Flow Algorithms for Parallel Query Optimization

Conference Paper · May 2008

DOI: 10.1109/ICDE.2008.4497484 · Source: IEEE Xplore

2 authors, including:

Amol Deshpande
University of Maryland, College Park
123 PUBLICATIONS **8,285** CITATIONS

Flow Algorithms for Parallel Query Optimization

Amol Deshpande
amol@cs.umd.edu
University of Maryland

Lisa Hellerstein
hstein@cis.poly.edu
Polytechnic University

August 22, 2007

Abstract

In this paper we address the problem of minimizing the response time of a multi-way join query using *pipelined* (*inter-operator*) parallelism, in a parallel or a distributed environment. We observe that in order to fully exploit the parallelism in the system, we must consider a new class of "interleaving" plans, where multiple query plans are used simultaneously to minimize the response time of a query (or maximize the tuple-throughput of the system). We cast the query planning problem in this environment as a "flow maximization problem", and present polynomial-time algorithms that (statically) find the optimal set of plans to use for a large class of multi-way join queries. Our proposed algorithms also naturally extend to query optimization over web services. Finally we present an extensive experimental evaluation that demonstrates both the need to consider such plans in parallel query processing and the effectiveness of our proposed algorithms.

1 Introduction

Parallelism has long been recognized as the most cost-effective approach to scaling up the performance of database query processing [11, 13, 18, 21]. Over the years, this has led to the development of a host of query processing and optimization algorithms for parallel databases, aimed toward maximizing the query-throughput of the system or minimizing the response time of a single large query. Broadly speaking, the parallelism in a parallel database can be exploited in three ways during query processing [19, 24]. Different query operators that do not depend on each other can be executed in parallel on different processors (*independent* parallelism). Two operators in a producer-consumer relationship can be run in parallel by pipelining the output of the producer to the consumer (*pipelined* or *inter-operator* parallelism). Finally, copies of the same query operator may be run on multiple processors simultaneously, each operating on a partition of the data (*partitioned* or *intra-operator* parallelism). Typically, most systems use a combination of these, depending on the available resources, the data placement (in a shared-nothing system), and the execution plan itself (some execution plans are naturally more parallelizable than others). For example, partitioned parallelism can exploit the available processors maximally, and should be used if the number of processors exceeds the number of operators. Partitioned parallelism, however, suffers from higher communication overhead, is sensitive to *data skew* [12], and is typically more complicated to set up. Pipelined parallelism is typically considered easier to implement and reason about, and results in less communication overhead; however, it enables limited parallelism since the number of operators in a database query is typically small.

In this paper, we consider the problem of minimizing the response time of a multi-way join query being executed using a left-deep pipelined plan with each join operator being evaluated on a separate processor.

This type of execution appears naturally in many settings, especially in shared-nothing systems where the query relations might be stored at different machines. In shared-memory or shared-disk environments, such execution might lead to better cache locality. Further, as Srivastava et al. [26] observe, query optimization over web services reduces to this problem as well: each web service can be thought of as a separate processor.

Perhaps the biggest disadvantage of this type of execution is that, in general, one of the processors in the system, most likely the processor executing the first join operator, will quickly become the *bottleneck* with the rest of the processors sitting idle [26]. We propose to remedy this problem by exploiting a new class of plans, called *interleaving* plans, where multiple regular query plans are used simultaneously to fully exploit the parallelism in the system. We would like to note that despite the superficial similarities to adaptive query processing techniques such as *eddies* [2], interleaving plans are not adaptive; we are instead addressing the static optimization problem of finding an optimal interleaving plan, assuming that the operator characteristics (selectivities and costs) are known.

Our algorithms are based on a characterization of query execution as *tuple flows* that we proposed in a prior work [7]; that work considers the problem of *selection ordering* in a parallel setting, and presents an algorithm to find an optimal solution by casting the problem as a *flow maximization* problem. In this paper, we first generalize that work by designing an $O(n^3)$ algorithm to find the optimal interleaving plan (that minimizes the response time) to execute a selection ordering query with *tree-structured precedence constraints* (where n is the number of operators). Our algorithm has the additional, highly desirable, *sparsity* property in that the number of different regular plans used is at most $O(n)$. We then extend this algorithm to obtain algorithms for a large class of multi-way join queries with acyclic query graphs, by reducing the latter type of queries to precedence-constrained selection ordering [22, 23, 26].

Outline

We begin with a discussion of related work in parallel query optimization (Section 2). We then present our execution model for executing a multi-way join query and present a reduction of this problem to precedence-constrained selection ordering (Section 3). We then present our main algorithm for finding an optimal interleaving plan for the case when all operators are *selective*, ie., have selectivity < 1 (Section 4). We then discuss how this algorithm can be extended to solving a large class of general multi-way join queries, which may result in *non-selective* operators (Section 5). We then present an extensive simulation-based experimental study that demonstrates the practical benefits of using interleaving plans (Section 6). In Section 7, we consider an alternative caching-aware approach to evaluating non-selective operators. The proofs of the lemmas and theorems in the body of the paper are presented in the Appendix.

2 Related Work

There has been much work in the database literature on designing query processing and optimization algorithms that can effectively exploit multi-processor parallelism. Apers et al. [1] were the first, to our knowledge, to explicitly study the tradeoffs between response time optimization (that minimizes the total time taken to execute a query) and total work optimization (that minimizes the total work across all processors), in the context of distributed query processing. Ganguly et al. [15] formalize these tradeoffs and show that optimizing for response time is *much harder* than optimizing for total work, since the response time metric does not obey the *principle of optimality*. To make the parallel query optimization problem tractable, Hong and Stonebraker [21] present a two-phase approach that separates join order optimization from parallel scheduling issues. Wilschut et al. [30] adopt the two-phase approach and analyze several strategies for the second phase that exploit pipelined, non-pipelined and intra-operator parallelism in various ways. Hasan et

al. [20, 6] present communication-cost aware scheduling algorithms for the second phase. Garofalakis and Ioannidis [16, 17] consider the issues of choosing the degrees of intra-operator parallelism to best execute the plan in presence of resource constraints. However, to our knowledge, none of the prior work in parallel query optimization has considered using interleaving plans to minimize response time. To our knowledge, none of the prior work in parallel query optimization has considered using interleaving plans to minimize response time.

Our work is closely related to two recent papers. In our prior work (Condon et al. [7]), we introduced the notion of interleaving plans for parallel selection ordering. In this work, we substantially extend that work by generalizing the algorithms to executing multi-way join queries; we also present an extensive experimental evaluation to demonstrate the practical benefits of interleaving plans. Srivastava et al. [26] study query optimization over web services. Although they focus on finding a single plan for all input tuples, they allow sending a tuple to multiple web services simultaneously in parallel. They don't, however, consider interleaving plans. In the extended version of this paper [10], we discuss an approach to combine these two classes of plans.

Interleaving plans bear a superficial similarity to tuple-routing query processors, particularly *eddies* [2]. The eddies approach treats query processing as routing of tuples through operators, and uses a different route for executing each tuple. Tian and DeWitt [27] considered the problem of designing tuple routing strategies for eddies in a distributed setting, where the operators reside on different nodes and the goal is to minimize the average response time or the maximum throughput (a setting similar to ours). They present an analytical formulation as well as several practical routing strategies for this problem. Eddies, however, use multiple plans for adaptivity purposes, with the aim of converging to a single optimal plan if the data characteristics are unchanged, whereas our goal is to find a statically optimal interleaving plan that minimizes the response time. In that sense, our work is closer in spirit to *conditional plans* [9], where the goal is to find a static plan with decision points to optimally execute a selection query over a sensor network. Our algorithms can be seen as a way to design routing policies in parallel and distributed setting for eddies.

The algorithms we present are based on a characterization of query execution as tuple flows, where the amount of flow may change as it travels through the network. Generalized maximum flow problems also have this property (cf. Fleischer [14]). We refer the reader to Condon et al. [7] for a discussion of how our problem differs from those problems.

3 Problem Formulation and Analysis

In this section, we formally define the problem of evaluating a multi-way join query using pipelined (inter-operator) parallelism and show how the problem can be reduced to precedence-constrained selection ordering. We then introduce the notion of interleaving plans.

3.1 Parallel Execution Model

Figure 1 shows a 5-way join query that we use as a running example throughout this paper. Executing such a query using pipelined parallelism requires us to designate one of the input relations as the *driver* relation[1]. The join operators are executed in parallel on different processors (Figure 2), and the tuples of the driver relation (along with matches found) are routed through the join operators one by one. We assume that the join operators are *independent* of each other.

[1] Although it is possible to drive execution using multiple driver relations through use of symmetric hash join operators [29], most database systems do not support such plans.

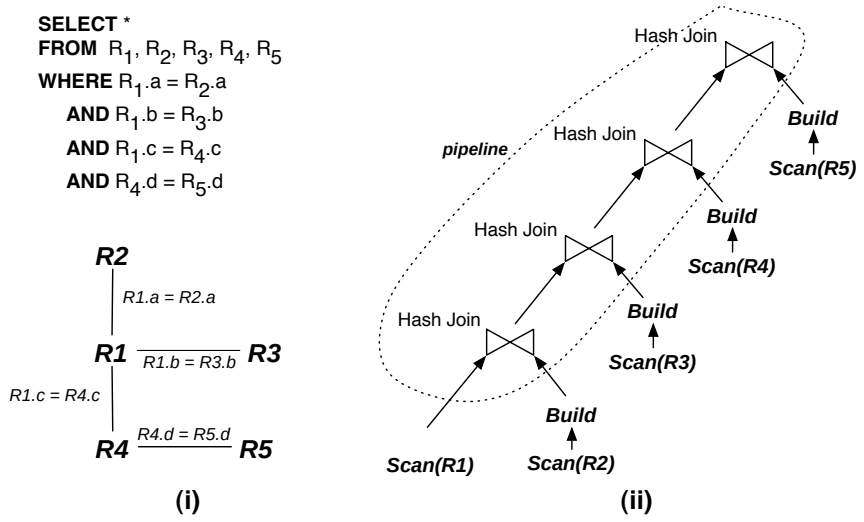


Figure 1: (i) A 5-way join query, and the corresponding query graph. (ii) A pipelined plan where R_1 is the driver relation.

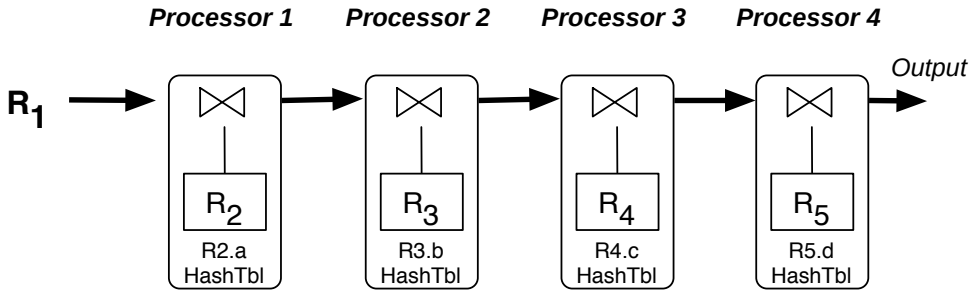


Figure 2: Executing query in Figure 1 using pipelined parallelism with each join running on a different processor. The join with R_5 is done using the R_4 matches found earlier.

The joins can be done as hash joins (as shown in Figure 1), index joins (based on the availability of indexes and the sizes of the relations), or nested-loops joins. The techniques we propose are invariant to this choice; in the rest of the paper, we assume all joins are executed using hash joins. For clarity, we focus only on the "probe" costs of the join operators, ie., the cost of looking up the matches given a tuple. We assume that these costs are constant and do not depend on the tuples being used to probe. We ignore the (constant) cost of scanning the driven relations and/or building indexes on them, and assume that the routing and communication costs are negligible – it is fairly easy to extend our cost model to include these costs (e.g., by adding the per-tuple routing and communication overhead to the probe costs of the operators).

operators have a uniform cost structure across tuples (in other words, we assume that the probe cost per tuple is constant and does not depend on the values of the tuple attributes). In general, join operators may exhibit a non-uniform cost structure. For example, if the join is being performed using a hash join and the hash table does not fit into memory, then the probe cost depends on whether the corresponding partition is in memory or not [28, 5]. Pipelined plans should not be used in such cases, and alternatives like blocking plans (using multi-pass hash joins or sort-merge joins) or XJoins [28] (which employ sophisticated scheduling logic to handle large tables) should be considered. In this paper, we assume uniform cost structure across

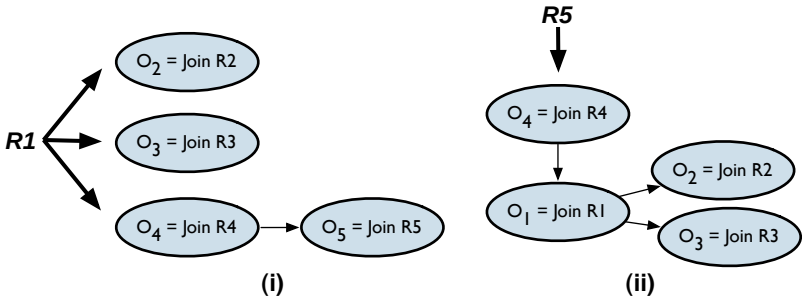


Figure 3: Reducing the query in Figure 1 to precedence-constrained selection ordering for driver choices R_1 and R_5.

c_{R_i}	Probe cost for relation R_i
$fanout_{R_i}$	Fanout of a probe into R_i
O_i	Selection oper. corresponding to join with R_i
c_i	Execution cost of selection operator O_i
p_i	Selectivity of selection operator O_i
r_i	Rate limit of O_i ($\propto 1/c_i$)

Table 1: Notation used in the paper

tuples; in other words, we assume that the probe cost per tuple is constant and does not depend on the values of the tuple attributes.

3.2 Reduction to Precedence-Constrained Selection Ordering

Assuming that the driver relation choice has been made, the problem of ordering the "driven" relations bears many similarities to selection ordering. In essence, the join operations can be thought of as selections applied to the tuples from the driver relation (even though some of the joins may be done using components from other relations - cf. Figure 2); the precedence constraints arise from the desire to avoid Cartesian products [22, 23, 26].

Given a multi-way join query over n relations, $R_1, \ldots, R_n$, with one of the relations designated as the driver relation, the reduction begins with creating a selection operator for each of the driven relations, and setting up appropriate precedence constraints to ensure that no Cartesian products are required (Figure 3). We denote the selection operator corresponding to relation R_i by O_i. For acyclic query graphs (that we focus on in this paper), the resulting precedence graph will be a forest of trees. The cost of O_i, denoted by c_i, is set to be the probe cost into the hash table on R_i, whereas the selectivity of O_i, denoted by p_i, is set to be the "fanout" of the join with R_i. Figures 3 (i) and (ii) show two examples of this reduction for example query in Figure 1.

Unlike selection operators, the join fanouts may be > 1.

Definition: We call the operators with selectivity < 1 *selective* operators, whereas an operator with selectivity ≥ 1 is called a *non-selective* operator.

3.3 Execution Plan Space

A *serial* plan for executing a selection ordering query specifies a single *permutation of the operators* (that obeys the precedence constraints) in which to apply the operators to the tuples of the relation. In a single-processor system, where the goal is to minimize the total work done, the *rank ordering* algorithm [23] can be used to find the optimal serial plan (shown in Figure 4). Srivastava et al. [26] present an algorithm to minimize the response time when each operator is being executed on a different processor in parallel. As they show, when the selectivities are all ≤ 1 and the processors are identical, the optimal algorithm, called *BOTTLENECK* (Figure 4), simply orders the operators by their execution costs (more generally, by their *tuple rate limits*, r_i's). A somewhat unintuitive feature of this algorithm is that the actual operator selectivities are irrelevant.

Algorithms: OPT-SEQ (minimize total work) & BOTTLENECK (minimize response time)

1. Let S denote the set of operators that can be applied to the tuples while obeying precedence constraints.
2. Choose next operator in the serial plan to be the one with:

   OPT-SEQ: min $c_i/(1-p_i)$ among S.

   BOTTLENECK: min c_i (equiv. max $r_i = \frac{1}{c_i}$) among S.
3. Add newly valid operators (if any) to S.
4. Repeat.

Figure 4: Algorithms for finding optimal serial plans for selective operators (assuming identical processors)

Although the BOTTLENECK algorithm finds the best serial plan for executing the query, it is easy to see that it will not exploit the full parallelism in the system in most cases. Figure 5 illustrates this through an example. In this case, the query consists of three identical operators, with probabilities 0.2 and costs 0.1. The best serial plan can process only 10 tuples in unit time. On the other hand, if we use three serial plans simultaneously by routing 1/3 of the tuples through each of them, the total expected number of tuples processed in unit time increases to about 24.19.

We call such plans *interleaving* plans. In general, an interleaving plan consists of a set of permutations (serial plans) along with a probability weight for each of the permutations (the probabilities sum up to 1). When a new tuple enters the system, it is assigned one of these permutations, chosen randomly according to the weights (called the *routing for that tuple*). The tuple is then sent through the operators in that order, and is either discarded by some operator, or it is output from the system with a designation specifying it has satisfied all predicates (for the original multi-way join query, a set of result tuples will be output instead).

4 MTTC Algorithm: Selective Operators

In this section, we present our algorithm for finding the optimal interleaving plan for executing a precedence-constrained selection ordering problem for tree-structured precedence constraints, when all operators are selective. The algorithm actually maximizes the *tuple throughput*, i.e. the number of tuples of the driver relation that can be processed in unit time. We call this the *MTTC* problem (*max-throughput problem with tree-structured precedence constraints*). We begin with a formal problem definition, followed by an algorithm for the special case when the precedence constraints are a *forest of chains*. We then present an algorithm for the general case that recursively reduces arbitrary tree-stuctured constraints to forests of chains. Proofs of the results in this section can be found in the appendix.

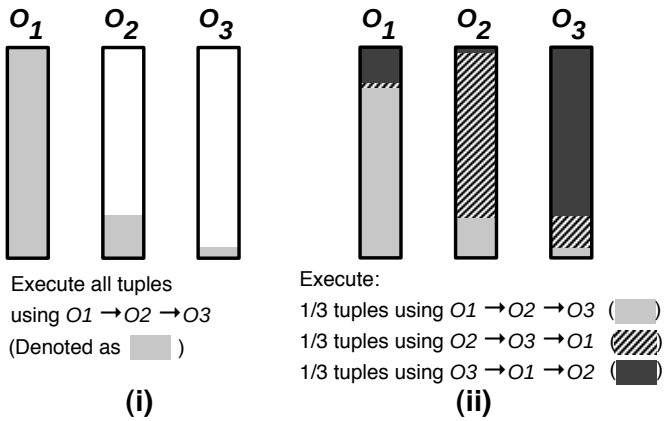


Figure 5: Given 3 identical operators, O_1, O_2, O_3, with $p = 0.2, c = 0.1$ and no precedence constraints, (i) the best serial plan processes 10 tuples in unit time; (ii) an interleaving plan that uses 3 serial plans equally can process 24.19 tuples.

4.1 Definition of the MTTC Problem

The input to the MTTC problem is a list of n selection operators, $O_1, \ldots, O_n$, associated selectivities $p_1, \ldots, p_n$ and rate limits $r_1, \ldots, r_n$, and a precedence graph G. The p_i and r_i are real values satisfying $0 < p_i < 1$ and $r_i > 0$. Rate limit $r_i = 1/c_i$ is the number of tuples O_i can process per unit time. Table 1 summarizes this notation. Graph G is a forest of rooted trees.

The goal in the MTTC problem is to find an optimal tuple routing that maximizes throughput. The routing specifies, for each permutation of the operators, the number of tuples to be sent along that permutation per unit time[2]. The routing must obey the precedence constraints defined by G: for each distinct O_i, O_j, if O_j is a descendant of O_i in G, then tuples must be sent to O_i before they are sent to O_j. The routing must also respect the rate limits of the operators: for each operator O_i, the expected number of tuples reaching O_i per unit time cannot exceeed r_i.

Below we give a linear program formally defining the MTTC problem. We use the following notation. Let π be a permutation of a finite set S. The kth element of π is denoted by $\pi(k)$. Let $\phi(n)$ be the set of all $n!$ permutations of $\{O_1, \ldots, O_n\}$. For $i \in \{1, \ldots, n\}$ and $\pi \in \phi_n$, $g(\pi, i)$ denotes the probability that a tuple sent according to permutation π reaches operator O_i without being eliminated. Thus if $\pi(1) = O_i$, then $g(\pi, i) = 1$; otherwise, $g(\pi, i) = p_{\pi(1)} p_{\pi(2)} \ldots p_{\pi(m-1)}$, where $\pi(m) = O_i$. Define $n!$ real-valued variables f_π, one for each $\pi \in \phi(n)$, where each f_π represents the number of tuples routed along permutation π per unit time. We call the f_π *flow variables*.

Formally, the MTTC problem is to find an optimal assignment K to the flow variables in the following LP:

MTTC LP: Given $r_1, \ldots, r_n > 0$, $p_1 \ldots, p_n \in (0, 1)$, and a precedence graph G on $\{O_1, \ldots, O_n\}$ that is a forest of trees, maximize

$$F = \sum_{\pi \in \phi(n)} f_\pi$$

[2]These values are normalized using the total throughput at the end, to obtain probabilities to be used for actual routing during execution.

subject to the constraints:

(1) $\sum_{\pi \in \phi(n)} f_\pi g(\pi, i) \leq r_i$ for all $i \in \{1, \ldots, n\}$,

(2) $f_\pi \geq 0$ for all $\pi \in \phi(n)$, and

(3) $f_\pi = 0$ if π violates some constraint of G.

Definition: We refer to the constraints involving the r_i as *rate constraints*. If assignment K satisfies the rate constraint for r_i with equality, we say O_i is *saturated* by K. The value F achieved by K is called the *throughput* of K, and we call K a routing.

4.2 Fundamental Lemmas and Definitions

Given operators O_i and O_j, we say that O_i *can saturate* O_j if $r_i p_i \geq r_j$. If $r_i p_i = r_j$ we say that O_i can *exactly saturate* O_j, and if $r_i p_i > r_j$ we say that O_i can *over-saturate* O_j. A *chain* is a tree in which each node has exactly one child. If C is a chain in the precedence graph of an MTTC instance, we say that C is *proper* if each non-leaf node in the chain can saturate its child.

Let K be a feasible solution to an MTTC instance, and let $\mathcal{O} = \{O_1, \ldots, O_n\}$. We say that K has the *saturated suffix property* if for some non-empty subset $Q \subseteq \mathcal{O}$, (1) the operators in Q are saturated by K and (2) if K assigns a positive value to f_π, then the elements of $\mathcal{O} - Q$ precede the elements of Q in π (in other words, no tuples ever flow from an operator in Q to an operator in $\mathcal{O} - Q$). We call Q a *saturated suffix* of K.

The following important lemma was proved in [7] for the MTTC problem with no precedence constraints. The proof also holds with the precedence constraints.

Lemma 4.1 *[7] (The saturated suffix lemma) If feasible solution K to the MTTC LP has the saturated-suffix property with saturated suffix Q, then K is an optimal solution to the max-throughput problem and achieves throughput*

$$F^* = \frac{\sum_{O_i \in Q} r_i(1 - p_i)}{(\prod_{O_j \in \mathcal{O} - Q} p_j)(1 - \prod_{O_i \in Q} p_i)}$$

The following is a strengthening of results from [7, 8].

Lemma 4.2 *Let I be an instance of the MTTC problem in which there are no precedence constraints, and let the operators in I be numbered so that $r_1 \geq r_2 \ldots \geq r_n$. Let*

$$F^* = min_{k \in \{1, \ldots, n\}} \frac{\sum_{i=k}^{n} r_i(1 - p_i)}{(\prod_{j=1}^{k-1} p_j)(1 - \prod_{i=k^n} p_i)}$$

Then F^* is the optimal value of the objective function for instance I. If k' is the largest value of k achieving the value F^*, then there exists an optimal routing for which $\{O_{k'}, O_{k'+1}, \ldots, O_n\}$ is a saturated suffix. If the optimal value F^* is achieved at $k = 1$, then every feasible routing K saturates all operators.

The above two lemmas are the basis for our algorithms and their correctness proofs. The idea behind our algorithms is to construct a routing with the saturated suffix property. This may be impossible due to precedence constraints; as a simple example, consider a two-operator instance where O_1 must precede O_2, but O_1 cannot saturate O_2. However, by reducing rate limits of certain operators, we can construct a routing with the saturated suffix property that is also optimal with respect to the original rate limits.

4.3 An MTTC algorithm for chains

We now present an algorithm for the special case of the MTTC problem in which the precedence graph G is a forest of chains. This algorithm generalizes the algorithm in [8] for the unconstrained case. The idea is to find subsets of operators that can be combined into *superoperators*. A superoperator is a permutation π' of a subset of the operators, such that each operator in the permutation (but the last) can saturate the next one. We treat a superoperator as a new single operator, formed by pasting together its component operators in the given order. We define the selectivity of π', denoted $\sigma(\pi')$, to be the product of the selectivities of its component operators, and its rate limit, denoted $\rho(\pi')$, to be the rate limit of its first operator. Note that flow sent through the operators in π' in the given order will either saturate all those operators, or none of them.

4.3.1 Preprocessing procedure

To preprocess the input, we first make all chains proper. We do this by performing the following operation on each operator O_i in the chain but the last, beginning from the top operator in the chain and proceeding downward: *Let O_j be the operator after O_i in the chain. If O_i cannot saturate O_j, then reset the rate limit r_j of O_j to be $r_i p_i$.*

Note that when the above procedure reduces the rate limit of an operator O_j, it does not reduce the maximum throughput attainable. Under any optimal routing, all flow into O_j must pass through its parent O_i first, so at most $r_i p_i$ flow can reach O_j, even under the optimal routing. Once the chains are proper, we sort all of the operators in descending order of their rate limits. Let π be the resulting permutation. We renumber the operators so that $\pi = (O_1, \ldots, O_n)$.

4.3.2 The RouteChains procedure

Following the preprocessing, we invoke a recursive procedure that we call *RouteChains*. RouteChains recursively constructs a routing K. It consists of pairs of the form (π, x) indicating that x amount of flow is to be sent along permutation π. Each recursive call adds one pair to this routing, and we stop when at least one operator is saturated. The inputs to RouteChains are as follows.

RouteChains: Inputs

1. Rate limits $r_1, \ldots, r_n$ and selectivities $p_1, \ldots, p_n$, for a set of operators $\mathcal{O} = \{O_1, \ldots, O_n\}$,
2. A precedence graph G with vertex set $\mathcal{O}$ consisting of proper chains.
3. A permutation π of the operators obeying the constraints of G.
4. An ordered partition $P = (\pi_1, \ldots, \pi_m)$ of π into subpermutations, such that each π_i is a superoperator.

The inputs to RouteChains must also obey the following **"can't saturate" precondition**[3]: for $2 \leq j \leq m$, $\rho(\pi_j)\sigma(\pi_j) \leq \rho(\pi_{j-1})$, which says that each superoperator in the partition either cannot saturate its *predecessor*, or can only saturate it exactly.

In the initial call to RouteChains made by the MTTC algorithm for chains (following preprocessing), the first input is set to the rate limits and selectivities of the operators $O_1, \ldots, O_n$, and the second to the precedence graph for the operators. Permutation π is set to $(O_1, \ldots, O_n)$ and P to the trivial partition $((O_1), \ldots, (O_n))$. Because the chains are proper, π obeys the precedence constraints. The sort performed in preprocessing ensures that the "can't saturate" precondition holds for the initial call, since each p_i is < 1.

[3]This should actually be called the "can't over-saturate" precondition

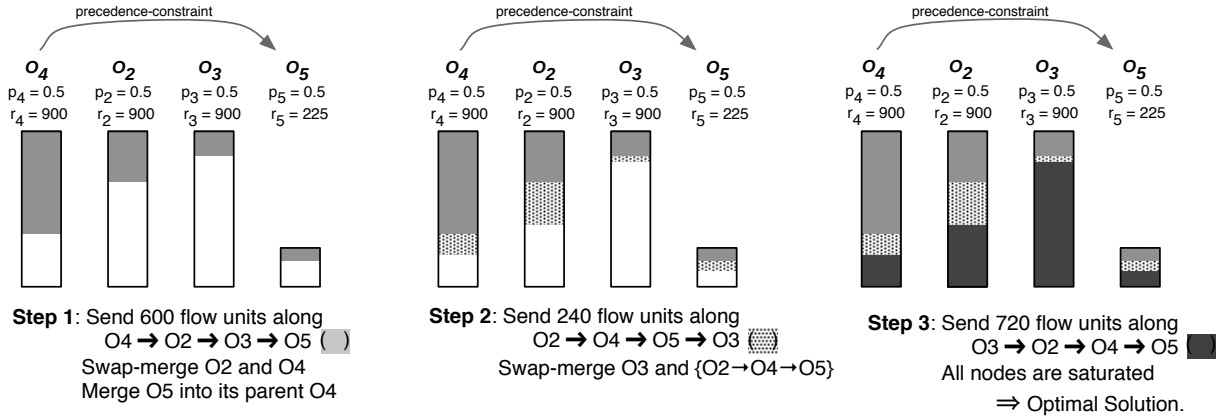


Figure 6: Illustration of the algorithm for the case shown in Figure 3 (ii); O_i corresponds to the join with R_i.

RouteChains: Output RouteChains returns a routing K that obeys the rate limits $r_1, \ldots, r_n$ and the precedence constraints defined by G, and saturates at least one operator in $\mathcal{O}$.

RouteChains: Execution

RouteChains begins by calculating an amount x of flow to send along permutation π, where π is the permutation specified in the input to RouteChains. More particularly, $x \geq 0$ is the minimum amount of flow that, sent along permutation π, triggers one of the following stopping conditions:

1. Some operator is saturated.

2. Some superoperator π_i, $2 \leq i \leq m$, can exactly saturate its predecessor π_{i-1}.

3. Some operator O_i can exactly saturate O_j, where O_j is the child of O_i in a chain of G, and O_j is not the successor of O_i in a superoperator π_l in P.

Conditions 2 and 3 are with respect to the *residual rate limits of the operators*, after x amount of flow is sent along permutation π. The residual rate limit of operator O_i is $r_i - xg(\pi, i)$. The value of x is easily calculated based on the following observations. Consider sending y flow along permutation π. Operator O_j becomes saturated at $y = \frac{r_j}{\prod_{k=1}^{j-1} \sigma(\pi_k)}$. For $2 \leq j \leq m$, superoperator π_j becomes able to exactly saturate π_{j-1} at $y = \frac{\rho(\pi_j)\sigma(\pi_j) - \rho(\pi_{j-1})}{\prod_{k=1}^{j} \sigma(\pi_k) - \prod_{k=1}^{j-2} \sigma(\pi_k)}$. For any $1 \leq i < j \leq n$, operator O_i becomes able to exactly saturate operator O_j at $y = \frac{r_i p_i - r_j}{\prod_{k=1}^{i} p_k - \prod_{k=1}^{j-1} p_k}$. Thus x can be calculated by taking the minimum of $O(n)$ values for y.

After RouteChains computes x, what it does next is determined by the lowest-numbered stopping condition that was triggered by sending x flow along permutation π.

If Stopping Condition 1 was triggered, then RouteChains returns $K = (\pi, x)$.

Else, if Stopping Condition 2 was triggered for some superoperator π_i, then RouteChains chooses one such π_i (arbitrarily, if there are multiple such π_i). It swaps π_i and π_{i-1} in $(\pi_1, \ldots, \pi_m)$ and concatenates them together into a single superoperator, yielding a new partition into superoperators $P' = (\pi_1, \ldots, \pi_{i-2}, \pi_i\pi_{i-1}, \pi_{i+1}, \ldots, \pi_n)$ and a new permutation $\pi' = (\pi_1\pi_2 \ldots \pi_{i-2}\pi_i\pi_{i-1}\pi_{i+1} \ldots, \pi_n)$. We call this operation a *swap-merge*. RouteChains then calls itself recursively, setting P to P', π to π', the r_i's to the residual rate limits, and keeping all other input parameters the same. The new inputs satisfy the "can't saturate" precondition and the other input specifications (assuming the initial inputs did). The recursive call returns a set K' of flow assignments. RouteChains returns the union of K' and $\{(\pi, x)\}$.

Else if Stopping Condition 3 was triggered by a parent-child pair O_i, O_j, then RouteChains chooses such a pair. It then *absorbs O_j into its parent O_i* as follows. Operators O_i and O_j cannot be contained in the same superoperator, since if they were, O_j would have to be the successor of O_i. If O_i and O_j are contained in superoperators (O_i) and (O_j), each containing no other operators, then absorbing O_j into O_i is simple; RouteChains deletes the superoperator (O_j), and adds O_j to the end of the superoperator containing O_i, to form the updated superoperator (O_i, O_j).

Otherwise, if the superoperators containing O_i and O_j contain other operators, let w, z be such that O_i is in π_w and O_j is in π_z. Let a, b be such that $\pi_w(a) = O_i$ and $\pi_z(b) = O_j$. RouteChains splits π_w into two parts, $A = (\pi_w(1), \ldots, \pi_w(a))$ and $B = (\pi_w(a+1), \ldots, \pi_w(s))$ where $s = |\pi_w|$. It splits π_z into three parts, $C = (\pi_z(1), \ldots, \pi_w(b-1))$, $D = (\pi_z(b), \ldots, \pi_w(c-1))$, and $E = (\pi_z(c), \ldots, \pi_w(t))$, where $t = |\pi_z|$ and c is the minimum value in $\{b+1, \ldots, t\}$ such that $\pi_z(c)$ is not a member of the same precedence chain as O_j; if no such c exists, it sets c to be equal to $t+1$ and E to be empty. RouteChains adds D to the end of A, forming four superoperators AD, B, C, E out of π_w and π_z. It then forms a new partition P' from P by replacing π_w in P by AD, B, in that order, and π_z by C, E in that order. If any elements of P' are empty, RouteChains removes them. Let π' denote the concatenation of the superoperators in P'.

Partition P' may not satisfy the "can't saturate" precondition, with respect to residual rate limits. (For example, in P' the precondition might be violated by superoperator B and its successor.) In this case, RouteChains performs a modified topological sort on P'. It forms a directed graph G' whose vertices are the superoperators in P', such that there is a directed edge from one superoperator to a second if there is an operator O_i in the first superoperator, and an operator O_j in the second, such that O_j is a descendant of O_i in G. Since π' obeys the precedence constraints, G' is a directed acyclic graph. RouteChains then sorts the superoperators in P' by executing the following step until G' is empty: *Let S be the set of vertices (superoperators) in G' with no incoming edges. Choose the element of S with highest residual rate limit, output it, and delete it and its outgoing edges from G'.* RouteChains re-sets P' to be the superoperators listed in the order output by the sort, and π' to be the concatenation of those superoperators.

RouteChains then executes a recursive call, using the initial set of operators, precedence constraints, and selectivities, setting $\pi = \pi'$, $P = P'$ and the rate limits of the operators to be equal to their residual rate limits. The recursive call returns a set K' of flow assignments. RouteChains returns the union of K' and $\{(\pi, x)\}$.

4.3.3 Analysis of the chains algorithm

The correctness of the MTTC algorithm for chains, and the analysis of the algorithm, is stated in the result below.

Theorem 4.1 *Suppose RouteChains is run on inputs having the specified properties, and obeying the "can't saturate" precondition. Let I be the MTTC instance whose rate limits, selectivities, and precedence graph are given by those inputs. Then RouteChains produces a routing that is optimal for I, and would also be optimal if we removed all precedence constraints from I. RouteChains runs in time $O(n^2 \log n)$, and produces a routing that uses at most $4n - 3$ distinct permutations.*

4.3.4 Example

We illustrate our algorithm using the 4-operator selection ordering query shown in Figure 3 (ii), which has three chains, and one precedence constraint, between O_4 and O_5 (Figure 6). We will assume the rate limits

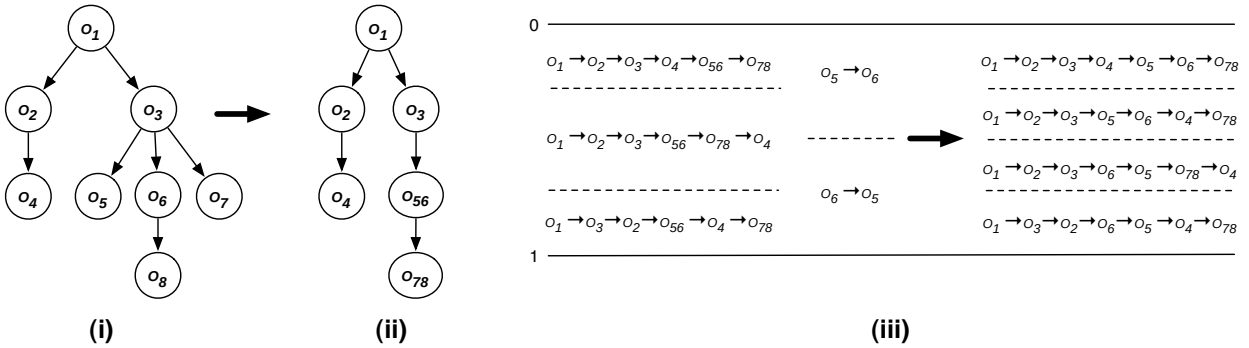


Figure 7: (i) An example precedence graph; (ii) The forest of chains below operator O_3 is replaced by a single chain to obtain a new problem; (iii) The solution for the new problem and for the operator O_{56} are combined together.

of 900 for operators O_2, O_3 and O_4, and a rate limit of 225 for O_5. The selectivity of each operator is set to be 0.5.

- We arbitrarily break the ties, and pick the permutation $O_4 \rightarrow O_2 \rightarrow O_3 \rightarrow O_5$ to start adding flow.
- When 600 units of flow have been added, O_2 exactly saturates O_4 (**stop. cond. 2**). We swap-merge O_2 and O_4 creating superoperator O_{24}.
- At the same time (after adding 0 units of flow), we find O_4 exactly saturates its child O_5 (**stop. cond. 3**). We absorb O_5 into its parent, creating superoperator O_{245}. There is no need to re-sort.
- After sending 240 units along $O_2 \rightarrow O_4 \rightarrow O_5 \rightarrow O_3$, we find that O_3 saturates O_{245} (**stop. cond. 2**). We swap-merge them to get a single super operator O_{3245}.
- We send 720 units along $O_3 \rightarrow O_2 \rightarrow O_4 \rightarrow O_5$, at which point all operators are saturated, and we achieve optimality (**stop. cond. 1**).

The value of the final solution is 1560 tuples; the best serial plan would have only been able to process 900 tuples per unit time.

4.4 MTTC Algorithm for General Trees

We now describe the MTTC algorithm for arbitrary tree-structured precedence graphs. Define a "fork" to be a node in G with at least two children; a chain has no forks. Intuitively, the algorithm works by recursively eliminating forks from G, bottom-up. Before describing the algorithm, we illustrate it with an example.

4.4.1 Example

We illustrate the execution of one recursive call to the MTTC algorithm. (We actually illustrate a simplified process to give the intuition; we discuss the actual process below.) Let the input graph be the one shown in Figure 7 (i). This graph has several forks; let the next fork we eliminate be at node O_3 (the choice is arbitrary). The subtrees under O_3 form a forest of three chains. A new set of operators is constructed from this forest of chains as follows:

- The three chains are made proper.
- RouteChains is used to find an optimal routing K' for these three chains. Suppose that $\{O_7, O_8\}$ is a saturated suffix of K'. Let K_{78} denote the routing (over O_7 and O_8) derived from K' that saturates O_7 and O_8.

- A new operator O_{78} is constructed corresponding to O_7 and O_8. Its rate limit is set to be the throughput achieved by K_{78}, and its selectivity is set to $p_7 p_8$.
- O_7 and O_8 are removed from the three chains, and **RouteChains** is applied again to the operators O_5 and O_6. Suppose the output routing K_{56} saturates both operators.
- A new operator O_{56} is constructed to contain K_5 and K_6. Its rate limit is set to the throughput of K_{56} and its selectivity is set to $p_5 p_6$.
- A new precedence graph is constructed as shown in Figure 7 (ii). Note that K' routes flow first through $\{O_5, O_6\}$ (where all but $p_5 p_6$ of it is eliminated), and then through $\{O_7, O_8\}$. Since K' saturates $\{O_5, O_6\}$, O_{56} can saturate O_{78}, and the new (sub)chain $O_{56}O_{78}$ is proper.

Having eliminated a fork from the graph, the resulting problem is recursively solved to obtain a routing K'', which is then combined with K_{56} and K_{78} to obtain a routing for the original problem, using a technique from [8]. We illustrate this with an example (Figure 7 (iii)).

Suppose K'', the optimal solution for the reduced problem (Figure 7 (ii)), uses three permutations, $(O_1, O_2, O_3, O_4, O_{56}, O_{78})$, $(O_1, O_2, O_3, O_{56}, O_{78}, O_4)$, and $(O_1, O_2, O_3, O_{56}, O_4, O_{78})$, and let the total flow be t. Further, suppose the first and third permutations each carry $\frac{1}{4}t$ flow, and the second permutation carries $\frac{1}{2}t$ flow. Similarly, suppose the routing K_{56} for O_{56} sends half the flow along permutation O_5, O_6 and half along permutation O_6, O_5. These two routings are shown graphically in the first two columns of Figure 7 (iii). In each column, the height of the region allocated to the three permutations indicates the fraction of flow allocated to that permutation by the associated routing. In the third column we superimpose the divisions from the first two columns. For each region R in the divided third column, we label it with the permutation obtained by taking the associated permutation from columns 1, and replacing O_{56} in it with the associated permutation from column 2. For example, the second region from the top in the third column is associated with $O_1, O_2, O_3, O_{56}, O_4$ from column 1 and O_5, O_6 from column 2, and is labeled by combining them. Column three represents a division of flow among permutations of all the operators, yielding a final routing that divides t units of flow proportionally according to this division. The resulting routing allocates $\frac{1}{4}t$ flow to each of four permutations. The same approach would be used to incorporate the routing for K_{78} into the overall routing.

4.4.2 Outline of the MTTC algorithm

We present a summary of the steps in the MTTC algorithm here for convenient reference, and explain the algorithm more fully in the next section.

1. If G is a forest of chains, make the chains proper (Section 4.3.1), use RouteChains to construct the optimal solution, and return the solution.
2. Otherwise, let O_i be a node such that no descendant of O_i has more than one child.
3. Let S denote the set of descendants of O_i, and let S denote the induced MTTC instance I restricted to the operators in S (with the corresponding subgraph of the precedence graph). The precedence graph G_S of I_S is a forest of chains.
4. Make the chains in I_S proper, and call $CombineChains(I_S)$ to get a partition $(A_1, \cdots, A_m)$ of the operators of I_S, and routings $K_{A_1}, \cdots, K_{A_m}$ corresponding to the partitions.
5. Create m new operators, $\mathcal{A}_1, \cdots, \mathcal{A}_m$ corresponding to the A_i's. For each $\mathcal{A}_i$, the rate limit of $\mathcal{A}_i$ is defined to be the throughput of K_{A_i}, and the selectivity is defined to be the product of the selectivities of the operators in $\mathcal{A}_i$.

6. Construct a new instance of the MTTC problem, I', by replacing the the chains below O_i with the single chain $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$.
7. Recursively solve I'. Let the routing returned by the recursive call be K'.
8. Use CombineRoutings to combine the K_{A_i} with K'. Return the resulting routing.

4.4.3 Algorithm Description

As described above, the MTTC algorithm is a recursive procedure that works by repeatedly eliminating forks in the precedence graph.

Base Case: The base case is a graph with no forks, i.e. a forest of chains, and it is solved using the algorithm for chains from Section 4.3.

Eliminating a fork with CombineChains: In the example above, we eliminated a fork by running RouteChains repeatedly on the chains emanating from that fork, each time removing the operators in a saturated suffix. The actual MTTC algorithm does something slightly different. It identifies the operators in a saturated suffix of some optimal routing; it then runs RouteChains just on the operators in that saturated suffix to produce a routing just for those operators. As in the example, it then removes the operators in the suffix, and repeats. We call this procedure CombineChains. Before running it, the MTTC algorithm makes the chains proper, if necessary (using the procedure in Section 4.3.1).

Formally, CombineChains solves the following problem: Given an MTTC instance I whose precedence graph G is a forest of proper chains, find an ordered partition $(A_1, \ldots, A_m)$ of the operators of I, and routings $K_1, \ldots, K_m$, such that the partition and routings satisfy the following properties:
For each A_i (1) the ancestors in G of the operators in A_i are all contained in $\bigcup_{j=1}^{i-1} A_j$, (2) K_{A_i} is a routing for just the operators in A_i that saturates all of them, and (3) if $i \neq 1$ then the maximum throughput attainable for just the operators in A_{i-1}, multiplied by the product of their selectivities, is at least as big as the maximum throughput attainable for just the operators in A_i.

CombineChains first sorts the operators in I in descending order of their rate limits. It (re)numbers them $O_1, \ldots, O_n$ so that $r_1 \geq \ldots \geq r_n$. This ordering obeys the precedence constraints because the chains of G are proper.

CombineChains then executes the following recursive procedure on I. It computes the value

$$F^* = min_{j \in \{1,\ldots,n\}} \frac{\sum_{i \in Q_j} r_i(1-p_i)}{(\prod_{k \notin Q_j} p_k)(1 - \prod_{i \in Q_j} p_i)}$$

It sets C_{j^*} to be $\{O_{j^*}, \ldots, O_n\}$, where j^* is the largest value of j achieving the minimum value F^*. It can be shown (Lemma 9.3 in the appendix and Lemma 4.2) that C_{j^*} is a saturated suffix in an optimal routing of the chains. CombineChains runs the MTTC algorithm for chains from Section 4.3 just on the (sub)chains of operators in C_{j^*}, to produce a routing K_{j^*}. (Since the chains are already proper, no rate limits are reduced in this process.)

CombineChains then removes the operators in C_{j_*} from I. It also removes them from the chains; note that the operators in C_{j_*} will always appear at the end of the chains, because of the precedence constraints. If no operators remain in the chains, CombineChains outputs the one-item list A_1 where $A_1 = C_{j_*}$, together with routing $K_1 = K_{j^*}$. Otherwise, CombineChains executes a recursive call on the remaining operators to produce a list of operator subsets $D = A_1, \ldots, A_{m-1}$, together with a corresponding list $K_1, \ldots, K_{m-1}$ of

routings for the operators in each of the A_i. It then sets $A_m = C_{j^*}$, appends it to the end of D, appends K_{j^*} to the end of the associated list of routings, and outputs the result.

Recursive call on tree with one fewer fork:

After running CombineChains on the chains descending from a fork, there is an ordered partition $D = (A_1, \ldots, A_m)$ of the operators in those chains, and there are routings $K_{A_1}, \ldots, K_{A_m}$ for the subsets A_i. In precedence graph G, the MTTC algorithm replaces the chains descending from the fork by a single chain of new operators corresponding to $A_1, \ldots, A_m$. For each A_i, it sets the rate limit of the corresponding new operator to be the throughput of the routing corresponding to A_i, and its selectivity to be the product of the selectivities of the operators in A_i. (It can be shown that the chain of new operators is proper, a fact that we use in proving the correctness of the MTTC algorithm.)

The MTTC algorithm then recursively finds an optimal routing K' for the resulting MTTC instance, which has a tree with one fewer fork than before. It remains only to combine K' with $K_{A_1}, \ldots, K_{A_m}$ to produce an optimal routing for the original operators.

Combining Routings: The MTTC algorithm uses the following procedure (also used in [8]) to combine the recursively computed routing K' with the routings $K_1, \ldots, K_m$, into a routing for the operators in the current call. We call the procedure *CombineRoutings*. The MTTC algorithm ends by returning the routing computed by CombineRoutings.

CombineRoutings divides the real interval $[0, 1]$ once for K', and once for each K_{A_i}. The division for a routing divides $[0, 1]$ into as many subintervals as there are permutations used in the routing, each of these permutations is associated with an interval, and the length of each subinterval is equal to the proportion of flow the routing sends along the corresponding permutation. It then superimposes the divisions to produce a new division which yields a combined routing for K' and the K_{A_i}, in a manner analogous to what was shown in the example. The number of permutations that are used in the combined routing at most the sum of the number of permutations used in K' and the total number of permutations used in the K_{A_i}'s.

The combined routing satisfies the following properties, which are needed for its use in the MTTC algorithm: (1) for each permutation π of the A_i's, if K' sends fraction α of its flow along permutation π, then K'' sends α of its flow along permutations of the O_j's respecting the ordering π (2) for each permutation π of the O_j's in an A_i, if K_{A_i} sends a fraction β of its total flow along that permutation, then K'' does also. (3) the throughput of K'' is equal to the throughput of K'.

4.4.4 Analysis

The analysis of the MTTC algorithm is summarized in the following theorem.

Theorem 4.2 *When run on an MTTC instance I, the MTTC algorithm runs in time $O(n^3)$ and outputs an optimal routing for I. The output routing uses fewer than $4n$ distinct permutations.*

5 The generalized MTTC problem: selective and non-selective operators

Multi-way join queries may contain non-selective operators with fanouts larger than 1. In this section, we discuss how the previous algorithm for tree-structured precedence constraints can be extended to handle such cases. We note that, in such cases, it might be preferable to consider an alternative plan space [26] or a caching-based approach. We will discuss this in detail in Section 7.

We define the *generalized* MTTC problem to be the generalization of the MTTC problem in which operators may be either selective ($0 < p_i < 1$) or non-selective ($p_i \geq 1$). We begin by considering the case in which *all* operators are non-selective; we then consider the scenario in which there are selective and non-selective operators.

5.1 All non-selective operators

If all operators are non-selective, then we construct an equivalent problem with only selective operators as follows:

- The selectivity p_i of an operator O_i is replaced by $1/p_i$.
- All precedence constraints are reversed.

After solving this new problem (which only contains selective operators), we reverse each of the permutations in the resulting routing to obtain a valid routing for the original problem.

Note that the precedence graph for the new problem may be an "inverted tree". Define an "inverted fork" to be a node that has more than one parent in the precedence graph.

The MTTC algorithm described in Section 4 can be used to solve such instances in a straightforward manner. The only difference is that, instead of eliminating forks in a bottom-up order, we instead eliminate the "inverted forks" in a top-down order. We call this the MTTC-INV algorithm. We can prove an analogous statement to Theorem 4.2.

Theorem 5.1 *When run on an MTTC instance I whose precedence graph is an inverted tree and which contains only selective operators, the MTTC-INV algorithm runs in time $O(n^3)$ and outputs an optimal routing for I. The output routing uses fewer than $4n$ distinct permutations.*

5.2 Mixture of Selective and Non-Selective Operators

However, if the problem instance contains a mixture of selective and non-selective operators, the problem is more complex. Next we present an optimal algorithm for the case in which the precedence graph is a forest of chains (or there are no constraints). The algorithm is based on following lemma.

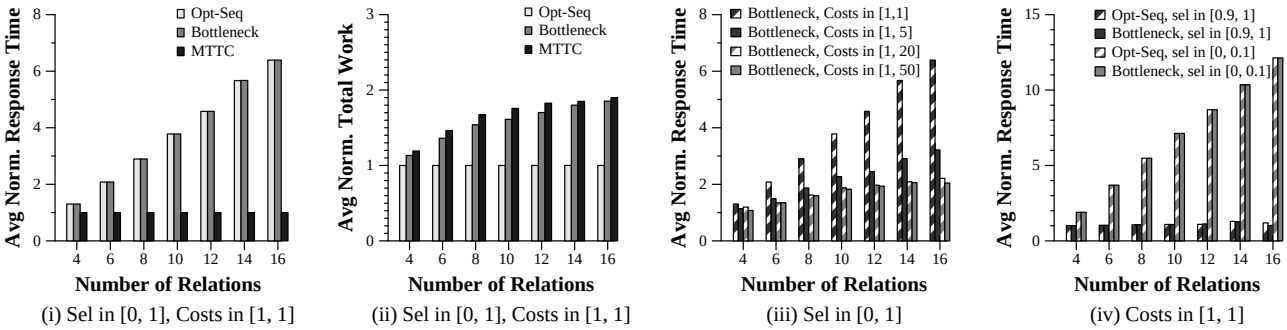
Lemma 5.1 *Given an instance I containing both selective and non-selective operators, there exists an optimal solution K satisfying the following property: for all O_i, O_j such that $p_i \geq 1$ and $p_j < 1$:*

- ***(A)** If O_i and O_j have no precedence constraint between them, then O_j does not immediately follow O_i in any permutation used in the routing.*
- ***(B)** If O_j is the only child of O_i, then O_j immediately follows O_i in every permutation used in the routing.*

5.2.1 Case: Forest of Chains

Let I denote a problem instance such that the precedence graph is a forest of chains (or there are no precedence constraints). We use the above lemma to solve this case as follows:

- **Pre-processing Step:** If there is a parent-child pair, O_i, O_j, such that $p_i \geq 1$ and $p_j < 1$, then replace the two operators with a new operator O_{ij} with selectivity $p_i p_j$ and rate limit $\min(r_i, r_j/p_i)$. From Lemma 5.1 (B), this does not preclude attainment of the optimal solution.



(i) Sel in [0, 1], Costs in [1, 1] (ii) Sel in [0, 1], Costs in [1, 1] (iii) Sel in [0, 1] (iv) Costs in [1, 1]

Figure 8: *Star* query experiments: Comparing (i) response times (normalized using the MTTC solution) and (ii) total work (normalized using the OPT-SEQ solution) for the three algorithms; (iii) With high variance in the operator costs, benefits of interleaving plans are lower; (iv) Interleaving plans are most beneficial when selectivities are low.

- Repeat until there are no such pairs. In the resulting problem, I', no non-selective operator precedes a selective operator.
- Split the problem into two problems, I'_s and I'_{ns}, such that I'_s contains the selective operators (along with the precedence constraints between the selective operators), and I'_{ns} contains the non-selective operators (along with the precedence constraints between them).
- From Lemma 5.1 (B), we can infer that there exists an optimal solution such that in every permutation used, the selective operators ($\in I'_s$) precede the non-selective operators ($\in I'_{ns}$).
- Solve the two problems separately to obtain routings K'_s and K'_{ns}.
- Combine the two routings in a fashion similar to CombineRoutings (Section 4.4).

The correctness follows from Lemma 5.1.

5.2.2 Case: Arbitrary Tree-Structured Precedence Constraints

Given the above algorithm for forests of chains, we can once again apply the procedure described in Section 4.4 to obtain an algorithm for solving instances with tree-structured precedence constraints. We conjecture that the algorithm returns an optimal routing; however, we have been unable to prove this so far.

6 Experimental Study

In this section, we present an extensive performance evaluation of the algorithms presented in the paper demonstrating both the benefits of using interleaving plans to reduce the response times and the effectiveness of our algorithms at finding such plans. We compare three planning algorithms:

- **OPT-SEQ [23]:** The optimal serial plan for the centralized case, that minimizes the total work done, found using the *rank ordering* algorithm (Section 3.3).
- **BOTTLENECK [26]:** The serial plan that minimizes the response time (bottleneck) using a serial plan, found using the Bottleneck Algorithm (Section 3.3).
- **MTTC:** The optimal interleaving plan found by our algorithm presented in Section 4.

We have implemented the above algorithms in a single-threaded simulation framework (implemented in Java) that simulates execution of a multi-way join query using pipelined parallelism. To execute a query with n relations, $n-1$ processors are instantiated, with each processor handling one of the driven relations. We use hash joins for executing the queries. We control the join selectivities by appropriately choosing the sizes of the driven relations; the join attributes are all set to have a domain of size 1000, and to simulate a selectivity of p, the corresponding relation is set to have $1000p$ randomly chosen tuples.

Each plotted data point in our graphs corresponds to 50 random runs. In all but one of the graphs, we plot the average value of the normalized response time (response time of the plan found using MTTC is used as the normalizing factor). In one of the graphs (Figure 8 (ii)), we plot the average value of the normalized total work (total work done by OPT-SEQ is used as the normalizing factor).

The effectiveness of interleaving plans depends heavily on the query graph shape; queries with shallow precedence graphs can exploit the parallelism more effectively than queries with deep precedence graphs. To illustrate this, we show results for 3 types of query graphs: (1) *star*, (2) *path*, and (3) *randomly-generated* query graphs.

Star Queries

For our first set of experiments, we use star queries with the central relation being the driver relation. First, we compare the normalized response times of the three planning algorithms for a range of query sizes (Figure 8 (i)). For this experiment, the operator selectivities were chosen randomly between 0 and 1, and the operator costs were assumed to be identical (which corresponds to the common case of homogeneous processors). As we can see, the interleaving plans found by the MTTC algorithm perform much better than any serial plan (in many cases, by a factor of 5 or more). Note that, since all operator costs are identical, the plan found OPT-SEQ is the optimal serial plan for response time as well.

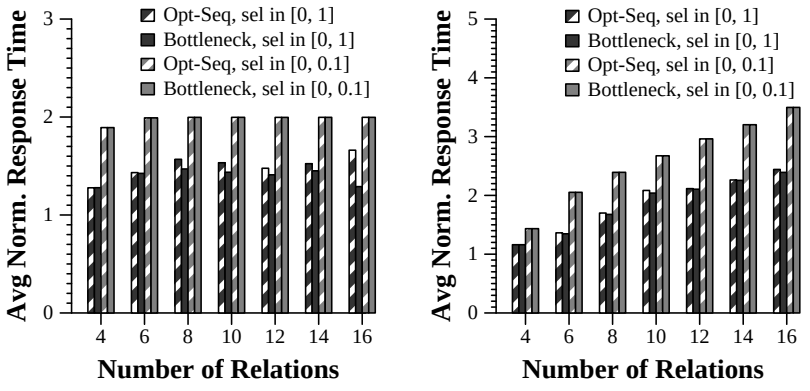
We next compare the total work done by the plans found by these algorithms (Figure 8 (ii)). As expected, the interleaving plans do more work than the OPT-SEQ plan (by up to a factor of 2), but the additional work done is not very high compared to the benefits in the response time that we get by using an interleaving plan. Interestingly, the BOTTLENECK plans also perform a lot more work than the OPT-SEQ plans, even though their response times are identical. Since all operator costs (c_i's) are equal, the BOTTLENECK algorithm essentially picks arbitrary plans (cf. Section 3.3); although optimal for the response time metric, those plans behave unpredictably with respect to the total work metric.

With the next experiment, we illustrate the effects of heterogeneity in the operator costs. For this experiment, we choose the operator costs randomly between 1 and X, where $X \in \{1, 5, 20, 50\}$. As we can see in Figure 8 (iii), with increasing heterogeneity in the operator costs, the benefits of using interleaving plans go down. This is because the total idle time across the operators, which the interleaving plans exploit, goes down significantly in such cases.

Finally, we illustrate how operator selectivities affect the performance of the algorithms. Figure 8 (iv) compares the performance of the three algorithms when the operator selectivities are high (chosen randomly between 0.9 and 1), and when they are low (chosen randomly between 0 and 0.1). As we can see, the benefits of interleaving plans are highest when the selectivities are low. This is because low selectivities result in higher overall idle time across the processors. On the other hand, when the selectivities are very high, the benefits of using interleaving plans are very low (around 10-20%). We note that star queries where all join selectivities are 0 form the best-case scenario for interleaving plans.

Path Queries

Next we compare the performance of the three algorithms when the query graph shape is a path (line), and the relation in the middle of the graph is chosen as the driver. This query essentially results in two long



(i) Path Queries (ii) Random Queries

Figure 9: Comparing the three planning algorithms for (i) path query graphs and (ii) randomly-generated query graphs.

precedence chains, and does not offer much parallelism. In fact, it is easy to show that the response time of the BOTTLENECK algorithm is within a factor of 2 of the best interleaving plan. We compare the three algorithms for two sets of selectivities. As we can see in Figure 9 (i), the interleaving plans can achieve close to a factor of 2 when the operator selectivities are low. When the selectivities are drawn randomly between 0 and 1, the benefits range from about 30% for small query sizes to about 60% for large queries.

Randomly-generated Queries

Finally, we run experiments on random query graphs generated by randomly choosing a parent for each node in the graph, while ensuring that the resulting graph is a tree. Figure 9 (ii) shows the results for this set of experiments. As we can see, even for queries with as few as 6 relations, we get significant benefits, even when the operator selectivities are chosen between 0 and 1. For low selectivity operators and for higher query sizes, the benefits of using interleaving plans are much higher.

7 Caching-aware Reduction for Non-selective Operators

As discussed in Section 5, the MTTC algorithm can still be applied when there are non-selective operators (also called *proliferative* [26]). However, as we illustrate below, non-selective operators lead to redundant work, and result in more probes into the join operators than minimally required (in fact, this could be true for selective operators as well). These can be avoided by either using an alternative plan space (as Srivastava et al. [26] demonstrate), or using a more careful executor. We explore the latter approach in this section.

7.1 Redundancy with Non-selective Operators

We illustrate this using an example. Consider the execution plan shown in Figure 1, and consider a tuple $x_0 \in R_1$ that produces 10 tuples after the join with R_2, $(x_0, y_0), \ldots, (x_0, y_9)$. According to the execution plan, these tuples should next be used to find matches in R_3. However, note that the matches with R_3 depend only on the R_1 component of these tuples (ie., on x_0), and the same matches (or none) will be returned from the probe into R_3 for all of these 10 tuples; such redundant probes should clearly be avoided.

There are two ways to achieve this.

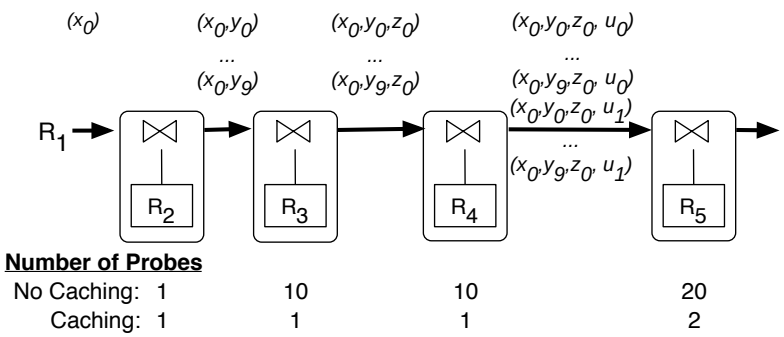


Figure 10: We can avoid the redundant probes into R_3, R_4 and R_5 using a caching-based execution model.

- **Caching:** From the engineering perspective, perhaps the easiest way to avoid these redundant probes is to cache the results of the probes in the join operator temporarily. If an identical probe is made into the table, then those results can be returned back without probing into the hash table or the index [25]. The duration for which these cached results should be kept depends on the behavior of the router.

  For the example shown in Figure 10, the first probe into R_3 using (x_0, y_0) will result in z_0 being cached in the join operator. Each of the subsequent probes will return the same result without executing a probe (since the probe into R_3 is done using x_0).

- **Independent/parallel probes:** Another option is to probe into the join operators independently and perform a cross-product of the probe results at the end. Several works have explored this for *star* queries [3, 4, 5]. Extending this to non-star queries is fairly straightforward.

Although some prior work has considered this issue for certain classes of queries, we are not aware of any general treatment of caching for query processing. Parallel plans, as explored by Srivastava et al. [26], behave quite differently from an ideal caching-based or independent probe-based approach; they cannot eliminate the redundancy entirely.

7.2 Reduction to Selection Ordering

In this section, we will consider how a query being evaluated using an ideal caching-based approach or the independent probes-based approach may still be reduced to precedence-constrained selection ordering. Unfortunately, the general case of this problem does not admit a direct reduction to selection ordering.

Before going on, we recap some of the earlier definitions and define the following additional terms.

For a join operator on relation R_i, let c_i denote the cost of probing into the corresponding hash table. Let $fanout_{R_i}$ denote the fanout of the join operator (which would have been used as the selectivity s_i in the prior reduction).

Further, let q_{R_i} denote the *match probability*, ie., the probability that a probing tuple finds a match (Figure 11 (i)). And let m_{R_i} denote the number of matches per successful probe. Note that, even if $fanout_{R_i} > 1$, we have that $q_{R_i} \leq 1$. Also, $fanout_{R_i} = q_{R_i} \times m_{R_i}$. Figure 11 (i) illustrates this with an example.

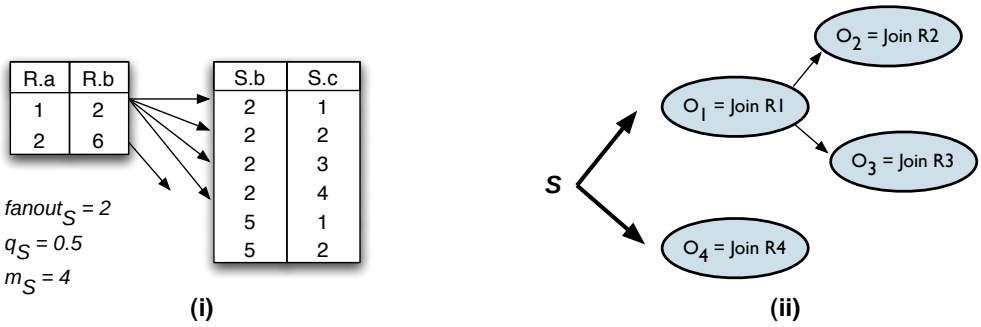


Figure 11: (i) Even though the fanout of the join with S is 2, the $match$ probability is only 0.5; (ii) Smallest graph for which a direct caching-aware reduction to selection ordering is not possible.

7.2.1 Star Query Graphs

To reduce a star multi-way join query with the center relation being the driver, we create the selection operators O_i as before, and set the cost to be the probe cost as before. There are no precedence constraints between the operators. The selectivity of the operator is set to be q_{R_i} (instead of $fanout_{R_i}$). It is easy to see that this captures the behavior of the ideal caching-based approach.

7.2.2 General Graphs

Unfortunately, aside from this and some additional special cases (a direct reduction also exists if the precedence graph is a forest of chains), arbitrary precedence-constrained trees cannot be directly reduced to selection ordering. Figure 11 (ii) shows the simplest graph for which a direct reduction to selection ordering is not possible.

8 Conclusions and Future Work

In this paper, we considered the problem of executing a multi-way join query using pipelined parallelism, and presented algorithms to optimally exploit the parallelism in the system through use of interleaving plans for a large class of join queries. Our experimental results demonstrate that the interleaving plans can effectively exploit the parallelism in the system to minimize query response times, sometimes by orders of magnitude. Our work so far has opened up a number of interesting future research directions, such as handling correlated predicates, non-uniform join costs, and multiple driver tables (a scenario common in data streams), that we are planning to pursue in future.

References

[1] P. Apers, A. Hevner, and S. Yao. Optimization algorithms for distributed queries. *IEEE Trans. Software Eng.*, 9(1), 1983.

[2] Ron Avnur and Joe Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD*, 2000.

[3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, 2004.

[4] S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *ICDE*, 2005.

[5] Pedro Bizarro and David DeWitt. Adaptive and robust query processing with sharp. Technical Report 1562, University of Wisconsin - Madison. CS Dept., 2006.

[6] C. Chekuri, W. Hasan, and R. Motwani. Scheduling problems in parallel query optimization. In *PODS*, 1995.

[7] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Flow algorithms for two pipelined filter ordering problems. In *PODS*, 2006.

[8] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Algorithms for distributional and adversarial pipelined filter ordering problems. Unpublished manuscript, 2007.

[9] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, 2005.

[10] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. Technical Report CS-TR-4873, Univ of Maryland, 2007.

[11] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *CACM*, 35(6), 1992.

[12] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.

[13] XXX D. J. DeWitt. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.

[14] L. Fleischer and K. Wayne. Fast and simple approximation schemes for generalized flow. *Mathematical Programming*, 91(2), 2002.

[15] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, 1992.

[16] M. Garofalakis and Y. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *SIGMOD*, 1996.

[17] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, 1997.

[18] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, 1990.

[19] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 1996.

[20] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *VLDB*, 1995.

[21] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in xprs. In *PDIS*, 1991.

[22] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.

[23] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.

[24] Bin Liu and Elke A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, 2005.

[25] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, 2003.

[26] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, 2006.

[27] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.

[28] Tolga Urhan and Michael J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

[29] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.

[30] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. In *SIGMOD*, 1995.

9 Appendix

Below we provide proofs of the lemmas and theorems in the body of the paper. We also present additional lemmas and propositions needed for those proofs. Throughout the appendix, for the convenience of the reader, we will restate any lemmas or theorems from the body of the paper in addition to giving the proofs.

9.1 Proofs: Fundamental lemmas

We will use the following additional lemma in the proof of Lemma 4.2.

Lemma 9.1 *Let F^* be the optimal value of the objective function in the MTTC problem. Let $Z = \frac{\sum_{i=1}^{n} r_i(1-p_i)}{(1-\prod_{i=1}^{n} p_i)}$. If $F^* = Z$, then every optimal routing saturates all the operators. Further, if there exists a routing that saturates all the operators, then that routing is optimal, achives throughput Z, and all optimal routings saturate all the operators.*

Proof. For any operator O_i, since operator O_i has a rate limit of r_i, O_i can process at most r_i flow units per unit time. Since O_i has selectivity p_i, it discards at most $r_i(1 - p_i)$ amount of flow per unit time. Thus under any feasible routing, the total amount of flow discarded by all the processors per unit time is at most $\sum_{i=1}^{n} r_i(1 - p_i)$; this is the precise amount if all processors are saturated, and is a strict upper bound otherwise.

Consider a routing achieving throughput F^*. Of the F^* amount of flow that is sent into the system in this routing, the amount that successfully travels through all the processors (i.e. is not discarded by any) is $F^* \prod_{i=1}^{n} p_i$. Thus the total amount that is discarded by some processor, per unit time, is $F^*(1 - \prod_{i=1}^{n} p_i)$. By the above, $F^*(1 - \prod_{i=1}^{n} p_i) \leq \sum_{i=1}^{n} r_i(1 - p_i)$, with equality iff the processors are all saturated. Since $Z = \frac{\sum_{i=1^n} r_i(1-p_i)}{(1-\prod_{i=1}^{n} p_i)}$, $F^* = Z$ implies that all operators are saturated.

Finally, given a routing that saturates all the operators, by the above arguments the routing achieves throughput Z. The lemma follows. □

Lemma 4.2 *Let I be an instance of the MTTC problem in which there are no precedence constraints, and let the operators in I be numbered so that $r_1 \geq r_2 \ldots \geq r_n$. Let*

$$F^* = min_{k \in \{1,\ldots,n\}} \frac{\sum_{i=k}^{n} r_i(1 - p_i)}{(\prod_{j=1}^{k-1} p_j)(1 - \prod_{i=k^n} p_i)}$$

Then F^* is the optimal value of the objective function for instance I. If k' is the largest value of k achieving the value F^*, then there exists an optimal routing for which $\{O_{k'}, O_{k'+1}, \ldots, O_n\}$ is a saturated suffix. If the optimal value F^* is achieved at $k = 1$, then every feasible routing K saturates all operators.

Proof of Lemma 4.2: For $a, b \in \{1, \ldots, n\}$, let $S_{a,b} = \sum_{i=a}^{b} r_i(1 - p_i)$, $P_{a,b} = \prod_{i=a}^{b} p_i$, and $h_{a,b} = \frac{S_{a,b}}{(1-P_{a,b})}$. Thus $F^* = min_{k \in \{1,\ldots,n\}} \frac{1}{P_{1,k-1}} h_{k,n}$.

By the results of [7, 8], F^* is the optimal value of the objective function for the given MTTC instance I, and for *some* k^* such that $F^* = \frac{1}{P_{1,k^*-1}} h_{k^*,n}$, there exists an optimal routing for which $\{O_{k^*}, O_{k^*+1}, \ldots, O_n\}$ is a saturated suffix. It remains only to show that what holds for k^* also holds for k'. If $k' = k^*$, then this is trivially true.

Suppose $k' \neq k^*$. Then $k' > k^*$. Consider the optimal routing for which $\{O_{k^*}, O_{k^*+1}, \ldots, O_n\}$ is a saturated suffix. Call it K. Routing K sends F^* amount of flow into the system, with all flow traveling first through the operators in $Q_{pref} = \{O_1, \ldots, O_{k^*-1}\}$ and then through the operators in $Q_{suff} = \{O_{k^*}, O_{k^*+1}, \ldots, O_n\}$. Of the F^* flow sent into the system, $P_{1,k^*-1}F^*$ of it reaches Q_{suff}, and it saturates those operators. Let I_1 be the induced MTTC instance created by keeping only the operators in Q_{suff} (i.e. by discarding the other operators and removing them from the precedence graph). It follows from the above that there is a saturating routing for I_1 that achieves throughput $P_{1,k^*-1}F^*$. By Lemma 9.1, $P_{1,k^*-1}F^* = h_{k^*,n}$.

We will show that, in fact, there exists a saturating routing $\hat{K}$ for I_1 that has $\{O_{k'}, O_{k'+1}, \ldots, O_n\}$ as a saturated suffix. Routing $\hat{K}$ can be used to construct the following routing for the original MTTC instance on $O_1, \ldots, O_n$: Send F^* flow through Q_{pref} as specified by K, and then send the $P_{1,k^*-1}F^*$ surviving (i.e. not eliminated) flow through Q_{suff} as specified by $\hat{K}$. Since $\{O_{k'}, \ldots, O_n\}$ is a saturated suffix of this routing, this proves the lemma.

It remains to show that $\hat{K}$ exists. We will use the following claim, which is easily shown by algebraic manipulation, the definitions of P and h, and the fact that the selectivities p_i are strictly between 0 and 1.

Claim: Let $a, k, b \in \{1, \ldots, n\}$ where $a \leq k \leq b$.

$$\frac{1}{P_{a,k-1}} h_{k,b} \leq h_{a,b}$$

iff

$$\frac{1}{P_{a,k-1}} h_{k,b} \leq h_{a,k-1},$$

and if one inequality holds strictly, so does the other.

Let I_3 be the instance induced from I_1 by keeping only the operators in $\{O_{k'}, \ldots, O_n\}$. By the definition of k', for all j such that $k' < j < n$, $\frac{1}{P_{1,k'-1}} h_{k',n} < \frac{1}{P_{1,j-1}} h_{j,n}$, and hence $h_{k',n} < \frac{1}{P_{k',j-1}} h_{j,n}$. Therefore, by the result stated at the start of this proof, there is an optimal routing for I_3 for which $\{O_{k'}, \ldots, O_n\}$ is a saturated suffix, and hence this routing is saturating for I_3 and achieves a throughput of $h_{k',n}$. Let K_3 denote this routing.

Let I_2 be the MTTC instance induced from I_1 by keeping only the operators in $\{O_{k^*}, \ldots, O_{k'-1}\}$. We show that there is a routing achieving throughput $h_{k^*,k'-1} = h_{k^*,n}$ on this instance. Assume not. Then by the properties of F^* given at the start of this proof, there exists i such that $k^* < i \leq k'$ and

$$\frac{1}{P_{k^*,i-1}} h_{i,k'-1} < h_{k^*,k'-1} \tag{1}$$

By the definitions of k' and k^*, $\frac{1}{P_{1,k^*-1}} h_{k^*,n} = \frac{1}{P_{1,k'-1}} h_{k',n}$, and multiplying both sides by P_{1,k^*-1} we get that

$$h_{k^*,n} = \frac{1}{P_{k^*,k'-1}} h_{k',n} \tag{2}$$

Applying the claim to Equation 2, we get that

$$h_{k^*,k'-1} = \frac{1}{P_{k^*,k'-1}} h_{k',n} \tag{3}$$

Combining Equation 3 and Inequality 1, we get

$$\frac{1}{P_{k^*,i-1}} h_{i,k'-1} < \frac{1}{P_{k^*,k'-1}} h_{k',n} \tag{4}$$

Multiplying both sides by $P_{k^*,i-1}$, we get that

$$h_{i,k'-1} < \frac{1}{P_{i,k'-1}} h_{k',n} \tag{5}$$

Applying the claim to Inequality 5 yields

$$h_{i,n} < \frac{1}{P_{i,k'-1}} h_{k',n} \tag{6}$$

Multiplying both sides of the above equation by $\frac{1}{P_{1,i-1}}$, we get that

$$\frac{1}{P_{1,i-1}} h_{i,n} < \frac{1}{P_{1,k'-1}} h_{k',n} \tag{7}$$

But this contradicts the definition of k' in the statement of the lemma, since in the definition for F^*, setting k' to k must minimize the given expression, and thus setting k to i cannot achieve a smaller value for it. So $h_{k^*,k'-1}$ is the value of the maximum throughput for I_2, and there is a routing K_2 achieving this throughput on I_2. By Lemma 9.1, this routing is saturating for I_2.

□

9.2 Proofs: The MTTC algorithm for chains

For $y > 0$, we define the residual rate limit of an operator O_i *relative to* (π, y) to be $r_i - g(\pi, y)$.

We will use the following proposition. Informally, the first part of the proposition says that if an operator O_j cannot saturate an operator O_i or can only saturate it exactly, but adding $z > 0$ amount of flow along permutation π will cause it to be able to over-saturate O_j, then for some $0 \leq w < z$, adding w amount of flow instead will cause O_j to be able to exactly saturate O_i. The rest of the proposition is similar.

Proposition: For $y > 0$, let $r_i^{(y)}$ and $r_j^{(y)}$ denote the residual rate limits of O_i and O_j respectively, relative to (π, y). If $r_j p_j \leq r_i$ and $z > 0$ is such that $r_j^{(z)} p_j > r_i^{(z)}$, then there exists $0 \leq w < z$ such that $r_j^{(w)} p_j = r_i^{(w)}$. Similarly if $r_j p_j \geq r_i$, and $z > 0$ is such that $r_j^{(z)} p_j <\leq r_i^{(z)}$, then there exists $0 \leq w < z$ such that $r_j^{(w)} p_j = r_i^{(w)}$. The analogous proposition holds for superoperators.

The proposition is easily proved, using the observation that a continuous increase in the amount of flow y sent along permutation π results in a continuous decrease in the residual capacity of each operator.

The correctness of RouteChains relies on the following invariants.

Lemma 9.2 *Suppose that RouteChains is called with inputs obeying the properties in the input specification (RouteChains:Inputs), and the "can't saturate" pre-condition. Then the same holds for the inputs to the next recursive call of RouteChains.*

Proof: Assume that RouteChains is called with inputs satisfying the conditions of the lemma. Thus in particular,

(a) Each π_i in P is a superoperator,

(b) The "can't saturate" precondition holds

(c) π satisfies all precedence constraints of G

(d) The chains in G are proper.

We show that these conditions hold on the next recursive call to RouteChains.

There are two cases, depending on whether a swap-merge or a parent-child absorption is performed before the next recursive call.

Case 1: Swap-merge

Suppose that just prior to the next recursive call, a swap-merge is performed with superoperators π_{i-1} and π_i.

We first show that Condition (a) holds for the inputs to the next recursive call. Consider any two adjacent operators O_{j-1} and O_j in π. Since they are adjacent, $g(\pi, j-1)p_{j-1} = g(\pi, j)$. Thus if $r_{j-1}p_{j-1} = r_j$ then $(r_{j-1} - xg(\pi, j-1))p_{j-1} = r_j - xg(\pi, j)$. In other words, if O_{j-1} can exactly saturate O_j, then this remains true (with respect to the residual capacity) after x units of flow are sent along permutation π. It follows that every input superoperator π_j not involved in the swap-merge (i.e. $j \notin \{i-1, i\}$), remains a superoperator after the addition of x units of flow along permutation π.

Further, after the addition of x units of flow along permutation π, π_i can exactly saturate π_{i-1}. Since the selectivity of π_i is the product of the selectivity of its component operators, following the addition of x units of flow along permutation π, every operator within the merged superoperator $\pi_i\pi_{i-1}$ can exactly saturate its successor in $\pi_i\pi_{i-1}$. Thus Condition (a) holds for the next recursive call.

We now show that the inputs to the next recursive call satisfy Condition (b). Suppose not. Then in the inputs to the next recursive call, some superoperator can over-saturate its predecessor. The superoperators in the next recursive call are the same as in the current call, except for the merged superoperator $\pi_i\pi_{i-1}$. For

$1 < j \leq m$ such that $j \notin \{i-1, i, i+1\}$, π_j couldn't over-saturate its predecessor π_{j-1} at the start of the current call. If π_j can over-saturate its predecessor after the addition of x flow along permutation π, then by the proposition, there must be $0 \leq x' < x$ such that adding x' flow along permutation π would cause π_j to be able to exactly saturate π_{j-1}. This contradicts the minimality of x, proving that pi_j couldn't over-saturate π_{j-1} after the addition of x flow along permutation π.

We now show that in the input to the next call, merged operator $\pi_i\pi_{i-1}$ can't over-saturate π_{i-2}. Let ρ'_i and ρ'_{i-1} denote the residual rate limits of O_i and O_{i-1} relative to (x, π). Then $\rho'_{i-1}\sigma(\pi_{i-1}) = \rho'_i$. By the same argument as in the previous paragraph, since π_{i-1} couldn't over-saturate π_{i-2} at the start of the current call, it can't over-saturate π_{i-2} following the addition of x flow along permutation π. Thus $\rho'_{i-1}\sigma(\pi_{i-1}) \leq \rho'_{i-2}$. Since π_i and π_{i-1} were the ones involved in the swap-merge, $\rho'_i\sigma(\pi_i) = \rho'_{i-1}$. Thus $\rho'_i\sigma(\pi_{i-1}) \leq \rho'_{i-2}$, and since $0 < \sigma(\pi_i) < 1$, $\rho'_i\sigma(\pi_{i-1})\sigma(\pi_i) \leq \rho'_{i-2}$. But since $\sigma(\pi_{i-1})\sigma(\pi_i)$ is the selectivity of the merged operator, and ρ'_i is its rate limit, it follows that in the input to the next recursive call, the merged superoperator $\pi_i\pi_{i-1}$ cannot saturate its predecessor.

Similarly, defining ρ'_{i+1} analogously to ρ_i and ρ_{i-1}, one can use the proposition to show that since π_{i+1} couldn't over-saturate π_i at the start of the current call, it can't over-saturate π_i following the addition of x flow along permutation π. Since ρ'_i is precisely the rate limit of the merged superoperator at the start of the next recursive call, π_{i+1} cannot over-saturate its predecessor at the start of the next recursive call.

We now consider Precondition (c). Suppose the inputs to the next recursive call do not satisfy (c). Then there are operators $O_k \in \pi_{i-1}$ and $O_l \in \pi_i$ such that O_k is an ancestor of O_l in a chain of G. Since (c) holds at the start of the given recursive call, on the subchain of G from O_k to O_l, there must exist a parent O_{k^*} and child O_{l^*} such that O_{k_*} is in π_{i-1} and O_{l^*} is in π_i. As already shown, after the addition of x flow along permutation π, each operator in the merged superoperator $\pi_i\pi_{i-1}$ can exactly saturate its successor within that superoperator (if any). Since selectivities are less than one, it follows that $r'_{l^*} > r'_{k^*}$ and thus $r'_{k^*}p_{k^*} < r'_{l^*}$. So after the addition of x flow along permutation π, O_{k^*} cannot saturate O_{l^*}. But by Condition (d), all chains were proper at the start of the current call, and so O_{k^*} could saturate O_{l^*}. By the proposition, for some $0 \leq x^* < x$, adding x^* flow along permutation π would have made O_{k^*} exactly saturate O_{l^*}, contradicting the minimality of x. Thus (c) is satisfied for the next recursive call.

Finally, consider (d). Let O_k and O_l be such that O_l is the child of O_k in a precedence chain of G. Since (d) was satisfied at the start of the current call, O_k could saturate O_l. If O_k cannot saturate O_l for the next recursive call, by the proposition, there exists $x^* < x$ such that sending x flow along π would cause O_k to be able to exactly saturate O_l. By the minimality of x, x^* did not satisfy Stopping Condition 3, despite the exact saturation, so O_l must have been the successor of O_k in some π_j. But then O_k could exactly saturate O_l both initially and after the addition of x units of flow along permutation x, contradiction. Thus (d) is satisfied for the next recursive call.

Case 2: Absorbing child into parent

Suppose that just prior to the next recursive call, operator O_j is absorbed into its parent O_i.

The parent-child merge of O_i and O_j leaves most superoperators intact, but does some cutting and pasting of the superoperators containing O_i and O_j.

We show that Precondition (a) holds for the next call. Since Precondition (a) holds at the start of the given recursive call, if O_l is the successor of O_k in a superoperator, then O_k can exactly saturate O_l. After the addition of x amount of flow along π, O_k can still exactly saturate O_l.

In P', every superoperator is either equal to a superoperator π_z from P, or a prefix or suffix of some such π_z. The one exception is the superoperator AD, where O_i is the last operator in A and O_j is the first operator in D. Since $r'_i p'_i = r'_j$, Precondition (a) holds for the next recursive call.

As in Case 1, it can be shown that (d) must also hold for the next recursive call, because otherwise a

parent-child absorption would have been triggered at some $x^* < x$.

Conditions (b) and (c) hold for P' as a result of the properties of the sort. □

We are now ready to give the proof of Theorem 4.1, which establishes the correctness and bounds of RouteChains.

Theorem 4.1 *Suppose RouteChains is run on inputs having the specified properties, and obeying the "can't saturate" precondition. Let I be the MTTC instance whose rate limits, selectivities, and precedence graph are given by those inputs. Then RouteChains produces a routing that is optimal for I, and would also be optimal if we removed all precedence constraints from I. RouteChains runs in time $O(n^2 \log n)$, and produces a routing that uses at most $4n - 3$ distinct permutations.*

Proof of Theorem 4.1: By induction on the number of parent-child absorptions. In the base case, there are none. The proof for the base case is essentially the same as the correctness proof in [8]. For completeness, we repeat the argument here. We show that in this case the output routing K has the saturated suffix property and hence this lemma follows from Lemma 4.1.

For contradiction, assume K does not have the saturated suffix property. Then there are operators O_i and O_j such that O_i appears before O_j in some permutation used in K, and O_i is saturated by K, but O_j is not. If O_j precedes O_i in some other permutation used in K, then there must have been a swap-merge of the superoperators containing O_i and O_j, since we have assumed no parent-child absorptions. But a swap-merge would put O_i and O_j in the same superoperator, and with no parent-child absorptions, O_i and O_j would be in the same superoperator in the final recursive call and hence would both be saturated by K, or both not. Therefore O_i precedes O_j in every permutation used by K.

We now claim that the set of operators saturated in the final recursive call consists of a suffix of the permutation π used in the final recursive call. Consider any two successive superoperators π_{k-1} and π_k in P in the final recursive call. If π_{k-1} is saturated by the flow (π, x) computed in the final recursive call, and π_k isn't, then the residual capacity of π_{k-1} relative to (π, x) is 0, while the residual capacity of π_k relative to (π, x) is greater than 0. Thus after x flow is routed along π, π_k can over-saturate π_{k-1}. But by the "can't saturate" precondition, at the start of the final recursive call, $\rho(\pi_k)\sigma_k \leq \rho(\pi_{k-1})$. By the same argument used in the proof of Lemma 9.2, there is some $x^* < x$ that would have triggered Stopping Condition 2. Contradiction. Therefore if π_{k-1} is saturated by the final (π, x), so is π_k, and K saturates a suffix of the π used in the final recursive call. Since O_i precedes O_j in every permutation in K, this contradicts the original assumption that O_i is saturated and O_j is not. Thus K obeys the saturated suffix property. This completes the proof of the base case.

For the induction step, we assume the theorem holds if the number of parent-child absorptions is $i \geq 1$, and show it holds if the number of parent-child absorptions is $i + 1$.

Suppose the inputs L to RouteChains are such that RouteChains performs exactly $i + 1$ parent-child absorptions. Let q be such that the first parent-child absorption occurs in the qth call of the recursive procedure. Since the $q + 1$st recursive call results (recursively) in only i absorptions, by induction it outputs an optimal routing K_2 relative to the rate limits specified for the operators at the start of the $q + 1$st call.

Let L' be L with the precedence constraints removed. The MTTC instance associated with L is I. Let I' be the MTTC instance associated with L'. Compare the execution of $RouteChains(L)$ and $RouteChains(L')$. Let $\hat{K}$ denote the set of flow assignments (π, x) that are computed in the first q recursive calls made by $RouteChains(L')$. In the first $q - 1$ recursive calls, the computed flow assignments (π, x) are the same for L and L'. The (π, x) computed in the qth call for L and L' both use the same permutation π; the value of x may be greater for L' (since a parent-child absorption was not triggered), but cannot be less. Let J denote the MTTC instance that is derived from I' by taking the rate limits in J to be the residual rate limits of the

operators in I relative to $\hat{K}$. Since running RouteChains on L' is guaranteed to produce an optimal routing for I' (by the correctness of the base case), there is a way to augment $\hat{K}$ with additional flow to obtain an optimal routing for I'. It follows that if $\tilde{K}$ is an optimal routing for J, $\tilde{K} \bigcup \hat{K}$ must be an optimal routing for I. When running RouteChains on instance I, starting from the $q+1$st recursive call, the algorithm is attempting to solve J, but with precedence constraints. By induction, it finds a routing that is optimal even without the precedence constraints. It follows that this routing, combined with routing $\hat{K}$, must be optimal for both I and I'.

We now prove the bounds on the runtime analysis and number of permutations used. Each recursive call adds one (π, x) to the output routing and can easily be implemented to run in time at most $O(n \log n)$. Let r be the number of parent-child absorptions, and t the number of swap-merges. The number of recursive calls is $r + t + 1$. Each swap-merge decreases the number of superoperators by 1 and each parent-child absorption increases it by at most 2. The initial number of superoperators is n, the final number is at least 1, so $n + 2r - t \geq 1$. Since an absorbed child always remains with its parent, $r \leq n - 1$ and hence $t \leq 3n - 3$. The stated bounds follow. □

The correctness of the MTTC algorithm relies on certain properties of the CombineChains algorithm, which are proved in the following lemma. In the lemma and its proof, for set S of operators, we use $\sigma(S)$ to denote the product of the selectivities of the operators in S.

Lemma 9.3 *Given an MTTC instance I whose precedence graph G is a forest of proper chains, CombineChains outputs an ordered partition $(A_1, \ldots, A_m)$ of the operators of I, and routings $K_{A_1}, \ldots, K_{A_m}$, such that the partition and routings satisfy the following properties:*
For each A_i (1) the ancestors in G of the operators in A_i are all contained in $\bigcup_{j=1}^{i-1} A_j$, (2) K_{A_i} is a routing for the operators in A_i that saturates all of them, and (3) if $i \neq 1$ then $\tau(K_{i-1})\sigma(A_{i-1}) \geq \tau(K_i)$, where $\tau(K_{i-1})$ and $\tau(K_i)$ are the throughputs of K_{i-1} and K_i.

Proof: By the observations in the description of CombineChains, Property (1) holds for the output partition. For (2) and (3), consider the recursive call to CombineChains that produced set A_i of the output partition. In this call, $A_i = Q_{j^*}$. Let I_{A_i} denote the induced MTTC instance that keeps only the operators in A_i. Let I^0 denote I with precedence constraints removed, and $I^0_{A_i}$ denote I_{A_i} with precedence constraints removed.

By Lemma 4.2, there is an optimal routing K for I^0 for which A_i is a saturated suffix. By taking the subrouting through saturated suffix A_i, we get a routing for $I^0_{A_i}$ that saturates all the operators in A_i. By Lemma 9.1, it follows that all optimal routings for $I^0_{A_i}$ saturated all the operators in A_i. When CombineChains runs the MTTC algorithm for chains from Section 4.3 on the constrained instance I_{A_i} which already has proper chains, the preprocessing step does not reduce any rate limits. Following the preprocessing step, CombineChains runs RouteChains, which outputs the routing K_{A_i}. By the properties of RouteChains (Theorem 4.1), K_{A_i} is optimal both for I_{A_i} and $I^0_{A_i}$. It follows by Lemma 9.1 that K_{A_i} saturates all operators in A_i. Thus Property (2) holds for A_i.

Since A_i is a saturated suffix of optimal routing K for I^0, K sends all flow first through the operators not in A_i (these operators will comprise $A_1, \ldots, A_{i-1}$) and then through the operators in A_i, which it saturates. Let F_K denote the throughput of K. Let Q' denote the operators not in A_i. Thus the amount of flow reaching suffix A_i in K is $\sigma(Q')F_K$. By similar arguments as above, there is an optimal routing K' for the operators in Q' for which A_{i-1} is a saturated suffix.

Let $B = Q' - \{A_{i-1}\}$. Routing K sends flow first through Q' and then through suffix A_i. Thus the max-throughput attainable just on Q' is at least as big as F_K. Since K' is optimal for Q', its throughput is at least F_K. So routing K' routes at least F_K flow first into B and then B and then into A_{i-1}, causing at

least $\sigma(B)F_K$ flow to reach A_{i-1} and saturate the operators in it. It follows that $\tau(K_{A_{i-1}}) \geq\geq \sigma(B)F_K$ and hence $\tau(K_{A_{i-1}})\sigma(A_{i-1}) \geq \sigma(B)\sigma(A_{i-1})F_K = \sigma(Q')F_K = \tau(K_{A_i})$. Thus Property (3) holds for A_i. □

9.3 Proof: The general MTTC algorithm

We are now ready to prove the main theorem in the paper.

Theorem 4.2 *When run on an MTTC instance I, the MTTC algorithm runs in time $O(n^3)$ and outputs an optimal routing for I. The output routing uses fewer than $4n$ distinct permutations.*

We break the proof up by proving related lemmas, from which the proof of the theorem follows immediately. First, we present some notation.

Consider a run of the MTTC algorithm on an instance I. Suppose the run includes k recursive calls (include the initial call). Let $I_1, I_2, \ldots, I_k$ be the MTTC instances in each of these calls (so $I_1 = I$ denotes the initial instance). We will refer to the operators in $I_1 = I$ as the *original operators*.

For $1 \leq j \leq k$, let O_i^j denote the ith operator in I_j. Let r_i^j denote the rate limit of O_i^j in I_j, and let p_i^j denote its selectivity. Let G^j denote the precedence graph in I_j. Let R_i^j denote the rate limit of O_i^j at the *end* of the jth call; $R_i^j < r_i^j$ if the rate limit of O_i^j was reduced when chains were made proper in the jth call (immediately prior to running CombineChains), otherwise $R_i^j = r_i^j$. For $2 \leq j \leq k$, each O_{i_j} either corresponds directly to an operator in I_{j-1}, or corresponds to a subset of operators from I_{j-1} that were grouped together into a subpartition when CombineChains was run during the processing of instance I_{j-1}. Hence each operator O_i^j in I_j corresponds inductively to an original operator or group of original operators. We view operator O_i^j as "containing" the original operators to which it corresponds and denote by S_i^j the original operators contained in O_i^j.

Lemma 9.4 *Assume the MTTC algorithm is run on an instance I. Let $1 \leq j \leq k$, where k is the total number of recursive calls executed (including the initial call). For each operator O_i^j in I_j, there exists a routing K_i^j for just the operators in S_i^j achieving throughput r_i^j, and this routing obeys the rate limits and precedence constraints of I.*

Proof: The proof is by induction on j. It is trivially true for $j = 1$, since in this case each S_i^j just consists of a single original operator. Suppose it is true for some $1 \leq j \leq k-1$; we will show it is true for $j+1$. By induction, at the beginning of the jth iteration, for each operator S_i^j in I_j, there there exists a routing K_i^j for the operators S_i^j achieving throughput r_i^j (and obeying the necessary constraints). Consider the execution of the jth iteration, and the resulting instance I_{j+1} constructed for the $j+1$st iteration. Every operator O_i^{j+1} in I_{j+1} either corresponds directly to an operator in I_j, or corresponds to a subset $\hat{A}$ of operators in I_j that were grouped together by CombineChains. In the first case, the inductive hypothesis guarantees that there is a routing for O_i^{j+1} achieving throughput r_i^{j+1}. In the second case, CombineChains outputs a routing $K_{\hat{A}}$ for the operators in $\hat{A}$ achieving throughput r_i^{j+1}. By induction, for each of the operators O_a^j in $\hat{A}$, there is a routing K_a^j on the operators in S_a^j achieving throughput r_a^j. Applying CombineRoutings to $K_{\hat{A}}$ and the K_a^j yields a routing for the operators in S_i^{j+1} achieving throughput r_i^{j+1} (which is the throughput of $K_{\hat{A}}$) and obeying the necessary constraints. □

Lemma 9.5 *When run on an MTTC instance I, the MTTC algorithm outputs an optimal routing for I.*

Proof: Since each recursive call eliminates a fork in the original precedence graph, the algorithm terminates and the total number of recursive calls is at most $n-1$. It is easy to see that the algorithm outputs a routing that satisfies all the rate constraints and precedence constraints. We now show that it achieves the maximum possible throughput. We do this by showing that the output routing has the saturated suffix property, not with respect to I, but with respect to a modified version of I with some reduced rate limits. The trick is to show that the modified version of I has the same max-throughput value as I itself.

Let K_i^j be as defined in the statement of Lemma 9.4. Let L_i^j be the routing produced from K_i^j by multiplying the flow along each permutation in K_i^j by a factor of R_i^j/r_i^j. (Thus L_i^j is a scaled version of K_i^j achieving throughput R_i^j; it is equal to K_i^j if O_i^j's rate limit is not changed in the jth call.) For each S_i^j, and for each original operator O_u in S_i^j, let ξ_u^j be the amount of flow reaching operator O_u in routing L_i^j. Let $I(j)$ be the modification of I produced by setting the rate limits of each operator O_i in I to be ξ_u^j.

Consider the final, kth recursive call. The input I_k to this call consists of a forest of chains. The call makes these chains proper and runs RouteChains. Let I'_k denote instance I_k with the rate limit of each operator O_i^k in I_k set to R_i^k, so that it reflects any rate reduction performed while the chains were made proper. By the analysis of RouteChains, the routing K^k constructed for I_k has the saturated suffix property with respect to the rate limits of I'_k. The final output routing K of the MTTC algorithm is formed by recursively combining the routing for I'_k with routings constructed in previous recursive calls. It is not hard to verify that, for every operator O_i^k of I'_k in the saturated suffix of K^k, routing K sends a total of ξ_u^k amount of flow to every original operator O_u in O_i^k. Further, K obeys the rate and precedence constraints of $I(k)$, and has the saturated suffix property for $I(k)$. Thus K is an optimal routing for $I(k)$. It remains to show that K is also an optimal routing for I. Recall that I is identical to $I(k)$ except that some of the rate limits in $I(k)$ are lower. Thus we must consider the ways in which these rate limits are reduced.

The reduction in rate limits from $I(1)$ to $I(2)$ and on to $I(k)$ are caused when chains are made proper immediately prior to a call to CombineChains. We will show that for any $1 \leq j \leq k$, each time a chain is made proper during the jth recursive call, the resulting reductions in rate limits, from $I(j-1)$ to $I(j)$, do not change the maximum throughput attainable. From this it follows that the max-throughput of $I(k)$ is equal to the max-throughput of I, proving the optimality of the routing output by the MTTC algorithm.

The procedure for making a chain proper proceeds downward on the chain. Consider the procedure. Label each node in the chain according to whether its rate limit was reduced. The labels partition the chain into maximal contiguous subchains of reduced nodes and unreduced nodes. The reductions in each such subchain of reduced nodes are due to the node immediately preceding the subchain; this node caused a reduction in the rate limit of the first node in the subchain which was then propogated through to the last node in the subchain. If the node preceding the subchain is O_{i_1}, and the subchain is $O_{i_2}, O_{i_3}, \ldots, O_{i_m}$, then the reduced rate limit of each O_{i_l} in the subchain is $r_{i_1} p_{i_1} p_{i_2} \ldots p_{i_{l-1}}$ (where the r_{i_l} and p_{i_l} are the rate limits and selectivities of the nodes, prior to the procedure making the chain proper). Let us call the node preceeding the subchain a "bottleneck" node.

In the execution of the MTTC algorithm, operators are only grouped together through execution of CombineChains. By the properties of CombineChains (Lemma 9.3), each time a new chain of operators is formed, the chain is proper, that is, every operator in that chain can saturate its successor. Since every bottleneck node in the jth iteration cannot saturate its successor, it follows that every bottleneck node in the iteration corresponds directly to a single original node that was never involved in a CombineChains operation. Further, since rate limits of operators are reduced only when those operators are about to be used as input to CombineChains, the rate limits of bottleneck nodes are equal to the initial rate limits of the corresponding original nodes.

Let $I_\infty(j)$ denote the modification of $I(j)$ in which, for every reduced operator from the jth iteration,

the rate limits for the original operators are removed, giving those operators infinite capacity. The max-throughput of $I_\infty(j)$ is clearly at least as large as the max-throughput of I_∞.

Consider a bottleneck node in I^j and the maximal subchain of reduced nodes after it. Let $O^j_{i_1}$ denote the bottleneck node and $O^j_{i_2}, \ldots, O^j_{i_m}$ denote the reduced nodes. Let O_b be the original node corresponding to $O^j_{i_1}$. Let S' denote the set of all original operators contained in $O^j_{i_2}, \ldots, O^j_{i_m}$; these are the original descendants of the node in the precedence graph $G = G^1$. Consider an optimal routing K_∞ for I_∞. By the precedence constraints, in every permutation used in the routing, each of these operators in S' must appear somewhere after O_b. Without loss of generality, we can assume they appear appear consecutively (in some order) immediately after O_b; if not, they may be moved forward in the permutation (while keeping the same relative order between them) to appear in this position, while still maintaining the precedence constraints, since their rate limits are infinity and moving them forward can only reduce the flow into other nodes. We can thus consider the operators in S' to form a "block" in the routing. The amount of flow entering this block is at most $r_b p_b$ (the product of the initial rate limit and selectivity of O_b). By an argument above, for each node $O^j_{i_q}$ in the reduced subchain ($2 \leq q \leq m$) there is a routing $K^j_{i_q}$ on the contained original operators achieving throughput $r^j_{i_q}$. Since the nodes in the subchain form a maximal subchain of reduced nodes, it follows that the rate limit $r^j_{i_2}$ of $O^j_{i_2}$ is $r_b p_b$, the rate limit $r^j_{i_3}$ of $O^j_{i_3}$ is $r_b p_b p^j_{i_2}$, and in general, the rate limit $r^j_{i_q}$ of $O^j_{i_q}$ in the reduced subchain is $r_b p_b p^j_{i_2} p^j_{i_3} \ldots p^j_{i_{q-1}}$. Thus in K, the at most $r_b p_b$ flow entering the block could be routed instead (if it isn't already) first through the original operators in $O^j_{i_2}$ (by scaling $K^j_{i_1}$), then the remaining $r_b p_b p^j_{i_2}$ flow could be routed through the original operators in $O^j_{i_3}$ (by scaling $K^j_{i_2}$), and so on, while still obeying precedence constraints and rate constraints of I_∞. Under this routing, each original operator O_i in a reduced $O^j_{i_q}$ in the subchain would get a total amount of flow not exceeding R^j_i, the amount of flow it receives under routing $K^j_{i_q}$.

Applying this same argument to all the maximal reduced subchains from iteration (j), we find that there is an optimal routing for I_∞ which does not put more that R^j_i flow through any original operator O_i contained in a reduced node. It follows that this optimal routing for I_∞ is also feasible for $I(j)$. Since the max-throughput of I_∞ is clearly greater than or equal to the max-throughput of $I(j)$, they have the same max-throughput. Finally, since $I(j)$ differs from I only in its reduced rate limits, the max-throughput of $I(j)$ is at most the max-throughput of I, which is at most the max-throughput of I_∞. Thus the max-throughput of $I(j)$ equals the max-throughput of $I(j-1)$, as claimed. □

Lemma 9.6 *The MTTC algorithm runs in time $O(n^3)$ and outputs a routing that uses fewer than $4n$ distinct permutations.*

Proof: We show that the total time spent over all runs of RouteChains is $O(n^2 \log n)$. All other processing takes time at most $O(n^2)$ per node in G, for a total time of $O(n^3)$.

The routing output by the MTTC algorithm is constructed hierarchically, through calls to CombineRoutings. We represent this hierarchical construction by a tree T, with each non-leaf node labeled by a routing K output by a call to CombineRoutings. The children of node K are the nodes labeled by the routings K_{A_i} that were combined together with a routing K' to form K. The routing K' was formed by a call to RouteChains. The leaves of the tree correspond to the n operators of I, and each is labeled with the trivial routings for its corresponding operator. Thus the number of edges in the tree is $n-1$.

If a non-leaf node corresponding to routing K in T has m children, then the routing K' used in forming it was produced by a call to RouteChains on m operators, taking time $O(m^2 \log m)$. If m_N denotes the number of children of node N, then the total time spent running RouteChains is upper bounded by the sum,

over all non-leaf nodes N in the tree, of $cm_N^2 \log m_k$, for some constant c. Since the sum of the m_K's is the number of edges in the tree, $n-1$, it follows that the total time spent in all calls to RouteChains is upper bounded by the maximum of $c(n_1^2 \log n_1 + n_2^2 \log n^2 + \ldots + n_t^2 \log n_t)$ over all (multi)-sets of positive integers $\{n_1, \ldots, n_t\}$, where t is arbitrary, such that $n_1 + \ldots n_t = n-1$. Since $n_1^2 + \ldots + n_t^2 < n^2$, the total time spent in executing RouteChains is $O(n^2 \log n)$. The time spent in RouteChains dominates the running time of the algorithm, which thus is $O(n^2 \log n)$ as claimed.

To bound the number of permutations used in the output routing, we again consider the tree described above. Each non-leaf node K in the tree is the result of combining a routing K' with routings on $K'_{A_i}s$. Call the K' an outer routing, and the K_{A_i}'s inner routings. The combined routing K uses a number of permutations that is at most the sum of (the number of permutations in the outer routing) + (the total number used in the inner routings). But the inner routings are also produced by a call to CombineRoutings that combines an outer routing with inner routings, except for children that are leaves.

It follows that the total number of routings used is equal to the sum, over all non-leaf nodes, of the number of permutations used by the associated outer routings, plus any final contribution due to incorporating the inner routings for the leaf nodes into the outer routings of their parents. But this final contribution is 0, since incorporating the single permutation of a leaf node into the the outer routing of its parent does not increase the number of permutations used.

By Theorem 4.1, the number of permutations used for the outer routing corresponding to a node N is at most 4 times the number of its children. The sum, over all non-leaf nodes N, of the number of children of N, is equal to the total number of edges, $n-1$. Hence the total number of permutations used by the outer routings is at most $4(n-1)$. □

9.4 Proofs: Non-Selective Operators

Theorem 5.1 *When run on an MTTC instance I, whose precedence graph is an inverted tree and which contains only selective operators, the MTTC-INV algorithm algorithm runs in time $O(n^3)$ and outputs an optimal routing for I. The output routing uses fewer than $4n$ distinct permutations.*

Proof: The proof is almost the same as the proof of the analogous theorem for ordinary trees. The only real difference is in justifying the reductions in rate limits when chains are made proper.

Consider running the MTTC-INV algorithm on an instance I with precedence graph G. Each time a chain is made proper, it is either done immediately prior to eliminating a fork (i.e. prior to running CombineChains) or immediately prior to finding a routing on the chains in the final recursive call.

In either case, the chain that is being made proper can easily be shown to consist of a prefix of composite nodes (possibly empty) produced by a run of CombineChains, followed by a suffix (possibly empty) consisting of original nodes from G. By the analysis of CombineChains, the chain of composite nodes is already proper. Suppose the suffix is non-empty. We argue that if a rate reduction is performed on the first node in the suffix, then this rate reduction does not reduce the maximum throughput attainable (for the underlying, original problem instance I). It follows that any rate reductions to the other original operators in the suffix will not reduce the maximum throughput attainable, since all flow into these nodes must first pass through the operators above them in the suffix.

Note first that, from the above, it immediately follows that all rate reductions performed by the algorithm are performed on original nodes; the rate limits of new, composite operators are never reduced.

Let $\mathcal{B}_1, \ldots, \mathcal{B}_m$ denote the chain of operators in the prefix of the current chain, and let $B_1, \ldots, B_m$ denote the underlying sets of original nodes. Let O_{frst} denote the first operator in the suffix. Let $N =$

$\bigcup_{i=1}^{m} B_i$. If the rate limit of O_{frst} is reduced when the chain is made proper, then it is re-set to be the product of the rate limit of $\mathcal{B}_m$ times its selectivity. We use the following claim.

Claim: In any feasible routing for G, the most flow that could ever reach O_{frst} is the product of the maximum-throughput attainable using just the operators in N, times $\sigma(N)$.

Recall that $\sigma(N)$ denotes the product of the selectivities of the operators in N. To prove the claim, consider a modified instance produced by eliminating the rate constraints for all operators in G other than O_{frst} and the operators in N. Consider a feasible routing K_{maxA} for this modified instance that maximizes the total amount of flow into O_{frst}. The total amount of flow into A in K_{maxA} is an upper bound on the maximum amount of flow into O_{frst} that would be attainable under the original constraints. Let $S = \{O_{frst}\} \bigcup N$. In routing K_{maxA}, if there is a permutation in which an operator not in S appears immediately after an operator in N, the order of these two operators can be switched in the permutation without without affecting the total amount of flow into O_{frst}, and without violating any rate constraints in the modified instance. Thus we may assume without loss of generality that in routing K_{maxA}, all permutations begin with operators not in S, followed by the operators in N, followed immediately by O_{frst}. The total amount of flow that reaches the operators in N, over all these permutations, cannot be more than the maximum amount of flow that could be routed through just the operators in N. Before reaching O_{frst}, this flow is reduced by $\sigma(N)$. The claim follows.

By Lemma 4.2 and the description of CombineChains, it can be shown inductively that B_m is a saturated suffix of an optimal routing through the operators in N (where saturation is defined with respect to any rate reductions already performed on the operators in N), and the throughput achieved by this optimal routing (and hence by any optimal routing) is the rate limit of the composite operator $\mathcal{B}_m$ times $1/\prod_{i=1}^{m-1} \sigma(B_i)$. It immediately follows from the claim that the maximum total flow into O_{frst} in any feasible routing is at most the rate limit of $\mathcal{B}_m$ times $\sigma(B_m)$. But $\sigma(B_m)$ is the selectivity of $\mathcal{B}_m$, so this is precisely the value to which O_{frst}'s rate limit was re-set. Since total flow into O_{frst} cannot exceed this value, the re-setting does not reduce the maximum throughput attainable. □

Lemma 5.1 *Given a problem instance I containing both selective and non-selective operators, there exists an optimal solution K satisfying the following: for all O_i, O_j such that $p_i \geq 1$ and $p_j < 1$:*

- ***(A)** If O_i and O_j have no precedence constraint between them, then O_j does not immediately follow O_i in any permutation used in the routing.*
- ***(B)** If O_j is the only child of O_i, then O_j immediately follows O_i in every permutation used in the routing.*

Proof: We prove this by contradiction.

Part A: Let K be an optimal solution to the problem instance I that minimizes the number of violations of **A**. Let π denote a permutation used in the solution, in which, for some O_i and O_j satisfying the properties listed above, O_j immediately follows O_i. It is easy to see that switching O_j and O_i in this permutation reduces the amount of flow through O_i and either reduces or keeps constant the amount of flow through O_j. Thus we can reduce the number of violations by one, contradicting the assumption.

Part B: Let K be an optimal solution to the problem instance I that minimizes the number of violations of **B**. Let π denote a permutation used in the solution, in which, for some O_i and O_j satisfying the properties listed above, O_j does not immediately follow O_i. Note that O_j must follow O_i in π (since O_j is the child of O_i). Let $O'_1, \cdots, O'_l$ denote the operators between O_i and O_j. There can be no precedence constraints between O'_k and O_i or O_j for any k. This follows from O_j being the only child of O_i.

Let $p' = p'_1 \times \cdots \times p'_k$ denote the combined selectivity of these k operators. We replace the part $(O_i, O'_1, \cdots, O'_k, O_j)$ of π with:

- $O'_1, \cdots, O'_k, O_i, O_j$, if $p' < 1$ (ie., we move O_i to after O'_k).
- $O_i, O_j, O'_1, \cdots, O'_k$, if $p' \geq 1$ (ie., we move O_j to before O'_1).

It is easy to see that the flow through all of the operators either reduces or remains constant after this substitution, thus contradicting the assumption made. □