Join Reordering by Join Simulation

Chaitanya Mishra , Nick Koudas

University of Toronto
Toronto, Canada
cmishra@cs.toronto.edu
koudas@cs.toronto.edu

***Abstract*— We introduce a framework for reordering join pipelines at runtime in a database system. This framework incorporates novel techniques for simulating the execution of a join pipeline using random samples and statistical summaries. Our simulation techniques provide accurate runtime cardinality estimates along all alternative execution paths of a join pipeline. These estimates are then utilized to compare costs of alternative execution paths in a dynamic fashion, and reorder the pipeline if a better alternative path is found. We describe simulation techniques for pipelines of different kinds of join operators. We present an experimental evaluation of a prototype implementation of our framework in an open source data manager. The results demonstrate the feasibility and utility of the approach presented herein.**

I. Introduction

Despite many years of research, database query optimizers often generate sub-optimal query evaluation plans. These errors typically arise due to incorrect cardinality estimates obtained from incomplete statistics and uniformity/independence assumptions. Much research has been done to address these problems, including (to give a few examples) improvements to histograms [1], sampling [2], and new schemes for collecting and maintaining statistics [3]. However, all these techniques are fundamentally limited by the design of traditional database engines which separates the optimization and execution phases of query processing.

In order to overcome these obstacles, many adaptive query processing techniques have been proposed in recent years [4]. These techniques interleave the optimization and execution phases of query processing, enabling reoptimization of sub-optimal query plans during query execution. Typically, such systems deploy a monitoring component which collects statistics during query execution and checks certain plan switching conditions, which if satisfied, trigger reoptimization.

Adaptive query processing techniques differ in terms of the scope of reoptimization considered. *Global reoptimization* techniques [5], [6] invoke the query optimizer with new statistics collected during query execution, and generate a new evaluation plan for the entire query, which may or may not incorporate the partial results produced by the current plan. The goal of these techniques is to generate the best possible query plan, while utilizing partial statistics obtained for the subset of relations accessed at the point of reoptimization. Since these techniques utilize partial local information to perform a global plan transformation, they run the risk of cardinality estimation errors similar to those associated with traditional query optimization. In contrast, *local reoptimization* techniques such as the switchable plans in Rio [7], and nested loops join reordering [8], avoid such risks by altering the current plan through smaller localized changes. The goal of these techniques is to switch to a better plan, in a robust, lightweight, and risk-free manner.

In this paper we present the XS (*Execution Simulation*) framework for reordering join pipelines during query execution. XS is a local reoptimization technique which detects sub-optimality of the currently executing join pipeline, and if so, reorders its inputs to obtain a better join pipeline. We introduce the concept of *simulation* of join pipelines using random samples and statistical summaries, and demonstrate how simulation can be used for cardinality estimation. We derive confidence guarantees, and show that our techniques provably converge to correct cardinality estimates with increasing random sample size. We present techniques for simulating all alternative execution paths of a given join pipeline efficiently *without making inter-table independence assumptions* enabling us to quickly identify the optimal alternative join pipeline.

II. Related Work

Adaptive query processing has recently received much attention from the research community. Babu and Bizarro [9] provide an overview of this area and define a taxonomy of such systems. More recently Deshpande et al. [4] also provide a comprehensive survey of this area.

Two systems similar to XS are the join reordering framework introduced by Li et al. [8] and the proactive reoptimization system, Rio [7]. Both these systems advocate lightweight local reoptimization at runtime, avoiding optimizer invocations for the entire plan. Li et al. [8] introduce a reordering technique for pipelines of Nested Loops Joins. The system monitors operator selectivities throughout query execution and reorder operators to execute the more selective joins first. However, the technique assumes independence between the selectivity of join operators, and can therefore make estimation errors and thrash between plans. In contrast, XS supports hash join pipelines as well, and avoids inter-table independence assumptions. Additionally, XS performs sampling based cardinality estimation at well defined points of symmetry during query execution, avoiding the overhead of continuous monitoring. Like the XS framework, the Rio system [7] also performs random sample processing at runtime to switch between a set of plans. However, the notion of switchable plans is limited

to switching the inner and outer inputs of a join operator. In contrast, our technique supports arbitrary reordering of the inner inputs of a join pipeline. Moreover, XS does not require any modifications to the query optimizer to produce switchable plans as required by the Rio system.

The Re-Opt [5] and POP [6] frameworks support global reoptimization of the current query evaluation plan. These systems detect cardinality estimation errors by monitoring the count of tuples at materialization points in the query execution plan. If the observed count is significantly different from the estimated count, reoptimization is triggered, and the optimizer is invoked with newly collected statistics. The new plan generated may or may not use intermediate results produced by the previous plan. The XS framework complements such global reoptimization techniques with local reordering techniques that do not require full plan reoptimization. In addition, we introduce advanced monitoring techniques in the form of constructing histograms at blocking points, which could also benefit global reoptimization techniques such as POP.

The Oracle RDB system [10] runs multiple plans competitively before selecting a final query evaluation plan in order to disambiguate uncertainty in execution costs. XS also compares the performance of alternative execution plans for the pipeline through lightweight simulation techniques as opposed to actually executing competing plans. Tuple routing techniques like the Eddies framework [11] also involve exploration and evaluation of different execution plans. This comparison is performed by a central eddy operator which routes tuples and monitors costs of different operators in the plan. However, Eddies require join operators with frequent moments of symmetry, and involve the overhead of maintaining routing information for each tuple.

More recently, in the context of query execution feedback, Chaudhuri et al. [12] introduced techniques to estimate cardinalities along alternate execution paths of a query. Their solutions are limited to 2-way key-foreign key joins with additional restrictions on the order in which the relations are processed. Our monitoring techniques apply to general multiway equijoins , and can be applied in the context of extracting statistics for future query executions as well.

III. The XS Framework

A. Problem Statement

A *pipeline* is a set of concurrently executing operators in a query execution plan. A plan is split into multiple pipelines by blocking operators. A *join pipeline* is a set of concurrently executing join operators. If the operators are the same, the join pipeline is *homogeneous*. A pipeline of k join operators has k inner inputs and 1 outer input. The outer input of a pipeline *drives* its execution. We define the *alternative orders* of a pipeline as a set of pipelines with the same operators and inputs which have the *same outer input*. For example, the pipelines shown in Figures 1(a)-(c) are all alternative orders of each other (the outer input is shown on the left). We restrict the set of alternative orders to join orders not involving Cartesian products (unless unavoidable) since such join orders are typically inefficient. Alternative orders therefore capture the set of pipelines which can be easily transformed into one another by reordering *only their inner inputs*. We represent the set of alternative orders of a pipeline as a *join lattice*. Each node in the lattice represents an intermediate point along one or more join paths. The lattice shown in Figure 1(d) corresponds to the alternative orders shown in Figures 1(a),(b) and (c). We annotate nodes with an estimate of the cardinality of the corresponding intermediate join result, and edges with an estimate of the cost of the corresponding join operation. The lattice structure is described in more detail in Section III-C

We define our problem of *join reordering* as the identification of the optimal alternative order *during execution* of a given join pipeline. This definition of reordering focuses the scope of our framework to switching the order of inners in a join pipeline. Such a focus enables utilization of a well defined *point of symmetry* at which reoptimization may be performed. This is the point when the preprocessing of the inner inputs is complete (e.g. loading the indices or sorting/hashing the inner relations), and the first *getnext* call is performed for the outer input. Reordering inner inputs at this point has no effect on the outer input (which may be a base table or materialized result of a lower pipeline) or the result of the pipeline (which may be an input for the next pipeline to be executed). Moreover, no processing of inners needs to be repeated since we do not alter the join operator and/or the outer input. Additionally, we are able to perform cardinality estimation without making independence assumptions, and without *processing join inputs out of order*. Avoiding independence assumptions is essential for accuracy purposes, while processing inputs in a given order allows our techniques to easily apply to arbitrary join pipelines in which inner and outer inputs are produced by a different processing pipeline. Our experimental evaluation in Section IV demonstrates that our model of reordering significantly speeds up query execution.

In this paper, we focus on the problem of reordering homogeneous join pipelines. In particular, we present techniques for reordering pipelines of hash joins and index nested loops joins. In Section III-F, we present a brief discussion on how these techniques could potentially be extended to other join operators, and the challenges associated with non-homogeneous join pipelines.

B. Overview

Given a query evaluation plan, our system partitions it into a set of pipelines using blocking operators as partition boundaries. For each pipeline, we use the join conditions to infer the corresponding join lattice. The goal of our simulation procedures is to obtain accurate cardinality estimates *at each node of the lattice*. These cardinality estimates can then be utilized by cost estimation functions to obtain cost estimates for the join operations. We annotate the corresponding edges of the join lattice with these cost estimates. Given such an annotated join lattice, the problem of identifying the optimal

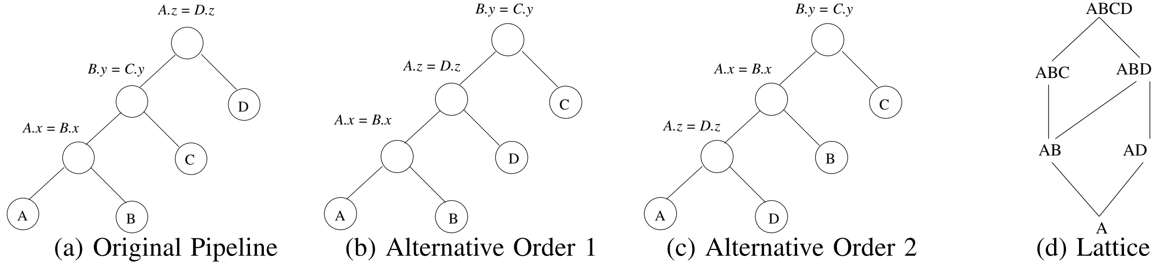


Fig. 1. Join Pipelines (Outer Input on the left)

alternative execution order is equivalent to computing the *shortest path* from the bottom to the top of the lattice. In this section, we present a high level overview of our techniques, with particular emphasis on the statistical estimation procedures. Our technique proceeds in 3 steps:

[Step 1] Obtaining a random sample of the outer input: Our framework relies on random samples in order to obtain fast and accurate cardinality estimates. Random samples may either be precomputed (for base tables), or generated on the fly at runtime. At blocking points, we perform reservoir sampling [13] to generate samples of a certain size. Unlike the Rio framework [7], we do not require modifications to join operators to generate random samples of their outputs.

[Step 2] All-paths simulation: Suppose we have a join operator with outer input S and inner input R and join condition $R.x = S.x$. We assume the presence of a random sample of S obtained using the techniques described above. Let D_n denote our estimate of the size of $R \bowtie S$ when n tuples of a random sample of S have been joined with R. Let the frequency of value i in column $S.x$ in the first n tuples of the sample be $f_i^{S_n}$. Similarly, let f_i^R be the frequency of i in the column $R.x$. Now $\frac{f_i^{S_n}}{n}$ is an unbiased estimate of the fraction of tuples in relation S that have value i in column $S.x$. By treating this as a Bernoulli random variable, we can derive $\alpha-$percentile confidence intervals $(1 \pm \frac{\beta}{2})$ as $\frac{\beta}{2} = \frac{Z_\alpha}{2\sqrt{n}}$ where Z_α can be obtained from standardized normal tables (e.g., for α=99.99%, $Z_\alpha = 4$). Given this, it can be seen that

$$D_n = \sum_{i \in R.x \cap S.x} \frac{f_i^R f_i^{S_n}}{n} |S| \pm \frac{f_i^R |S| \beta}{2}$$

We note that this is a typical estimation procedure for join cardinality estimation, and stronger confidence bounds derived in the literature [2], [14] can also be applied here. An advantage of this estimation technique is that it is incrementally maintainable i.e it is possible to express D_{n+1} in terms of D_n. If the $(n+1)$th tuple has the value i on its join attribute, it produces f_i^R tuples in the join output. Therefore (ignoring confidence bounds) we have:

$$D_{n+1} = \frac{D_n \cdot n + f_i^R\,|S|}{n+1}$$

We use this formula to obtain incremental cardinality estimates and associated confidence bounds at all nodes in the join lattice. Relation S corresponds to the outer node of the join pipeline. Relation R corresponds to a join of the inner inputs at that node in the lattice. Given a tuple t of the outer relation, the primary technical challenge is to obtain the number of tuples produced f_i^R for *each node of the lattice*. We describe our techniques to do so for general join lattices in more detail in Section III-C.

Given this estimation procedure, we define stopping conditions that specify when the simulation process is to be halted. To define a stopping condition, we could utilize confidence bounds derived using limit theorems for join cardinality estimation, which however can be rather conservative. Instead, we define stopping conditions based on the *stability* of the estimates. At regular intervals, we call optimizer cost estimation functions with the new cardinality estimates obtained through simulation. If the resulting cost estimates remain stable within a pre-specified threshold $(1 \pm \delta)$, we conclude that they have converged to the correct values. This defines a stopping condition which enables early termination of our simulation procedure. We also bound the size of the random sample processed to ensure that the time spent in simulation is low. If the estimates haven't stabilized during the processing of the sample, simulation is halted, and the pipeline proceeds as before without checking whether we should switch to an alternative execution plan.

[Step 3] Comparing alternative orders and reordering: The stopping conditions as defined previously compute the costs of joins at regular intervals and update the edges of the join lattice. In the face of these cost updates, we show in Section III-G, that it is easy to dynamically maintain the cheapest execution path on the join lattice. Once our stopping condition is satisfied, we compare the optimal alternative execution order having cost $C(Alt)$ to the current join order having cost $C(Cur)$. These costs are obtained by invoking cost functions with the new cardinality estimates. We switch to the alternative order if $C(Alt) < \tau \times C(Cur)$ where the parameter τ is a tunable threshold between 0 and 1 which represents how sensitive the reordering strategy is to the newly obtained cost estimates. This parameter is required since even with correct cardinality estimates, the cost estimation function of an optimizer is not a perfect predictor of the execution time of an operator. Once the switching decision is made, execution of the (possibly reordered) pipeline proceeds without simulating alternative orders.

Effectively, the XS framework pauses regular query execution at a point of symmetry to explore alternative execution

orders using simulation. Having simulated the alternative orders, XS selects the optimal alternative execution order, and reorders the join pipeline, if required.

C. Join Simulation

In Section III-B we described our cardinality estimation procedure **[Step 2]** for a 2 table join. The primary challenge we face is to obtain the number of tuples f_i^R of the inner relation R that join with a single tuple of the outer relation S with value i on the join column. For multiway joins with more than 2 relations, we require such estimates for all the nodes of the join lattice. In this section, we formally describe our simulation procedure, leaving details of specific instantiations of the procedure for different types of joins to Sections III-D and III-E. We use as a running example the 3 join lattices shown in Figure 2. We assume that relation A is the outer relation and the join initially proceeds as per the pipeline $(((A \bowtie B) \bowtie C) \bowtie D)$. The 3 join lattices represent the cases when the outer relation joins with 3, 2, and 1 inner relations respectively.

We have access to a stream of random samples of the outer relation A. For each tuple $t(A)$ processed, we require the number of tuples it will produce at each node of the lattice. This quantity, which corresponds to f_i^R is termed as the *tuple-contribution* $N_t(X)$ of the tuple t at the node X. For instance $N_t(ABC) = |t(A) \bowtie (B \bowtie C)|$ is the number of tuples produced at node ABC by joining tuple $t(A)$ with the inner relations B and C. The notion of *tuple-contributions* also extends to tuples produced as the result of a join operation. Thus, for example, if a tuple $t(AB) \in \{t(A) \bowtie B\}$ joins with relation C to produce 3 tuples, then we write that $N_{t^B}(ABC) = 3$. We use N_{t^X} to denote tuple-contributions of a tuple $t(AX) \in (t(A) \bowtie X)$ where X is a relation or join of relations. Our goal is to compute tuple-contributions at each node of the lattice for each tuple $t(A)$ of the random sample of A that we process.

To compute the tuple-contributions at all nodes of the lattice, we utilize information about the structure of the lattice. Each edge of a lattice represents a join operation. We annotate edges of the lattice with (*source, target*) pairs. The *target* relation represents the new relation being joined at the edge. The *source* relation represents the relation at the lower node of the edge from which the join attribute for the edge is extracted. Thus, for example, in Figure 2(b), the edge between AD and ABD is annotated with (A, B) since it corresponds to the join condition $A.x = B.x$. In case of multiple choices of the source relation, we select the source relation that is lower in the join tree (the outer relation being the lowest).

Multiple edges of a lattice can have the same annotations. We term edges with the same annotations as being *equivalent*. Thus in figure 2(b), the edges $A - AB$ and $AD - ABD$ are equivalent. Equivalent edges capture dependencies between the tuple-contributions at each node of a lattice. For example, in Figure 2 (b) suppose that the current outer tuple $t(A)$ joins with 3 tuples of relation B i.e $N_t(AB) = 3$. Then, we know that each tuple $t(AD) \in \{t(A) \bowtie D\}$ would join with 3 tuples of relation B as well. If we have single equijoin conditions at each operator of the join pipeline, then a lattice representing the alternate orders of a k operator join pipeline can have its edges split into at most k equivalence classes due to the k join conditions.

Equivalence classes represent relationships between tuple-contributions at different nodes of a lattice. We maintain such relationships as a *Dependency Table*, which expresses the relationships between nodes in terms of sums and products. For example, consider *Lattice1* in Figure 2(a). In this lattice, the outer relation A joins with all 3 inner relations. Therefore, we only need to compute $N_t(AB), N_t(AC)$ and $N_t(AD)$. The tuple-contributions at all other nodes follow from these values. For instance, $N_t(ABC) = N_t(AB) \times N_t(AC)$, because the edges between A and ABC belong to different equivalence classes (A, B) and (A, C) which have the same source node A. Note that this expression *does not assume independence*, since it represents the tuple contribution of a *single tuple*. Thus, if a tuple $t(A)$ joins with 2 tuples of relation B and 3 tuples from relation C, it will produce 6 tuples at node ABC.

Lattice1 illustrates the case when dependencies can be expressed as only products. We now present *Lattice2* from Figure 2(b) where sums are required as well. In this example, $t(A)$ joins with relations B and D giving values of $N_t(AB)$ and $N_t(AD)$. However, the edge between relations B and C is annotated as (B, C) corresponding to the join condition $B.y = C.y$. Therefore, for each $t(AB) \in (t(A) \bowtie B)$, we compute the number of tuples of relation C that join with it. This is its tuple-contribution $N_{t^B}(ABC)$. *Summing over* the values of $N_{t^B}(ABC)$ for all tuples $t(AB) \in (t(A) \bowtie B)$ produced by the current outer tuple $t(A)$ provides us with the value of $N_t(ABC)$.

Lattice3 from Figure 2(c) represents the extreme case when the outer relation A joins with only one inner relation B. The resulting tuples $t(AB)$ can then be used to compute $N_{t^B}(ABC)$, $N_{t^B}(ABD)$ and $N_{t^B}(ABCD) = N_{t^B}(ABC) \times N_{t^B}(ABD)$. Summing over these values provides us with the tuple contributions N_t due to the outer tuple.

Effectively, the dependency table expresses relationships between tuple-contributions in terms of sums and products. Product terms are a result of multiple edges having the *same source relation*. Sum terms are due to summation over tuple contributions produced by tuples from intermediate join results. Computing tuple-contributions correctly is at the core of our cardinality estimation procedures. We next describe how to perform such computation for index nested loops (Section III-D) and hash (Section III-E) joins.

D. Index Nested Loops Joins

In this section, we assume that the join lattices in Figure 2 represent pipelines of index nested loops joins (INLJs). We let the original join pipeline be $(((A \bowtie B) \bowtie C) \bowtie D)$, with indices on the join attribute for the inner relations B, C and D, and a random sample stream of tuples from the outer relation A.

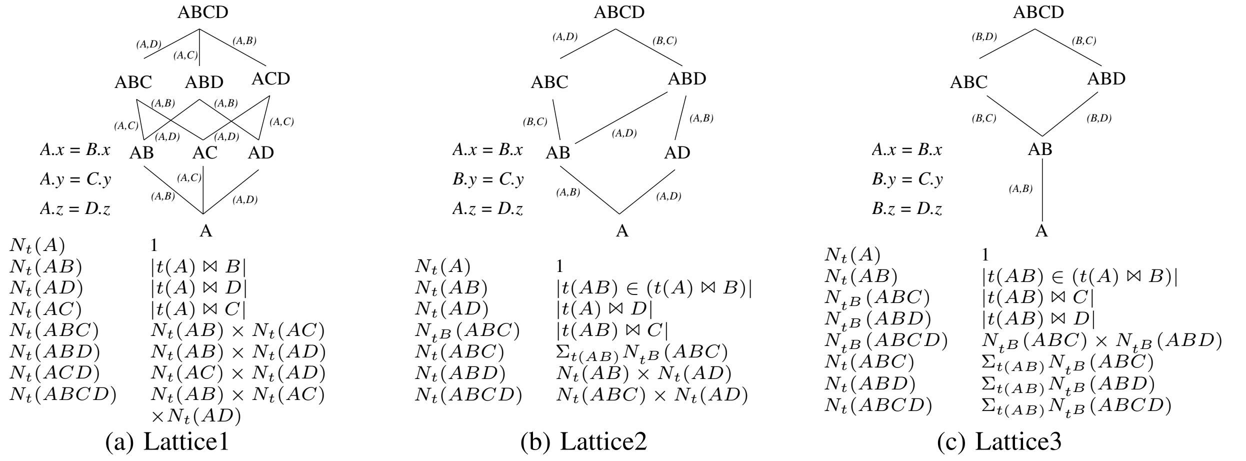


Fig. 2. Join Lattices and Tuple Contribution Dependency Tables

Our goal is to efficiently compute tuple contributions $N_t(X)$ at each node X of the join lattice given a tuple $t(A)$ of the outer relation A. Note that computing tuple-contributions at nodes A, AB, ABC and $ABCD$ is straightforward. We simply execute the given join pipeline and count the number of tuples produced at each intermediate node. In contrast, computing tuple-contributions at the remaining nodes (e.g. AD, ABD) is not easy since these nodes are not observed during normal execution of the join pipeline. We can compute tuple-contributions at nodes that are not observed during normal execution of the pipeline by performing additional index probes. For example, in *Lattice1* shown in Figure 2(a), $N_t(AD)$ can be computed by probing the index on D with $t(A)$.

In order to minimize the number of index probes made, we utilize the *dependency table* computed for the join lattice. Consider the dependency table for *Lattice1* shown in Figure 2(a). Given a tuple $t(A)$ of the outer relation, we observe that there are 3 entries in the dependency table that correspond to the join of $t(A)$ with another relation. For example $N_t(AC) = |t(A) \bowtie C|$. Therefore, we perform index probes on B, C and D to compute $N_t(AB)$, $N_t(AC)$ and $N_t(AD)$. The remaining tuple-contributions can be computed from these values.

As another example, consider the dependency table for *Lattice2* shown in Figure 2(b). Unlike *Lattice1* the outer relation in *Lattice2* does not join with all inner relations. In this case, we perform index probes on relations B and D with the outer tuple $t(A)$ to compute $N_t(AB)$ and $N_t(AD)$. Additionally, the tuples $t(AB) \in (t(A) \bowtie B)$ probe the index on relation C to compute $N_{t^B}(ABC)$. The remaining tuple-contributions can be computed by appropriately summing and multiplying these values as per the dependency table. Similar computations can be performed for *Lattice3* as well.

These examples illustrate our estimation procedure for pipelines of INLJs. We perform index-probes for each entry of the dependency table that takes the form $t \bowtie R$ where t is an outer or intermediate tuple, and R is an inner relation. The equations encoded in the dependency table enable us to compute the remaining values of tuple-contributions appropriately. Performing such estimation allows us to obtain tuple-contributions along *all alternative execution orders* without actually executing all of them.

E. Hash Joins

In the previous section, we described an estimation procedure for pipelines of INLJs that hinged on the fact that we have access to tuples of the inner relation through indices. If we had a pipeline of hash joins on small inputs (the entire inner hash table fits into memory), then we could have followed a similar strategy for estimation. After building in-memory hashtables for each of the inner relations, we could utilize these hashtables as indices and proceed as with INLJs. However, on larger inputs, hash join algorithms like Grace Join or Hybrid Hash Join, partition the inputs into batches and perform the join in a batchwise manner. With such algorithms, the hash join pipeline exhibits limited buffering since an outer input could either be immediately joined or written to a temporary buffer depending on the corresponding hash partition. In such cases, we cannot apply the estimation procedure in its current form since only a part of the inner hash table is accessible in memory at any point during the execution of the join.

Our estimation procedure for a pipeline of hash join operators utilizes the fact that although inner hash tables are usually too large to fit in memory, it is possible to build a *statistical summary structure* on the join column which can be kept in memory for estimation purposes. We note that this summary structure is kept separately in memory from the hashtable which may be partitioned and spill to disk. The construction of this summary structure proceeds along with the hashtable creation step for the inner relations. Given the appropriate summary structures at each node of the join lattice, we can *simulate* the join using a random sample of the outer input as detailed in Section III-B. In the discussion that follows, the terms histogram and summary structure are used interchangeably for ease of presentation. We defer a discussion on the appropriate choice of summary structures to Section III-

E.1. As before, we use as a running example the join lattices described in Figure 2 and assume that the initial join order is $(((A \bowtie B) \bowtie C) \bowtie D)$.

Given a join pipeline, our goal is to identify the set of statistical summary structures to be constructed for simulation of all alternative execution paths. An important consideration to note is that the original join pipeline decides the order in which relations are read. For example, given the original join order as $(((A \bowtie B) \bowtie C) \bowtie D)$, the pipeline will proceed by first building hashtables on relations D, C and B in order. If these relations are larger than memory size, only some partitions are kept in memory, while the rest is spilled back to disk. This is the *build* phase of the hash join pipeline.

Consider *Lattice1* from Figure 2(a). Suppose that while building inputs D, C, and B we construct histograms on $D.z$ and $C.y$ and $B.x$ Now, for each tuple $t(A)$ of the outer relation A, we can utilize these histograms for estimating $N_t(AB) = |t(A) \bowtie B|$ and similarly $N_t(AC)$ and $N_t(AD)$. These tuple-contributions can then used to compute tuple-contributions at the remaining nodes of the lattice.

As a somewhat different example, consider the join lattice *Lattice2* in Figure 2(b). As before, while building inputs D and C we construct histograms on join columns $D.z$ and $C.y$. Similarly, while reading B, we construct a histogram on the column $B.x$. Additionally however, we notice that there is a join condition $B.y = C.y$ which involves B and *does not involve* the outer relation A. Therefore we simulate the join of B and C using the histogram on $C.y$ to construct a new histogram on $(B \bowtie_y C).x$. This new histogram is constructed by probing the existing histogram on $C.y$ with the value $t(B).y$ for each tuple $t(B)$ of relation B to get an estimate of $|t(B) \bowtie_y C|$. This value is then used to update the frequency count of $t(B).x$ in the histogram $(B \bowtie_y C).x$. Given a tuple $t(A)$, we can probe this histogram to compute $N_t(ABC)$. We can compute $N_t(AD)$ and $N_t(AB)$ using the histograms on $D.z$ and $B.x$ and use these values to compute tuple-contributions at the remaining nodes.

The above examples describe our summary structure computation algorithm. When we build a hashtable on a new inner relation R on join column x, we also build a histogram on $R.x$. Additionally, we check the dependency table for any entries marked with $N_{t^{XR}}$ i.e entries that require tuples produced by a join with R. For each of these entries, we generate histograms by simulating joins corresponding to these entries. For example, consider *Lattice3* from Figure 2(c). As with *Lattice2*, we first build histograms on join columns $D.z$ and $C.y$. While reading in relation B, we construct a histogram on $B.x$. Additionally, we observe entries of the form $N_{t^B}(ABC), N_{t^B}(ABD), N_{t^B}(ABCD)$. Correspondingly, we simulate joins with $D.z$ and $C.y$ to construct the histograms $(B \bowtie C).x$, $(B \bowtie D).x$ and $(B \bowtie C \bowtie D).x$, which suffice for cardinality estimation. Given a join pipeline involving $k+1$ relations, our procedure requires between k (e.g. *Lattice1*) and 2^{k-1} (e.g. *Lattice3*) histograms in memory. The actual number depends on the structure of the original join pipeline and the join conditions therein. In practice, hash join pipelines which involve repartitioning have a small number of relations, and therefore few summary structures are required in general.

To summarize, we piggyback the construction of these histograms on the *build* phase of the join operators. Once the inner hash tables have been built, the probe phase of the join pipeline begins. In this step, we read in a random sample of the outer input, and simulate a join of the sample and the histograms constructed on the inner relations. Therefore, the online estimation algorithm for the case of hash joins does not involve any actual join computation. Once the sample has been processed and the stopping conditions triggered, we obtain accurate cardinality and cost estimates We switch to our predicted optimal join order, and then perform the actual join processing. The estimation phase thus fits in between the building of the hash tables, and the join processing which involves probing the hash tables.

In the next section, we discuss our algorithms to construct approximate summary structures for the purposes of join simulation.

1) Approximate Summary structures: The discussion in the previous section assumed exact histograms as summary structures. While this is feasible for small relations, it may not be possible to store the exact distribution for larger tables. As a result, we require a technique to approximate the frequency distribution along the join column. The technique should construct the summary structure in a *single pass* over the data. The summary structure must answer frequency and membership queries with high accuracy.

One might argue that utilizing an approximate summary structure exposes our technique to the same pitfalls associated with traditional query optimization. However, our techniques construct *query specific* summary structures, as opposed to traditional database catalog statistics, which are designed for use by all queries. As a result, we require fewer summary structures, and therefore have a larger space budget for the structures. Additionally, we construct our summary structures on the tuples after they have passed the selection predicates on the base relations, hence removing cardinality estimation errors due to the selection predicates. Finally, we note that our algorithms construct summary structures on joins of relations *without making independence assumptions*, hence removing a major source of error in join cardinality estimation.

In our current system, we utilize as a summary structure an approximation of an end-biased histogram constructed by a combination of bloom filters [15] and exact counts for the high frequency elements. We detect the high frequency elements using multistage filters [16], a technique to detect large flows in high volume network traffic. We note that our selection of these techniques is driven primarily by ease of implementation, and there are several other techniques (e.g [17], [18]) for detecting high frequency elements in a stream. We now provide a brief overview of our technique.

Bloom Filters: Bloom filters are a space efficient data structure for answering membership queries on a set. In particular, they have the property that they do not allow false negatives,

while having a tunable expected false positive rate. A bloom filter is a bit vector B of m bits, all initially set to 0. It uses k *independent hash functions* $h_1(x), \ldots, h_k(x)$ that map a value to a position in the bit vector. Each value of the set is hashed using the k functions, and each of the k bits is set to 1. Thus, for each inner tuple, we apply the k hash functions on the value of its join attribute to set k positions in the bit array. Membership queries on a bloom filter are performed similarly by hashing the value with the k hash functions and checking the corresponding bit positions. If all the k positions have bits set to 1, then the bloom filter returns *True*, else it returns *False*.

Multistage filters: Multistage filters are a technique for detecting high frequency values in a stream. Like bloom filters, they utilize k independent hash functions. Each hash function is associated with an array of counters, which are all initialized to 0. Figure 3 describes the multistage algorithm with 3 hash functions. Each value i is hashed using hash function $h_j(i)$ to array A_j and the counter $A_j[h_j(i)]$ is incremented. If the counter exceeds the threshold T for each array j, the value is identified as a high frequency element, and its count is explicitly maintained separately. Because high frequency elements are identified only if *each* counter is above the threshold T, the probability of false positives is low. Like bloom filters, multistage filters guarantee no false negatives. As with bloom filters, probabilistic guarantees can be derived for the multistage filters technique [16].

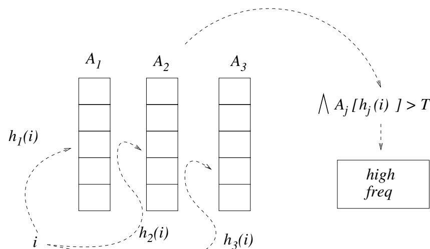


Fig. 3. Multistage Filters Algorithm

Algorithm 1 describes our algorithm for updating and maintaining a combined summary structure that uses a bloom filter and a multistage filter. When the inner relation is being built, the function *UpdateFilters()* is invoked with the value i on the join column. The function updates a bloom filter and multistage filter using a set of hash functions $h_j()$. In addition, the function also checks if the value is new, using the bloom filter, and whether it is a probable high frequency element using the multistage filter. If the value is new, the set of distinct values is updated. Likewise, if the value is identified as a high frequency element , its count is maintained separately. Finally, the average frequency of the low frequency elements is updated.

For the purposes of join simulation, when the outer relation is read in, the function *ProbeFilters()* is invoked for each tuple to find the number of matching elements in the inner relation. *ProbeFilters* first checks the bloom filter for membership of the value in the inner relation, and if present, checks the list of high frequency elements. If the value is in this list, it returns the stored frequency, otherwise it returns the average frequency of the remaining elements.

Algorithm 1 Updating and Probing Bloom and Multistage filters

```
UpdateFilters(Value i)
new = false
highfreq = true
numvals++
for all h_j ∈ hashfunctions do
    hashval = h_j(i)
    bit = hashval%bitarraysize
    if bitarray[bit] == 0 then
        new = true
        bitarray[bit] = 1
    end if
    index = hashval%arraysize
    A_j[index]++
    if A_j[index] ≤ T then
        highfreq = false
    end if
end for
if new == true then
    distinctvals ++
end if
if highfreq = true then
    UpdateCounters(i)
end if
UpdateAvg(i)

ProbeFilters(Value i)
if checkBloomFilters(i) == false then
    return 0
else if checkHighFreq(i) == false then
    return Avg
else
    return Counter(i)
end if
```

There remains one final issue to be resolved. We need to initialize parameters of the bloom filters and multistage filters appropriately. A bloom filter requires us to initialize a bit array to a certain size, which is given by the size of the set multiplied by the number of bits per tuple. Similarly, a multistage filter requires us to set an array size and a corresponding threshold, which is specified as a fraction of the size of the inner relation. Both these techniques require us to estimate the size of the inner relation. We describe our size estimation technique in the context of our procedure for producing a more optimal partition of the hashtable which we describe next.

2) Hashtable Partitioning: Hash Joins typically decide the number of buckets and partitions of the inner/build hash tables using the optimizer estimates multiplied by a small fudge factor. However, optimizer estimates may possibly be very inaccurate. In our experiments we found that non-optimal partitioning often leads to many tuples per bucket and significantly slows down hash join processing.

If the build input of the hash join is a relation scan with filter conditions, then we can obtain accurate estimates of the selectivities of the filters by streaming a random sample of the relation through the filters. As in the case of join pipelines, we estimate the size of the relation S passing through the selectivity predicate P using a random sample. If the $(n+1)$th tuple has the value i

$$D_{n+1} = \frac{D_n \cdot n + P(i)\,|S|}{n+1}$$

where $P(i)$ is 1 if i satisfies the predicate P, and 0 otherwise. Confidence bounds similar to those for join estimation can be derived as well. Using this online estimate, we can appropriately decide the partitioning mechanism for the build input, and improve the performance of the join algorithm. Additionally, this online estimation procedure also enables us to appropriately allocate space for the bloom filters and multistage filters. We note that if the inner input is the result of a join, we utilize online cardinality estimation procedures developed for progress estimation [19] in this setting.

F. General Pipelines

1) Other Join Algorithms: We have described techniques to perform all-paths simulation for join pipelines utilizing either existing summary structures (INLJs) or summary structures constructed at runtime (HJs). Other join algorithms fall into one or the other of these categories. For example, in a nested loops join (without indices), the entire inner input is read for each tuple (or block) of the outer. Therefore, we can create summary structures as in a pipeline of hash joins to enable all paths estimation. In a sort-merge join, the input is either sorted in advance or sorted during query processing. In the latter case, we can build summary structures during the sorting phase to enable estimation. If the input is already sorted, there is usually an index on the join column. This index can be used to do estimation as in pipelines of INLJs.

2) Multiple Join Conditions: We have described our estimation framework using single equijoin conditions at each node. We can also proceed similarly for multiple join conditions per node. In the case of pipelines of INLJs, the simulation procedure remains essentially unchanged since we have full access to tuples. For pipelines of Hash Joins, the sole change is that we now build multidimensional summary structures at a relation instead. These are built on all the join columns of the relation at that node. In the presence of space constraints, we can construct appropriate approximate histograms using techniques as described in the literature [20]. Similar extensions to the framework can also be defined for non-equijoin conditions as well.

3) Non-homogeneous pipelines: A natural question that arises in this context is the extension of join reordering techniques to non-homogeneous pipelines consisting of different kinds of join operators. The simulation techniques defined in the previous section extend to this setting as well, utilizing indexes when present, and constructing summary structures at runtime if required. However, join reordering in this setting also includes the related problem of operator selection, wherein we may also change the operator being used at some point in the pipeline. As a consequence, reordering no longer remains a low-cost operation since operators might be switched at runtime, and new indices and hashtables may need to be loaded into memory. A detailed treatment of this question is beyond the scope of this paper and is left for future work.

G. Dynamic Shortest Paths

The final step (Step 3) of our join reordering procedure requires us to compute the shortest paths on the join lattice. The edges of this join lattice are regularly updated with new cost estimates obtained from the updated cardinality estimates. We now describe our algorithm, which takes advantage of the lattice structure to maintain shortest paths inexpensively.

Suppose we ran a shortest path algorithm on the graph, and come up with a distances $d[u]$ for each vertex u. Now some changes happen in the graph (weights decrease or increase). Define $rhs[u]$ as

$$rhs[u] = min_{v \in pred(u)} d[v] + l(v, u)$$

where $l(v, u)$ is the (possibly updated) distance of the edge from v to u. We use the set $pred(u)$ to denote the set of vertices with edges incident on u. A vertex is *inconsistent* if $rhs[u] \neq d[u]$. A shortest path maintenance algorithm needs to define an order in which vertices are to be processed in order to remove inconsistencies. Since our graph is a lattice, we process nodes in order of their *level*, where the level of a node in the lattice is the number of edges separating the node from the bottom. In the example lattice from Figure 1, the bottom (A) is at level 0 while the top node ($ABCD$) is at level 3. At each level, the algorithm first identifies inconsistent nodes, and then corrects the shortest path to such nodes, propagating inconsistencies up to the next level. This ensures that shortest path computation takes linear time in the number of vertices. We recompute shortest paths at regular intervals in order to maintain the optimal alternative execution order for the current join pipeline.

H. Summary

To summarize, XS provides a framework for obtaining cardinality estimates along multiple alternative execution orders of a join pipeline by simulation. Our simulation techniques take the form of per-tuple cardinality estimation encapsulated in the notion of *tuple-contributions*. We abstractly describe our techniques in the form of operations on a join lattice, and then provide concrete instantiations for pipelines of INLJs and HJs. Our simulation procedures utilize a *point of symmetry* where reordering inner inputs in a lightweight fashion is feasible. In the context of INLJs, this point is when indices have been loaded and the outer relation is about to be read. In the context of HJs, this is the point where the inner relations have been built, and processing of the outer (probe) relation is to commence. At these points, we process a random sample of the outer relation and perform *all-paths simulation* in order to evaluate the alternative execution orders of the join pipeline. We note that this is a lightweight step, as demonstrated in our evaluation in the next section. Following the simulation step, the inner inputs are reordered if necessary, and the join is processed as in normal query execution.

IV. EVALUATION

In this section, we first describe our implementation and experimental setup for evaluating the performance of the XS

framework. We then describe an experimental evaluation of our techniques.

A. Implementation and Setup

We implemented the XS framework inside the Postgresql 8.0 database engine and added support for automatic pipeline detection, all paths simulation, and shortest path maintenance. We let our tables be prefixed with a small random sample of the whole table. The table scan operators return tuples from the random sample before reading in the rest of the table. The actual code for reordering was implemented by switching pointers to inner indices/hashtables and modifying join conditions appropriately. Since no tuples have actually been joined at the reordering point, no state is required to be maintained after reordering. For hash joins, we implemented both exact histograms and approximate summary structures for join simulation. Unless otherwise specified, our experiments were conducted using exact histograms.

In our experiments, we utilized two standard test databases. We generated a 2.5 GB database using the DMV data generator [6], [7]. The DMV dataset consists of 4 tables. The table sizes of *Owner (O)* and *Car (C)* are 2.5 million tuples; the size of *Demographics (D)* is approximately 3.6 million tuples; and the size of *Accidents (A)* is approximately 10.7 million tuples. The dataset was generated with the correlations flag set on which makes cardinality estimation highly challenging. We utilize this dataset for all our experiments except the ones in Section IV-B.5. For the experiments in Section IV-B.5, we generated a 10 GB TPCH database. This database was generated with Zipfian skew 1 using a publicly available tool [21]. The experimental evaluation was conducted on a lightly loaded machine running Suse Linux with 4 GB memory and 3.60GHz clock speed. All queries were run with a cold cache.

Before describing the methodology of our experiments, we introduce some notation for pipelines. We label a pipeline with the order of its inputs starting with the outermost input. Thus, the pipeline shown in Figure 1 (a) is labeled as $ABCD$, and the ones in Figures 1 (b) and (c) are labeled $ABDC$ and $ADBC$ respectively. In addition, we denote the tables *Owner*, *Car*, *Demographics* and *Accidents* by O, C, D and A respectively.

There are two primary goals of our experimental evaluation

- To show that the XS framework can detect a better alternative execution path for the pipeline if one exists.
- To show that if the current query execution plan is optimal, the XS framework detects its optimality without imposing a significant overhead on query execution.

We demonstrate this by running XS on plans generated by the Postgresql optimizer. Our experimental evaluation shows that the framework optimizes suboptimal query plans without imposing a significant overhead on queries executing with the optimal plan.

B. Experiments

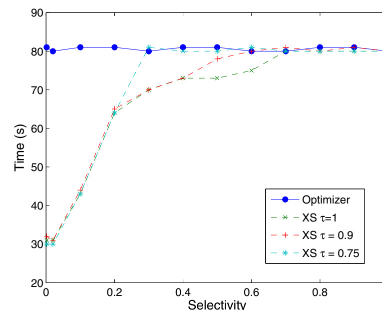


Fig. 4. 3 Way INL Joins

1) 3 way INL Join pipeline: Consider the query given below:

```
SELECT city, COUNT(*), avg(assets)
FROM owner O, car C, demographics D
WHERE C.ownerid = O.id AND D.ownerid = O.id
AND C.make = 'BMW' AND C.model = '318' AND D.salary > value
AND O.country1 = 'Germany' AND O.country3 = 'GM'
GROUP BY city ;
```

The query joins the 3 relations on the $ownerid$ attribute, and has additional selectivity conditions on the various relations. We vary the selectivity of the predicate on demographics ($salary > value$) by setting appropriate values. The selectivities of the other predicates are kept fixed.

Given this query, the Postgresql optimizer generates a pipeline with C as the outer relation, and index probes on O and D in order. This corresponds to the pipeline COD. The alternative plan has the order of index probes switched and is denoted as CDO. The optimizer selects plan COD irrespective of the selectivity of the predicate on D because it significantly underestimates the selectivities of the predicates on C and O due to independence assumptions between the correlated predicates `make = 'BMW'` and `model = '318'` and `country3 = 'GM'`.

The execution times for plan COD selected by the optimizer for the current query can be seen in Figure 4 as the line labeled *Optimizer*. The X axis is marked with the selectivity of the predicate on the *demographics* relation. All other predicates are kept fixed. The query is then executed with the XS framework enabled. We test the switching condition after a random sample of the outer relation containing at least 1% of the relation is read with an additional constraint that at least 200 tuples pass the filter condition on the outer relation.

The different graphs in Figure 4 marked as $XS\ \tau = val$ correspond to the XS framework executed with different values of the switching parameter τ. For the different settings of τ between 0.75 and 1, we see that XS switches to the alternative plan CDO at different selectivities. Recall that we switch to an alternative plan Alt if $C(Alt) < \tau \times C(Cur)$ where Cur is the current plan. With $\tau = 1$, the framework switches when the selectivity on *demographics* relation drops below

0.6. At this point, the estimated cost of plan COD, given the new statistics, is 307812 units of work (as defined by the optimizer cost function), while the cost of the alternative plan CDO is 307696 units. In this case, the benefit of switching is low since the current plan COD takes $80s$ to execute, while the alternative plan CDO takes $75s$ to execute. Thus, setting the threshold condition to $\tau = 1$ defines an aggressive switching policy that forces a change in the execution plan as soon as any potential benefit is detected. However, a policy like this may lead to an incorrect plan choice since optimizer cost functions are not perfect predictors of query execution times even with accurate statistics. At the other end, when we set $\tau = 0.75$, the framework switches from COD to CDO only when the selectivity on *demographics* drops below 0.2. At this point, the original plan COD takes $81s$ to execute while the alternative plan CDO takes $64s$ to execute. With this policy, switching takes place only when a significant reduction in query execution time is predicted with the alternative plan. However, a conservative policy like this may miss opportunities to reorder the pipeline to a better one. In practice, we take a middle path and use a policy between these extremes, setting $\tau = 0.9$ in our remaining experiments. This enables us to utilize opportunities for reordering, while acknowledging the uncertainty in optimizer cost estimates. Finally, we note that for the queries where XS did not reorder the join pipeline, the overheads of simulation are negligible. This is because our framework performs additional index probes for only a small random sample of the outer relation.

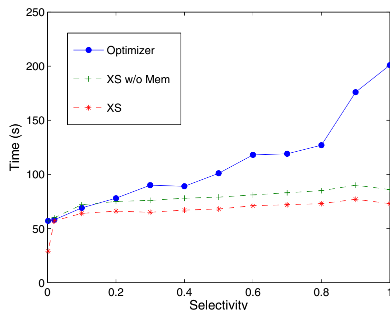


Fig. 5. 3 Way Hash Joins

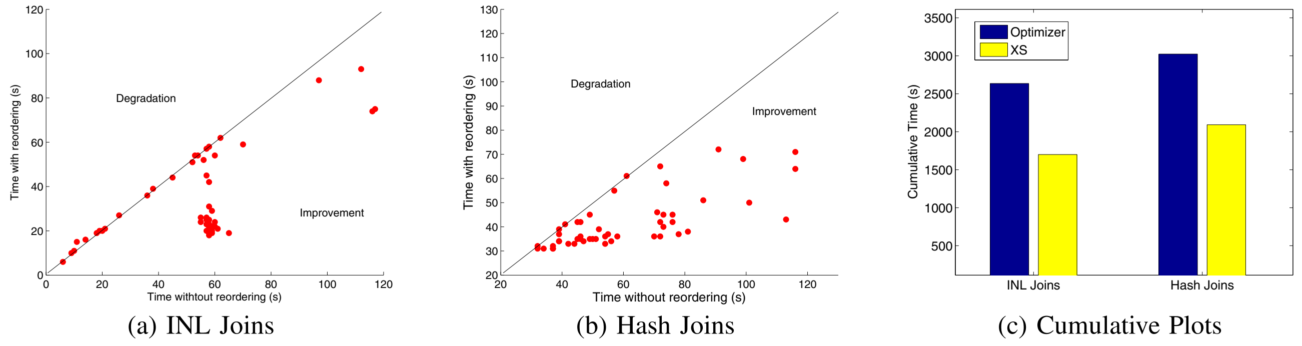
2) 3 way Hash Join Pipeline: Figure 5 describes the execution of a similar query on O, C and D using a hash join pipeline . We vary the selectivity of the predicate on D while keeping other predicates constant. Across the range of selectivities presented in the Figure 5, the optimizer always selects the plan ODC as the best pipelined hash join plan. This plot is marked as *Optimizer* in the figure.

The plot marked with *XS w/o Mem* corresponds to executing the query with the join reordering enabled, but adaptive hashtable partitioning disabled. For the range of selectivity above 0.2, we can see that XS switches to the alternative plan OCD which is significantly faster than the original query plan. The maximum improvement is at the right end of the graph for selectivity 1, where a query that originally took 201 s to execute now runs in 86 s.

The plot marked as *XS* shows the execution times when we enable both reordering and adaptive hashtable partitioning as described in Section III-E.2 illustrating the additional benefits of this technique. In addition to causing significant performance gains, hashtable partitioning also ensures that our cost estimates for the different alternative execution plans are accurate. As illustrated for Nested Loops joins in Section IV-B.1, the optimizer cost functions are not perfect predictors of the query execution time. For the particular case of hash joins, the cost functions used by Postgresql assume that there are around 10 tuples per bucket of the hashtable. If the hashtable is improperly partitioned, the number of tuples per bucket may be higher. In this case, the cost function estimates do not scale properly with the number of tuples per bucket, and the function becomes a poor predictor of the cost of the join. This dependence of the cost function on the partitioning scheme illustrates an important difference between cost function requirements for a traditional query optimizer, and an adaptive query processing engine. The optimizer accepts the cardinality estimates, designs a hashtable partitioning scheme based on these estimates, and then estimates the cost of the hash join given the hashtable partitioning scheme. The adaptive query processing engine needs to estimate the costs given the *pre-existing* partitioning scheme designed by the query optimizer using the incorrect statistics from the catalogs. Our adaptive hashtable partitioning technique allows the execution engine to break away from the constraints imposed by the optimizer, and do a better job of partitioning the data appropriately.

3) 3 Way Joins: Multiple Queries: We now test our framework on workloads of randomly generated queries. The queries were generated according to a fixed template with random values inserted to change the selectivities of the predicates on the different relations. For INL and Hash joins, we ran 50 queries which involve a join between O, C and D. Figures 6 (a) and 6 (b) show scatter plots of the execution times for the queries with and without reordering. The X-axis shows the time taken to execute the query on an unmodified Postgresql system, when executed with the *plan selected by the optimizer*. The Y-axis shows the time when the query is executed using the same plan, but with the XS framework enabled. The line between the axes partitions the space into queries whose performance has improved, and queries which have become slower. Figure 6(c) shows the cumulative execution times for the query workloads for INL and Hash Joins.

In the case of the workload for INL Joins, 30 of the 50 queries were reordered to an alternative execution plan, with a maximum reduction of 70%. Only one query was slowed down from 12 s to 16 s due to incorrect estimates from the cost function. The cumulative execution time for the workload dropped by 36%. In the case of hash joins, we measure the benefits of reordering with adaptive hashtable partitioning. 46 of the 50 queries had execution times reduced, with 38 queries having the execution order changed, and the maximum reduction in execution time being 62%.



(a) INL Joins (b) Hash Joins (c) Cumulative Plots

Fig. 6. Multiple Queries

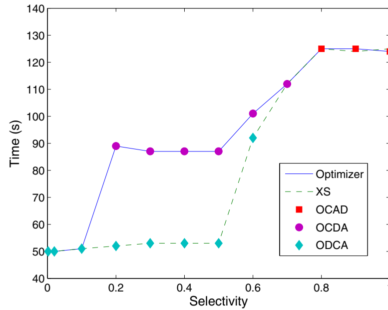


Fig. 7. 4 way INL Joins

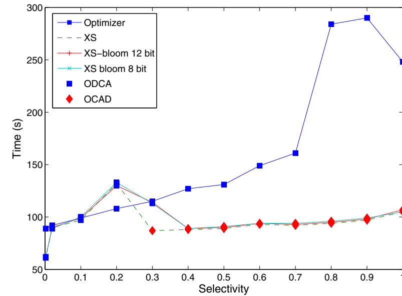


Fig. 8. 4 way Hash Joins

4) 4 way Joins: We tested our framework on 4 way join pipelines. The join condition was a join of O, C, and D on the *ownerid* attribute, and of C and A on the *carid* attribute. Note that a join with the A relation cannot take place until the relation C has been joined. This is a pipeline in which the outer input does not join each inner input. For both INL and Hash joins, we varied the selectivity of the predicate on D, while keeping the selection conditions on other relations fixed.

Figure 7 describes the results of our experiment with a 4 way INL join pipeline. The plot labeled *Optimizer* represents the execution time of the query using the best nested-loops plan selected by the optimizer. The optimizer selected the plan $OCAD$ for the range of selectivity ≥ 0.8, the plan $OCDA$ for selectivity between 0.2 and 0.8, and the plan $ODCA$ for selectivity less than 0.2. These are marked in Figure 7 using different markers for different plans. Since the outer relation remains constant over the range of selectivities, these are the 3 possible alternative plans.

When the queries are executed with the XS framework, using the plan selected by the optimizer the execution times can be seen in the dashed line in the figure marked *XS*. For selectivity less than 0.8, the original optimizer plan is reordered to the optimal $ODCA$ plan. This is because the predicate on D becomes sufficiently selective. Over the range of selectivities between 0.2 and 0.5, the query execution times are reduced by almost half.

Figure 8 describes the results of our experiments with 4 way hash join pipelines. Irrespective of the selectivity of the predicate on D, the optimizer generated the plan $ODCA$ for all the queries. When the queries were executed with reordering enabled, XS reordered the queries above Selectivity = 0.2 to the $OCAD$ plan. This is represented by the line marked as *XS* in the figure. The maximum reduction in execution time is at selectivity 0.9 where the optimizer plan takes 290 s to execute, while the reordered plan takes 97 s.

For selectivities at and below 0.2, XS does not reorder the plan selected by the optimizer. At selectivity 0.2, the optimizer plan $OCAD$ takes 108 s to execute while it takes 132 s if the XS framework is enabled even though it is not reordered. This is a rare case of memory management through adaptive hashtable partitioning causing more harm than good. On further investigation, we found that even though the estimates we obtain at runtime for hashtable partitioning are correct, the partitioning produced is sub-optimal due to the presence of additional unused memory in the system. Even though the optimizer originally had incorrect estimates, its hashtable partitioning was better since it had an incorrect estimate of the usable memory in the system as well.

Additionally, we also experimented with our approximate summary structures in this setting. Figure 8 shows the results of using our framework with approximate summary structures. We utilized bloom filters with 8 and 12 bits per tuple, and 3 hash functions. The threshold for the multistage filters was set as 0.1% of the size of the relation. The results are identical to the reordering algorithm using exact histograms except at one

point, where the selectivity is 0.3. Here the summary structures overestimate the cost of the alternative path, and therefore the algorithm does not reorder the query plan.

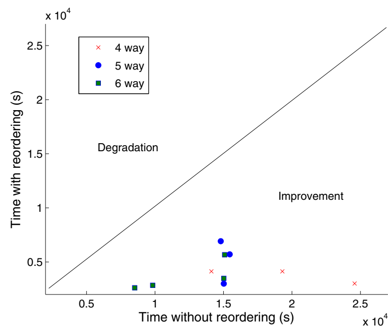


Fig. 9. Multiway INL Joins

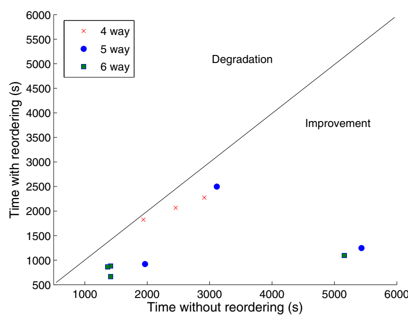


Fig. 10. Multiway Hash Joins

5) Multiway joins on a 10GB TPCH database: Finally, we present an evaluation of our techniques on a 10 GB TPCH database generated with Zipfian skew 1 on the selection columns. Figures 9 and 10 shows scatterplots displaying the improvements of our techniques relative to the optimizer for workloads of 10 queries containing 4 way, 5 way and 6 way join queries. The 4 way join queries were on the *Orders*, *Lineitem*, *Customer* and *Part* tables of the TPCH schema. The 5 way queries included the *Partsupp* table as well, and the 6 way queries additionally included the *Supplier* table.

Figure 9 plots the execution times with and without reordering for queries executed with an INL Join pipeline. The figure clearly shows the benefits of our technique, with all the queries showing significant improvements in execution time. The best improvement is shown by a 4 way join query that executes in 24571 s using the join pipeline selected by the query optimizer, and in 3007 s with XS enabled, demonstrating a reduction in query execution time by a factor of 8.

Similarly, Figure 10 displays the execution times with and without reordering for queries executed with a pipeline of hybrid hash joins. The original hash join pipeline was selected by the query optimizer. In this experiment, we utilized our approximate summary structures described in Section III-E.1. We set 12 bits per tuple for the bloom filter, 3 hash functions, and threshold of 0.1% of the relation size for the multistage filters. Our experiments demonstrate the utility of our technique for pipelined hash joins, with significant improvements for all queries. The best improvement is demonstrated by a 6 way join query whose execution time is reduced by a factor of almost 5, from 5157 s to 1096 s due to the XS framework.

V. Conclusions and Future Work

In this paper, we introduce the idea of *simulation* in the context of query execution, and demonstrate that simulation can be used to reorder join pipelines at runtime. We show that a local reoptimization technique like pipeline reordering can correct optimizer errors leading to significant reduction in query execution time. We envisage our local reoptimization technique working in tandem with a global reoptimization framework, with a decision making component which defines the risk-opportunity tradeoffs associated with each framework. Additionally, we intend to further explore techniques for reordering non-homogeneous join pipelines, including effects of operator switching at runtime.

References

[1] Y. E. Ioannidis, "The history of histograms (abridged)." *VLDB*, 2003.
[2] P. Haas and A. Swami, "Sequential Sampling Procedures for Query Size Estimation," *SIGMOD*, 1992.
[3] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil, "LEO - DB2's LEarning Optimizer," *VLDB*, 2001.
[4] A. Deshpande, Z. Ives, and V. Raman, "Adaptive query processing," *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 2007.
[5] N. Kabra and D. DeWitt, "Efficient Mid Query Reoptimization of Sub-Optimal Query Execution Plans," *SIGMOD*, 1998.
[6] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic, "Robust Query Processing Through Progressive Optimization," *SIGMOD*, 2004.
[7] S. Babu, P. Bizarro, and D. DeWitt, "Proactive Re-Optimization," *SIGMOD*, 2005.
[8] Q. Li, M. Shao, V. Markl, K. Beyer, L. Colby, and G. Lohman, "Adaptively reordering joins during query execution," *ICDE*, 2007.
[9] S. Babu and P. Bizarro, "Adaptive Query Processing in the Looking Glass," *CIDR*, 2005.
[10] G. Antoshenkov and M. Ziauddin, "Query Processing and Optimization in Oracle RDB," *VLDB Journal, 5(4)*, 1996.
[11] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," *SIGMOD*, 2000.
[12] S. Chaudhuri, V. Narassaya, and R. Ramamurthy, "A pay-as-you-go framework for query execution feedback," *VLDB*, 2008.
[13] J. S. Vitter, "Random sampling with a reservoir," *ACM Trans. Math. Softw.*, vol. 11, no. 1, pp. 37–57, 1985.
[14] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami, "Selectivity and cost estimation for joins based on random sampling." *J. Comput. Syst. Sci.*, vol. 52, no. 3, pp. 550–569, 1996.
[15] B. Bloom, "Space/time trade-offs in hash coding with allowable errors," *CACM*, vol. 13, no. 7, 1970.
[16] C. Estan and G. Varghese, "New directions in traffic measurement and accounting." *SIGCOMM*, 2002.
[17] G. S. Manku and R. Motwani, "Approximate frequency counts over data streams." *VLDB*, 2002.
[18] R. M. Karp, S. Shenker, and C. H. Papadimitriou, "A simple algorithm for finding frequent elements in streams and bags," *ACM Trans. Database Syst.*, vol. 28, pp. 51–55, 2003.
[19] C. Mishra and N. Koudas, "A lightweight online framework for query progress indicators," *ICDE*, 2007.
[20] S. Guha, P. Indyk, N. Koudas, and N.Thaper, "Dynamic Multidimensional Histograms," *SIGMOD*, 2002.
[21] S. Chaudhuri and V. Narasayya, "Program for TPC-D Data generation with skew," *ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew*.