Appears in Proc. of the Fifth Int'l Conf. on Information and Knowledge Management, 45-52, 1996.

Processing Queries for First-Few Answers

Roberto J. Bayardo Jr.
Department of Computer Sciences and Applied Research Laboratories
The University of Texas at Austin
bayardo@cs.utexas.edu
http://www.cs.utexas.edu/users/bayardo

Daniel P. Miranker
Department of Computer Sciences and Applied Research Laboratories
The University of Texas at Austin
miranker@cs.utexas.edu
http://www.cs.utexas.edu/users/miranker

Abstract

Special support for quickly finding the first-few answers of a query is already appearing in commercial database systems. This support is useful in active databases, when dealing with potentially unmanageable query results, and as a declarative alternative to navigational techniques. In this paper, we discuss query processing techniques for first-answer queries. We provide a method for predicting the cost of a first-answer query plan under an execution model that attempts to reduce wasted effort in join pipelining. We define new statistics necessary for accurate cost prediction, and discuss techniques for obtaining the statistics through traditional statistical measures (e.g. selectivity) and semantic data properties commonly specified through modern OODB and relational schemas. The proposed techniques also apply to all-answer query processing when optimizing for fast delivery of the initial query results.

1 Introduction

Traditional methods for query processing, primarily those based on the relational model, process queries with the goal of materializing the set of all answer tuples with minimal cost. Several applications instead require only the first answer or first-few answers to particular queries, or require the first answers of a query to be delivered as quickly as possible. This is evidenced by increasing support for first answer query optimization in modern relational systems [11, 16]. First-answer query support is also important in active databases based on production system models, where fast match algorithms lazily enumerate answers to a query one at a time [15]. Object-oriented database systems and knowledge-representation systems support complex structures allowing data to be retrieved through navigation as well as querying. Navigation is often preferable over querying for locating a single object since query engines, usually geared around set-oriented constructs, inevitably touch more data than necessary. A declarative query language with first-answer support can enable more understandable code than navigation, and potentially faster retrieval due to cost-based optimization. Finally, there will always be cases when producing the entire query result is simply too costly. Various search engines (including those for the world wide web) provide functionality for lazily enumerating answers in case of overly general search criteria. In this domain one might argue that all-answer query responses may take infinitely long or an input "table" may be a stream with no known end. Thus only depth first, first solution methods are applicable.

This paper presents our work on query processing techniques specifically geared for optimizing and executing first-answer join queries. The techniques also apply to optimizing all-answer queries when the goal is to minimize latency of first-answer delivery instead of overall throughput. The analysis is independent of any storage model, and therefore applies should the database be disk resident, main memory resident, or distributed.

We begin by providing a modified pipelined join algorithm that remedies performance problems sometimes exhibited by naive join pipelining. We then present a probabilistic technique for predicting query-plan cost under this modified pipelined join execution model. Though the cost-estimation technique requires database statistics not typically maintained by traditional centralized database systems, the statistics are derivable from those commonly maintained by distributed query processors. We also show how they can often be derived or estimated from selectivity information and semantic information often specified in the form of cardinality constraints (such as existence and functional dependencies) in modern relational, object-oriented, and knowledge-base systems.

2 Preliminaries

We make several assumptions in order to simplify the presentation and limit the scope of our investigation. We discuss consequences of loosening the assumptions when appropriate. First, we assume that queries are tree structured. Tree-structured queries are often regarded as the most common class of queries, and have several properties that make them easier to process [5, 14]. Next, we concentrate on the problem of optimizing and executing multi-join queries, and ignore the project and selection operators for the moment. We focus on joins because they are common and they are typically the most costly of the query operators. Lastly, we only consider the problem of determining a single answer to a particular query since the techniques are easily generalized to finding more answers.

A *join query* is a collection of *base relations* and *join predicates.* Base relations consist of distinct data elements called *tuples,* and join predicates involve two base relations and specify which pairs of tuples, one from each base relation, are compatible. A tuple in one relation is said to *join* with a tuple in another relation if the two tuples satisfy all join predicates involving the two relations. Tuples from relations that have no predicates between them are said to join trivially.

Answering a join query involves finding a set of tuples, one from each base relation, such that each tuple joins with all the others. A query *answer* is the tuple formed by combining these tuples into one. A *query graph* represents each base relation with a node, and each join predicate with an edge connecting the nodes corresponding to the predicate arguments. A query graph is *tree-structured* if it consists of a single connected component without any cycles. We will from here on assume that all query graphs are tree-structured. Without loss of generality, we further assume that each join predicate involves a distinct pair of relations. This implies that given n join predicates, there are exactly $n + 1$ relations mentioned in the query.

```
    PIPELINE-JOIN(R_1, R_2, ..., R_{n+1})
1   i = 1
2   C_i = {t | t ∈ R_i, ∀j < i, t ⋈ t_j}
3   if C_i non-empty then
4     Remove a tuple t from C_i and let t_i = t.
5     if i = n + 1 then return (t_1, t_2, ..., t_{n+1})
6     i = i + 1
7     goto 2
8   else
9     if i = 1 then return ∅
10    i = i – 1
11    goto 3
```

FIGURE 1. Pseudo-code for join pipelining.

Pipelining of multiple joins is an effective technique for executing first-answer queries since it can avoid touching large portions of the base relations. The *pipelined join algorithm* (Figure 1) requires a *join order* of the $n + 1$ relations be specified. According to usual heuristics, we assume the order is selected so that cross products are avoided. Since we assume the query graph is tree-structured, this implies each relation (other than the first) is preceded in the ordering by exactly one relation that is connected to it in the query graph. The pipelined join algorithm advances and retreats along the join order, filling the pipe with joining tuples until either an answer is found or none are determined to exist. The algorithm can be trivially extended to return all answers instead of a first answer by simply continuing pipelining instead of halting after an answer is produced.

An important goal of query optimization is to find a join order likely to provide good performance. A query plan thereby specifies the particular ordering used by the pipelined join algorithm. More generally, a query plan would also specify the indices and algorithms used to identify the set of joining tuples at each stage of the pipe.

3 Finding the First Answer by Pipelined Joins

When providing a single answer to a query, DB2 and Oracle exploit pipelining to its fullest extent by avoiding any access plan requiring a sort [11, 16]. Through pipelining, the execution engine can usually avoid touching large segments of each relation when only one answer is needed, particularly when relations are indexed on their join columns. Without pipelining, intermediate results need to be fully materialized, which requires touching most if not all of the relation contents.

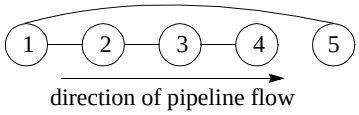

FIGURE 2. An ordered query graph.

There is one significant problem with naive join pipelining which can easily lead to poor performance. Suppose we are given the query whose query graph appears ordered along a pipeline in Figure 2. Further, suppose the pipelined join algorithm reaches the final stage in the pipe, only to find that there are no tuples that join with the previous tuple set ($C_5 = \emptyset$). In this case, the algorithm will back up to the previous stage. However, the previous stage had nothing to do with the failure at the final stage, since the only predicate incident upon the final relation connects it to the first relation in the pipe. The result is the algorithm performs unnecessary work before eventually backing up to the first relation where the situation can potentially be remedied. The amount of unnecessary work performed is exponential in the number of stages between the failure stage and the stage at which the failure can be remedied, with the exponent equal to the number of joining tuples identified at each stage. The common "star query" is particularly prone to this pathology since its query graph consists of a center node connecting to several nodes of degree one.

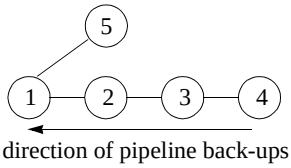

FIGURE 3. Partial order used for determining back-up destinations.

A solution to the problem is relatively straightforward: instead of backing up according to the join order, back up according to the *rooted tree* implied by the choice of initial relation and the structure of the query graph (Figure 3). Given a dead end in the pipeline, we thereby immediately back up to the stage that is filled with the tuple responsible for the failure. Whenever the pipeline backs up to a previous stage r that has more than one child, not only do we

select a new tuple to fill stage r, but we must also un-fill every stage that is a descendent of r in the rooted tree. For example, when backing up from stage 5 in Figure 3, we must un-fill the contents of stages 2 through 4. The algorithm still advances along the pipeline according to the join ordering. We call this variant of pipelined join *rooted-tree pipelining*. Pseudo-code for the algorithm appears in Figure 4.

```
    ROOTED-TREE-PIPELINE-JOIN(R_1, R_2, ..., R_{n+1})
1   i = 1, ∀j where 1 ≤ j ≤ n + 1, let t_j = ∅
2   C_i = {t | t ∈ R_i, ∀j < i, t ⋈ t_j}
3   if C_i non-empty then
4     Remove a tuple t from C_i and let t_i = t
5     if i = n + 1 then return (t_1, t_2, ..., t_{n+1})
6     Advance i to the next stage j where t_j = ∅
7     goto 2
8   else
9     if i = 1 then return ∅
10    i = parent_of(i)
11    ∀j where j is a descendent of i, let t_j = ∅
12    goto 3
```

FIGURE 4. Pseudo-code for rooted-tree pipelining.

There are several other known pipeline optimizations that can be used to further improve performance. For example, we could mark as "nogood" any tuple encountered in some stage of the pipe that fails to join with any tuples in some relation. This way, should the tuple be encountered again, it can be immediately skipped since it is certain to lead to the same failure. This and related "marking" optimizations have been proven useful in AI and deductive database domains [2, 6]. If the relations involved in a first-answer query are large, then it appears unlikely that the same tuple will arise in the pipe more than once. The utility of tuple markers in this environment is therefore unclear, so we leave their consideration in the context of first-answer query optimization open to future work.

4 Determining Query Plan Cost

Query optimization requires we have a method for determining the cost of a particular query plan given the available set of physical query operators. This section describes how to predict first-answer query plan cost under a rooted-tree pipelined join execution model. The primary operation of rooted-tree pipelining is the *tuple lookup* operator that identifies a set of joining tuples C_i at each pipeline stage (line 2 of Figure 4). Our measure of query plan cost is the net cost of tuple lookup since the other algorithm steps are negligible in comparison.

We begin by defining the database statistics required by our cost estimation procedure, and then show how they are used in determining the expected number of applications of the tuple lookup operator along any predicate within a given query. The presentation makes extensive use of the query plan illustrated in Figure 5. Relations (nodes) are ordered according to the join order, and predicates (edges) are ordered according to the order in which they are considered in the pipeline. This implies the second relation linked by a predicate numbered j always appears at stage $j + 1$ in the pipe.

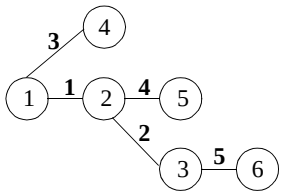

FIGURE 5. An example query plan.

A tuple lookup is said to be *attempted along a predicate* j when the pipeline reaches the stage corresponding to the second relation linked by j (stage $j + 1$) and joining tuples are identified. We say that a tuple lookup attempt along a predicate j *eventually fails* if it has to be repeated in order to produce an answer to the query.

4.1 Necessary Statistics

Query optimizers make use of statistics that summarize the contents of the database in order to make cost estimation possible. Here we define the statistical parameters necessary for accurately predicting the cost of a first-answer query plan. Section 5 provides details on obtaining these statistics and describes how they relate to statistical measurements appearing elsewhere in the query optimization literature.

When every tuple joins with at least one tuple from another relation, the tuple lookup operator is applied exactly once for each of n joins, and then an answer is produced. Unfortunately, since joins are often *lossy*, we need to know the probability that the tuple lookup will fail to identify any joining tuples in order to accurately determine the expected number of times it is applied. We assume that for each join predicate j, we know the probability p_j that a tuple in the first relation fails to join with any tuples in the second relation. The "order" implied by our use of "first" and "second" depends on the join order. The *first* relation linked by a join predicate is the parent of the *second* relation in the rooted tree. Note that p_j, like our definitions of first and second, is join-order dependent.

Another statistic required for our cost analysis is the expected number of tuples that join with a given tuple along some predicate j. Let b_j be defined as follows: if a tuple t in the first relation linked by a predicate j joins with at least one tuple in the second relation, then on average it will join with b_j tuples.

4.2 Notation

Before delving into the details of determining query plan cost, we review the notation defined thus far and present additional notation necessary for simplifying the upcoming presentation. As done in Figure 5, predicates will be identi-

fied according to the order they are considered in the pipeline.

1. n denotes the number of predicates in the query (implying the number of relations is $n + 1$).
2. p_j denotes the probability a tuple lookup along predicate j fails to produce any joining tuples.
3. b_j denotes the mean number of tuples returned should a lookup along predicate j be successful.

We will also make extensive use of the following additional notation:

4. $c_j(1), c_j(2), \ldots, c_j(m)$ denote the *child* predicates of predicate j in the rooted tree. Child predicates are those other predicates connecting to the second relation linked by predicate j. For instance, in Figure 5, the child predicates of predicate 1 are predicates 2 and 4.
5. $a_j(1), a_j(2), \ldots, a_j(m)$ denote those predicates whose first relation only is an ancestor of predicate j in the rooted tree. We call these the *back-up* predicates of predicate j. For instance, the back-up predicates of predicate 4 in Figure 5 are predicates 2, 3, and 4. Note that a predicate is always a back-up predicate of itself.

We also make use of two probabilistic functions. Given a set of i independent events that happen with probability $p_1, p_2, \ldots, p_i$ respectively, the probability that one or more events happen will be denoted with $\text{Any}(p_1, p_2, \ldots, p_i)$, the value of which is $1 - (1 - p_1)(1 - p_2)\ldots(1 - p_i)$ [4]. Given a task that fails with probability p_f, we denote the average number of attempts required for success at the task with $E(p_f)$, the value of which is $1/(1 - p_f)$.

4.3 Probability of Tuple Lookup Failure

If we can compute the probability T_j that a tuple lookup along predicate j eventually fails, then, if the query has an answer, the expected number of tuple lookups along predicate j is $E(T_j)$. This section discusses how to compute T_j for each predicate in the query, culminating in a procedure for estimating the cost of a first-answer query.

Once the pipeline reaches stage $j + 1$, the tuple lookup along predicate j eventually fails if the join algorithm is forced to back up to any stage that is an ancestor of predicate j in the rooted tree. There are multiple scenarios which could lead to such a back-up, the most trivial of which is failure of the tuple lookup attempt along predicate j to identify any joining tuples in R_{j+1}.

The first back-up to an ancestor of predicate j after reaching stage $j + 1$ (should one occur) must take place along some back-up predicate $a_j(1), a_j(2), \ldots, a_j(m)$. Let $P_i(j)$ denote the probability that a subtree rooted at predicate i, when considered independently from other subtrees in the query plan, does not lead to an answer after the pipeline has been advanced to stage $j + 1$. We then have that

$$T_j = \text{Any}(P_{a_j(1)}(j), P_{a_j(2)}(j), \ldots, P_{a_j(m)}(j)).$$

We first describe how to calculate the probability $P_i(j)$ when $i \geq j$. For this case, once pipeline flow reaches stage $j + 1$, by the structure of the ordering and the rooted tree arrangement, we know the pipeline portion corresponding to the subtree rooted at predicate i is completely unfilled. This is because any descendent of predicate i must be numbered higher than i.

The subtree rooted at predicate i will not lead to a answer if the lookup along predicate i fails. Also, the subtree will not lead to a answer if the lookup along predicate i succeeds in identifying b_i tuples, but then we encounter failure at any subtree rooted beneath predicate i for each of the b_i tuples. Therefore, when $i \geq j$,

$$P_i(j) = \text{Any}(p_i, \text{Any}(P_{c_i(1)}(j), \ldots, P_{c_i(m)}(j))^{b_i}).$$

If the predicate has no children, then $P_i(j) = p_i$.

Let us illustrate the idea through a concrete example. From Figure 5, suppose we are trying to determine $P_1(1)$ -- the probability the subtree rooted at predicate 1 leads to an answer after the pipeline advances to the second relation linked by predicate 1. Should the lookup along predicate 1 fail (which will happen with probability p_1), a back-up along predicate 1 takes place and an answer to the subtree is not produced. Now suppose the lookup along predicate 1 succeeds and identifies exactly b_1 tuples. In this case, an answer to the subtree will not be produced if the predicate lookups along predicate 2 or 4 eventually fail for every tuple identified. We thus have the following:

$$P_1(1) = \text{Any}(p_1, (\text{Any}(P_2(1), P_4(1)))^{b_1}).$$

Now consider computing $P_i(j)$ when $i < j$. The situation is complicated by the fact that some stages of the subtree rooted by predicate i are initially filled after the pipeline has advanced to stage $j + 1$. For the first tuple identified by the lookup along predicate i, a subtree rooted at a child c of i fails to lead to an answer with probability $P_c(j)$. If we need to consider the other tuples identified by the lookup along predicate i, then a back-up took place to stage $i + 1$, un-filling the contents of all its descendents. In this case, the subtree rooted at a child c of i is completely empty, so it fails to lead to an answer with probability $P_c(i)$. Setting the parameter to i instead of j prevents the calculation of failure from assuming any initially filled stages. We can now conclude that when $i < j$,

$$P_i(j) = \text{Any}(P_{c_i(1)}(j), \ldots, P_{c_i(m)}(j)) \times (\text{Any}(P_{c_i(1)}(i), \ldots, P_{c_i(m)}(i)))^{b_i - 1}.$$

If there are no child predicates of predicate i, then $P_i(j) = 0$.

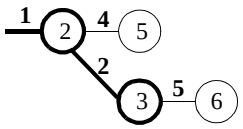

FIGURE 6. State of the subtree rooted at predicate 1 when the pipeline reaches stage 4. Filled stages correspond to the highlighted segments.

From Figure 5, suppose we are calculating $P_1(3)$ -- the probability that the subtree rooted at predicate 1 fails to lead to an answer after the pipeline reaches stage 4 (the second stage linked by predicate 3). For this case, when the pipeline

reaches stage 4, we have successful lookups along predicates 1 and 2 within the subtree rooted at predicate 1 (as illustrated in Figure 6). Therefore,
$P_1(3) = \text{Any}(P_2(3), P_4(3)) \times (\text{Any}(P_2(1), P_4(1)))^{b_1 - 1}$.

4.4 Query Plan Cost

Given the above method for computing the probability T_j that a tuple lookup along predicate j eventually fails, query plan cost can be determined by calculating the expected number of tuple lookups using $E(T_j)$, weighting each such value by the cost w_j of each tuple lookup along the predicate, and summing up the values:

$$\text{Cost(Query Plan)} = \sum_{i=1}^{n} \frac{w_i}{1 - T_i}$$

Cost of tuple lookup along a particular predicate can depend on multiple factors including sizes of the involved relations, the access paths available, clustering, and implementation details of the algorithms used to perform tuple lookup. Like p_i and b_i, the value of w_i is dependent on the direction of tuple lookup.

4.5 Summary

- Cost(Query Plan) $= \sum_{i=1}^{n} \frac{w_i}{1 - T_i}$ where w_i denotes the tuple lookup cost along predicate i.
- T_j denotes the probability that a lookup along predicate j eventually fails, and
  $T_j = \text{Any}(P_{a_j(1)}(j), P_{a_j(2)}(j), \ldots, P_{a_j(m)}(j))$.
- $P_i(j)$ denotes the probability that a back-up takes place along predicate i after the pipeline has reached stage $j+1$, assuming the subtree rooted at predicate i is considered in isolation from the other subtrees in the query.
  When $i \geq j$, if i connects to a leaf node, then $P_i(j) = p_i$, otherwise
  $P_i(j) = \text{Any}(p_i, \text{Any}(P_{c_i(1)}(j), \ldots, P_{c_i(m)}(j))^{b_i})$.
  When $i < j$, if i connects to a leaf node, then $P_i(j) = 0$, otherwise
  $P_i(j) = \text{Any}(P_{c_i(1)}(j), \ldots, P_{c_i(m)}(j)) \times (\text{Any}(P_{c_i(1)}(i), \ldots, P_{c_i(m)}(i)))^{b_i - 1}$.
- Definitions of c_i, a_i, p_i and b_i appear in Section 4.2.

5 Statistical Issues

The above-described method for determining the cost of first-answer query plans makes use of the statistical parameters b_j and p_j which are not explicitly maintained by existing database systems. Current database systems typically provide *join selectivity*, defined as the expected fraction of tuples in one relation which join with a tuple in another relation. Consider the join selectivity s_j of a predicate j linking relations R_k and R_{j+1}:

$$s_j = \frac{|R_k \bowtie R_{j+1}|}{|R_k||R_{j+1}|} = \frac{b_j(1 - p_j)}{|R_{j+1}|}.$$

Unfortunately, it is not possible to compute b_j and p_j from selectivity and relation cardinality information alone. Some database systems also maintain statistics on the most and least frequently occurring values in a column, or histograms of column value distributions. It is possible that histograms may be useful in estimating p_j and b_j.

Interestingly, while these statistics are not derivable from those typically maintained by centralized databases, they are derivable from those usually required for cost estimation in a distributed environment. For instance, a statistic necessary for estimating the result of a semi-join is the semi-join selectivity of a predicate, which can be defined as the fraction of tuples remaining in the relation following a semi-join. Semi-join selectivity is also sometimes used in detailed cost modeling of join algorithms [3]. Supposing a predicate j links relation R_k with R_{j+1}, the semi-join selectivity z_j of predicate j is defined as:

$$z_j = \frac{|R_k \ltimes R_{j+1}|}{|R_k|}.$$

Note then, that $p_j = 1 - z_j$, and $b_j = (s_j|R_{j+1}|)/z_j$.

Semantic schema information provides an alternative method for obtaining estimates of these statistics. Knowledge bases, object-oriented databases, and modern relational bases are often augmented with semantic data. In particular, we are interested in *cardinality constraints* -- those that impose restrictions on the number of links a tuple in one relation can have to tuples in another. A common type of cardinality constraint already supported by many relational systems is the existence constraint. An existence constraint can specify that a particular column value must correspond to some column value in another relation. Cardinality constraints are also often specified in knowledge-representation languages [8] and object-oriented databases [1].

An existence constraint between two relations implies that the join predicate between them has a p_j of 0 in the direction of the constraint. If a cardinality constraint is given as a range [0-m] of tuples that may be linked by a predicate, then p_j can be estimated by assuming a uniform (or other) distribution. For instance, given that a tuple joins with [0-3] tuples in another relation, the join predicate has a p_j of .25 if we assume a uniform distribution. The join selectivity of predicate j can then be used to determine b_j given p_j.

6 Experiments

We performed experiments for the purpose of demonstrating the accuracy of the cost-estimation formula, the effects of rooted-tree versus naive pipelining, and the expected effects of first-answer query optimization on several classes

of queries. For the evaluation, queries were generated by randomly creating a tree-structured query graph with a specified number of joins, and randomly generating two pairs (p_j, b_j) for each predicate. Two pairs are required for each predicate since the values differ depending on the direction of tuple lookup. To model real-world queries which frequently contain lossless joins and joins on key columns, we selected the values according to the following 4 distributions. A distribution was chosen with equal probability each time a value pair was generated.

1) $p_j = 0$ and $b_j = 1$
2) $p_j = 0$ and $b_j = [1-5]$
3) $p_j = p$ and $b_j = [1-5]$
4) $p_j = p$ and $b_j = 1$

Values in the specified ranges were selected according to a random uniform distribution. The parameter p for distributions 3 and 4 was dependent on the experiment. We assume a fixed unit tuple lookup cost for each predicate in order to better isolate the effects of varying p and the number of joins n. Allowing tuple lookup cost to vary typically results in wider distributions of query plan costs given different join orders, so our numbers can be considered conservative.

6.1 Accuracy of Cost Estimation

In order to verify the accuracy of the cost-estimation procedure, we implemented a simulator which would accept a query plan and simulate execution of rooted-tree pipelining on the query plan. Given a query, we compared the estimated cost produced by the optimizer to the "actual" cost produced by the execution simulator.

For the first experiment, we had the simulator identify exactly b_j tuples with each successful tuple lookup along predicate j. Queries were generated according to the above-described method with values $n = 7$ and $p = .25$. Expected error, averaging across 1, 10, and 100 query runs appear in Figure 7. Expected error for a single query run is 41%, though the error rapidly diminishes as we average across more and more runs of the query (which could correspond to either the user running the query more than once, or the user requiring more than a single query answer). We verified that expected error approaches zero when averaging over enough runs for several values of n and p. Averaging over 1000 runs, the error was always within 1%. Though this does not constitute a formal proof, it does provide reasonable evidence that the cost estimation procedure is correct.

In real-world data, we expect b_j would reflect the mean number of tuples identified with a successful tuple lookup, rather than the exact number. We therefore repeated the experiment after modifying the simulator to identify a number of tuples selected from a distribution whose mean was b_j. For a particular b_j, we used a uniform random distribution within the range $[1 - b_j \times 2]$. This data skew did not appreciably affect the outcome (Figure 8).

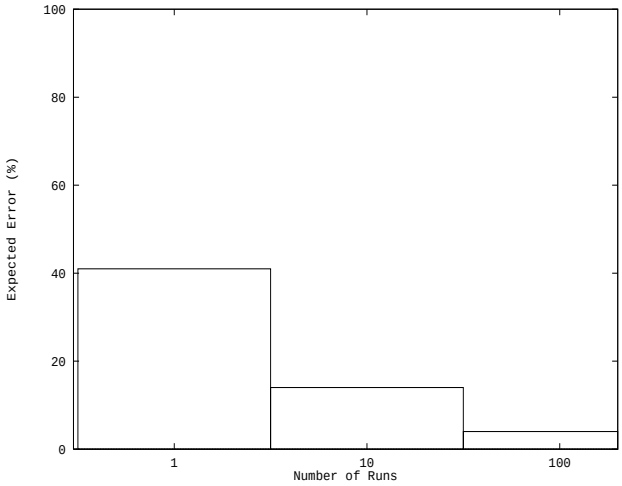

FIGURE 7. Expected Error, no data skew, $n = 7$, $p = .25$.

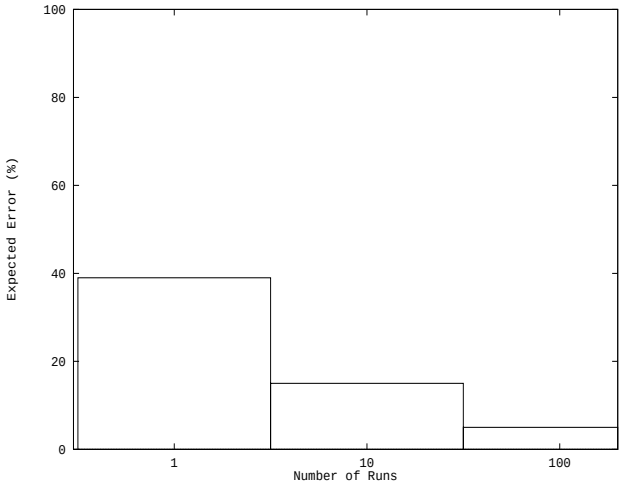

FIGURE 8. Expected error, skewed data, $n = 7$, $p = .25$.

Unlike cost-estimation routines for all-answer queries [12], we found the accuracy of the cost estimates do not degrade as the number of joins increases. Though error can be non-negligible for single-run, single-answer queries due to random variance, in general the lowest-cost plan has a higher probability of performing well than a plan with higher estimated cost.

6.2 Naive vs. Rooted-Tree Pipelining

A simulator of naive pipelining was also implemented for comparison with rooted-tree pipelining. We generated 100 queries at several values of p and n, and randomly generated 1000 plans for each query. We selected the best plan from this set according to our estimation procedure, and simulated the execution of both rooted-tree and naive pipelining on these plans. Even though our cost-estimation procedure is intended for use with rooted-tree pipelining, we found it to accurately rank plans according to performance of naive pipelining as well. We did not exhaustively search the space of all query plans because of computational limits

at larger values of n. The plan generation scheme we used corresponds to that in [10].

The results of the experiment are plotted in the following figures. First we fixed n at 7 and varied p (Figure 9). Then we fixed p at .25 and varied n (Figure 10). Each point represents the mean of the costs computed for each of the 100 queries.

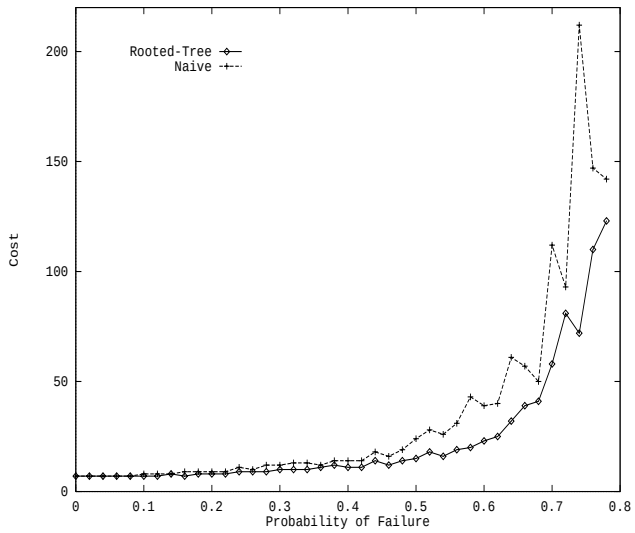

FIGURE 9. Rooted-tree vs. naive pipelining, $n = 7$.

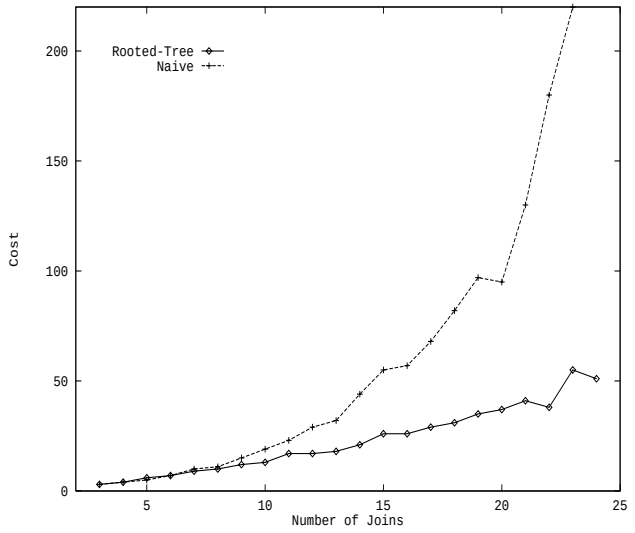

FIGURE 10. Rooted-tree vs. naive pipelining, $p = .25$.

The plot for naive pipelining at high values of p is noisy even though each point is the average of over 100 queries. The cause is that naive pipelining occasionally performs extremely poorly (orders of magnitude worse than the median) due to the problem described in Section 3. These cases arise with higher probability at larger values of p and n, hence the increase in noise at those values. Rooted-tree pipelining, on the other hand, exhibits relatively noise-free performance. It also performs better, particularly at larger values of p and n.

6.3 Benefits of First-Answer Query Optimization

Here we show the effects of first-answer query optimization across a wide variety of queries. As done in Section 6.2, a point n, p was chosen,100 queries generated per point, and 1000 plans per query. Figure 11 and Figure 12 plot the expected mean query plan cost, expected minimum query plan cost, and expected maximum query plan cost for a particular point n, p. The first figure scales p, and the second scales n.

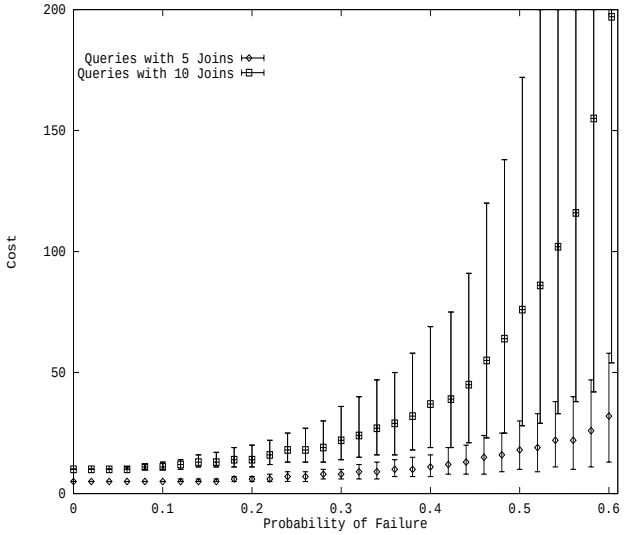

FIGURE 11. Expected mean, minimum, and maximum query plan costs when varying p.

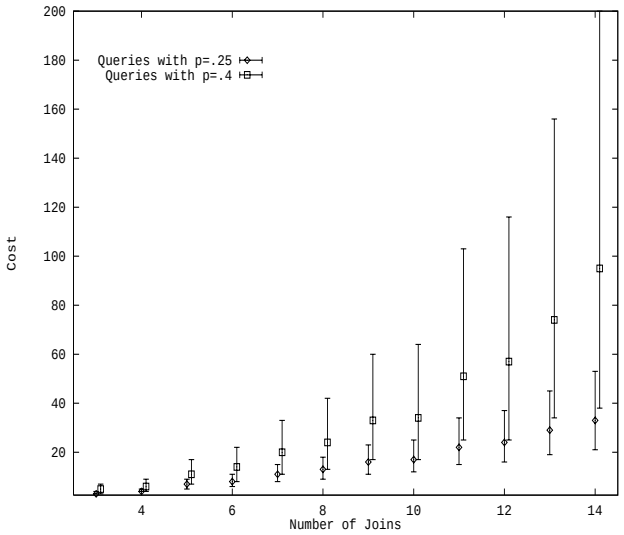

FIGURE 12. Expected mean, minimum, and maximum query plan costs when varying n.

The bottom end of each error bar represents the expected cost of the best plan found, and thereby represents the average cost of a plan selected by an optimizer employing our technique. If we were to select a plan at random, then the expected cost is designated by the tic at the center of each bar. The highest-most point at each bar represents how badly we may do when selecting an arbitrary plan. In gen-

eral, the mean speedup of the best plans over the average or worst plans increases rapidly as both n and p are scaled upwards, though there is appreciable speedup across all values examined with the exception of the smallest values of n and p.

7 Conclusions and Future Work

The need for first-answer query processing is likely to grow as data sets become larger since query result size may grow proportionally. This paper has addressed the specific needs of optimizing and executing multi-join first-answer queries. Our technique, like existing commercial implementations, exploits pipelining as much as possible in order to avoid touching large segments of most relations. Since naive pipelining is prone to pathological behavior, we suggested an alternative pipelining algorithm that backs up according to a rooted-tree arrangement of the relations implied by the query graph. Under this execution model, we then presented techniques for predicting cost through probabilities of tuple-lookup failure. Finally, we discussed methods for obtaining the required set of database statistics from traditional centralized database statistics, distributed database statistics, and semantic constraints specified as part of the database schema. Ideally, we believe a system providing thorough support for first-answer query processing should strive to maintain the specific statistics we have described.

This work remains to be extended to cyclic queries, though we have ideas on how this might be accomplished. In the cyclic case, query graphs can still be arranged into rooted trees for rooted-tree pipelining [9]. The probabilistic analysis will have to be extended to consider that tuple-lookups along more than one predicate may be required at any one stage in the pipe. The work could also be extended to consider cases where non-pipelined methods may be appropriate. For instance, two small relations may be better off joined greedily instead of lazily through pipelining.

We have avoided issues involving selection and projection, though there are trivial methods for dealing with these operators for first-answer query processing which seem reasonable. It is unlikely that the heuristic of "pushing selections/projections down as far as possible" will be beneficial when finding a first answer, since executing the initial selections or projections would likely require touching large portions of the involved relations. A more attractive alternative is to perform the selections and projections on the fly during pipelining. While some selections may be redundant using this approach, we suspect the amount of redundancy will be small compared to the cost of performing all selections before the joins.

On a related note, while we have modeled the cost of a pipelined join algorithm that identifies a set of joining tuples at each stage, it may be beneficial in some circumstances to instead lazily fetch joining tuples one-at-a-time on an as-needed basis. For instance, consider the case when an index is available on the join column of a relation, but the tuples are not clustered on the join attribute. The cost of fetching the set of joining tuples is proportional to the number of tuples identified. If only a few of the entire set are required in order to produce an answer, work will be wasted by identifying them all.

8 References

1. A. Analdo, G. Ghelli, and R. Orsini, A relationship mechanism for strongly typed Object-Oriented database programming languages. In *Proc. of the 17th Int. Conf. on Very Large Data Bases*, 565-575, 1991.
2. R. J. Bayardo Jr. and D. P. Miranker, An optimal backtrack algorithm for tree-structured constraint satisfaction problems**.** *Artificial Intelligence***,** 71(1), 159-181, 1994.
3. J. A. Blakeley and N. L. Martin, Join index, materialized view, and hybrid-hash join: A performance analysis. In *Proc. of the Sixth Int'l Conf. on Data Engineering*, 256-263, 1990.
4. G. Blom, *Probability and Statistics -- Theory and Applications*, Springer-Verlag, New York, N.Y, 1989.
5. C. Beeri, R. Fagin, D. Maier and M. Yannakakis, On the desirability of acyclic database schemes. *Journal of the Association for Computing Machinery*, 30(3), 479-512, 1983.
6. D. Chimenti, R. Gamboa, and R. Krishnamurthy, Abstract machine for LDL. In *Lecture Notes in Computer Science* v416 (Proceedings EDBT-90), 153-168, Springer, Verlag, New York, N.Y., 1990.
7. D. Calvanese and M. Lenzerini, On the interaction between ISA and cardinality constraints. In *Proc. of the Tenth Intl. Conf. on Data Engineering*, 204-213, 1994.
8. R. Fikes and T. Kehler, The role of frame-based representation in reasoning. *Communications of the ACM*, 28(9), 904-920, 1991.
9. E. C. Freuder, and M. J. Quinn, Taking advantage of stable sets of variables in constraint satisfaction problems. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, 1076-1078, 1985.
10. Galindo-Legaria, C., Pellenkoft, A., and Kersten, M. Fast, randomized join-order selection--why use transformations? In *Proc. of the 20th Int'l Conf. on Very Large Data Bases*, Santiago, Chile, 1994.
11. P. Gassner, G. M. Lohman, K. B. Schiefer, and Y. Wang, Query optimization in the IBM DB2 family. In *Bulletin of the Technical Committee on Data Engineering*, 16(4), 14-18, 1993.
12. Y. E. Ioannidis and S. Christodoulakis, On the propagation of errors in the size of join results. In *Proc. of the 1991 ACM-SIGMOD Conference*, 268-277, 1991
13. Y. E. Ioannidis and Y. C. Kang, Randomized algorithms for optimizing large join queries. In *Proc. of the 1990 ACM-SIGMOD Int'l Conf. on Management of Data*, 312-321, 1990.
14. R. Krishnamurthy, H. Boral, and C. Zaniolo, Optimization of nonrecursive queries. In *Proc. of the Twelfth International Conference on Very Large Data Bases*, Kyoto, Japan, 128-137, 1986.
15. D. P. Miranker and D. A. Brant, An algorithmic basis for integrating production systems and large databases. In *Proc. of the Sixth Int. Conf. on Data Engineering*, 353-360, 1990.
16. Oracle Corp., *Oracle7 Server Application Developer's Guide*. 1991.
17. P. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, Access path selection in a relational database management system. In *Proc. of the 1979 ACM-SIGMOD Conference*, 23-34, 1979.