresolved group G_{j^*}. Therefore, the expected cost of any fixed strategy is $\Omega(n) = \Omega(\mu)$.

```
Algorithm GreedyVariant($\delta_1, \delta_3$)
 1. $\bar{\mathcal{Q}} \leftarrow$ set of unresolved queries
 2. $\mathcal{Q}_f \leftarrow \{Q_i \in \bar{\mathcal{Q}} \mid r_i \geq \delta_3\}$, $\mathcal{Q}_t \leftarrow \bar{\mathcal{Q}} - \mathcal{Q}_f$
 3. $\mathcal{F}_l \leftarrow \{F \mid F \in Q \in \mathcal{Q}_t$ and $F$ unevaluated$\}$, $\mathcal{F}_h \leftarrow \phi$
 4. for each $Q_i \in \mathcal{Q}_f$, $\bar{r}_i \leftarrow r_i$
 5. while ($\mathcal{Q}_f$ not empty)
 6.     pRank($F_j$) $\leftarrow \frac{c_j}{\sum_{i \mid F_j \in Q_i \in \mathcal{Q}_f} \min(\bar{r}_i, 1-s_j)}$
 7.     $\mathcal{F}_h \leftarrow \mathcal{F}_h \cup \{$unevaluated filter $F_{j^*} \notin F_h$ with min pRank$\}$
 8.     for each $Q_i \in \mathcal{Q}_f$
 9.         if $F_{j^*} \in Q_i$ then $\bar{r}_i \leftarrow \bar{r}_i - (1 - s_{j^*})$
10.         if $Q_i$ is $\delta_1$-satisfied by $\mathcal{F}_h$ then $\mathcal{Q}_f \leftarrow \mathcal{Q}_f - \{Q_i\}$
11. pick all filters in $\mathcal{F}_l \cup \mathcal{F}_h$ to be evaluated
```

Figure 13: Analyzed variant of *Greedy*

B. GREEDY ALGORITHM VARIANT

In this section, we analyze a variant of *Greedy* (recall Section 4.4) known as *GreedyVariant* shown in Figure 13 (written assuming it will be invoked repeatedly by Line 5 of Algorithm *ExecuteStrategy* in Figure 1). We refer to each invocation of *GreedyVariant* as a *phase*. *GreedyVariant* differs from *Greedy* in that in every phase, it returns a set of filters to be evaluated (Line 11) rather than a single filter at a time as returned by *Greedy*.

Recall the definition of r_i from (6), the probability that query Q_i resolves to false. At any general stage of execution when some filters have already been evaluated, the current value of this probability is given by:

$$r_i = 1 - \prod_{j \mid F_j \in Q_i \wedge F_j \text{ unevaluated}} s_j \qquad (9)$$

We also define the following:

Definition B.1 (α-Satisfaction). *A set $\mathcal{F}'$ of filters α-satisfies a query Q if $\sum_{j \mid F_j \in Q \cap \mathcal{F}'} (1 - s_j) \geq \alpha$.* □

Definition B.2 (α-Cover). *An α-cover of a collection of queries $\mathcal{Q}'$ is any set $\mathcal{F}'$ of filters such that $\mathcal{F}'$ α-satisfies Q for every query $Q \in \mathcal{Q}'$. The cost of the cover is $\sum_{j \mid F_j \in \mathcal{F}'} c_j$.* □

Fix three constants $0 < \delta_1 < \delta_2 < \delta_3 < 1$ such that $\delta_3 - \delta_2 \geq 0.1$ and $\delta_2 - \delta_1 \geq 0.1$. For concreteness, take them to be 0.25, 0.35 and 0.5 respectively. δ_2 is used only in the analysis and not in the algorithm. First, we partition the set of unresolved queries $\bar{\mathcal{Q}}$ into the set of queries $\mathcal{Q}_f$ that probably resolve to false (indicated by a high value of $r_i \geq \delta_3$), and the remaining set of queries $\mathcal{Q}_t$ that probably resolve to true (Line 2). Most of the unevaluated filters in the queries in $\mathcal{Q}_t$ will need to be evaluated any way, so we pick all these filters to be evaluated (Line 3). Lines 6-7 are similar to greedily choosing the filter with the minimum pRank in *Greedy* (recall Figure 3), i.e., the filter that has the minimum ratio of cost to expected number of queries resolved. However, to calculate the expected number of queries resolved by filter F_j, we cannot simply say, as we did in *Greedy*, that filter F_j resolves every unresolved query it is part of with probability $1 - s_j$: We are choosing a set of filters to evaluate rather than a single filter, and the probability that a query is unresolved may have been substantially reduced due to filters already chosen in the current phase. Thus, we maintain a current estimate $\bar{r}_i$ of the probability that Q_i is unresolved and will eventually resolve to false. This estimate is updated whenever a filter is added to the set of filters chosen to be evaluated. Then, if $F_j \in Q_i$, F_j resolves Q_i only with probability $\min(\bar{r}_i, 1 - s_j)$ rather than with probability $1 - s_j$. Clearly, $\bar{r}_i$ is initially r_i (Line 4) and reduces by $1 - s_{j^*}$ when $F_{j^*} \in Q_i$ is chosen to be evaluated (Line 9). We continue choosing filters with the minimum pRank in this way until the probability that Q_i is unresolved has decreased sufficiently for every query $Q_i \in \mathcal{Q}_f$ (Line 10).

B.1 Analysis

Let OPT denote the expected total cost of filters evaluated by the optimal adaptive strategy. Intuitively, the proof rests on proving the following two main points:

1. In every phase, the cost of filters chosen to be evaluated is at most $O(\log^2 m)$ times OPT.
2. After every phase, the number of unresolved queries decreases by at least a constant factor, thus the number of phases is logarithmically bounded.

Combining the above two, we shall show that *GreedyVariant* is a $O(\log^2 m \log n)$ approximation. Let a *partial decision tree* be a decision tree where the leaf nodes may have unresolved queries.

LEMMA B.3. *Let T be any partial decision tree. The optimal expected cost of extending T to a complete decision tree (where all queries are resolved at any leaf node) is $\leq$ OPT.*

PROOF SKETCH. This lemma follows from the intuitive fact that even if some filters have been evaluated and some queries resolved (as given by the partial decision tree T), we can still follow the optimal strategy to resolve all the queries, skipping the evaluation of filters that had already been evaluated in T and reusing the results of evaluation from T. Clearly, the cost of doing so cannot be greater than the original optimal cost OPT, since the evaluation of some filters is skipped. $\square$

LEMMA B.4. *The cost of the filters in $\mathcal{F}_l$ is at most* $\frac{1}{1-\delta_3}$OPT.

PROOF. Any query Q in $\mathcal{Q}_t$ resolves to true with probability at least $1 - \delta_3$. Thus the optimal strategy evaluates all filters in the queries in $\mathcal{Q}_t$ with probability at least $1 - \delta_3$. Thus, $\text{OPT} \geq (1 - \delta_3) \sum_{j \mid F_j \in Q \in \mathcal{Q}_t} c_j = (1 - \delta_3) \sum_{j \mid F_j \in \mathcal{F}_l} c_j$. $\square$

LEMMA B.5. *There exists a δ_2-cover for $\mathcal{Q}_f$ with cost is at most* $\text{OPT} \cdot O(\log m)$.

PROOF. Consider the fractional e_j obtained by solving the linear program in (7). As shown in Section 4.3, the optimal value of this linear program is a lower bound on OPT. We first set $e'_j \leftarrow \min(1, 30 e_j \log m)$. Clearly, the cost of the resulting solution $\sum_{j=1}^{n} c_j e'_j \leq 30 \log m \cdot \text{OPT}$. Let $\mathcal{F}_A$ be the set of filters F_j for which $e'_j = 1$, and $\mathcal{F}_B$ be the set of remaining filters, i.e., $\mathcal{F}_B = \mathcal{F} - \mathcal{F}_A$.

If for some query $Q_i \in \mathcal{Q}_f$, $\sum_{j \mid F_j \in \mathcal{F}_A \cap Q_i} (1 - s_j) \geq \delta_2$, then Q_i is already δ_2-satisfied by $\mathcal{F}_A$ alone. Now consider the remaining queries $Q_i \in \mathcal{Q}_f$. For these queries, we have:

$$\sum_{j \mid F_j \in \mathcal{F}_A \cap Q_i} (1 - s_j) < \delta_2 \tag{10}$$

By the first constraint in the linear program in (7), and because $r_i \geq \delta_3$, and for all j, $e_j \leq 1$, we have:

$$\sum_{j \mid F_j \in \mathcal{F}_A \cap Q_i} (1 - s_j) + \sum_{j \mid F_j \in \mathcal{F}_B \cap Q_i} (1 - s_j) e_j \geq \delta_3$$

Combining the above with (10), and because $\delta_3 - \delta_2 \geq 0.1$, we get $\sum_{j \mid F_j \in \mathcal{F}_B \cap Q_i} (1 - s_j) e_j \geq 0.1$. Finally, since for filters $F_j \in \mathcal{F}_B$ $e'_j = 30 e_j \log m$, we get:

$$\sum_{j \mid F_j \in \mathcal{F}_B \cap Q_i} (1 - s_j) e'_j > 3 \log m \tag{11}$$

For filters in $\mathcal{F}_B$, we perform randomized rounding [22] of e'_j, setting e'_j to 1 with probability equal to e'_j and to 0 otherwise. The expected cost of the resulting solution $\sum_{j=1}^{n} c_j e'_j$ remains the same, i.e., $\leq 30 \log m \cdot \text{OPT}$. By Chernoff bound [17] and using (11), for any $Q_i \in \mathcal{Q}_f$ that was not already δ_2-satisfied by $\mathcal{F}_A$, the probability that after rounding $\sum_{j \mid F_j \in \mathcal{F}_B \cap Q_i} (1 - s_j) e'_j < 1$ is $O(1/m^2)$. Let $\mathcal{F}_C = \mathcal{F}_A \cup \{F_j \in \mathcal{F}_B \mid e'_j \text{ was rounded to } 1\}$. Thus all queries in $\mathcal{Q}_f$ are δ_2-satisfied by $\mathcal{F}_C$ with probability at least $1 - 1/m$. Since the cost of $\mathcal{F}_C$ is at most $\text{OPT} \cdot O(\log m)$, it is the required δ_2-cover. $\square$

LEMMA B.6. *At the end of the first phase, the cost of the filters in $\mathcal{F}_h$ is at most $O(\log m)$ times the cost of the optimal δ_2-cover.*

PROOF. Let C be the cost of the optimal δ_2-cover. The sum $U = \sum_{Q_i \in \mathcal{Q}_f} \bar{r}_i$ is at most m initially. By the same argument as the proof for greedy set cover [22], it can be shown that the greedy step in Line 6 (Figure 13) ensures that every time filters of cost at most C are added to $\mathcal{F}_h$, U reduces by at least a factor half. When U falls below $\delta_3 - \delta_1$, the loop in lines 5-10 terminates since $\mathcal{F}_h$ is now a δ_1-cover for the queries in $\mathcal{Q}_f$. This is because $U \leq \delta_3 - \delta_1$ implies that for each query $Q_i \in \mathcal{Q}_f$, $\bar{r}_i \leq \delta_3 - \delta_1$, i.e., $\bar{r}_i$ has decreased by at least δ_1 from its initial value, thereby implying that $\mathcal{F}_h$ δ_1-satisfies Q_i. Thus, U needs to be reduced by a factor half at most $\log(\frac{m}{\delta_3 - \delta_1})$ times. Since each reduction adds filters of cost at most C to $\mathcal{F}_h$, the cost of $\mathcal{F}_h$ at the end is at most $O(C \log m)$. $\square$

LEMMA B.7. *The cost of filters evaluated in the first phase is at most* $\text{OPT} \cdot O(\log^2 m)$.

PROOF. From Lemmas B.5 and B.6, the cost of filters in $\mathcal{F}_h$ is at most $\text{OPT} \cdot O(\log^2 m)$. Combining with Lemma B.4, we get the result. $\square$

LEMMA B.8. *The expected number of unresolved queries at the end of the first phase is at most $e^{-\delta_1} \mathcal{Q}$.*

PROOF. All the queries in $\mathcal{Q}_t$ are definitely resolved after the first phase since all their unevaluated filters ($\mathcal{F}_l$) are chosen to be evaluated. Now consider the queries in $\mathcal{Q}_f$. Since $\mathcal{F}_h$ is a δ_1-cover for $\mathcal{Q}_f$, for every query $Q_i \in \mathcal{Q}_f$, we have $\sum_{j \mid F_j \in Q_i \cap \mathcal{F}_h} (1 - s_j) \geq \delta_1$. Thus the probability that Q_i is left unresolved after evaluating all filters in $\mathcal{F}_h$ is at most $\prod_{j \mid F_j \in Q_i \cap \mathcal{F}_h} s_j \leq e^{-\sum_{j \mid F_j \in Q_i \cap \mathcal{F}_h} (1 - s_j)} \leq e^{-\delta_1}$. Thus the number of unresolved queries at the end of the first phase is at most $e^{-\delta_1} \mathcal{Q}$. $\square$

THEOREM B.9. *GreedyVariant is a $O(\log^2 m \log n)$ approximation to the optimal adaptive solution.*

PROOF. By Lemma B.8, the expected number of unresolved queries reduces by at least a constant factor. Since this reduction is independent of the queries unresolved at the beginning, Lemma B.8 can be applied to any phase. Suppose $c_{\max}$ and $c_{\min}$ are the maximum and minimum filter costs. After $\log(nc_{\max}/c_{\min})$ phases, the expected number of unresolved queries is at most $O(\frac{c_{\min}}{nc_{\max}})$. Then, even if we apply all filters non-adaptively, we could apply at most n filters of cost at most $c_{\max}$. The expected cost of this application is therefore at most $O(c_{\min})$, which can be ignored since OPT is also at least c_{min}.

By Lemma B.3, the expected cost of the optimal solution given a partial decision tree is at most OPT. Combining with Lemma B.7, we obtain that for every phase, the expected cost is at most $O(\mathrm{OPT} \cdot \log^2 m)$. Therefore, *GreedyVariant* is a $O(\log^2 m \cdot \log \frac{nc_{\max}}{c_{\min}})$ approximation to OPT. If $c_{\max}/c_{\min}$ is polynomially bounded in n, the approximation ratio is $O(\log^2 m \log n)$.

To get an approximation ratio independent of the costs of the filters in the general case, we group the filters based on cost into powers of n^3 (assuming by scaling that $c_{min} = 1$). There are at most n groups, since there are n filters. We then separately bound the contribution of each cost group to the cost of *GreedyVariant*, thereby obtaining a final approximation ratio of $O(\log^2 m \log n)$. The details are omitted. $\square$