Flow Algorithms for Parallel Query Optimization

Amol Deshpande [#1], Lisa Hellerstein [*2]

[#] *University of Maryland, College Park, MD USA*

[1] amol@cs.umd.edu

[*] *Polytechnic University, Brooklyn, NY USA*

[2] hstein@cis.poly.edu

***Abstract*— 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 to 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 given query, 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 algorithms.**

I. Introduction

Parallelism has long been recognized as the most cost-effective approach to scaling up the performance of database query processing [8], [10], [14], [17]. 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 [15], [20]. 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* [9], 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 operation 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. [21] 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 one of the processors, most likely the one executing the first join, may quickly become the *bottleneck*, with the rest of the processors sitting idle [21]. 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. Despite 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 prior work [4], [5]; 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 giving 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 property that it produces a *sparse* solution using at most $O(n)$ different regular plans. 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 [18], [19], [21].

Outline

We begin with a discussion of related work in parallel query optimization (Section II). We then present our execution model for executing a multi-way join query and a reduction of this problem to precedence-constrained selection ordering (Section III). We present our main algorithm for finding an optimal interleaving plan for the case when all operators are *selective*, ie., have selectivity < 1 (Section IV). We discuss how this algorithm can be extended to solve a large class of general multi-way join queries, which may result in *non-selective* operators

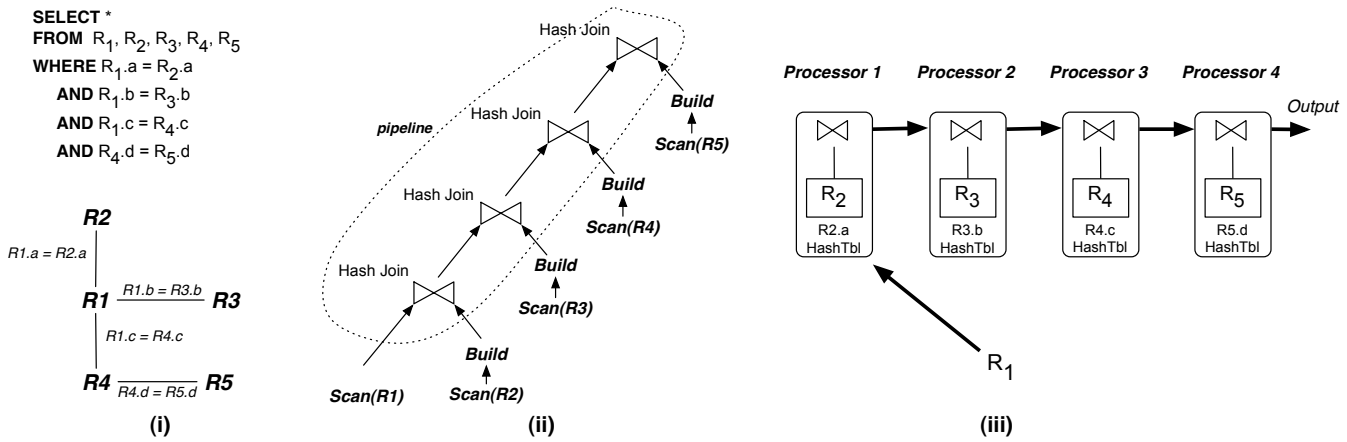

Fig. 1. (i) A 5-way join query, and the corresponding query graph. (ii) A pipelined plan where R_1 is the driver relation. (iii) Executing the query using pipelined parallelism with each join running on a different processor. The join with R_5 is done using the R_4 matches found earlier.

(Section V). Finally, we present an extensive simulation-based experimental study that demonstrates the practical benefits of using interleaving plans (Section VI).

II. Related Work

There has been much work in the database literature on 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 (minimizing the total time taken to execute a query) and total work optimization (minimizing the total work across all processors), in the context of distributed query processing. Ganguly et al. [11] 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 [17] present a two-phase approach that separates join order optimization from parallel scheduling issues; several researchers have since looked at the scheduling and optimization issues in the second phase in more detail (e.g. [24], [16], [3], [12], [13]). 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 prior work (Condon et al. [4], [5]), we introduced the notion of interleaving plans for parallel selection ordering. Here we substantially extend that work by generalizing the algorithms to multi-way join queries; we also present an extensive experimental evaluation demonstrating the benefits of interleaving plans. Srivastava et al. [21] 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 in parallel. They don't, however, consider interleaving plans. In the extended version of this paper [7], 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 [22] considered the problem of designing tuple routing strategies for eddies in a distributed setting, where operators reside on different nodes and the goal is to minimize average response time or maximize throughput (a setting similar to ours). They present an analytical formulation and 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* [6], where the goal is to find a static plan augmented with decision points to optimally execute a selection query. Our algorithms can be seen as a way to design routing policies in a parallel and distributed setting for eddies.

The algorithms we present are based on a characterization of query execution as tuple flows. We refer the reader to Condon et al. [4], [5] for a discussion of related work on flow algorithms.

III. Problem Formulation and Analysis

We begin with formally defining 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.

A. Parallel Execution Model

Figure 1 shows a 5-way join query that we use as a running example. 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 1 (iii)), 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]It is possible to drive execution using multiple driver relations and symmetric hash join operators [23], but most database systems do not support such plans.

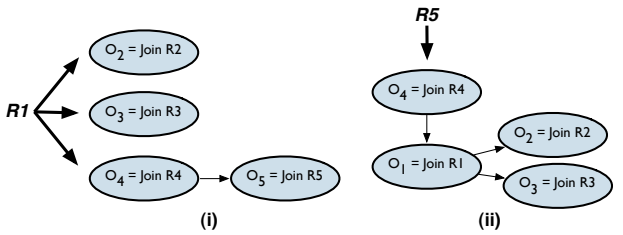

Fig. 2. Reducing the query in Figure 1 to precedence-constrained selection ordering for driver choices R_1 and R_5.

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).

B. Reduction to Precedence-Constrained Selection Ordering

Once 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 1 (iii)); the precedence constraints arise from the desire to avoid Cartesian products [18], [19], [21].

Given a multi-way join query over relations $R_1, \ldots, R_n$, with one relation designated as the driver, the reduction begins with creating a selection operator for each of the driven relations, and setting up precedence constraints to prevent Cartesian products (Figure 2). We denote the selection operator corresponding to relation R_i by O_i. For acyclic query graphs (our focus in this paper), the resulting precedence graph is 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, and the selectivity of O_i, denoted by p_i, is set to be the "fanout" of the join with R_i. Figures 2 (i) and (ii) show two examples of this reduction for the example query in Figure 1.

Join fanouts, unlike selection fanouts, may be > 1.

Definition: We say that an operator is *selective* if it has selectivity < 1, and *non-selective* if it has selectivity ≥ 1.

C. Execution Plan Space

A *serial* plan for executing a selection ordering query specifies a single *permutation of the operators* (obeying 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, the *rank ordering* algorithm [19] can be used to find the optimal serial plan; this algorithm simply orders the operators by their values of $c_i/(1-p_i)$ (Figure 3). Srivastava et al. [21] present an algorithm to minimize response time when each operator is executed on a different processor in parallel. As they show, when selectivities are all ≤ 1 and the processors are identical, the optimal algorithm, called *BOTTLENECK* (Figure 3), 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.

Fig. 3. Algorithms for finding optimal serial plans for selective operators (assuming identical processors)

Although BOTTLENECK finds the best serial plan for executing the query, it typically does not exploit the full parallelism in the system. Figure 4 illustrates this with an example in which the query consists of three identical operators with selectivities 0.2 and costs 0.1. The best serial plan can process only 10 tuples in unit time. If we instead use three serial plans simultaneously by routing 1/3 of the tuples through each, 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 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*

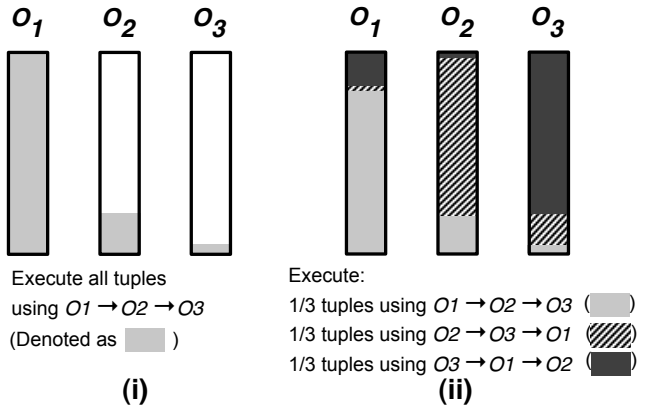

Fig. 4. Given 3 identical operators, O_1, O_2, O_3, with $p = 0.2, c = 0.1$ and no precedence constraints, (i) the best serial plan processes 10 tuples in unit time; (ii) an interleaving plan that uses 3 serial plans equally can process 24.19 tuples.

tuple). The tuple is then sent through the operators in that order, and is either discarded by some operator, or 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).

IV. 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 *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 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 extended version of this paper [7].

A. 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 on the operators. 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. 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]. A tuple sent along a permutation π travels through the operators in the order specified by π, until it is either eliminated by an operator or it has traveled through all the operators. A tuple is eliminated by operator O_i with probability $(1 - p_i)$. The routing must obey the precedence constraints defined by G: for each O_i, O_j pair, if O_j is a descendant of O_i in G, then tuples must be sent to O_i before 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 the operators $\mathcal{O} = \{O_1, \ldots, O_n\}$. The kth element of π is denoted by $\pi(k)$. The index of operator $\pi(k)$ is denoted by $\pi'(k)$, so $\pi(k) = O_{\pi'(k)}$. Let $\phi(n)$ be the set of all $n!$ permutations of $\mathcal{O}$. 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, $g(\pi, i) = p_{\pi'(1)} p_{\pi'(2)} \cdots p_{\pi'(m-1)}$, where m is such that $\pi(m) = O_i$. 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*.

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

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, find an assignment to the variables f_π, for all $\pi \in \phi(n)$, maximizing

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

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 to the f_π satisfies the rate constraint for r_i with equality, we say that O_i is *saturated* by K. The value F achieved by K is the *throughput* of K, and we call K a routing.

B. Preliminaries

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

Below we state the main lemma on which our algorithms are based. It was proved in [4], [5] for the MTTC problem with no precedence constraints, but it also holds with precedence constraints. Let K be a feasible solution to an MTTC instance, and let $\mathcal{O} = \{O_1, \ldots, O_n\}$. We say K has the *saturated suffix property* if for some non-empty $Q \subseteq \mathcal{O}$, (1) the operators in Q are saturated by K and (2) if $f_\pi > 0$ in K, then the elements of $\mathcal{O} - Q$ precede the elements of Q in π (i.e., no tuples flow from an operator in Q to an operator in $\mathcal{O} - Q$). We call Q a *saturated suffix* of K.

Lemma 4.1: [4], [5] (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 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 idea behind our algorithm 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 will construct a new problem instance with the same maximum throughput value, for which a routing with the saturated suffix property is achievable.

C. The MTTC problem with no precedence constraints

We begin by briefly reviewing the algorithm from [5] for the MTTC problem with no precedence constraints. In the algorithm, we build a routing K incrementally. The routing

consists of pairs (π, x) indicating that x amount of flow is to be sent along permutation π.

Let the operators be numbered such that $r_1 \geq \ldots \geq r_n$. Let π be the permutation $(O_1, \ldots, O_n)$. Since all selectivities are < 1, π obeys the property that each operator in the ordering can't overfill its predecessor; this "can't overfill" property is an invariant of the algorithm.

We execute the following recursive procedure. Conceptually, we start by sending flow through the operators according to ordering π, beginning with no flow, and increasing the amount at a continuous rate. As flow is increased, the residual capacity (rate limit) of the operators decreases. Suppose that before any operator is saturated, an operator O_i suddenly becomes able to exactly saturate its predecessor O_j in π (i.e. $r'_i p_i = r'_j$, where r'_i and r'_j are the residual capacities of O_i and O_j). We stop increasing the flow at this point. Let x be the resulting flow value. (We actually calculate x analytically.) No additional flow can be added along π without violating the "can't overfill" invariant. We place (π, x) into (initially empty) output routing K. We then modify π by swapping O_i and O_j. In addition, we "paste" O_i to the front of O_j, forming a "superoperator" (O_i, O_j). All subsequent flow sent into the superoperator will be sent first to O_i and then immediately to O_j. (Because O_i can exactly saturate O_j, this means that ultimately either both operators will be saturated, or neither.) We treat the superoperator as a new single operator. Its rate limit is defined to be the (residual) rate limit of O_i and its selectivity is defined to be $p_i p_j$. We now have a new ordering π on a set of $n-1$ operators. We set the rate limits of the operators to equal their residual rate limits. The "can't overfill" invariant holds for π, and we recurse on the $n-1$ operators and π. In each recursive iteration, we add another (π, x) to routing K.

The recursion stops when during some iteration, as we send increasing flow along π, some operator becomes fully saturated before any operator O_i becomes able to exactly saturate its predecessor O_j. In this case x becomes the amount of flow causing the saturation, (π, x) is added to K, and we terminate. It can be shown that, due to the "can't overfill" invariant, the final flow K has the saturated suffix property, and hence is optimal.

As an example, consider an instance with two operators O_1 and O_2, where $r_1 = 3$, $r_2 = 2$, and $p_1 = p_2 = 1/2$. Let $\pi = (O_1, O_2)$. After sending $x = 8/3$ flow along ordering π, the residual capacity of O_1 is $3 - 8/3 = 1/3$ and the residual capacity of O_2 is $2 - 1/2 * 8/3 = 2/3$, so O_2 can now exactly saturate O_1. We place $(\pi, 8/3)$ into our routing. We swap O_1 and O_2 in π and form a superoperator (O_2, O_1) with rate limit 2/3 and selectivity $1/2 * 1/2 = 1/4$. Treating it as a single operator in π, a trivial repetition of the above procedure (on one operator) finds that sending $2/3$ flow units results in saturation of (super)operator (O_2, O_1). These units are sent along permutation $\pi' = (O_2, O_1)$, so $(\pi', 2/3)$ is added to the routing. The result is an optimal routing saturating both operators, whose total throughput is $8/3 + 2/3 = 10/3$.

D. The MTTC algorithm for chains

We now present an MTTC algorithm for the special case in which precedence graph G is a forest of chains. It is a generalization of the algorithm just described. That algorithm does not work here because precedence constraints may be violated when (1) initially ordering the operators in decreasing order of rate limits and (2) swapping the order of some O_i and O_j in π. The first problem is handled via a preprocessing procedure. To avoid the second problem we add additional steps to the above algorithm, yielding a new procedure that we call RouteChains. Details are below.

1) Preprocessing procedure: In the preprocessing procedure, we first make each chain of G proper, as follows. For each non-leaf operator O_i in the chain, beginning from the top of the chain and proceeding downward, we execute the following step: *Let O_j be the child of O_i. If O_i cannot saturate O_j, then reset the rate limit r_j of O_j to be $r_i p_i$.*

Although the above procedure reduces the rate limits of some operators, it does not reduce the maximum throughput attainable. Because of the precedence constraints, all flow into an operator O_j must first pass through its parent O_i, so at most $r_i p_i$ flow can ever reach O_j.

Once the chains are proper, we sort all operators in descending order of their rate limits. Let $\pi = (O_{i_1}, \ldots, O_{i_n})$ be the resulting permutation, and let partition P of π be $((O_{i_1}), \ldots, (O_{i_n}))$.

2) The RouteChains procedure: Following preprocessing, we run a recursive procedure called *RouteChains*. RouteChains incrementally constructs a routing K, consisting of pairs of the form (π, x), indicating that x amount of flow is to be sent along permutation π.

Define a *superoperator* to be a permutation π' of a subset of the operators, such that with respect to the routing constructed so far, each operator in π' (but the last) can saturate its successor in π'. The selectivity of π', denoted $\sigma(\pi')$, is the product of the selectivities of its component operators, and its rate limit, denoted $\rho(\pi')$, is the (residual) rate limit of its first operator.

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 $\mathcal{O}$ obeying the constraints of G,
4. An ordered partition $P = (\pi_1, \ldots, \pi_m)$ of π into subpermutations π_i, where each π_i is a superoperator.

The inputs to RouteChains must obey the following **"can't overfill" precondition**: for $2 \leq j \leq m$, $\rho(\pi_j)\sigma(\pi_j) \leq \rho(\pi_{j-1})$. That is, each superoperator in P cannot overfill its predecessor.

For the initial call to RouteChains we use the values calculated during preprocessing. Because the chains are proper, π obeys the precedence constraints.

RouteChains: Execution

RouteChains first calculates the minimum $x \geq 0$ such that sending x flow units along permutation π triggers one of the following stopping conditions:

1. Some operator is saturated.

2. Some superoperator π_i, $2 \leq i \leq m$, becomes able to 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_i and O_j are contained in distinct superoperators of P.

The value of x is calculated based on the following observations. Suppose $\pi = (O_1, \ldots, O_n)$. For any operator O_j, if y flow units are sent along π, then $y\prod_{k=1}^{j-1} p_k$ units will reach operator O_j. The residual rate limit of O_j will then be $r_j - y\prod_{k=1}^{j-1} p_k$. Thus saturation of O_j occurs when $y = \frac{r_j}{\prod_{k=1}^{j-1} p_k}$. Similarly, it can be shown that for $2 \leq j \leq m$, superoperator π_j becomes able to exactly saturate π_{j-1} when $y = \frac{\rho(\pi_j)\sigma(\pi_j) - \rho(\pi_{j-1})}{\prod_{k=1}^{j} \sigma(\pi_k) - \prod_{k=1}^{j-2} \sigma(\pi_k)}$. Finally, for $1 \leq i < j \leq n$, O_i becomes able to exactly saturate O_j when $y = \frac{r_i p_i - r_j}{\prod_{k=1}^{i} p_k - \prod_{k=1}^{j-1} p_k}$. Thus x can be calculated by taking the minimum of $O(n)$ values.

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 π_i, then RouteChains chooses one such π_i. It swaps π_i and π_{i-1} in $(\pi_1, \ldots, \pi_m)$ and concatenates them 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 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 and *absorbs O_j into O_i* as follows.

If O_i and O_j are contained in superoperators (O_i) and (O_j), each containing no other operators, RouteChains deletes the superoperator (O_j), and adds O_j to the end of the (O_i), to form superoperator (O_i, O_j). Otherwise, 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 overfill" precondition with respect to the residual rate limits. (For example, it may 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', with 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 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, and 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 $\{(\pi, x)\}$.

Theorem 4.1: When run on an MTTC instance I whose precedence graph is a set of proper chains, RouteChains produces an optimal routing for I with the saturated suffix property. RouteChains runs in time $O(n^2 \log n)$, and produces a routing that uses at most $4n - 3$ distinct permutations. [3]

---

3) Example: We illustrate our algorithm using the 4-operator selection ordering query shown in Figure 2 (i), which has three chains, and one precedence constraint, between O_4 and O_5 (Figure 5). We will assume the rate limits 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 throughput achieved is 1560; the best serial plan can only process 900 tuples per unit time.

E. 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,

[3]The proofs of the theorems in this paper can be found in the extended version of the paper [7].

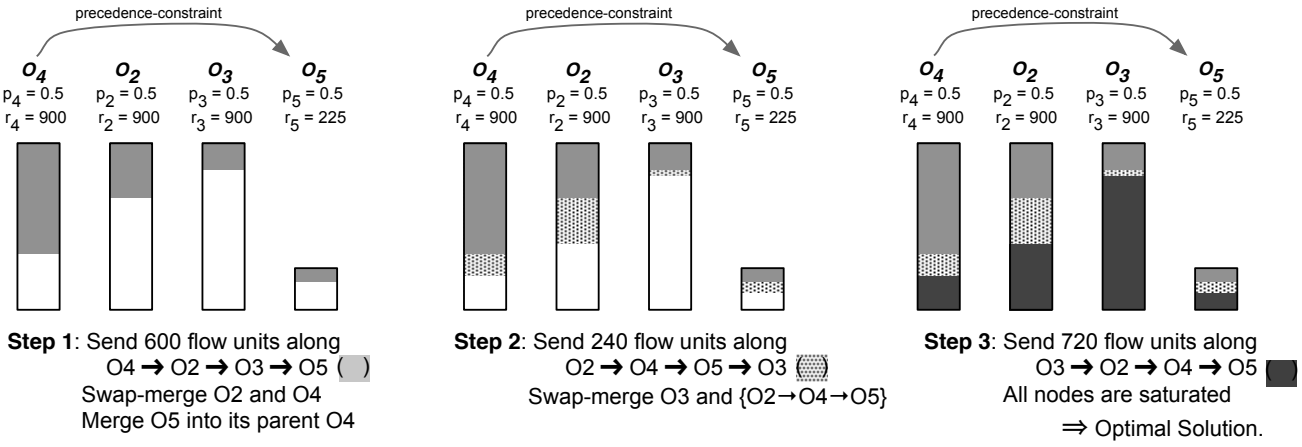

Fig. 5. Illustration of the algorithm for the case shown in Figure 2 (ii); O_i corresponds to the join with R_i.

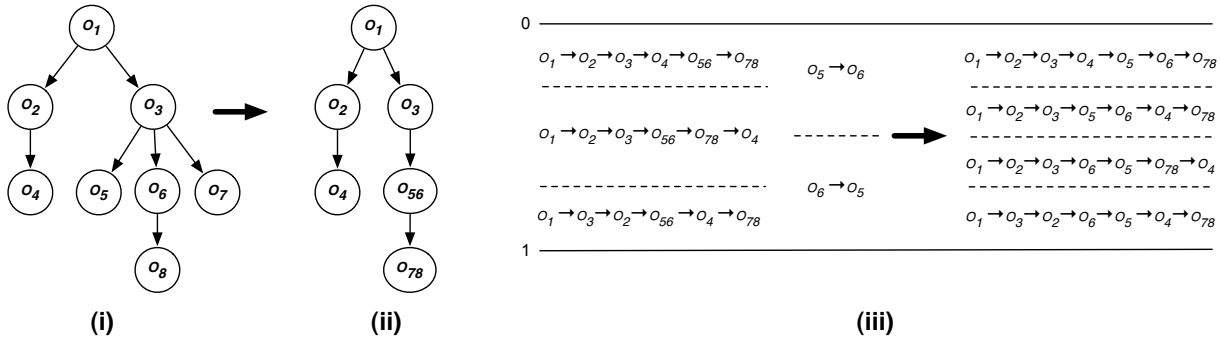

Fig. 6. (i) An example precedence graph; (ii) The forest of chains below operator O_3 is replaced by a single chain to obtain a new problem; (iii) The solution for the new problem and for the operator O_{56} are combined together.

the algorithm works by recursively eliminating forks from G, bottom-up. Before describing the algorithm, we illustrate it with an example.

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 6 (i). This graph has several forks; let the next fork we eliminate be at node O_3. 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 to 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 6 (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 $\{O_7, O_8\}$, 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 [5]. We illustrate this with an example (Figure 6 (iii)).

Suppose K'', the optimal solution for the reduced problem (Figure 6 (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 carries $\frac{1}{2}t$ flow. Similarly, suppose routing K_{56} for O_{56} sends half the flow along permutation O_5, O_6 and half along O_6, O_5. These two routings are shown graphically in the first two columns of Figure 6 (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 column 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.

2) MTTC algorithm description: The MTTC algorithm is a recursive procedure that works by repeatedly eliminating forks in the precedence graph. We describe the steps in the MTTC algorithm here. The algorithm uses two procedures, CombineRoutings and CombineChains, described below.

1. Base Case: If G is a forest of chains, apply the MTTC algorithm for chains (Section IV-D) and return the solution.
2. Otherwise, let O_i be a fork of G whose subtrees are all chains.
3. Let S denote the set of descendants of O_i, and let I_S denote the induced MTTC instance I restricted to the operators in S. The precedence graph G_S of I_S is thus a forest of chains.
4. Make the chains in I_S proper (Section IV-D.1), 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 precedence graph G' from G by replacing the chains below O_i with the single chain $(\mathcal{A}_1, \ldots, \mathcal{A}_m)$. Thus G' has one less fork than G.
7. Let I' be the resulting new MTTC instance, having precedence graph G'.
8. Recursively solve I'. Let K' be the routing returned by the recursive call.
9. Use CombineRoutings to combine the K_{A_i} with K'. Return the resulting routing.

CombineChains: CombineChains is used in fork elimination, to replace a set of chains emanating from a fork by a single chain. In the example above, to eliminate a fork we ran RouteChains repeatedly on the chains emanating from the fork, each time removing the operators in a saturated suffix. For efficiency, CombineChains does something slightly different. It uses another procedure (described below) to identify 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.

More specifically, let I be the input to CombineChains. 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 $r_1 \geq \ldots \geq r_n$. It 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 that C_{j^*} is a saturated suffix in an optimal routing of the chains. CombineChains runs RouteChains just on the (sub)chains of operators in C_{j^*}, to produce a routing K_{j^*}.

CombineChains then removes the operators in C_{j_*} from I; the operators in C_{j_*} will always appear at the end of the chains. 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.

Combining Routings: CombineRoutings is used to combine the K_A with A'. The approach is described in [5] and has already been illustrated with an example above. See [5] for more details.

The number of permutations used in the combined routing is 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.

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.

V. MTTC: Non-selective operators

Multi-way join queries may contain non-selective operators with fanouts larger than 1. Next we briefly sketch how the previous algorithm can be extended to handle such cases. Further details can be found in [7]. We also note that, in such cases, it might be preferable to consider an alternative plan space [21] or a caching-based approach [7].

A. All non-selective operators

If all operators are non-selective, then we construct an equivalent problem with only selective operators, by (1) replacing the selectivity, p_i, of an operator O_i with $1/p_i$, and (2) reversing all the precedence constraints. After solving this new problem (which only contains selective operators), we reverse each of the permutations in the resulting routing to obtain a routing for the original problem.

Note that the precedence graph for the new problem may be an "inverted tree". The approach used in Section IV-E (wherein we replace forests of chains by a single chain) can be extended to handling such precendence graphs [7].

B. Mixture of Selective and Non-Selective Operators

However, if the problem instance contains both selective and non-selective operators, the problem is more complex. If the precedence graph is a forest of chains, then we can solve the problem optimally 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 $p_i p_j$

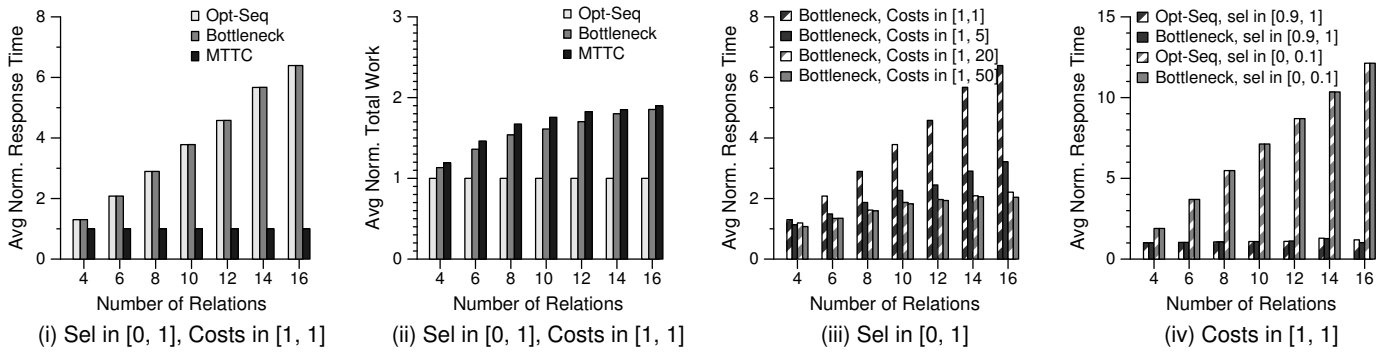

Fig. 7. *Star* query experiments: Comparing (i) response times (normalized using the MTTC solution) and (ii) total work (normalized using the OPT-SEQ solution) for the three algorithms; (iii) With high variance in the operator costs, benefits of interleaving plans are lower; (iv) Interleaving plans are most beneficial when selectivities are low.

and rate limit $\min(r_i, r_j/p_i)$. 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).
- Solve these two problems separately to obtain routings K'_s and K'_{ns}, and combine them (as described in Section IV-E).

The optimality proof can be found in [7].

Given the above algorithm for forests of chains, we can once again apply the procedure described in Section IV-E 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.

VI. Experimental Study

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 [19]:** The optimal serial plan for the centralized case, that minimizes the total work done, found using the *rank ordering* algorithm (Section III-C).
- **BOTTLENECK [21]:** The serial plan that minimizes the response time (bottleneck) using a serial plan, found using the Bottleneck Algorithm (Section III-C).
- **MTTC:** The optimal interleaving plan found by our algorithm presented in Section IV.

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 graph, 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 graph (Figure 7 (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. We first compare the normalized response times of the three planning algorithms for a range of query sizes (Figure 7 (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). 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 7 (ii)). As expected, the interleaving plans do more work than the OPT-SEQ plan (by up to a factor of 2), but the amount of additional work not large compared to the benefits in the response time obtained 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 III-C); 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

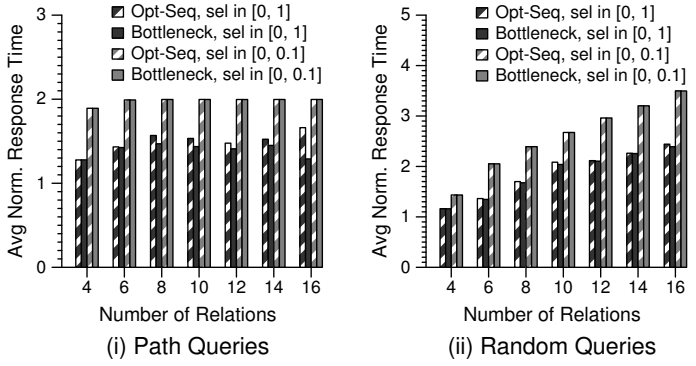

Fig. 8. Comparing the three planning algorithms for (i) path query graphs and (ii) randomly-generated query graphs.

$X \in \{1, 5, 20, 50\}$. As we see in Figure 7 (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 7 (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 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 8 (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 8 (ii) shows the results for this set of experiments. Even for queries with as few as 6 relations, we get significant benefits, including when 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.

VII. 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.

Acknowledgment

This work was supported by NSF Grants IIS-0546136 and ITR-0205647.

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] C. Chekuri, W. Hasan, and R. Motwani. Scheduling problems in parallel query optimization. In *PODS*, 1995.
[4] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Flow algorithms for two pipelined filter ordering problems. In *PODS*, 2006.
[5] A. Condon, A. Deshpande, L. Hellerstein, and N. Wu. Algorithms for distributional and adversarial pipelined filter ordering problems. *To Appear in Transactions on Algorithms*, 2008.
[6] A. Deshpande, C. Guestrin, W. Hong, and S. Madden. Exploiting correlated attributes in acquisitional query processing. In *ICDE*, 2005.
[7] A. Deshpande and L. Hellerstein. Flow algorithms for parallel query optimization. Technical Report CS-TR-4873, Univ of Maryland, 2007.
[8] D. DeWitt and J. Gray. Parallel database systems: the future of high performance database systems. *CACM*, 35(6), 1992.
[9] D. DeWitt, J. Naughton, D. Schneider, and S. Seshadri. Practical skew handling in parallel joins. In *VLDB*, 1992.
[10] D. J. DeWitt et al. The Gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), 1990.
[11] S. Ganguly, W. Hasan, and R. Krishnamurthy. Query optimization for parallel execution. In *SIGMOD*, 1992.
[12] M. Garofalakis and Y. Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *SIGMOD*, 1996.
[13] M. Garofalakis and Y. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *VLDB*, 1997.
[14] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD*, 1990.
[15] W. Hasan, D. Florescu, and P. Valduriez. Open issues in parallel query optimization. *SIGMOD Record*, 1996.
[16] W. Hasan and R. Motwani. Coloring away communication in parallel query optimization. In *VLDB*, 1995.
[17] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in xprs. In *PDIS*, 1991.
[18] T. Ibaraki and T. Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, 1984.
[19] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of nonrecursive queries. In *VLDB*, 1986.
[20] Bin Liu and Elke A. Rundensteiner. Revisiting pipelined parallelism in multi-join query processing. In *VLDB*, 2005.
[21] U. Srivastava, K. Munagala, J. Widom, and R. Motwani. Query optimization over web services. In *VLDB*, 2006.
[22] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, 2003.
[23] S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB*, 2003.
[24] Annita N. Wilschut, Jan Flokstra, and Peter M. G. Apers. Parallel evaluation of multi-join queries. In *SIGMOD*, 1995.