The Pipelined Set Cover Problem

Kamesh Munagala*, Shivnath Babu**, Rajeev Motwani***, and Jennifer Widom†

Abstract. A classical problem in query optimization is to find the optimal ordering of a set of possibly correlated selections. We provide an abstraction of this problem as a generalization of set cover called *pipelined set cover*, where the sets are applied sequentially to the elements to be covered and the elements covered at each stage are discarded. We show that several natural heuristics for this NP-hard problem, such as the greedy set-cover heuristic and a local-search heuristic, can be analyzed using a linear-programming framework. These heuristics lead to efficient algorithms for pipelined set cover that can be applied to order possibly correlated selections in conventional database systems as well as data-stream processing systems. We use our linear-programming framework to show that the greedy and local-search algorithms are 4-approximations for pipelined set cover. We extend our analysis to minimize the l_p-norm of the costs paid by the sets, where $p \geq 2$ is an integer, to examine the improvement in performance when the total cost has increasing contribution from initial sets in the pipeline. Finally, we consider the *online* version of pipelined set cover and present a competitive algorithm with a logarithmic performance guarantee. Our analysis framework may be applicable to other problems in query optimization where it is important to account for correlations.

1 Motivation

A common operation in database query processing is to find the subset of records in a relation that satisfy a given set of selection conditions. To execute this operation efficiently, a query processor prefers to determine the optimal order in which to evaluate the individual selection conditions, so we call this operation *pipelined filters* [2, 4, 12, 18]. Optimality in pipelined filters is usually with respect to minimizing the total processing time [4, 12].

For example, consider a relation `packets`, where each record contains the header and an initial part of the payload of network packets logged by a network

* Computer Science Department, Duke University. Part of this work was done while the author was at Stanford University supported by NIH 1HFZ465. kamesh@cs.duke.edu

** Computer Science Department, Stanford University. Supported by NSF under grants IIS-0118173 and IIS-9817799. shivnath@cs.stanford.edu

*** Computer Science Department, Stanford University. Supported in part by NSF Grants IIS-0118173 and EIA-0137761, NSF ITR Award Number 0331640, and grants from Microsoft and Veritas. rajeev@cs.stanford.edu

† Computer Science Department, Stanford University. Supported by NSF under grants IIS-0118173 and IIS-9817799. widom@cs.stanford.edu

router. Suppose a query needs to compute the subset of `packets` where each record r in the result satisfies the following three conditions:

1. p_1: `destPort` = 80, where `destPort` is the destination port field of r.
2. p_2: *domain*(`destAddr`) = "yahoo.com", where `destAddr` is the destination address field of r, and *domain* is a function that returns the Internet domain name of an address passed as input.
3. p_3: The payload of r contains the regular expression "^[^\\n]*HTTP/1.*" [9].

A query processor might use three selection operators on `packets`, denoted O_{p_1}, O_{p_2}, and O_{p_3}, to evaluate these three conditions respectively. In this case the query processor might choose to apply O_{p_1} first on each record in `packets` so that O_{p_2} and O_{p_3} need only process records that are selected by O_{p_1}. Since both O_{p_2} and O_{p_3} involve complex functions, applying either of them before O_{p_1} could increase the total processing time by orders of magnitude. Further, the query processor may choose to process O_{p_2} before O_{p_3}, since packets selected by O_{p_1} are also likely to be selected by O_{p_3}. As this example shows, it is important to choose a good, if not the optimal, order for applying the selection operators on the records of the input relation. Also note that both the expected fraction of records selected (the *selectivity*) and the record-processing time of each selection operator must be taken into account.

Suppose the selection conditions are independent; that is, the selectivity s of any operator O among the records that O processes is independent of the operators that appear before O in the order. Under this assumption, computing the order that minimizes total processing time is easy: We simply order the operators in nonincreasing ratio of $1 - s$ and the record-processing time. Most previous work on the selection-ordering problem and on related problems make the independence assumption and use this ordering technique [4, 12, 18, 23].

The independence assumption reduces the complexity of finding the optimal order, but it is often violated in practice [6, 25]. It can be shown that when the independence assumption does not hold, the total processing time can be $O(n)$ times worse than optimal when n operators are ordered in nonincreasing ratio of $1 - s$ and the record-processing time. Without the independence assumption, the problem is NP-hard. Previous work [17, 22, 23] on ordering of dependent (correlated) operators either uses exhaustive search—which requires selectivity estimates for an exponentially large number of operator subsequences—or proposes simple heuristics with no provable performance guarantees for the solution obtained. As databases are being extended to manage complex data types such as multimedia and XML, the use of expensive selection conditions are becoming frequent, making the problem of ordering dependent selections even more important [4, 12]. The pipelined filters problem also captures restricted types of relational joins and combinations of joins and selections; see [2].

Pipelined filters can be formulated as a generalization of the classical set cover problem [13, 15]: The relation represents the elements to be covered, and each selection operator is a set which drops (or covers) a certain number of records (or elements). The sets are applied sequentially to the elements to be covered, with each set removing the elements that it covers from further processing; the

cost of applying a set depends linearly on the number of elements that are still not covered when the set is applied. The solution desired is an ordering of the sets that minimizes the total cost of applying the sets sequentially. We call this problem *pipelined set cover*, the key difference with classical set cover being the cost function. The mapping from pipelined filters to pipelined set cover is straightforward: the operators map to the sets, and the operator ordering, or *pipeline*, maps to the ordering of the sets.

2 Our Contribution

Pipelined set cover has been considered previously in a non-database context by Feige et al. [11] and by Cohen and Kaplan [7]. They show that the uniform cost version of this problem is MAX-SNP hard and develop a greedy 4-approximation algorithm for the uniform cost version. In addition to showing the application of pipelined set cover to classical optimization problems in database and data-stream processing, we extend previous work significantly in this paper, as follows.

2.1 Approximation Algorithms for Pipelined Set Cover

We provide two approximation algorithms for pipelined set cover, one based on the greedy heuristic for classical set cover and another based on an intuitive local-search heuristic. (In separate work we have implemented both algorithms efficiently in a data-stream processing system [2].) Using a different and more general analysis technique from previous work, we show that both these algorithms are 4-approximations, even when the linear cost function depends on the set. This relatively new analysis technique is based on formulating the worst-case performance of the algorithms as linear programs. (This technique was first used by Jain, Mahdian, and Saberi to analyze the performance of a dual-fitting algorithm for facility location [14].) This technique has several advantages. In addition to bounding the approximation ratio, the linear program can be used to analyze running time, e.g., the rate of convergence of the local search heuristic. The linear program gives new insights about the approximation algorithms, with strong implications for query optimization: The bound on approximation depends on the number of sets (operators) n; for $n \leq 20$, this bound ≤ 2.35. Furthermore, this technique can be used to analyze other algorithms for pipelined set cover, including a simple *move-to-front* algorithm which can be implemented very efficiently.

We can view our problem as minimizing the l_1-norm of the vector of the number of elements processed (or the cost paid) by each set. The classical set cover problem can be viewed as minimizing the l_0-norm[1]—it gives a cost to any set that is independent of the number of elements it processes, so long as that set processes at least one element. For set cover, the performance of the greedy algorithm is logarithmic [13, 15, 24], and this approximation factor is optimal [10], assuming $P \neq NP$. The approximation ratio improves to 4 for our l_1-norm formulation, where the cost of each set is weighted by the number of elements it

[1] Of course, technically speaking, there is no such norm. However, we can adopt the view that the set cover objective function is minimizing a Hamming measure, which is sometimes treated as a substitute for the l_0-norm [8].

processes. A natural question to ask is what happens to the approximation ratio when the goal is to minimize the l_p-norm of the costs paid by the sets, for integers $p \geq 2$. As p increases, this formulation gives increasing weight to sets at the start of the pipeline that process more elements. The intuition is that the performance of the greedy algorithm should improve with increasing p, and it should reach the optimal solution when we are minimizing the l_∞-norm. Since the objective function is nonlinear, linear programming techniques fail to apply. We develop a Lagrangian-relaxation analysis technique for $p \geq 2$ to show that the approximation ratio of the greedy algorithm is $9^{\frac{1}{p}}$ when the processing costs are uniform (independent of the set), and that local search is a $4^{\frac{1}{p}}$-approximation when the processing costs are nonuniform. The improvement in performance of greedy confirms the intuition that as we skew the total cost in favor of the initial sets chosen, greedy's performance should improve for uniform processing costs.

2.2 Online Pipelined Set Cover

Our original motivation for defining and analyzing pipelined set cover came from our work on processing pipelined filters in a data-stream query processor [2]. A stream, as opposed to a relation, is a continuous unbounded flow of records arriving at a stream-processing system [1]. Example streams include network packets, stock tickers, and sensor observations. Pipelined filters are common in stream processing, e.g., `packets` may be a stream in our example query introduced at the beginning of this section. Another common example of pipelined filters in stream processing is a join of a stream S with a set of relations $R_1, R_2, \ldots, R_k$: For each record s arriving in S, we need to find $R_i^{'} \subseteq R_i$, $1 \leq i \leq k$, such that each record $r_i \in R_i^{'}$ satisfies $r_i.A = s.A$ where A is a field that is common among $S, R_1, R_2, \ldots, R_k$. (We have defined a restricted version of the problem for succinctness [2].) The join output for s is the set of concatenated records $s \cdot r_1 \cdot r_2 \cdots r_k$ for each combination of $r_1 \in R_1^{'}, r_2 \in R_2^{'}, \ldots, r_k \in R_k^{'}$. If any of the $R_i^{'}$'s are empty, then s produces no join output and we say that s is *dropped*. For processing the join efficiently, we must order $R_1, R_2, \ldots, R_k$ for computing $R_1^{'}, R_2^{'}, \ldots, R_k^{'}$ such that records in S that get dropped eventually consume minimal processing time. Note that the processing required for records that are not dropped is independent of the ordering.

Pipelined filters over data streams motivate the *online* version of pipelined set cover. In online pipelined set cover, some number of elements arrive at each time step. Our online algorithm has to choose an ordering of the sets in advance at every time step, and process the incoming elements according to this ordering. The performance of our online algorithm is compared against the performance of the best possible offline algorithm that does not change its ordering for the entire course of the request sequence. For online pipelined set cover, we present an $O(\log n)$ competitive algorithm for the uniform cost case, where n is the number of sets. This algorithm can be extended to an $O(\log n + \log \frac{c_{\max}}{c_{\min}})$ competitive algorithm for the nonuniform cost case, where $c_{\max}$ is the largest per-element processing cost among all sets, and $c_{\min}$ is the smallest such cost.

2.3 Implementation

In a companion paper [2], we describe our implementation of some of the approximation algorithms for pipelined set cover proposed in this paper for optimizing pipelined filters in a Data Stream Management System [20]. We propose and evaluate techniques to compute selectivity estimates of operator subsequences needed by our approximation algorithms with minimal overhead, as part of query processing itself. While previous work [22, 23] on provably good algorithms for dependent pipelined filters required selectivity estimates for an exponentially large number of operator subsequences, our algorithms require only $O(n^2)$ estimates. (In a conventional database setting, a sample of records from the input relation can be used to estimate these selectivities with low overhead [3, 19].) Furthermore, because data and arrival characteristics of streams can change over time, in [2] we introduce adaptive versions of the algorithms that modify orderings as statistics change, converging on the static solution when statistics do not change. The need to adapt forces us to optimize the pipeline continuously, which motivates the low-overhead heuristics we consider such as greedy and local search.

2.4 Organization

The rest of the paper is organized as follows:

- Section 3 presents the formal problem statement. In Section 4, we introduce and use our linear-programming framework to analyze the greedy set cover algorithm applied to pipelined set cover.
- We move on to local search heuristics in Section 5, showing that our analysis technique carries over to this case, leading to bounds not only on the approximation ratio, but also on the rate of convergence. In the full version [21] we describe simpler implementations of the local search algorithm using limited amount of state, and analyze the resulting performance degradation.
- In Section 6, we present a Lagrangian-relaxation method for analyzing the performance of the greedy and local search algorithms when we optimize the l_p-norm of the cost paid by the sets. The detailed analysis of the local search heuristic, showing that it is a $4^{\frac{1}{p}}$-approximation, is relegated to the full version [21].
- We finally present the online algorithm and its analysis in Section 7.

We omit a separate section on related work because of space constraints. However, related work is referenced appropriately in all sections in the paper.

3 Preliminaries

We are given a set cover instance with n elements denoted $U = \{e_1, e_2, \ldots, e_n\}$, and a collection of sets $A = \{S_1, S_2, \ldots, S_k\}$. Set S_i has a *processing cost* per

unit element of c_i. Let $\pi(A)$ denote the set of all possible orderings (permutations) of the sets $S_1, S_2, \ldots, S_k$. The goal is to choose an ordering of the sets, $(S_{p_1}, S_{p_2}, \ldots, S_{p_k}) \in \pi(A)$, so as to minimize the *pipelined cost*:

$$\sum_{i=1}^{k} c_{p_i} |U - \cup_{j=1}^{i-1} S_{p_j}|$$

This cost reflects the cost of a sequence of selection operations in a relational schema, and the goal is to find the optimal such sequence of operations. If the c_i's are equal, we call the instance *uniform*. Note that in this formulation, each element can have a weight associated with it, so that the size of a set is simply the sum of the weights of the elements. We call this problem the *Pipelined Set Cover* problem. Feige et al [11] show that this problem does not admit to better than a 4 approximation unless $P = NP$.

4 Greedy Algorithm

We now analyze the greedy set cover algorithm for this problem. At step i, let n denote the total number (weight) of uncovered elements. Let n_j be the number (respectively weight) of uncovered elements that get covered by set j. We choose the set that minimizes the *cost ratio* $\frac{c_j n}{n_j}$. The uniform cost version of this algorithm was analyzed in [11] and in [7]. We provide a different analysis that handles nonuniform costs which are important in databases since different operators in a pipeline can have different costs.

We can formulate the worst-case performance of greedy as a linear program. Consider any optimal solution whose sets are pipelined $\{O_1, O_2, \ldots, O_k\}$. Suppose we scale down the problem size by scaling down the weights of the elements so that the pipelined cost of the optimal solution is 1. Without loss of generality, assume that the sets in the optimal solution are disjoint, else O_i denotes the residual part of the set after the application of $O_1, \ldots, O_{i-1}$. We denote $|O_i|$ by a_i, and the processing cost of O_i by c_{oi}. The cost of the optimal solution is:

$$OPT = \sum_{i=1}^{k} \left(a_i \cdot \sum_{s=1}^{i} c_{os} \right)$$

Let us denote the sets chosen by greedy as $\{G_1, G_2, \ldots, G_k\}$. Again, assume without loss of generality that they are disjoint, else G_i denotes the residual part of the set after application of $G_1, \ldots, G_{i-1}$. Let $b_{ij} = |O_i \cap G_j|$, so that $a_i = \sum_{j=1}^{k} b_{ij}$. Let the processing cost of G_j be c_{gj}. The cost of greedy is:

$$GREEDY = \sum_{j=1}^{k} \left(\sum_{r=1}^{j} c_{gr} \cdot \sum_{s=1}^{k} b_{sj} \right)$$

Since greedy maximizes the weight of uncovered elements at each time, we have for every stage j of greedy and every set O_i of OPT, the cost ratio of the residual part of O_i after $j-1$ stages of greedy must be at least the cost ratio of G_j. This gives us:

$$\frac{\sum_{s=1}^{k} b_{sj}}{c_{gj}} \geq \frac{\sum_{r=j}^{k} b_{ir}}{c_{oi}}$$

We can now formulate the worst possible approximation ratio that greedy can achieve as the following linear program:

$$\text{maximize} \quad \sum_{j=1}^{k} \left(\sum_{r=1}^{j} c_{gr} \cdot \sum_{s=1}^{k} b_{sj} \right), \text{ subject to:}$$

$$\begin{array}{rl} \sum_{i=1}^{k} (\sum_{s=1}^{i} c_{os} \cdot \sum_{r=1}^{k} b_{ir}) \leq 1 & \\ c_{gj} \cdot \sum_{r=j}^{k} b_{ir} \leq c_{oi} \cdot \sum_{s=1}^{k} b_{sj} & \forall i, j \\ b_{ij} \geq 0 & \forall i, j \end{array}$$

For all the processing costs being uniform, we can compute the precise worst-case ratios. For $k = 20$, the worst-case ratio is 2.35. For $k = 100$, this climbs to 2.61, and for $k = 200$, this is around 2.80.

The upper bound on this approximation ratio for any possible value of the processing costs would be an upper bound on the worst-case performance of the greedy algorithm. For this purpose, we take the dual of this linear program:

$$\text{minimize} \quad \gamma, \text{ subject to:}$$

$$\begin{array}{rl} \gamma \sum_{s=1}^{i} c_{os} + \sum_{r=1}^{j} \alpha_{ir} c_{gr} \geq \sum_{s=1}^{k} \alpha_{sj} c_{os} + \sum_{r=1}^{j} c_{gr} & \forall i, j \\ \alpha_{ij} \geq 0 & \forall i, j \end{array}$$

By linear programming duality, for any choice of the processing costs, the objective function value for *any* feasible solution for the dual problem would be an upper bound on the optimal solution to the primal problem for those processing costs. The maximum of this value over all possible choices of the processing costs would therefore be a bound on the worst-case approximation ratio for the greedy algorithm.

Fix a choice of the processing costs. We show that there is a feasible solution to the dual with $\gamma = 4$. Let $P_i = \sum_{s=1}^{i} c_{os}$ and $Q_j = \sum_{r=1}^{j} c_{gr}$. We set $\alpha_{ij} = 2$ if $P_i \leq \frac{Q_j}{2}$ and 0 otherwise. For any i, j, if $P_i \leq \frac{Q_j}{2}$, then $\sum_{r=1}^{j} \alpha_{ir} c_{gr} \geq 2(Q_j - 2P_i) = 2Q_j - 4P_i$. This implies $4 \sum_{s=1}^{i} c_{os} + \sum_{r=1}^{j} \alpha_{ir} \geq 4P_i + 2Q_j - 4P_i = 2Q_j$. In the other case, if $P_i > \frac{Q_j}{2}$, then $\sum_{r=1}^{j} \alpha_{ir} c_{gr} = 0$, implying $4 \sum_{s=1}^{i} c_{os} + \sum_{s=1}^{j} \alpha_{is} = 4P_i \geq 2Q_j$. We also have for all j, $\sum_{s=1}^{k} \alpha_{sj} c_{os} \leq 2\frac{Q_j}{2} = Q_j$, implying $\sum_{r=1}^{j} c_{gr} + \sum_{s=1}^{k} \alpha_{sj} c_{os} \leq 2Q_j$. We have for all i, j:

$$4 \sum_{s=1}^{i} c_{os} + \sum_{r=1}^{j} \alpha_{ir} c_{gr} \geq \sum_{s=1}^{k} \alpha_{sj} c_{os} + \sum_{r=1}^{j} c_{gr}$$

Our choice of α_{ij} forms a feasible solution for the dual with an objective value of $\gamma = 4$ for every choice of values for the processing costs. Therefore, the greedy algorithm always has an approximation ratio of at most 4.

Theorem 1. *The greedy algorithm is a 4-approximation to the pipelined set cover problem.*

4.1 Approximate Greedy Algorithm

At every step, suppose the greedy algorithm does not pick the best set, but any set that covers a fraction $\sigma \leq 1$ times as many elements covered by the best possible set (for the case with uniform processing costs). We can express the worst case of this algorithm as a similar linear program, and derive an approximation ratio of $\frac{4}{\sigma}$. The argument generalizes naturally to the case with arbitrary processing costs and yields the same approximation ratio. The parameter σ was useful in our implementation [2] to avoid pipeline *thrashing*: we do not want to change the current ordering unless we detect a set (operator) that covers a significant fraction more of the elements at some earlier stage in the pipeline than the current set (operator) at that position.

5 Local Search

We will now analyze the following local search heuristic: We start with an arbitrary complete pipeline. We insert a set into the pipeline (in other words, move it up the pipeline) if it improves the cost of the solution. We repeat till no insert operation improves the cost of the solution. As before, we denote the residual part of the i^{th} set in the optimal solution by O_i, and the residual part of the j^{th} set in the current solution by L_j.

Let the processing cost of set O_i be c_{oi} and the cost of set L_j be c_{lj}. Let $Q_j = \sum_{r=1}^{j} c_{lr}$ and $P_i = \sum_{s=1}^{i} c_{os}$. Define $Q_0 = 0$.

We will show that local search is a $(4+\epsilon)$-approximation. As before, we can formulate the problem as a linear program. The constraint to enforce is that inserting O_i at position j does not help the current solution:

$$\sum_{r=1}^{k}\left(Q_r \sum_{s=1}^{k} b_{sr}\right) \leq \sum_{r=1}^{j-1}\left(Q_r \sum_{s=1}^{k} b_{sr}\right) + \sum_{r=j}^{k}\left((c_{oi}+Q_{j-1})b_{ir} + (c_{oi}+Q_r)\sum_{\substack{s=1 \\ s\neq i}}^{k} b_{sr}\right)$$

This simplifies to:

$$\sum_{r=j}^{k}(Q_r - Q_{j-1})b_{ir} \leq c_{oi}\sum_{r=j}^{k}\sum_{s=1}^{k} b_{sr}$$

We can now write the linear program as:

$$\text{maximize} \quad \sum_{j=1}^{k}\left(Q_j \sum_{i=1}^{k} b_{ij}\right)$$

subject to:

$$\begin{aligned} \textstyle\sum_{r=j}^{k}(Q_r - Q_{j-1})b_{ir} &\leq c_{oi}\textstyle\sum_{r=j}^{k}\sum_{s=1}^{k} b_{sr} && \forall i,j \\ \textstyle\sum_{i=1}^{k}(P_i \sum_{j=1}^{k} b_{ij}) &\leq 1 \\ b_{ij} &\geq 0 && \forall i,j \end{aligned}$$

We now take the dual of this program:

$$\text{minimize} \quad \gamma$$

subject to:

$$\begin{array}{rl} \gamma P_i + \sum_{r=1}^{j}(Q_j - Q_{r-1})\alpha_{ir} \geq Q_j + \sum_{r=1}^{j}\sum_{s=1}^{k}\alpha_{sr}c_{os} & \forall i,j \\ \alpha_{ij}, \gamma \geq 0 & \forall i,j \end{array}$$

For every i, let $z(i)$ denote the first value j such that $Q_j \geq 2P_i$. We set $\alpha_{iz(i)} = 2$. For every other j, we set $\alpha_{ij} = 0$. Therefore, $\sum_{r=1}^{j}(Q_j - Q_{r-1})\alpha_{ir} \geq 2(Q_j - 2P_i)$ for all j. In addition, $\sum_{r=1}^{j}\sum_{s=1}^{k}\alpha_{sr}c_{os} \leq Q_j$ for all j. Therefore, $\gamma = 4$ is feasible for the dual.

5.1 Convergence Analysis

We now examine the number of iterations required by local search. Suppose the current solution is an M-approximation to the optimal solution. We will compute the smallest amount by which the approximation factor improves with the best possible local move. This can be formulated as the following linear program (note that M is not a variable):

$$\text{minimize} \quad A$$

subject to:

$$\begin{array}{rl} \sum_{r=j}^{k}(Q_r - Q_{j-1})b_{ir} \leq A + c_{oi}\sum_{r=j}^{k}\sum_{s=1}^{k}b_{sr} & \forall i,j \\ \sum_{j=1}^{k}(\sum_{j=1}^{k}Q_j\sum_{i=1}^{k}b_{ij}) \geq M & \\ \sum_{i=1}^{k}(P_i\sum_{j=1}^{k}b_{ij}) \leq 1 & \\ b_{ij} \geq 0 & \forall i,j \end{array}$$

We take the dual as before, and set $\alpha_{iz(i)} = \frac{1}{k}$. This yields a dual value of $\frac{M-4}{2k}$. Therefore, the reduction in approximation ratio is $\frac{M-4}{2k}$.

Fix any $\epsilon > 0$. Suppose we stop when we achieve an approximation ratio of $(4+\epsilon)$. It is easy to start with a solution of cost at most $nk \cdot OPT$. Therefore, the number of iterations is at most $2k \log \frac{nk}{\epsilon}$. We have therefore shown:

Theorem 2. *The local search heuristic produces a $(4+\epsilon)$-approximation in $O(k \log \frac{nk}{\epsilon})$ operations.*

We provide simpler implementations of the local search algorithm using limited amount of state, and show the degradation in performance in the full version.

6 Extensions to Higher l_p Norms

Consider the problem of finding a pipeline $S_{t_1}, S_{t_2}, \ldots, S_{t_k}$ which minimizes the l_p-norm ($p \geq 1$):

$$\left(\sum_{i=1}^{k}(c_{t_i}|U-\cup_{j=1}^{i-1}S_{t_j}|)^p\right)^{\frac{1}{p}}$$

We will analyze the greedy and local search algorithms for this cost function. Note that the greedy algorithm gives a $O(\log n)$-approximation for classical set cover which be viewed as seeking to minimize an l_0-norm (see Footnote 1), as the cost of a set is independent of the number of elements it processes, as long as it is nonzero. When $p = 1$, the cost of a set is weighted by the number of elements it processes (so that initial sets in the ordering are more important), and this ratio goes down to 4. Clearly, the greedy algorithm could have an approximation ratio as bad as $\frac{c_{\max}}{c_{\min}}$ for the l_∞-norm, as the cost of the first set chosen dominates. We will consider the *uniform* case where $c_i = 1$ for all i, and show that for integers $p \geq 2$ the approximation ratio for the greedy algorithm is at most $9^{\frac{1}{p}}$, using a Lagrangian-relaxation analysis. This proves the intuitive claim that the performance of the greedy algorithm improves as we skew the objective function more and more in favor of the initial sets in the ordering. The analysis can be tightened by a better choice of constants; our only goal here is to show that the performance ratio improves dramatically with increasing p. We leave the problem of computing the approximation ratio for arbitrary monotone cost functions as an interesting open problem.

We will analyze the local search heuristic in the full version and show that it is a $4^{\frac{1}{p}}$-approximation minimizing the l_p-norm for the nonuniform case when $p \geq 1$ is an integer.

Consider the objective function of the form $\sum_{i=1}^{k}|U-\cup_{j=1}^{i-1}S_{t_j}|^p$. For this case, the worst-case performance of greedy can be formulated as a nonlinear program:

$$\text{maximize} \quad \sum_{j=1}^{k}\left(\sum_{i=1}^{k}\sum_{r=j}^{k}b_{ir}\right)^p$$

subject to:

$$\begin{aligned}\textstyle\sum_{i=1}^{k}(\sum_{s=i}^{k}\sum_{r=1}^{k}b_{sr})^p &= 1 \\ \textstyle\sum_{r=j}^{k}b_{ir} &\leq \textstyle\sum_{s=1}^{k}b_{sj} \qquad \forall i,j \\ b_{ij} &\geq 0 \qquad \forall i,j\end{aligned}$$

We now write the Lagrangian relaxation of this formulation, using nonnegative α_{ij} and γ:

$$\sum_{j=1}^{k}\left(\sum_{i=1}^{k}\sum_{r=j}^{k}b_{ir}\right)^p+\gamma\left(1-\sum_{i=1}^{k}\left(\sum_{s=i}^{k}\sum_{r=1}^{k}b_{sr}\right)^p\right)+\sum_{i,j}\alpha_{ij}\left(\sum_{s=1}^{k}b_{sj}-\sum_{r=j}^{k}b_{ir}\right)$$

Given any setting of the b_{ij}, we will find a setting for the α_{ij} and γ so that the Lagrangian is at most 9. We will use a simple method for setting the variables – we will ensure that the coefficients for all the b_{ij} variables in the Lagrangian are negative or zero. If this were true for $\gamma = 9$, we would be able to easily establish

a 9-approximation. Let $L_{ij} = \sum_{r=j+1}^{k} \sum_{s=i}^{k} b_{sr}$. We set $\alpha_{ij} = 3pL_{ij}^{p-1}$ if $i \leq \frac{j}{3}$, and 0 otherwise.

We will now compute the coefficient of a general term of the form $b_{i_1 j_1}^{p_1} b_{i_2 j_2}^{p_2} \ldots b_{i_t j_t}^{p_t}$, where $p_1 + p_2 + \ldots + p_t = p$. Let $i = \min(i_1, i_2, \ldots, i_t)$ and $j = \min(j_1, j_2, \ldots, j_t)$. Let b_{im_i} and $b_{m_j j}$ be the relevant terms. The relevant nonzero terms are present in the following sum; note that there are more terms, but these would make the sum only smaller.

$$\sum_{j=1}^{k} \left(\sum_{i=1}^{k} \sum_{r=j}^{k} b_{ir} \right)^{p} - \gamma \left(\sum_{i=1}^{k} \left(\sum_{s=i}^{k} \sum_{r=1}^{k} b_{sr} \right)^{p} \right) - \sum_{r=1}^{j} \alpha_{ir} b_{im_i} + \sum_{s=1}^{k} \alpha_{sj} b_{m_j j}$$

Let $H = \frac{p!}{p_1! p_2! \cdots p_t!}$. The coefficient from the first two terms in the summation is $H(j - 9i)$. Let $n(j)$ denote the sum of the exponents of the terms in the product $b_{i_1 j_1}^{p_1} b_{i_2 j_2}^{p_2} \ldots b_{i_t j_t}^{p_t}$ of the form b_{sj}. The coefficient of the product depends on whether $n_j = 1$ or not. Let $n(i) \geq 1$ denote the total power of terms in the product of the form b_{ir}. We consider four cases:

Case 1: If $n(j) = 1$ and $i \leq j/3$, the coefficient is: $-3H \times n(i) \times \max(j - 3i - 1, 0) + 3H \times \frac{j}{3} \leq H(9i - 2j + 3) \leq H(9i - j)$, as $j \geq 3$ in this case.
Case 2: If $n(j) = 1$ and $i > j/3$, the coefficient is: $3H \times \frac{j}{3} \leq H(3i) \leq H(9i - j)$.
Case 3: If $n(j) > 1$ and $i \leq j/3$, the coefficient is: $-3H \times n(i) \times \max(j - 3i - 1, 0) \leq H(9i + 3 - 3j) \leq H(9i - j)$ since $j \geq 3$ for this case.
Case 4: If $n(j) > 1$ and $i > j/3$, the coefficient is $0 \leq H(9i - j)$.

Therefore, in all cases, the net coefficient of $b_{i_1 j_1}^{p_1} b_{i_2 j_2}^{p_2} \ldots b_{i_t j_t}^{p_t}$ is negative or zero, showing that the primal problem has objective value at most 9. We have therefore proved the following theorem:

Theorem 3. *The greedy algorithm is a $9^{\frac{1}{p}}$-approximation for minimizing the l_p-norm (for integer $p \geq 1$) of the costs in the uniform cost model.*

7 Online Problem

We now consider the pipelined set cover problem in the online setting which arises in data-stream processing [1]. At each time step, the algorithm is presented with a collection of elements. The algorithm has to choose an ordering of the sets without knowledge of these newly-arriving elements, and use this ordering to process the elements. The goal is to be competitive against the best possible algorithm that does not change its ordering for the entire request sequence.

We will begin by assuming that the incoming elements are chosen from a domain containing a relatively small number of distinct elements $\{e_1, e_2, \ldots, e_d\}$. This assumption will be dropped in Section 7.3. Each element $e_i \in \{e_1, e_2, \ldots, e_d\}$ is dropped by zero or more sets, not dropped by the others, and this behavior does not change over time. In the rest of this section, we will use the phrase "count of elements till time t" to refer to the vector $\{s_{e_1}, s_{e_2}, \ldots, s_{e_d}\}$ where s_{e_i}

is the number of times the element e_i has arrived till t time steps. "Count of elements at time t" is defined similarly.

We give an $O(\log n)$ competitive algorithm for the uniform cost version of online pipelined set cover, where n is the number of sets. Our algorithm uses a technique introduced by Kalai and Vempala in [16]. (Our algorithm and proof extend to the case with arbitrary processing costs; we omit the discussion because of space constraints.) The basic idea behind this technique is the observation that if we knew the counts of the elements at time t in advance, then using the optimal solution for the counts till time t to process the elements at time t, for all t, gives the optimal solution for the online case. Since we do not know the counts at time t in advance, we use the counts till time $t-1$. To prevent the adversary from being malicious, we add a large random value to these counts. The first argument is that adding randomness does not affect the cost of the solution too much, provided the randomness is "small". The second argument is that for any choice of the counts at time t, the expected cost of the solution we pick will be good, since the distribution of the counts till $t-1$ with randomness and till time t with randomness are almost identical, if the randomness is sufficiently "large". The analysis framework from [16] can be used to find the optimal amount of randomness to add. The only catch is that for approximation algorithms, this analysis works only for algorithms that provide a lower bound on the cost of the optimal solution, and provide approximation guarantees on the cost of processing every element against the fractional cost of processing the same element. (The reason for this condition can be found in [16].) The greedy and local search algorithms from Sections 4 and 5 respectively, do not provide per element guarantees, so they cannot be used here.

As a first step, we need a technique to lower bound the optimal solution for a certain set of counts. We do this by writing a linear program. Suppose the count for element e is s_e. We have a variable x_{ij} which is set to 1 if set S_i is placed in position j. We also have a variable y_{ej} which is 1 if element e passes through j stages of the pipeline. We have the following integer programming constraints:

$$\begin{array}{rll} \sum_j x_{ij} = 1 & & \forall\ S_i \\ \sum_i x_{ij} = 1 & & \forall\ j \\ y_{ej} \geq 1 - \sum_{j' \leq j} \sum_{e \in S_i} x_{ij'} & & \forall\ e, j \\ x_{ij}, y_{ej} \in \{0,1\} & & \forall\ S_i, j, e \end{array}$$

The optimal solution minimizes the objective function:

$$\text{Objective Function} = \sum_{e,j} s_e y_{ej}$$

7.1 Offline Solution

We first present an offline randomized rounding algorithm for the problem, and then show how to convert it to an online algorithm. We solve the linear relaxation of the integer program described above. For each position j, we pick $2 \log n$ sets independently at random, the probability of picking set S_i at each trial being equal to x_{ij}. We repeat this for every j. If a set gets picked more than once,

we place it at the earliest position at which it got picked. Note that we pick $2 \log n$ sets for each position, which implies the solution is "stretched" by the same factor. We therefore pay a cost of $2 \log n$ per element (instead of unit cost) for each position j.

Lemma 1. *For any element e, let z_{ej} denote the indicator variable showing element e "survived" until position j. If $y_{ej} < 0.25$, then:*

$$\Pr[z_{ej} = 1] \leq \frac{1}{n^{1.5}}$$

Proof. An element survives if none of the sets containing it are picked at that or the previous positions. We divide the picking of the sets into $3 \log n$ independent trials, in each of which we pick one set per position. Consider element e and position j. For one of the trials, the probability that no set containing it was picked is at most $(1 - \frac{1-y_{ej}}{j})^j \leq \exp(y_{ej} - 1)$. If this experiment is repeated $2 \log n$ times, the probability that no set was picked is at most $\exp(-2(1 - y_{ej}) \log n) \leq \frac{1}{n^{1.5}}$ assuming $y_{ej} < 0.25$.

It is easy to bound the cost of the solution now. In expectation, if $y_{ej} > 0.25$, we pay a cost of $2 \log n$ for that stage with probability 1. Otherwise, we pay a cost of $2 \log n$ with probability $\frac{1}{n^{1.5}}$. Since an element can pass through at most n sets, the contribution to the expected cost from the second set of terms is negligible. Therefore, we have a $O(\log n)$ approximation algorithm. Note that the guarantee holds for the processing cost of every element versus its fractional processing cost.

7.2 Online Solution

We convert this offline algorithm to an online algorithm exactly as in [16]. Let s_{et} denote the count for element e given the input at time t. Let p_{et} denote a number chosen uniformly at random in $[0, \frac{\sqrt{t}}{\delta}]$, where δ is a function of the input [16]. Set $S_{et} := \sum_{t'=0}^{t-1} s_{et'} + p_{et}$. Note that we do not know s_{et}, and therefore can only compute the sum till time $t - 1$. We find the solution using S_{et} as the counts in the above integer program, and use this solution at time t. Using the same proof idea as in [16], it is easy to show that since this algorithm provides a $O(\log n)$ approximation to the fractional cost of every element, it converges to within $O(\log n)$ of optimal fixed offline solution with an additive error of $O(\sqrt{T})$ at time T. In other words, in the limit as $T \to \infty$, the cost of this algorithm converges to within $O(\log n)$ of the cost of the optimal offline solution.

7.3 Incomplete Information Model

So far we assumed that the incoming elements are chosen from a small domain so that we can keep track of the counts of the arrived elements. We now drop this assumption and show that our algorithm from above can be used unchanged in this case except now we use sampling-based estimates instead of the actual

counts of the elements. We sample the incoming elements with probability $\frac{1}{n^2}$. For each element e in the sample, we pass e through all n sets to categorize e such that elements that are dropped by exactly the same sets belong to a specific category. This sampling process gives us an estimate of the counts of elements till time t. We use this estimate to find the optimal online solution for the remaining elements using the algorithm described previously. The sampled elements, which are processed by all the sets, usually add little extra overhead to the overall cost; see [2]. Furthermore, by Chernoff bounds, if $T \gg n$, the estimates of the large counts converge to the true values, and therefore, the error due to sampling is negligible. Details are omitted for lack of space.

8 Conclusions and Future Work

We identified the relevance of pipelined set cover to query optimization and presented efficient approximation algorithms for this NP-Hard problem. We also considered the online version of pipelined set cover and presented a competitive algorithm with a logarithmic performance guarantee. An interesting open problem is to incorporate precedence constraints on the sets that are required to handle non-commutative operators. Natural extensions of the algorithms mentioned in this paper do not yield constant factor approximations to this variant. While we focused on a single pipeline, an interesting avenue for future work is to consider approximation algorithms for optimizing a set of pipelines, which, e.g., is applicable in a publish-subscribe setting [5]. If the pipelines are optimized independently in such a setting, e.g., using the greedy algorithm from Section 4, then the resulting overall plan may be far from optimal because it misses opportunities for sharing computation using operator sequences that are common among the pipelines.

Acknowledgments

We would like to thank Pankaj Agarwal, Arvind Arasu, Arpita Ghosh, Chandra Nair, Serge Plotkin, and Jun Yang for helpful discussions.

References

1. 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.
2. S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, 2004.
3. S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 436–447, June 1998.
4. S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Transactions on Database Systems*, 24(2):177–228, 1999.
5. J. Chen, D. 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.

6. S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Transactions on Database Systems*, 9(2):163–186, 1984.
7. E. Cohen, A. Fiat, and H. Kaplan. Efficient sequences of trials. In *Proc. of the 2003 Annual ACM-SIAM Symp. on Discrete Algorithms*, 2003.
8. G. Cormode, M. Datar, P. Indyk, and S. Muthukrishnan. Comparing data streams using hamming norms (how to zero in). In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 335–345, August 2002.
9. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–651, June 2003.
10. U. Feige. A threshold of $\ln n$ for approximating set cover. *Journal of the ACM*, 45:634–652, 1998.
11. U. Feige, L. Lovász, and P. Tetali. Approximating min-sum set cover. *Algorithmica*, 2004.
12. J. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Transactions on Database Systems*, 23(2):113–157, 1998.
13. D. Hochbaum, editor. *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, Boston, MA, 1997.
14. K. Jain, M. Mahdian, and A. Saberi. A new greedy approach for facility location problems. In *Proc. of the 2002 Annual ACM Symp. on Theory of Computing*, May 2002.
15. D. Johnson. Approximation algorithms for combinatorial problems. *Journal of Computer and System Sciences*, 9:256–278, 1974.
16. A. Kalai and S. Vempala. Efficient algorithms for the online decision problem. In *Proc. of 16th Conf. on Computational Learning Theory*, 2003.
17. A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing boolean expressions in object-bases. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pages 79–90, August 1992.
18. R. Krishnamurthy, H. Boral, and C. Zaniolo. Optimization of nonrecursive queries. In *Proc. of the 1986 Intl. Conf. on Very Large Data Bases*, pages 128–137, August 1986.
19. Y. Ling and W. Sun. An evaluation of sampling-based size estimation methods for selections in database systems. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pages 532–539, March 1995.
20. R. Motwani, J. Widom, and et al. Query processing, resource management, and approximation in a data stream management system. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*, pages 245–256, January 2003.
21. K. Munagala, S. Babu, R. Motwani, and J. Widom. The pipelined set cover problem. *Stanford University Database Group Technical Report 2003-65*, 2003.
22. L. Reinwald and R. Soland. Conversion of limited-entry decision tables to optimal computer programs I: Minimum average processing time. *Journal of the ACM*, 13(3):339–358, 1966.
23. K. Ross. Conjunctive selection conditions in main memory. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, June 2002.
24. A. Srinivasan. Improved approximations of packing and covering problems. In *Proc. of the 1995 Annual ACM Symp. on Theory of Computing*, pages 268–276, June 1995.
25. M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 9–28, September 2001.