Ordering Pipelined Query Operators with Precedence Constraints

Jen Burge*
Duke University
jen@cs.duke.edu

Kamesh Munagala†
Duke University
kamesh@cs.duke.edu

Utkarsh Srivastava‡
Stanford University
usriv@cs.stanford.edu

ABSTRACT

We consider the problem of optimally arranging a collection of query operators into a pipelined execution plan in the presence of precedence constraints among the operators. The goal of our optimization is to maximize the rate at which input data items can be processed through the pipelined plan. We consider two different scenarios: one in which each operator is fixed to run on a separate machine, and the other in which all operators run on the same machine. Due to parallelism in the former scenario, the cost of a plan is given by the maximum (or *bottleneck*) cost incurred by any operator in the plan. In the latter scenario, the cost of a plan is given by the *sum* of the costs incurred by the operators in the plan. These two different cost metrics lead to fundamentally different optimization problems: Under the bottleneck cost metric, we give a general, polynomial-time greedy algorithm that always finds the optimal plan. However, under the sum cost metric, the problem is much harder: We show that it is unlikely that any polynomial-time algorithm can approximate the optimal plan to within a factor smaller than $O(n^\theta)$, where n is the number of operators, and θ is some positive constant. Finally, under the sum cost metric, for the special case when the selectivity of each operator lies in $[\epsilon, 1-\epsilon]$, we give an algorithm that produces a 2-approximation to the optimal plan but has running time exponential in $1/\epsilon$.

1. INTRODUCTION

We consider the problem of optimizing queries consisting of pipelined operators over a stream of input data items. We primarily focus on the case when there may be precedence constraints between the query operators. If there is a precedence constraint from operator f_i to operator f_j, then in any feasible query plan, f_j can be evaluated on a data item only after f_i has been evaluated. The goal of our optimization is to arrange the query operators, respecting all precedence constraints between them, into a pipelined execution plan that maximizes the rate at which input data items can be processed through the plan, i.e., the throughput of the plan.

In this paper, we consider two very different scenarios in which queries consist exclusively of pipelined operators. The first scenario is that of query processing over *web services* [21] as introduced in [19]. Most web services provide a function-call like interface $\mathcal{X} \rightarrow \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ are sets of attributes: given values for the attributes in $\mathcal{X}$, the web service returns values for the attributes in $\mathcal{Y}$. For example, a web service may take a zip code and return the current temperature at that zip code. Due to this very restricted interface, fully pipelined query processing is natural: The input data items are used one-by-one to query some web service. This web service may then provide additional attributes for each data item that may become a part of the final query result, or may be used to query another web service and so on (effectively performing joins across web services). Thus, in this scenario, each query operator is a call to a remote web service, and precedence constraints arise because the output from one web service may be required to query another one.

The other scenario we consider in this paper is that of processing *continuous queries* over data streams [2]. In this scenario, pipelining is essential because streams are potentially unbounded and query results are required in a continuous fashion. Here too, precedence constraints may arise for the same reason as in the web-services scenario: The output from one operator may be required as input to another operator.

The two scenarios described above differ in one fundamental aspect: In the web-services scenario, each operator is a remote web service and is thereby fixed to run on a separate machine, while in the continuous-query scenario, each operator runs on the same machine (assuming centralized query processing, and that no query operator makes calls to remote sources of data). This difference leads to fundamentally different cost metrics for the two scenarios:

1. **Bottleneck Metric**: The web-services scenario exhibits *pipelined parallelism* [6], i.e., multiple operators can execute in parallel. Hence, the throughput of a plan is limited by the slowest (or bottleneck) stage in the pipelined plan. Thus, the cost of a plan is given by the maximum cost incurred by any operator in the plan. This cost metric is referred to as the bottleneck metric.
2. **Sum Metric**: In the continuous query scenario, there is no parallelism. Hence the cost of a plan is given by the *sum* of the costs incurred by the operators in the plan. This cost metric is referred to as the sum metric.

Suppose for each operator f_i, we know the execution time of

*Supported by the James B. Duke Graduate Fellowship and a NSF Graduate Research Fellowship.
†Supported in part by NSF DDDAS-TMRP 0540347.
‡Supported by a Stanford Graduate Fellowship from Sequoia Capital and a fellowship from Microsoft Corporation.

f_i per input data item (referred to as the cost of f_i), and the average number of data items output by f_i per data item input to it (referred to as the selectivity of f_i). Under both the bottleneck and the sum cost metrics, the problem of optimally arranging pipelined operators, but in the absence of precedence constraints, has been addressed in previous work [5, 13, 19]. All these optimization techniques primarily rely on evaluating those operators early that either have low cost, or low selectivity (so that they reduce work for the latter operators in the plan).

However, the presence of precedence constraints makes the problem considerably harder: Suppose there is a precedence constraint from operator f_i having high cost to operator f_j having low selectivity. Without the precedence constraint, we would just place f_j before f_i, thereby utilizing the low selectivity of f_j, as well as avoiding the high cost of f_i. However, since this plan is infeasible due to the precedence constraint, we are faced with the tradeoff of either placing both operators early (thereby incurring the high cost of f_i but utilizing the benefit of the low selectivity of f_j), or placing both operators later (thereby avoiding the high cost of f_i but giving up the benefit of the low selectivity of f_j).

In this paper, under both the bottleneck as well as the sum cost metrics, we develop principled approaches to address the above general tradeoff without an exhaustive search through all possible query plans. Under the bottleneck metric, we give a greedy algorithm to arrive at the optimal plan. At each stage, our algorithm picks the operator having the minimum cost of adding it to the plan constructed so far, and adds it to the current plan. We show that the optimal way of adding an operator to a partial query plan can be found efficiently by solving an appropriate linear program through network-flow techniques. Under the sum cost metric, the problem of finding the optimal plan is much harder, and we primarily show a negative result: If a polynomial-time algorithm can approximate the optimal plan to within a factor smaller than $O(n^\theta)$, where n is the number of operators, and θ is some positive constant, it would imply an improvement in the best-known approximation ratio for the *densest k-subgraph* problem [8]. Since densest k-subgraph is widely conjectured to be hard to approximate, such an algorithm is unlikely to exist. Finally, under the sum cost metric, we consider the special case when the selectivity of each operator lies in the range $[\epsilon, 1-\epsilon]$, and give an algorithm that results in a 2-approximation to the optimal plan, but has running time exponential in $1/\epsilon$.

The rest of this paper is organized as follows. In the remainder of this section, we survey related work. In Section 2, we define the problem of optimally arranging operators into a pipelined execution plan, and formalize the bottleneck cost metric. In Section 3, we give our greedy algorithm to find the optimal plan under the bottleneck cost metric. In Section 4, we formalize the sum cost metric, show the hardness of finding the optimal plan under this metric, and give our algorithm for the special case when each operator has selectivity in the range $[\epsilon, 1-\epsilon]$. We conclude in Section 5.

1.1 Related Work

Under the bottleneck cost metric, we consider optimization of queries consisting of pipelined operators that are running on different machines. Hence our problem can be considered a special case of the more general problems of parallel and distributed query optimization, each of which has been addressed extensively in previous work [6, 14, 17]. However, our problem has a considerably smaller search space of query plans when compared with general distributed or parallel query processing. In our setting, operators are *fixed* to run on separate machines, and we do not have the flexibility to move operators around (e.g., when operators are web services). In traditional distributed or parallel query processing, we can choose which machines to execute which operator on. Operators may even be split across machines e.g., *fragment-replicate* joins [17]. These possibilities lead to a very large space of possible query plans in distributed or parallel query optimization. Consequently, most query optimization techniques in that setting are limited to a heuristic exploration of the search space, with no guarantees of optimality. Similarly, our focus only on fully pipelined query processing distinguishes our work from that on query optimization in data integration systems that considers general data sources, e.g. [9, 10, 16, 18, 22]. We are not aware of any work for the above general problems that, when applied to our specific problem, produces the optimal results. The special case of our problem when there are no precedence constraints between operators, or when the precedence constraint graph (see Section 2) is a tree has been considered in our earlier work [19].

Under the sum cost metric, due to our focus on fully pipelined query plans, our problem is a special case of general query optimization in traditional relational DBMSs. One specific case with fully pipelined processing is queries involving a conjunction of predicates, which has been considered for DBMSs in [5, 13], and for data stream systems in [4]. However, none of these contributions consider precedence constraints between operators. Even in the context of traditional DBMS query optimization, there are scenarios where the focus is only on pipelined query plans, e.g., for online aggregation [12]. It is an interesting direction for future work to investigate whether our results in this paper can be applied to such scenarios.

Finally, under the sum cost metric, our execution model resembles that of Eddies [1], and under the bottleneck cost metric, it resembles that of distributed Eddies [20]. However, unlike our work, the Eddies framework does not perform static optimization of queries. Furthermore, Eddies-style execution requires maintaining state in tuples, not needed in our one-time plan-selection approach.

2. PRELIMINARIES

Consider the optimization of a query Q consisting of a set $\mathcal{F}$ of operators $f_1, f_2, \ldots, f_n$ that are to be evaluated in a pipelined fashion on a stream of input data items. Query Q also specifies certain *precedence constraints* between the operators in $\mathcal{F}$: If there is a precedence constraint from f_i to f_j (denoted $f_i \prec f_j$), then in any feasible query plan, operator f_j can be evaluated on a data item only after operator f_i has been evaluated.

We assume that the precedence constraints are irreflexive (f_i does not have a precedence constraint with itself), asymmetric ($f_i \prec f_j$ and $f_j \prec f_i$ do not simultaneously exist), and

transitive ($f_i \prec f_j$ and $f_j \prec f_k \Rightarrow f_i \prec f_k$). Thus, the precedence constraints may be represented as a directed acyclic graph (DAG) $\mathcal{G}$ in which there is a node corresponding to each operator, and there is a directed edge from f_i to f_j if there is a precedence constraint $f_i \prec f_j$. We define M_i as the set of all operators that are prerequisites for f_i, i.e.,

$$M_i = \{f_j \mid f_j \prec f_i\} \quad (1)$$

We also give the following definition that is used in the description of our algorithms.

Definition 2.1. (Precedence-Closed Set). *A set of operators $\mathcal{S}$ is precedence-closed if f_i belonging to $\mathcal{S}$ implies that all the prerequisite operators for f_i also belong to $\mathcal{S}$, i.e., $f_i \in \mathcal{S} \Rightarrow M_i \subseteq \mathcal{S}$.* □

The goal of our optimization is to arrange the operators in $\mathcal{F}$ into a pipelined execution plan that respects all precedence constraints specified by $\mathcal{G}$, and maximizes the rate of processing of input data items through the plan, i.e., the throughput of the plan. To arrive at an expression for the throughput of a plan, we assume that for each operator $f_i \in \mathcal{F}$, the following two quantities are known:

1. **Cost**: The per-item processing time of operator f_i is referred to as the cost of f_i, and is denoted by c_i.
2. **Selectivity**: The average number of items output by operator f_i per data item input to it is referred to as the selectivity of f_i, and is denoted by σ_i. If $\sigma_i < 1$, e.g., if f_i is a filter, then out of the data items input to f_i, only a fraction σ_i are required to be processed further, while the rest are filtered out. In general, σ_i may be > 1, e.g., if f_i performs a one-to-many join of the input data item with a stored table.

We assume *independent operator selectivities*, i.e., the selectivity of an operator does not depend on which operators have already been evaluated. Extension to correlated operator selectivities (when the selectivity of an operator may change based on which operators have already been evaluated) is left as future work.

We note that the problem of finding the optimal plan in the special case when the precedence constraint graph $\mathcal{G}$ is a tree instead of a general DAG has been addressed in [19] under the bottleneck cost metric, and the algorithm there can be adapted to the sum metric as well. However, the algorithms in this paper are more general and can be applied for any general precedence constraint graph $\mathcal{G}$.

In the remainder of this section, we formalize the bottleneck cost metric. We defer the formalization of the sum cost metric to Section 4.

2.1 Bottleneck Cost Metric

Recall from Section 1 that the bottleneck cost metric arises when each query operator is remote, i.e., fixed to run on a separate machine, e.g., in query processing over remote web services [19]. We assume that there is a single site referred to as the *coordinator*, at which the input data items are available, and which coordinates query execution by making pipelined calls to these remote operators. We first describe the possible query plans in such a scenario.

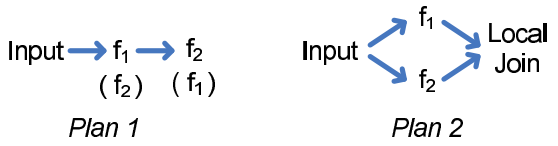

Figure 1: Two Basic Types of Plans

Consider a query consisting of two remote operators f_1 and f_2 with no precedence constraint between them. Figure 1 shows two possible ways of arranging these operators into an execution plan. In Plan 1, input data items are first processed through f_1 (or f_2) and the results are then processed through f_2 (or f_1) to obtain the final query results. However, in Plan 2, data items are dispatched in parallel (denoted by two outgoing arrows from the input) to both f_1 and f_2, and the results from both the operators are joined locally at the coordinator to obtain the final query result. Plan 1 is superior if at least one of f_1 or f_2 has selectivity < 1 so that placing it first decreases work for the latter operator. However, if both f_1 and f_2 have selectivity> 1, Plan 2 is superior (since placing any operator before the other will increase work for the latter operator).

Using the plans in Figure 1 as building blocks, it is easy to see that a general query plan is an arrangement of the query operators into a DAG with arbitrary parallel dispatch of data (denoted by multiple outgoing edges from a single operator), and rejoining of data (denoted by multiple incoming edges into an operator). The input data items are dispatched in parallel to all the source nodes in the DAG (all operators that do not have any incoming edges), and the final query result is obtained by joining the results from all sink nodes (all operators that do not have any outgoing edges).

Consider such a query plan $\mathcal{H}$ specified as a DAG on the query operators. We define the following:

$$P_i(\mathcal{H}) = \{f_j \mid f_j \text{ has a directed path to } f_i \text{ in } \mathcal{H}\} \quad (2)$$

Intuitively, $P_i(\mathcal{H})$ consists of all the predecessors of f_i in $\mathcal{H}$, i.e., all operators that are evaluated before operator f_i in the plan. Note that if plan $\mathcal{H}$ is feasible, we must have $M_i \subseteq P_i(\mathcal{H})$ for every operator f_i (recall definition of M_i from (1)).

Given a set $\mathcal{S}$ of operators, we define the combined selectivity of all the operators in $\mathcal{S}$ as $R[\mathcal{S}]$. Since operator selectivities are independent, $R[\mathcal{S}]$ is given by:

$$R[\mathcal{S}] = \prod_{i \mid f_i \in \mathcal{S}} \sigma_i \quad (3)$$

Then, for every data item input to plan $\mathcal{H}$, the average number of items that f_i needs to process is given by $R[P_i(\mathcal{H})]$. Hence the average processing time required by operator f_i (or intuitively, the cost incurred by f_i) per original input data item is $R[P_i(\mathcal{H})] \cdot c_i$. Due to parallelism among the

operators that are each running on a separate machine, the throughput of a pipelined plan is given by the slowest (or bottleneck) stage in the pipeline. Thus the cost of a plan is given by the maximum cost incurred by any operator in the plan (referred to as the bottleneck cost metric), i.e.,

$$\text{cost}(\mathcal{H}) = \max_{1 \leq i \leq n} \Big(R[P_i(\mathcal{H})] \cdot c_i \Big) \tag{4}$$

It may appear that the bottleneck cost metric ignores the cost of the local joins performed at the coordinator (recall Plan 2 in Figure 1). Formally, we can treat all the work done at the coordinator as just another call in the pipeline. The bottleneck cost metric and our algorithms are designed under the assumption that the pipeline stage constituted by the coordinator is never the bottleneck, which seems realistic since it is much more expensive to invoke remote web services than to perform local operations (shown by experiments in [19]).

We can now formally define the problem of optimizing a query under the bottleneck cost metric.

Definition 2.2. (Query Optimization Under the Bottleneck Cost Metric). *Given a query Q consisting of a set $\mathcal{F}$ of operators, and a DAG $\mathcal{G}$ of precedence constraints among the operators in $\mathcal{F}$, find a query plan arranging the operators in $\mathcal{F}$ into a DAG $\mathcal{H}$ such that for every $f_i \in \mathcal{F}$, $M_i \subseteq P_i(\mathcal{H})$, and cost($\mathcal{H}$) given by* (4) *is minimized.*

3. GREEDY ALGORITHM

In this section, we develop a general, polynomial-time algorithm for the problem specified by Definition 2.2. We first consider a special case the algorithm for which forms the crux of our general algorithm (Section 3.1). We then give our greedy algorithm for the general case in Section 3.2, and prove its correctness in Section 3.3.

3.1 Special Case: Single Expensive Operator

Consider the special case when there is exactly one operator f_e with non-zero cost, while other operators in $\mathcal{F}$ have cost 0. Then minimizing cost($\mathcal{H}$) according to (4) amounts to finding a set of operators $\mathcal{S}$ to place before f_e such that:

- $\mathcal{S}$ does not contain f_e or any operator f_i such that $f_e \prec f_i$ (since the operators in $\mathcal{S}$ have to be placed before f_e).
- $M_e \subseteq \mathcal{S}$ (so that it is feasible to place f_e after $\mathcal{S}$).
- $\mathcal{S}$ is precedence-closed (so that is feasible to place the operators in $\mathcal{S}$ before f_e, recall Definition 2.1).
- The combined selectivity $R[\mathcal{S}]$ is minimized (so that cost($\mathcal{H}$) is minimized).

The problem of finding such a set $\mathcal{S}$ can be solved using linear programming. Let $\mathcal{F}'$ be the set of operators $\mathcal{F}' = \mathcal{F} - (\{f_e\} \cup \{f_i \mid f_e \prec f_i\})$. We have a variable z_i for each operator $f_i \in \mathcal{F}'$. z_i is set to 1 if $f_i \in \mathcal{S}$ and to 0 otherwise. Then, from (3), we have:

$$R[\mathcal{S}] = \prod_{i \mid f_i \in \mathcal{F}'} (\sigma_i)^{z_i} \tag{5}$$

The linear program for finding $\mathcal{S}$ is as follows:

$$\begin{array}{lcll} \text{Minimize} & \sum_{i \mid f_i \in \mathcal{F}'} z_i \log \sigma_i & \text{subject to:} & \\ z_i & = & 1 & \forall i \mid f_i \in M_e \\ z_i & \geq & z_j & \forall i, j \mid f_i \prec f_j \\ z_i & \in & [0,1] & \forall i \end{array} \tag{6}$$

The objective function of the above linear program minimizes $\log(R[\mathcal{S}])$ that is equivalent to minimizing $R[\mathcal{S}]$. This is done to ensure that the objective function is linear. The constraints of the above linear program are easily seen to correspond directly to the requirements for $\mathcal{S}$ enumerated at the beginning of this section. The first constraint ensures that $M_e \subseteq \mathcal{S}$, and the second constraint ensures that $\mathcal{S}$ is precedence-closed. Note that the third constraint relaxes the linear program to include fractional z_i's instead of just integers. However, there always exists an optimal integer solution to the above linear program as shown by the following theorem.

Lemma 3.1. *The linear program* (6) *has an optimal solution where each z_i is set to either* 0 *or* 1.

Proof. Consider any optimal solution to the program. Choose r uniformly at random in the range $[0, 1]$. If $z_i > r$, set z_i to 1 and to 0 otherwise. Clearly, $E[z_i]$ is equal to the original value of z_i. Hence, the expected value of the constructed integer solution is the same as the cost of the optimal solution. Moreover, the constructed integer solution satisfies all the constraints of the linear program (6). Hence, there must exist an integer solution that is optimal. □

We will show in Section 3.1.1 that the optimal integer solution to (6) can be computed in $O(n^3)$ time (where n is the number of operators) by converting the linear program into a network flow problem. Once the optimal integer solution to (6) had been found, all operators $f_i \in \mathcal{F}'$ having $z_i = 1$ define the set $\mathcal{S}$. The optimal plan $\mathcal{H}$ then consists of the operators in $\mathcal{S}$ in any feasible order, followed by f_e, followed by all remaining operators in any feasible order.

3.1.1 Solution to Linear Program using Flows

To solve the linear program (6), we construct a network flow graph W as follows. In W, let there be a node for every operator $f_i \in \mathcal{F}'$, a source node s, and a sink node t. Let $D = \max_{i \mid f_i \in \mathcal{F}'} |\log \sigma_i|$. For every operator $f_i \in \mathcal{F}'$, add a directed edge $s \rightarrow f_i$ with capacity D, and a directed edge $f_i \rightarrow t$ with capacity $D + \log \sigma_i$. Note that the choice of D ensures that all capacities are non-negative. If for operators $f_i, f_j \in \mathcal{F}'$, there is a precedence constraint $f_i \prec f_j$, add a directed edge (f_j, f_i) with infinite capacity. For every i such that $f_i \in M_e$ (i.e., $f_i \prec f_e$), the capacity of the edge (s, f_i) is set to infinity. An example of how W is constructed is shown in Figure 2.

Lemma 3.2. *The minimum cut separating s from t in W yields the optimal integer solution to the linear program* (6).

Proof. Consider any feasible integer solution to program (6). Recall that the set of operators $f_i \in \mathcal{F}'$ having $z_i = 1$

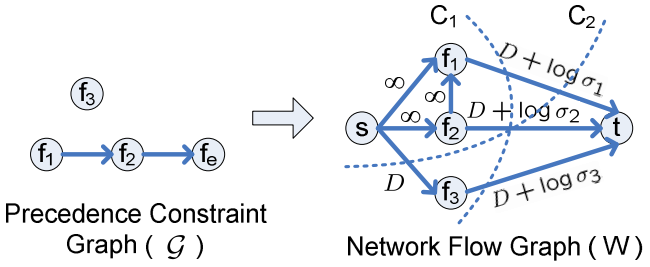

Figure 2: Solving (6) using Network Flows

define the set $\mathcal{S}$. Consider the cut in W defined by placing the operators in $\mathcal{S}$ on the same side as s and the remaining operators in $\mathcal{F}'$ (say $\bar{\mathcal{S}}$) on the other side, i.e., on the same side as t. Since $\mathcal{S}$ is precedence-closed, there is no directed edge (f_i, f_j) in W such that $f_i \in \mathcal{S}$ and $f_j \in \bar{\mathcal{S}}$. Therefore, no infinite capacity edge of the form (f_i, f_j) crosses the cut. Since $M_e \subseteq \mathcal{S}$, no infinite capacity edge of the form (s, f_i) crosses the cut. Therefore, the cut has finite capacity. The capacity of the cut is $\sum_{i \mid f_i \in \mathcal{S}} (D + \log \sigma_i) + \sum_{i \mid f_i \in \bar{\mathcal{S}}} D = nD + \sum_{i \mid f_i \in \mathcal{S}} \log \sigma_i$.

Conversely, consider any cut separating s and t in W where the set of operators on the side of s is $\mathcal{S}$ and that on the side of t is $\bar{\mathcal{S}}$. If the cut has finite capacity, the set $\mathcal{S}$ must be precedence-closed and $M_e \subseteq \mathcal{S}$. The capacity of this cut is precisely $nD + \sum_{f_i \in \mathcal{S}} \log \sigma_i$. Therefore, the minimum cut in W finds the set $\mathcal{S}$ that optimizes the linear program (6). □

For example, in Figure 2, the only cuts separating s and t in W that have finite capacity are C_1 and C_2 as shown. C_1 is the minimum cut if $\sigma_1\sigma_2\sigma_3$ is smaller than $\sigma_1\sigma_2$, otherwise C_2 is the minimum cut.

The only hurdle to using an efficient network flow algorithm to compute the minimum cut in W in $O(n^3)$ time (where n is the number of operators) is that the capacities need not be rational numbers since they involve logarithms. In practice, this is not a problem since numbers are represented using a fixed number of bits. However, to handle the general case, we perform discretization and set $\gamma_i = \lceil n^2 \log \sigma_i \rceil$, and $\tilde{D} = \max_i |\gamma_i|$. We then construct a modified network flow graph W' exactly as W except that we use γ_i instead of $\log \sigma_i$ and $\tilde{D}$ instead of D. This discretization does not result in any significant loss in accuracy as shown by the following theorem.

THEOREM 3.3. *The minimum cut in W' gives a $(1 + O(1/n))$ approximation to the optimal plan.*

PROOF. For each operator f_i, we have $\log \sigma_i \le \gamma_i/n^2 \le \log \sigma_i + 1/n^2$. The effective selectivity of f_i after the discretization is $\sigma'_i = e^{\gamma_i/n^2}$. Thus we have $\sigma_i \le \sigma'_i \le \sigma_i e^{1/n^2} \le \sigma_i(1 + 2/n^2)$. Therefore, for any set $\mathcal{S}$ of operators, we have:

$$\prod_{i \mid f_i \in \mathcal{S}} \sigma_i \le \prod_{i \mid f_i \in \mathcal{S}} \sigma'_i \le (1 + 3/n) \prod_{i \mid f_i \in \mathcal{S}} \sigma_i$$

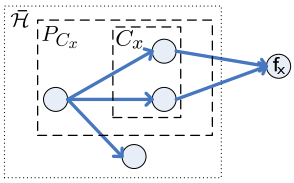

Figure 3: Placing f_x after a cut C_x in $\bar{\mathcal{H}}$

Thus, on using the effective selectivities σ'_i, we get a $(1+3/n)$ approximation to $R[\mathcal{S}]$, i.e., the cost of the optimal plan. □

Since the capacities in W' are all non-negative integers, the minimum cut in W' can be found in $O(n^3)$ time [15].

3.2 General Case

We now present a polynomial-time algorithm for the general case of the problem given by Definition 2.2. Our algorithm builds the plan DAG $\mathcal{H}$ incrementally by greedily augmenting it one operator at a time. At any stage the operator that is chosen for addition to $\mathcal{H}$ (say f_i) is the one that can be added to $\mathcal{H}$ with minimum cost, and all of whose prerequisite operators, i.e., all operators in M_i, have already been added to $\mathcal{H}$. The minimum cost of adding an operator to $\mathcal{H}$ is found by a procedure similar to that in Section 3.1.

Suppose we have constructed a partial plan DAG $\bar{\mathcal{H}}$. We first define the *frontier set* $T(\bar{\mathcal{H}})$ as the set of all operators that are candidates for addition to $\bar{\mathcal{H}}$, since all their prerequisite operators have already been added to $\bar{\mathcal{H}}$. Formally (overloading $\bar{\mathcal{H}}$ to also denote the set of operators in the DAG $\bar{\mathcal{H}}$):

$$T(\bar{\mathcal{H}}) = \{f_i \mid f_i \notin \bar{\mathcal{H}}, M_i \subseteq \bar{\mathcal{H}}\} \tag{7}$$

For every operator $f_x \in T(\bar{\mathcal{H}})$, we compute the optimal cost of adding f_x to $\bar{\mathcal{H}}$ (without modifying $\bar{\mathcal{H}}$) as follows: We compute the best *cut* C_x in the DAG $\bar{\mathcal{H}}$, such that on placing edges from the operators in C_x to f_x, the cost incurred by f_x is minimized. An example is shown in Figure 3. Formally, a cut C in a DAG $\mathcal{H}$ is defined as any set of operators in $\mathcal{H}$ such that there does not exist a directed path in $\mathcal{H}$ from one operator in the cut to another. Thus, if $f_i \in C$, then for all $f_j \in C$, $f_i \notin P_j(\mathcal{H})$ and $f_j \notin P_i(\mathcal{H})$ (recall definition of $P_i(\mathcal{H})$ from (2)). For the cut C_x in $\bar{\mathcal{H}}$, we also define the set P_{C_x} (also shown in Figure 3) as consisting of all the operators in C_x and all their predecessors in $\bar{\mathcal{H}}$, i.e.,

$$P_{C_x} = C_x \cup \{f_i \mid f_i \in P_j(\bar{\mathcal{H}}) \text{ for } f_j \in C_x\} \tag{8}$$

Note that given P_{C_x}, the cut C_x can be easily computed as only those operators in P_{C_x} that are not predecessors of some other operator in P_{C_x}.

Recall definition of $R[\mathcal{S}]$ from (3). When we place edges from the operators in C_x to f_x (as shown in Figure 3), the total cost incurred by f_x is given by $R[P_{C_x}] \cdot c_x$. To find the optimal set P_{C_x} (and hence the optimal cut C_x) such that the cost incurred by f_x is minimized, we solve a linear program. We have a variable z_i for every $f_i \in \bar{\mathcal{H}}$ that is set to 1 if $f_i \in P_{C_x}$, and to 0 otherwise. The linear program is as follows:

Algorithm ***Greedy***
1. $\bar{\mathcal{H}} \leftarrow \phi$; $T(\bar{\mathcal{H}}) \rightarrow \{f_i \mid M_i = \phi\}$
2. while ($\bar{\mathcal{H}}$ does not include all operators in $\mathcal{F}$)
3. for each operator f_x in $T(\bar{\mathcal{H}})$
4. $v_x \leftarrow$ optimal value of linear program (9)
5. $C_x \leftarrow$ optimal cut in $\bar{\mathcal{H}}$ from the solution to (9)
6. $f_{opt} \leftarrow$ operator f_x with minimum v_x
7. add f_{opt} to $\bar{\mathcal{H}}$ placing edges from C_{opt} to f_{opt}
8. update $T(\bar{\mathcal{H}})$ according to Equation (7)

Figure 4: Greedy Algorithm for Bottleneck Cost Metric

$$\begin{array}{|lcll|l|}\hline \text{Minimize} \log c_x + \sum_{i \mid f_i \in \bar{\mathcal{H}}} z_i \log \sigma_i \text{ subject to} & & & & \\ z_i & = & 1 & \forall i \mid f_i \in M_x & \\ z_i & \geq & z_j & \forall i,j \mid f_i \in P_j(\bar{\mathcal{H}}) & \\ z_i & \in & [0,1] & \forall i & \\ \hline \end{array} \quad (9)$$

Note that the above linear program is of the same form as (6) in Section 3.1. Similar to (6), the objective function of (9) minimizes $\log(R[P_{C_x}] \cdot c_x)$. The first constraint ensures that $M_x \subseteq P_{C_x}$ (so that it is feasible to add f_x after the cut C_x). The second constraint ensures that the set P_{C_x} is chosen according to the current structure of $\bar{\mathcal{H}}$, i.e., if an operator f_i is chosen in P_{C_x}, all predecessors of f_i in $\bar{\mathcal{H}}$ are also chosen in P_{C_x}. By a similar argument as in Lemma 3.1, the linear program (9) has an optimal integer solution. The operators with $z_i = 1$ in the solution define the optimal set P_{C_x}, which in turn defines the optimal cut C_x.

The overall greedy algorithm in shown in Figure 4. We start by initializing $\bar{\mathcal{H}}$ as empty, and the frontier set $T(\bar{\mathcal{H}})$ as all those operators that do not have any prerequisite operators (Line 1). Then for each operator $f_x \in T(\bar{\mathcal{H}})$, we solve the linear program (9) to determine the optimal cost of adding f_x to $\bar{\mathcal{H}}$ (Line 4). Let f_{opt} be the operator having least such cost (Line 6), and let the optimal cut for adding f_{opt} be C_{opt} as given by the solution to the linear program. f_{opt} is then added to $\bar{\mathcal{H}}$ by placing directed edges from the operators in cut C_{opt} to f_{opt} (Line 7). We update the frontier set $T(\bar{\mathcal{H}})$ according to Equation (7), and continue in this fashion until the DAG $\bar{\mathcal{H}}$ includes all the operators.

3.3 Analysis of Greedy Algorithm

In this section, we show the correctness of our algorithm *Greedy* (Figure 4) for the bottleneck cost metric. Note that since the cost of a plan is determined only by the bottleneck in the plan, in general there are many possible optimal plans. We show that our greedy algorithm finds an optimal plan. The proof is by induction on the number of operators added by *Greedy* to the partial plan $\bar{\mathcal{H}}$.

Our inductive hypothesis is that when k operators have been added to the DAG $\bar{\mathcal{H}}$ constructed by *Greedy*, $\bar{\mathcal{H}}$ agrees (in terms of edges placed) with some optimal solution restricted to just the operators in $\bar{\mathcal{H}}$, i.e., there exists an optimal solution that has $\bar{\mathcal{H}}$ as a subgraph. The base case for our induction is $k = 0$ which is trivially satisfied since the empty DAG is a subgraph of any DAG.

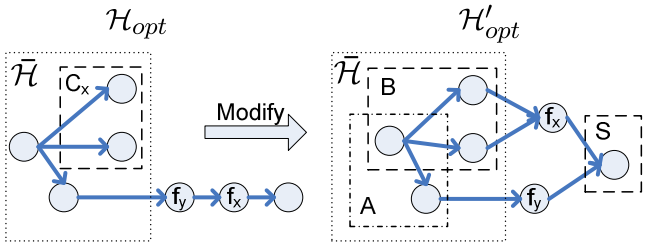

Figure 5: Modifying $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$

LEMMA 3.4. *When Greedy adds the $(k+1)$th operator, the inductive hypothesis still holds.*

PROOF. Let $\bar{\mathcal{H}}$ denote the partial DAG when k operators have been added by *Greedy*. Let $\bar{\mathcal{H}}$ be a subgraph of some optimal plan $\mathcal{H}_{opt}$ (by the inductive assumption). Suppose the $(k+1)$th operator chosen by *Greedy* to be added to $\bar{\mathcal{H}}$ is f_x. Let the optimal cut in $\bar{\mathcal{H}}$ for adding f_x be C_x. An example is shown in Figure 5.

Consider the position of f_x in $\mathcal{H}_{opt}$. Suppose $\mathcal{H}_{opt}$ places some other operator $f_y \in T(\bar{\mathcal{H}})$ with input only from operators in $\bar{\mathcal{H}}$, and places f_x so that $f_y \in P_x(\mathcal{H}_{opt})$ (see Figure 5). In general, there could be many such f_y's that are predecessors of f_x in $\mathcal{H}_{opt}$; the proof remains unchanged. Modify $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$ as follows. Remove the input to f_x in $\mathcal{H}_{opt}$ and make its input the cut C_x (just as *Greedy* does). The output of f_x in $\mathcal{H}_{opt}$ is replaced by the join of the output of f_x in $\mathcal{H}'_{opt}$ and the input to f_x in $\mathcal{H}_{opt}$. An example of this modification is shown in Figure 5. We now show that $\mathcal{H}'_{opt}$ is also an optimal plan.

CLAIM 3.5. *On modifying $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$, the cost incurred by any operator except f_x cannot increase.*

PROOF. The only operators except f_x whose cost in $\mathcal{H}'_{opt}$ may be different from their cost in $\mathcal{H}_{opt}$ are those for which f_x is a predecessor in $\mathcal{H}_{opt}$. Let $S = \{f_i \mid f_x \in P_i(\mathcal{H}_{opt})\}$ denote the set of these operators. Let A be the set $P_x(\mathcal{H}_{opt}) \cap \bar{\mathcal{H}}$ and B be the set $P_x(\mathcal{H}'_{opt})$. See Figure 5 for examples of the sets S, A, and B. Note that B is the same as P_{C_x}, i.e., the set that *Greedy* chooses to place before f_x. The combined selectivity of the set $B - A$, i.e., $R[B-A]$, can be at most one; if not, *Greedy* would have chosen P_{C_x} to be $B \cap A$ instead of B. Note that $P_{C_x} = B \cap A$ is feasible since A and B are both feasible sets of operators to place before f_x. In $\mathcal{H}_{opt}$, the operators in S had input from the set of operators $A \cup \{f_x\} \cup \{\text{other operators} \notin \bar{\mathcal{H}}\}$. In $\mathcal{H}'_{opt}$, the operators in S have input from the expanded set of operators $A \cup B \cup \{f_x\} \cup \{\text{same other operators} \notin \bar{\mathcal{H}}\}$. Since $R[B-A]$ is at most 1, the number of data items seen by operators in S in $\mathcal{H}'_{opt}$ is at most as many as in $\mathcal{H}_{opt}$. Thus the cost of any operator in S cannot increase on modifying $\mathcal{H}_{opt}$ to $\mathcal{H}'_{opt}$. □

Now consider the cost incurred by f_x in $\mathcal{H}'_{opt}$. If $R[P_x(\mathcal{H}'_{opt})] \leq R[P_x(\mathcal{H}_{opt})]$, the cost incurred by f_x also does not increase, hence combined with Claim 3.5, we have

$\text{cost}(\mathcal{H}'_{opt}) \leq \text{cost}(\mathcal{H}_{opt})$. If $R[P_x(\mathcal{H}'_{opt})] > R[P_x(\mathcal{H}_{opt})]$, there are two cases:

1. Suppose f_x is the bottleneck in $\mathcal{H}'_{opt}$. Then the cost incurred by any other operator, specifically by f_y in $\mathcal{H}'_{opt}$, is smaller. But then since $f_y \in T(\bar{\mathcal{H}})$, *Greedy* would have chosen f_y to add to $\bar{\mathcal{H}}$ instead of f_x. Hence this case is not possible.

2. If f_x is not the bottleneck in $\mathcal{H}'_{opt}$, then $\text{cost}(\mathcal{H}'_{opt})$ is given by the cost incurred by some other operator. Hence, by Claim 3.5, we have $\text{cost}(\mathcal{H}'_{opt}) \leq \text{cost}(\mathcal{H}_{opt})$.

Thus in all cases, $\text{cost}(\mathcal{H}'_{opt}) \leq \text{cost}(\mathcal{H}_{opt})$. Since $\mathcal{H}_{opt}$ is an optimal plan, $\mathcal{H}'_{opt}$ is also optimal. After *Greedy* adds f_x to $\bar{\mathcal{H}}$, $\bar{\mathcal{H}}$ is a subgraph of $\mathcal{H}'_{opt}$. Hence assuming that the inductive hypothesis holds when k operators have been added to $\bar{\mathcal{H}}$, it still holds on adding the $(k+1)$th operator. □

THEOREM 3.6. *Algorithm Greedy computes an optimal plan in $O(n^5)$ time where n is the number of operators.*

PROOF. The correctness is immediate from Lemma 3.4 by induction on the number of operators added to $\bar{\mathcal{H}}$. The running time of *Greedy* is at most the time taken to solve the linear program (9) $O(n^2)$ times. Since the linear program (9) is of the same form as the linear program (6) in Section 3.1, it can be solved in $O(n^3)$ time using the minimum cut algorithm in Section 3.1.1. Thus the total running time of *Greedy* is $O(n^5)$. □

4. SUM COST METRIC

Recall from Section 1 that the sum cost metric arises when each query operator runs on the same machine, e.g., in centralized processing of continuous queries over data streams [2].

Under the sum cost metric, we only consider the case when the selectivity of each operator is at most 1. This is sufficient to capture many common cases [3], e.g., queries with conjunctive predicates, foreign-key joins to stored tables, and aggregation. As will be shown in Sections 4.1 and 4.2, even this special case of the problem is extremely hard. Extension to the general case when the selectivity of operators may be > 1 (e.g., one-to-many joins with a stored table) is left as future work.

Given that $\sigma_i \leq 1$ for every operator f_i, we only need to consider plans that are a linear ordering of the operators into a pipeline (plans that are general DAGs need not be considered, recall Figure 1). For consistency with Section 3, we still use $\mathcal{H}$ to represent a query plan, but now $\mathcal{H}$ can only be a linear order instead of a general DAG as in Section 3.

Recall the definition of $P_i(\mathcal{H})$ (2) and the definition of $R[\mathcal{S}]$ from (3). Then, as in Section 2.1, the average processing time required by operator f_i (or intuitively, the cost incurred by f_i) per input data item is $R[P_i(\mathcal{H})] \cdot c_i$. Since all operators run on the same machine, the cost of the plan $\mathcal{H}$ is given by the sum of the costs incurred by the operators in the plan (referred to as the sum cost metric), i.e.,

$$\text{cost}(\mathcal{H}) = \sum_{i=1}^{n} \Big(R[P_i(\mathcal{H})] \cdot c_i \Big) \tag{10}$$

We can now formally define the problem of optimizing a query under the sum cost metric.

DEFINITION 4.1. (QUERY OPTIMIZATION UNDER THE SUM COST METRIC). *Given a query Q consisting of a set $\mathcal{F}$ of operators, and a DAG $\mathcal{G}$ of precedence constraints among the operators in $\mathcal{F}$, find a query plan arranging the operators in $\mathcal{F}$ into a linear order $\mathcal{H}$ such that for every $f_i \in \mathcal{F}$, $M_i \subseteq P_i(\mathcal{H})$, and cost($\mathcal{H}$) given by* (10) *is minimized.* □

We find that the above problem is much harder than the corresponding problem under the bottleneck cost metric given by Definition 2.2 (some intuition for this fact is provided in Section 4.2). Thus, under the sum cost metric, our main results are negative: We first show the NP-hardness of the above problem in Section 4.1, and then its hardness of approximation in Section 4.2. Finally, in Section 4.3, we give an algorithm for the special case when the selectivity of each operator lies in the range $[\epsilon, 1-\epsilon]$.

4.1 NP-hardness

We show the NP-hardness of our problem by reduction from the following problem known to be NP-hard [11].

DEFINITION 4.2. (AVERAGE COMPLETION TIME SCHEDULING). *Given jobs $1, \ldots, n$ with job i having processing time p_i (p_i's are positive integers), and precedence constraints between jobs, find an ordering $\pi = \pi(1), \ldots, \pi(n)$ of the jobs such that π respects all precedence constraints, and minimizes the sum of the completion times of the jobs, where the completion time of job $\pi(i)$ is defined as $\sum_{j=1}^{i} p_{\pi(j)}$.* □

THEOREM 4.3. *Query optimization under the sum cost metric (Definition 4.1) is NP-hard even when every operator has unit cost.*

PROOF. We reduce from the average completion time scheduling problem (Definition 4.2). Given an instance of the average completion time scheduling problem with n jobs, we construct a operator f_i for each job i. We reverse all the precedence constraints, i.e., if $i \prec j$ is a precedence constraint for the jobs, we place the constraint $f_j \prec f_i$. Let $p_{\max} = \max_{1 \leq i \leq n} p_i$. Let $P = n^4 p_{\max}^3$. For each operator f_i we set its cost $c_i = 1$, and its selectivity $\sigma_i = \frac{1}{1+p_i/P}$.

We show that finding the optimal ordering for the constructed operators (in terms of cost) is equivalent to finding the optimal ordering for the jobs (in terms of completion time). Consider the optimal operator ordering $\mathcal{H}_O$. Construct a job schedule $\mathcal{H}_J$ by reversing $\mathcal{H}_O$. Clearly, $\mathcal{H}_J$ is feasible since $\mathcal{H}_O$ was feasible and all precedence constraints were reversed for the operators. Let $R = \prod_{i=1}^{n} \sigma_i$ and let D_i denote the set of all operators appearing later than f_i

(including f_i) in $\mathcal{H}_O$. Then the cost incurred by f_i in $\mathcal{H}_O$ is given by (recall $c_i = 1$):

$$\bar{c}_i = R \cdot \prod_{j \mid f_j \in D_i} \frac{1}{\sigma_j} = R \cdot \prod_{j \mid f_j \in D_i} \left(1 + \frac{p_j}{P}\right) \quad (11)$$

Let t_i denote the completion time of job i in $\mathcal{H}_J$. Since $\mathcal{H}_J$ is reverse of $\mathcal{H}_O$, we have $t_i = \sum_{j \mid f_j \in D_i} p_j$. Then, from (11), $\bar{c}_i \geq R \cdot (1 + t_i/P)$ and also:

$$\bar{c}_i \leq R \cdot (1 + t_i/P + n^2 p_{\max}^2/P^2 + n^3 p_{\max}^3/P^3 + \ldots$$

Since $P = n^4 p_{\max}^3$, we have $\bar{c}_i \leq R \cdot (1 + t_i/P + \frac{1}{n^2 P})$. Since $\text{cost}(\mathcal{H}_O) = \sum_{i=1}^{n} \bar{c}_i$, we have:

$$R \cdot (n + \sum_{i=1}^{n} t_i/P) \leq \text{cost}(\mathcal{H}_O) \leq R \cdot (n + \sum_{i=1}^{n} t_i/P + \frac{1}{nP})$$

Since t_i's are integers the $\frac{1}{nP}$ term is insignificant, and optimizing $\text{cost}(\mathcal{H}_O)$ is the same as optimizing $\sum_{i=1}^{n} t_i$ which is precisely the sum of the completion times of the jobs. Thus, the problem of finding the optimal operator ordering is NP-hard. □

4.2 Hardness of Approximation

In this section, we show that the problem given by Definition 4.1 is hard to approximate by showing its connection to the hard graph problem *densest k-subgraph* [8].

Definition 4.4. (Densest k-subgraph Problem). *Given an undirected graph G and a number $k > 0$, find the subgraph of G having k vertices and maximum number of edges.* □

The densest k-subgraph problem is well-studied with its best known approximation ratio being $O(n^{-1/3})$ [8] where n is the number of vertices in the graph. It is widely conjectured that densest k-subgraph is hard to approximate to better than $n^{-\rho}$ for some constant $\rho > 0$.

Lemma 4.5. *Let ρ be a positive constant. If there exists a polynomial-time algorithm that, given an undirected graph G with n vertices, where the densest k-subgraph of G has m^* edges, finds a subgraph $\bar{G}$ of G with at most $n^\alpha k$ vertices and at least $(\alpha/2)m^*$ edges where $\alpha = \frac{\rho - o(1)}{\lg 6}$, then the densest k-subgraph has a better than $n^{-\rho}$ approximation.*

Proof. Consider a subgraph $\bar{G}$ of G with at most $n^\alpha k$ vertices and at least $(\alpha/2)m^*$ edges found by the polynomial-time algorithm. We repeatedly split $\bar{G}$ in half by using a greedy procedure from [8] to obtain a dense k-subgraph. Let $\bar{n}$ be the number of vertices, and $\bar{m}$ be the number of edges in $\bar{G}$. Construct a subgraph $\bar{G}'$ of $\bar{G}$ as follows. Add the set A consisting of the $\bar{n}/4$ highest-degree vertices in $\bar{G}$ to $\bar{G}'$. Sort the remaining vertices of $\bar{G}$ in decreasing order by the number of neighbors they have in A. Add the first $\bar{n}/4$ of the vertices in this sorted order to $\bar{G}'$.

Claim 4.6. *$\bar{G}'$ has $\bar{n}/2$ vertices and at least $\bar{m}/6$ edges.*

Proof. (Adapted from [8]) The number of nodes in $\bar{G}'$ is clearly $\bar{n}/2$ by construction. Let d_A be the average degree of a vertex in A before splitting, i.e., with respect to graph $\bar{G}$. Let m_A be the number of edges with both its vertices in A. The number of edges between vertices in A and vertices not in A is then $e_A = d_A(\bar{n}/4) - 2m_A$. The vertices in $\bar{G}' - A$ must make up at least $1/3$ of the other endpoints for these e_A edges. Thus the number of edges in $\bar{G}'$ is $\geq m_A + \frac{1}{3}\big(d_A(\bar{n}/4) - 2m_A\big) \geq \bar{n} d_A/12$. Since A consists of the highest degree vertices from $\bar{G}$, $d_A \geq 2\bar{m}/\bar{n}$. Thus the claim follows. □

We then continue the splitting procedure on the subgraph $\bar{G}'$. After repeating the splitting procedure $\alpha \lg n$ times, we are left with a subgraph having at most k vertices (since the number of vertices reduces by a factor of 2 at every step and $\bar{n} \leq n^\alpha k$) and $\bar{m}/n^{\alpha \lg 6}$ edges (since the number of vertices reduces by a factor of 6 at each step). Since $\bar{m} > (\alpha/2)m^*$ and $\alpha = \frac{\rho - o(1)}{\lg 6}$, the number of edges is $> n^{-\rho} m^* (\alpha/2) n^{o(1)} > n^{-\rho} m^*$. Thus, densest k-subgraph can be approximated to better than $n^{-\rho}$. □

Theorem 4.7. *Let ρ be a positive constant. If query optimization under the sum cost metric (Definition 4.1) has a better than $n^{\alpha/2}$ approximation where $\alpha = \frac{\rho - o(1)}{\lg 6}$, then the densest k-subgraph problem has a better than $n^{-\rho}$ approximation.*

Proof. Given an undirected graph G with n vertices and where the densest k-subgraph (say $\bar{G}$) of G has m^* edges, we construct an instance of our problem as follows. We have an "edge operator" f_e for every edge e with $c_e = 0$ and $\sigma_e = n^{-1/m^*}$, and a "vertex operator" f_v for every vertex v with $c_v = 1$ and $\sigma_v = 1$. Corresponding to every edge $e = (u, v)$, we have the precedence constraints $u \prec e$ and $v \prec e$.

Consider an ordering $\mathcal{H}$ of the operators in which we first place the operators corresponding to the vertices in $\bar{G}$, followed by the operators corresponding to the edges in $\bar{G}$, followed by the remaining operators in any feasible order. Clearly, this order is feasible. The combined selectivity of the operators corresponding to the vertices and edges of $\bar{G}$ is $(n^{-1/m^*})^{m^*} = 1/n$. Since the edge operators are free and all the vertex operators have unit cost, $\text{cost}(\mathcal{H})$ using (10) is $k + (n - k) \cdot 1/n \approx k$.

Now consider any ordering $\mathcal{H}'$ of the operators found in polynomial time. Consider the set $\mathcal{S}$ of the first $n^\alpha k$ vertex operators in $\mathcal{H}'$. Any edge operators that are placed in between the operators in $\mathcal{S}$ by $\mathcal{H}'$ must correspond to edges in the subgraph of G induced by the vertices that operators in $\mathcal{S}$ correspond to. Let $\bar{m}$ be the number of such edge operators, and their combined selectivity is hence $n^{-\bar{m}/m^*}$. By Lemma 4.5, unless densest k-subgraph has a better than $n^{-\rho}$ approximation, $\bar{m} \leq (\alpha/2)m^*$. Thus the combined selectivity of these edge operators is $\geq n^{-\alpha/2}$. Thus the cost incurred by the operators in $\mathcal{S}$ is $\geq n^\alpha k \cdot n^{-\alpha/2}$. Hence $\text{cost}(\mathcal{H}') \geq k \cdot n^{\alpha/2}$. Comparing with $\text{cost}(\mathcal{H})$, the approximation ratio is $\geq n^{\alpha/2}$. Doing any better would imply

that densest k-subgraph has a better than $n^{-\rho}$ approximation. □

Note that the above proof also provides intuition why our problem under the sum cost metric is much harder than under the bottleneck cost metric. Under the bottleneck cost metric, for the problem instance constructed in the above proof, we could have placed all the vertex operators in parallel, thereby incurring a bottleneck cost of only 1, and then placing the edge operators in any order. However, under the sum cost metric, we are forced to find a dense subgraph, which is hard.

Assuming the best possible approximation ratio for densest k-subgraph is $n^{-1/3}$ as known currently [8], Theorem 4.7 implies that the best possible approximation for our problem of query optimization under the sum cost metric is $n^{0.064}$.

4.3 Algorithm for Special Case

The hardness results in Sections 4.1 and 4.2 practically rule out any interesting approximation algorithms for the general case of the problem given by Definition 4.1. However, we note that all our hardness proofs required operators with selectivity either 1 or $1-o(1)$. Hence in this section, we consider the special case when the selectivity of each operator lies in the range $[\epsilon, 1-\epsilon]$ for some constant $\epsilon > 0$. We give a polynomial-time algorithm that is a 2-approximation for this special case. However, the running time of our algorithm is exponential in $\frac{1}{\epsilon}$.

Assume for simplicity that all operator selectivities are powers of $1-\epsilon$. This assumption is without loss of generality (we omit the proof of this claim). Our algorithm *SumMetric* is shown in Figure 6. First in Lines 1-5, for every operator f_i, if $\sigma_i = (1-\epsilon)^{n_i}$, we replace f_i with n_i "mini operators", each having selectivity $1-\epsilon$ (so that the combined selectivity remains unchanged). These mini operators for f_i have precedence constraints as a chain. The earliest mini operator for f_i has the entire cost of f_i, i.e., c_i, while the remaining mini operators for f_i have cost 0. Note that the number of mini operators n_i for any original operator f_i is at most $\frac{\log \epsilon}{\log(1-\epsilon)}$. Since ϵ is a constant, the number of mini operators is $O(n)$, where n is the number of original operators. Then in Lines 6-7, we add a precedence constraint between the first mini operator for f_i and the first mini operator for f_j if there is a precedence constraint $f_i \prec f_j$.

In a feasible ordering $\mathcal{H}$ of the mini operators, it is not necessary for all the mini operators corresponding to an operator f_i to occur consecutively. However, $\mathcal{H}$ can be used to construct a feasible ordering $\mathcal{H}'$ of the original operators such that $\text{cost}(\mathcal{H}) = \text{cost}(\mathcal{H}')$: $\mathcal{H}$ can be modified, without increase in cost, such that for every operator f_i, all the mini operators corresponding to f_i occur consecutively. Henceforth, we ignore this detail and focus on the problem of finding the optimal ordering for the set of all mini operators.

Assume there is an integer k such that $(1-\epsilon)^k = 0.5$. This assumption is again without loss of generality. Then $k = O(\frac{1}{\epsilon})$ and is a constant. In Lines 10-11 of Figure 6, we repeatedly solve the knapsack-type problem of finding the precedence-closed set $\mathcal{S}$ of exactly k mini operators (so that they have combined selectivity exactly 0.5) having minimum total cost when placed in their optimal order. Since k is a constant, $\mathcal{S}$ can be found by brute force enumeration in time polynomial in the number of operators (but exponential in k). These mini operators $\in \mathcal{S}$ are then placed in their best order in the plan $\mathcal{H}$ being constructed (Line 12). We continue in this fashion until $\mathcal{H}$ contains all mini operators.

Algorithm ***SumMetric***
1. for each operator f_i having $\sigma_i = (1-\epsilon)^{n_i}$
2. construct mini operators $f_i^1, \ldots, f_i^{n_i}$
3. selectivity$(f_i^j) = 1-\epsilon$ for all j
4. cost$(f_i^1) = c_i$; cost$(f_i^j) = 0$ for $j > 1$
5. add precedence constraints $f_i^1 \prec \ldots \prec f_i^{n_i}$
6. for each precedence constraint $f_i \prec f_j$
7. add precedence constraint $f_i^1 \prec f_j^1$
8. $\mathcal{H} = \phi$; $k = \log_{1-\epsilon}(0.5)$
9. while ($\mathcal{H}$ does not include all mini operators)
10. from all possible sets of k mini operators $\notin \mathcal{H}$
11. choose precedence-closed set $\mathcal{S}$ with least cost
12. add mini operators in $\mathcal{S}$ to $\mathcal{H}$ in optimal order

Figure 6: Algorithm for Special Case of Sum Cost Metric

THEOREM 4.8. *Algorithm SumMetric achieves a 2-approximation to the optimal ordering.*

PROOF. The proof idea is from [7]. Let $\mathcal{H}_{opt}$ be the optimal ordering of the original operators. We first replace each operator in $\mathcal{H}_{opt}$ by the sequence of its mini operators constructed just as in algorithm *SumMetric*. We then partition $\mathcal{H}_{opt}$ into contiguous segments of mini operators, with each segment having exactly k mini operators and hence combined selectivity exactly 0.5. Let m be the number of segments and let OPT_i denote the cost of the ith segment. Then from (10), $\text{cost}(\mathcal{H}_{opt}) = \sum_{i=1}^{m} \frac{1}{2^{i-1}} \text{OPT}_i$.

When *SumMetric* is choosing the ith set of k mini operators for $\mathcal{H}$, since so far it has only chosen $i-1$ sets of k mini operators each, there exists a set of at least k unchosen mini operators from the first i segments of $\mathcal{H}_{opt}$. Choose the earliest k of such operators from $\mathcal{H}_{opt}$, which is clearly a feasible choice for the ith set.

The cost of the ith set chosen by *SumMetric* is therefore at most $\sum_{j=1}^{i} \text{OPT}_j$. Thus the cost incurred by the ith set in the plan $\mathcal{H}$ is at most $\frac{1}{2^{i-1}} \sum_{j=1}^{i} \text{OPT}_j$. Thus $\text{cost}(\mathcal{H}) \leq \sum_{i=1}^{m} \sum_{j=1}^{i} \frac{1}{2^{i-1}} \text{OPT}_j$. Interchanging summations we get:

$$\begin{aligned} \text{cost}(\mathcal{H}) &\leq \sum_{j=1}^{m} \text{OPT}_j \sum_{i=j}^{m} \frac{1}{2^{i-1}} \\ &\leq 2 \sum_{j=1}^{m} \frac{1}{2^{j-1}} \text{OPT}_j = 2 \cdot \text{cost}(\mathcal{H}_{opt}) \quad \square \end{aligned}$$

5. CONCLUSIONS

We have considered the problem of finding the optimal execution plan for queries consisting of pipelined operators

over a stream of input data, where there may be precedence constraints between the operators. We considered two very different scenarios, one in which each operator is fixed to run on a separate machine (e.g., when an operator is a call to a web service), and the other in which each operator runs on the same machine (e.g., an operator in a query plan for a continuous query over a data stream). We showed that these two scenarios lead to two completely different cost metrics for optimization. When each operator runs on a separate machine, due to parallelism, the cost of a plan is given by the maximum cost incurred by any operator (the bottleneck metric). When each operator runs on the same machine, the cost of a plan is given by the sum of the costs incurred by the operators (the sum metric). We showed that under the bottleneck cost metric, the optimal plan can be found using a greedy algorithm in polynomial time. However, under the sum cost metric, we showed that the problem is NP-hard, and most likely, cannot be approximated to better than $O(n^\theta)$ for some positive constant θ. We also gave a polynomial-time algorithm for the special case when the operator selectivities lie in the range $[\epsilon, 1-\epsilon]$.

An obvious direction for future work is to do away with the assumption of independent selectivities that has been made throughout this paper, and to extend our algorithms to the case when operator selectivities may be correlated. It is also interesting to investigate whether the techniques developed in this paper can be used to optimize traditional DBMS query plans when the focus is only on pipelined processing, e.g., for online aggregation.

Acknowledgements

We are grateful to Shivnath Babu, Rajeev Motwani, and Jennifer Widom for helpful discussions and suggestions.

6. REFERENCES

[1] 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.

[2] 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.

[3] B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the 2004 Intl. Conf. on Data Engineering*, pages 350–361, 2004.

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

[6] D. DeWitt et al. The Gamma Database Machine Project. *IEEE Trans. on Knowledge and Data Engineering*, 2(1):44–62, 1990.

[7] 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.

[8] U. Feige, G. Kortsarz, and D. Peleg. The densest k-subgraph problem. In *Proc. of the 1993 Annual Symp. on Foundations of Computer Science*, pages 692–701, 1993.

[9] D. Florescu, A. Levy, I. Manolescu, and D. Suciu. Query optimization in the presence of limited access patterns. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 311–322, 1999.

[10] H. Garcia-Molina et al. The TSIMMIS approach to mediation: Data models and languages. *Journal of Intelligent Information Systems*, 8(2):117–132, 1997.

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

[12] J. Hellerstein, P. Haas, and H. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, May 1997.

[13] 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.

[14] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. In *Proceedings of the First Intl. Conf. on Parallel and Distributed Information Systems*, pages 218–225, 1991.

[15] A. Karzanov. Determining the maximal flow in a network by the method of preflows. *Soviet Math. Dokl.*, 15:434–437, 1974.

[16] M. Ouzzani and A. Bouguettaya. Query processing and optimization on the web. *Distributed and Parallel Databases*, 15(3):187–218, 2004.

[17] M. Ozsu and P. Valduriez. *Principles of distributed database systems*. Prentice-Hall, Inc., 1991.

[18] M. Roth and P. Schwarz. Don't Scrap It, Wrap It! A Wrapper Architecture for Legacy Data Sources. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 266–275, 1997.

[19] U. Srivastava, J. Widom, K. Munagala, and R. Motwani. Query optimization over web services. Technical report, Stanford University, October 2005. http://dbpubs.stanford.edu/pub/2005-30.

[20] F. Tian and D. DeWitt. Tuple routing strategies for distributed Eddies. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 333–344, 2003.

[21] Web Services, 2002. http://www.w3c.org/2002/ws.

[22] V. Zadorozhny et al. Efficient evaluation of queries in a mediator for websources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 85–96, 2002.