Least Expected Cost Query Optimization: An Exercise in Utility

Francis Chu* Joseph Y. Halpern* Praveen Seshadri†
Department of Computer Science
Upson Hall, Cornell University
Ithaca, NY 14853-7501, USA

{fcc,halpern,praveen}@cs.cornell.edu

Abstract

We identify two unreasonable, though standard, assumptions made by database query optimizers that can adversely affect the quality of the chosen evaluation plans. One assumption is that it is enough to optimize for the *expected* case—that is, the case where various parameters (like available memory) take on their expected value. The other assumption is that the parameters are constant throughout the execution of the query. We present an algorithm based on the "System R"-style query optimization algorithm that does not rely on these assumptions. The algorithm we present chooses the plan of the *least expected cost* instead of the plan of least cost given some fixed value of the parameters. In execution environments that exhibit a high degree of variability, our techniques should result in better performance.

1 Introduction

A database query is specified declaratively, not procedurally. Given a query, it is the job of the DBMS to choose an appropriate evaluation plan for it. This task is performed by a cost-based query optimizer. In theory, the task of the optimizer is simple: it performs a search among a large space of equivalent plans for the query, estimates the cost of each plan, and returns the plan of least cost.

In practice, things are not so simple. There are two major problems: (1) there are far too many possible plans for an optimizer to consider them all, and (2) accurate cost estimation is virtually impossible, since it requires detailed *a priori* knowledge of the nature of the data and the run-time environment. Because query optimization is such a critical component of a database system—queries are typically optimized once and then evaluated repeatedly, often over many months or years—much effort has gone into dealing well with these problems. The query optimizer has become one of the most complex software modules in a DBMS.

Dynamic programming techniques are typically used to deal with the first problem [SAC+79], although randomized algorithms have also been proposed [Swa89, IK90]. As we shall see, they apply in our approach too, so we focus here on the second problem. To accurately estimate the cost of executing a particular plan, we need to estimate the values of various parameters. Typically, these parameters can be divided into three categories:

1. Parameters representing properties of the data (cardinalities of tables, distributions of values, etc.). The DBMS typically maintains estimates of these parameters.
2. Parameters representing properties of the query components (e.g., sizes of groups, selectivities of predicates). Much research has focused on how to make accurate estimates of selectivities and result sizes. These techniques typically use histograms [PIHS96] (part of the data properties) or sampling [LNSS93].
3. Parameters representing properties of the run-time environment (e.g., amount of available memory, processor speed, multiprogramming level, access characteristics of secondary storage). These are gathered from observations of the realistic deployment environments.

If the value of a parameter cannot be exactly predicted (which is almost always the case, especially if it can change even during the execution of a query), it can at best be modeled by a distribution. Current optimizers simply approximate each distribution by using the mean or modal value. They then choose the plan that is cheapest under the assumption that the parameters actually take these specific values [SAC+79] and remain constant during execution. We call this the *least specific cost (LSC)* plan.

We propose a very different approach in this paper. We view the query optimizer as an agent trying to make a decision; it must choose among different plans. The standard decision-theoretic approach is to choose a plan that maximizes expected utility [Res87]. Here utility is essentially the negation of the cost: the lower the cost, the more attractive the plan. Thus, we argue that rather than choosing the LSC plan, query optimization algorithms should be directed towards finding the plan of *least expected cost (LEC)*.

1.1 A Motivating Example

The following example should help motivate the use of LEC plans. For pedagogical reasons, we focus in this example, and throughout most of the paper, on only one parameter, the amount of available memory, which is known to be difficult to predict accurately in practice [Loh98]. Thus we

*Work supported in part by NSF under grant IRI-96-25901.
†Work supported in part by NSF under grant IRI-96-?????.

assume that everything else (e.g., predicate selectivities) is known. We consider the case of more than one parameter in Section 3.6.

Example 1.1: Consider a query that requires a join between tables A and B, and the result needs to be ordered by the join column. A has 1,000,000 pages, B has 400,000 pages, and the result has 3000 pages. Consider the following two evaluation plans:

- *Plan 1*: Apply a sort-merge join to A and B. The optimizer cost formulas tell us that if the available buffer size is greater than 1000 pages (the square root of the larger relation), the join requires two passes over the relations [Sha86]. If there are fewer than 1000 pages available, it requires at least another pass. Each pass requires that 1,400,000 pages be read and written.

- *Plan 2*: Apply a Grace hash-join [Sha86] to A and B and then sort their result. We know that if the available buffer size is greater than 633 pages (the square root of the smaller relation), the hash join requires two passes over the input relations. The subsequent sort also incurs additional overhead, especially if the data does not fit in the buffer.

If the available buffer memory is accurately known, it is easy to choose between the two plans (Plan 1 when more than 1000 pages are available and Plan 2 otherwise). However, assume that the available memory is estimated to be 2000 pages 80% of the time and 700 pages 20% of the time. (This distribution is obtained by observing the actual query execution environment.) Current optimizers assume one specific memory value (in this case, 2000 pages as a modal value, or 1740 pages as a mean value). In either case, the plan chosen would be Plan 1. However, we claim that Plan 2 is likely to be cheaper on average across a large number of evaluations. The intuition is simple: In 80% of the runs, Plan 2 is slightly more expensive than Plan 1 (the extra expense arises in sorting the small result), whereas in 20% of the cases, Plan 1 is far more expensive than Plan 2 (the extra expense comes from an extra pass over the data). On average, we would expect Plan 2 to be preferable. ∎

Example 1.1 shows the flaw that arises if parameter distributions are characterized by a single expected value: The least cost plan found based on this value is not necessarily the plan of least expected cost. Indeed, whenever there are discontinuities in cost formulas (as is the case with database join algorithms), such an effect is likely to arise.

1.2 Contributions

The contributions of this paper are:

1. We introduce LEC plans, which are guaranteed to be at least as good as (and typically better than) any specific LSC plan.

2. We show how LEC query optimization can take into account parameters that have a constant value during any specific query execution (*static parameters*), and also those that vary during execution (*dynamic parameters*). Moreover, it can be applied either at compile-time or at start-up time.

3. We show how traditional dynamic programming query optimizers can be easily extended to produce the LEC plan. The extension increases the cost of query optimization by a factor depending on the granularity of the parameter distribution.

4. We extend LEC query optimization to handle queries where the selectivity of each predicate is a parameter modeled by a distribution.

Depending on the actual distribution of parameters that arises in practice, the LEC plan can be much more efficient on average than any specific LSC plan. The greater the run-time variation in the values of parameters that affect the cost of the query plan, the greater the cost advantage of the LEC plan is likely to be. If such parameter distributions are common, it should be well worth implementing this approach in commercial database systems.

The rest of this paper is organized as follows. In the next section, we review the traditional approaches to query optimization and discuss related work. In Section 3, we introduce LEC query optimization and discuss how LEC plans may be generated in practice by a typical DBMS. To this end, we consider extensions to the widely used System R query optimization algorithm. Our algorithms apply both in the case where parameters are static and (under some simplifying assumptions) the case where they are dynamic. In Section 4 we discuss the simplifying assumptions we make in the paper and possible future directions.

2 Background

2.1 Standard Query Optimization

There are three basic approaches proposed for query optimization algorithms:

- *Bottom-Up Optimization*: This is a synthetic approach in which a suitable plan is created by starting from the stored tables and building increasingly larger plans until a plan for the entire query is formed.

- *Top-Down Optimization*: This is a divide-and-conquer approach in which the entire query is divided into pieces, each piece is optimized, and then the pieces are put together to form the query plan.

- *Transformational Optimization*: This starts with some valid complete query plan, and repeatedly transforms it into a different valid complete plan. At every stage, the plan of least cost is retained.

Every query optimizer uses some element of each approach. Typically, a query is divided into a graph of "query blocks" and some transformational optimizations are performed on the query blocks [PHH92]. Each query block in the graph is then typically optimized almost independently. A specific kind of query block, the *SELECT-PROJECT-JOIN* or *SPJ* block, has received a lot of attention, because it occurs in many queries and involves expensive join operations. The optimization of an SPJ block itself could use any of the three basic approaches. Most commercial database systems use a bottom-up optimizer based on dynamic programming. This approach was first suggested in the System R project [SAC+79]. We now briefly describe how it works. In later sections of the paper, we describe variations of this algorithm that find LEC plans.

2.2 The System R Approach

Suppose we are given n relations, $A_1, \ldots, A_n$, whose join we want to compute. For ease of exposition, assume that there are join predicates between every pair of relations. (This is not very realistic, but one can always assume the existence of a trivially true predicate.) Three basic observations influence the algorithm:

1. Joins are commutative.
2. Joins are associative.
3. The result of a join does not depend on the algorithm used to compute it. Consequently, dynamic programming techniques may be applied.[1]

The System R optimizer also applies some heuristics that further limit the space of plans considered. Of particular relevance to this paper are the following two heuristics:

1. Only binary join algorithms are considered. Consequently, a three-relation join evaluation plan involves the combination (i.e., join) of a two-relation join result and a stored relation.
2. To find the best plan for a k-relation join, the only plans considered are those that first involve joining some subset of $k-1$ of these relations and then adding in the kth. Other possible approaches (for example, considering the best plan for joining a subset of $k-2$ of the relations, joining the remaining two relations, and then joining the results) are not considered.

Given these two heuristics, System R is essentially trying to find the permutation π of $\{1, \ldots, n\}$ that produces the best plan of the form

$$((\cdots((A_{\pi(1)} \bowtie A_{\pi(2)}) \bowtie A_{\pi(3)}) \cdots) \bowtie A_{\pi(n)}).$$

Such plans are called *left-deep* plans.

Conceptually, we can think of the System R optimizer as working on a dag with a single root (node of indegree 0). Each node in the dag is labeled by a subset S of $\{1, \ldots, n\}$. The label of the root is the empty set. The nodes at depth k are labeled by the subsets of $\{1, \ldots, n\}$ of cardinality k. There is an edge from a node at depth $k-1$ labeled by S' to a node at depth k labeled by S iff $S' \subseteq S$. Fix a setting of the parameters. Associated with the node labeled S is the best left-deep plan to compute the *join over* S (i.e., $\bowtie_{i \in S} A_i$, the join of the relations with indices in S) for that setting of the parameters. This plan is determined inductively as follows. Initially, the algorithm determines the best plan to access each of the individual relations. In the next step, the algorithm examines all possible joins of two relations. For each pair of relations, several different join algorithms are considered and the cheapest evaluation plan is retained. Assume inductively that we have associated with each node up to depth k the plan for computing the join associated with that node. To compute the best plan for a node S at depth $k+1$, consider each $j \in S$ and let $S_j = S - \{j\}$ and let $B_j = \bowtie_{i \in S_j} A_i$. For each j, let C_j be the sum of the cost of the best plan for accessing A_j, the cost of the best plan for computing B_j (which we have already determined), and the cost of the best plan for computing $B_j \bowtie A_j$. We then find the j for which C_j is minimal. This gives us the the best plan for computing $\bowtie_{i \in S} A_i$ as well as its cost. Note that at phase $(k+1)$, only results from phase k are utilized. At phase n, we label the root by the best plan to compute the join (and its cost). Although this approach takes time and space exponential in n, n is usually small enough in practice to make this approach feasible.

To summarize, we have:

Theorem 2.1: *The System R optimizer computes the LSC left-deep plan for a specific setting of the parameters.*

(We remark that a formal proof of Theorem 2.1 can be provided along the lines of the proof of Theorem 3.3 given below.)

2.3 Previous Work on Dealing with Uncertainty of Parameter Values

It is widely recognized that query optimizers often make poor decisions because their compile-time cost models use inaccurate estimates of various parameters. There have been several efforts in the past to address this issue. We categorize them as (a) strategies that make decisions at the start of query execution and (b) strategies that make decisions during query execution.

Let us assume that there are some parameters that cannot be predicted accurately at compile-time, but that are accurately known when a query begins execution. An example of such a parameter is the number of concurrent users. Let us further assume that the value of this parameter stays constant during the execution of the query. In this case, we are aware of three kinds of strategies:

- A trivial strategy is to perform query optimization just before query execution. This is the approach used in database systems like Illustra [Ill94], but is not particularly efficient, since the query may be executed repeatedly.
- Another strategy is to find the best execution plan for every possible run-time value of the parameter. This requires much additional work at compile-time, but very little work at query execution time (a simple table lookup to find the best plan for the current parameter value). In [INSS92], the authors suggest using randomized optimization to reduce the compile-time optimization effort.
- [GC94] suggests a hybrid strategy that performs some of the search activity at compile-time. Any decisions that are affected by the value of the parameter are deferred to start-up time through the use of "choice nodes" in the query evaluation plan.

For parameters that cannot be accurately predicted at the start of query execution (like predicate selectivities), these strategies are clearly inapplicable. We are aware of four other strategies that address this case; all involve a potential modification of query execution.

- [KD98] proposes using a regular query optimizer to generate a single plan, annotated with the expected cost and size statistics at all stages of the plan. These statistics are affected by the choice of parameter value. During actual query execution, the expected statistics are compared with the measured statistics. If there is a

[1] It is well-known that this is not quite accurate; the physical properties ("interesting orders") of the join result depend on the specific plan used to create it. This requires simple extensions of the optimization algorithm, as described in [SAC+79]. We ignore this issue here, since our solutions apply without change in the presence of these extensions.

significant difference, the query execution is suspended and re-optimization is performed using the more accurate measured value of the parameter.

- [Ant93] implements an interesting variant of this idea. In order to execute a query, multiple query plans are run in parallel. When one plan finishes or makes significant progress, the other competing plans are killed. This strategy assumes that resources are plentiful (and so can be wasted), and is applied only to subcomponents of the query (typically to individual table accesses).

- [UFA98] examines a very specific kind of parameter variation: the cost of accessing a table across an occasionally faulty network, such as the Internet. Their strategy reoptimizes the query when the system recognizes during query execution that the source of a table is not available. Instead of restarting the query like [KD98], the remainder of the query plan is adjusted ("scrambled") to try to make forward progress.

- [SBM93] focuses on uncertainties that can be reduced by sampling (more specifically, the uncertainty of the selectivity of a predicate). They use decision-theoretic methods to pre-compute scenarios where it may be worthwhile to do sampling (since sampling itself comes with a cost). If such a scenario arises, they do the sampling and modify the plan as appropriate, given the result of the sampling. We remark that this approach is the one perhaps closest to that advocated here in its view of query optimization as a decision problem and its aim of minimizing expected cost. The techniques of [SBM93] can be combined with those suggested here.

Note that these approaches all involve making some decision after compile-time. The way they deal with uncertainty is to wait until they have more information. We deal with uncertainty by treating the parameters as random variables, so our approach can be applied completely at compile-time, as well as at start-up time or run-time. When our approach is applied at compile-time, the size of the query plan created does not increase as with some of these approaches.

3 LEC Query Optimization

3.1 The Formal Model

Fix a query. Like [INSS92], we start by assuming that there is a cost function $\mathbb{C}$ that takes two arguments, a plan p and a vector $\mathbf{v}$ of values of relevant parameters, and returns a cost. Intuitively, $\mathbb{C}(\mathsf{P}, \mathbf{v})$ is the cost of executing plan p under the assumption that the relevant variables take the values $\mathbf{v}$. The standard (LSC) approach is to choose a fixed value of $\mathbf{v}$—usually the expected value of the parameters—and find a p with the least cost. We assume instead that there is a probability measure on the space $\mathcal{V}$ of possible values of the parameters. Given $\mathbb{C}$ and Pr, we can compute the expected cost for each plan p. Let $\mathbf{E}_{\mathbb{C}}(\mathsf{P})$ denote the expected cost of p (with respect to cost function $\mathbb{C}$ and probability Pr); as usual, we have

$$\mathbf{E}_{\mathbb{C}}(\mathsf{P}) = \sum_{\mathbf{v} \in \mathcal{V}} \mathbb{C}(\mathsf{P}, \mathbf{v}) \Pr(\mathbf{v}).$$

Following standard decision-theoretic approach, our goal is to find the *LEC plan*, the one whose expected cost is least. If the distribution Pr is an accurate model of the distribution of the parameters that is encountered at run-time, and the cost estimates $\mathbb{C}$ are accurate then, by definition, the expected execution cost of the LEC plan is at least as low as that of any specific LSC plan.

The goal of finding the LEC plan makes sense both at compile-time and at start-up time. At start-up time, the distribution of parameters will typically be different from the one at compile-time. For some parameters, the distribution may be more concentrated around one value. However, it is unlikely that there will be complete information about all the values of the relevant parameters, even at start-up time. This is particularly true about parameters whose values may change as we execute the query. Note that the LEC approach applies to such dynamic parameters as well. We need to use a probability distribution over all possible sequences of parameter values during the execution and then again compute the plan with least expected cost. We explore the details of this approach in Section 3.5. Until then, we assume that parameters do not change value during query execution.

This leads to some obvious questions:

1. How do we get the probability distributions?

2. How do we get the cost estimates?

3. How do we compute the LEC plan?

With regard to the first question, as we noted in the introduction, the DBMS in practice is constantly gathering statistical information. We believe that the statistics can be enhanced to provide reasonable estimates of the relevant probabilities, although this is certainly an area for further research. For the purposes of this paper, we assume without further comment that the probabilities are available. As for the second question, we are making the same assumptions about costs that are made by all standard query optimizers. Finally, with regard to the last question, in the remainder of this section, we provide a number of ways to modify the standard optimizers so as to produce the LEC plan (or a reasonable approximation to it). These approaches vary in their need to modify the underlying query optimizer, the quality of the plan produced, and the underlying assumptions about the distribution.

3.2 Algorithm A: Using a Standard Query Optimizer as a Black Box

For ease of exposition, we assume in the next few sections that the only relevant parameter is the amount of available memory, so we take $\mathbf{v}$ to represent this quantity. This assumption is dropped in Section 3.6 We assume that we can partition the distribution of the amount of available memory into a small number (say b) of buckets such that the cost of a plan is likely to remain relatively constant within a bucket. For example, in Example 1.1, the appropriate buckets are the intervals $[0, 633)$, $[633, 1000)$, and $[1000, \infty)$. Choosing the buckets appropriately can be nontrivial; we discuss this issue in more detail in Section 3.7. We can identify the standard approach to doing query optimization with the special case where there is only one bucket. Once we have chosen the buckets, we pick a representative from each bucket; call them $m_1, \ldots, m_b$. Finally, we assume that we have a probability measure Pr such that $\Pr(m_i)$ characterizes how likely we are to run the query in the ith bucket.

Given these assumptions, there is a straightforward approach to finding good approximations to the LEC plan that uses a standard query optimizer as a black box. Suppose we want to compute $A_1 \bowtie \cdots \bowtie A_n$. Assume that memory stays constant during the execution of the plan.

Algorithm A

> For each value m_i of the memory parameter, we run the optimizer under the assumption that m_i is the actual amount of memory available. This gives us b candidate plans. We then compute the expected cost of each candidate, and choose the one with least expected cost.

As long as the expected value of memory used by the traditional LSC approach is one of the b possible values we consider (and, without loss of generality, we can assume it is), then we are guaranteed to end up with a plan whose expected cost is no higher than that of the plan chosen by the traditional approach. We assume that, in practice, the actual LEC plan will have a cost close to the optimal plan for some value m_i of memory. To the extent that this is true, Algorithm A gives us a good approximation to the LEC plan.

The cost of Algorithm A is the cost of b invocations of the optimizer, plus the cost of evaluating the expected cost of each candidate plan. Each candidate plan has $n-1$ joins, and each has to be costed for b different memory sizes to determine the expected cost. There are b such candidates, leading to a total cost of $O((n-1)b^2)$, which should be much smaller than the cost of candidate generation, $O(bn2^{n-1})$. Consequently, the approximate cost of Algorithm A is b times the cost of a single optimizer invocation.

Note that it makes sense to use Algorithm A at start-up time as well as at compile-time; we simply use the appropriate distribution over memory sizes when checking to see which candidate plan is best. We can also combine these ideas with the parametric query optimization approach of [INSS92]. We can precompute the best expected plan under a number of possible distributions (ones that give good coverage of what we expect to encounter at run-time), and store these expected plans, for use at query execution time.

While this approach has the advantage of not requiring any change to the optimizer whatsoever, it has two major drawbacks. The first is that it requires us to prespecify the buckets (this point should become clearer in Section 3.7); the second is that it may not actually return the LEC plan. It is conceivable that a plan not optimal for any m_i actually does better on average than any candidate considered by the algorithm above. For example, the plan that is second-best for some memory size may do better on other memory sizes than the best plan for that memory size, and so may do better in expectation. We now present a simple modification of this approach that generates more candidate plans, although it has the disadvantage of requiring us to modify the basic query optimizer.

3.3 Algorithm B: Generating More Candidates

Suppose that rather than generating the best plan for each memory size m_i, we generate the top c plans, for some $c > 1$. It is relatively straightforward to modify existing query optimizers to do this. For concreteness, we show how this can be done with System R.

Algorithm B

> Assume inductively that we have associated with each node up to depth k in the dag the top c plans for computing the join associated with that node. To compute the top c plans for a node S at depth $k+1$, consider each $j \in S$ and let $S_j = S - \{j\}$, as before. We consider the top c plans for computing the join over S_j and the top c plans for accessing A_j; combining them using each possible join method gives us the top c plans for computing the join over S if we join A_j last.

While it seems that there are c^2 combinations of plans that need to be considered here for each join method, the actual number of combinations is lower.

Proposition 3.1: *It suffices to consider at most $c + c\log c$ combinations of plans for each join method to produce the top c plans.*

Proof: Suppose $s_1, \ldots, s_c$ are the top c plans for computing S_j (sorted in increasing order of cost) and $a_1, \ldots, a_c$ are the top c plans for accessing A_j (again sorted in increasing order of cost). Note that the cost of the combination (s_i, a_k) is no higher than the combination $(s_{i'}, a_{k'})$ if $i \le i'$ and $k \le k'$; so there are (at least) $ik-1$ combinations with cost no higher than (s_i, a_k). Thus we only have to consider (s_i, a_k) for $ik \le c$, since $ik > c$ implies we can get (at least) c plans at least as good as (s_i, a_k).

Note that $ik \le c$ implies $i \le c/k$, so we only need to consider the top $\lfloor c/k \rfloor$ entries of the kth column if we arrange the combinations in a $c \times c$ matrix. Thus the total number of entries we need to consider is

$$\sum_{k=1}^{c} \left\lfloor \frac{c}{k} \right\rfloor \le \sum_{k=1}^{c} \frac{c}{k} = c \sum_{k=1}^{c} \frac{1}{k}.$$

Recall that

$$\sum_{x=1}^{c} \frac{1}{x} < 1 + \int_1^c \frac{1}{x} dx = 1 + \log c.$$

Thus the total number of entries we need to consider is at most $c + c\log c$. ∎

To compute the best c plans using a particular join method for joining S_j and A_j, we must first evaluate the cost formula for the join method. Note that all the c variants of each input have the very same properties, and so behave identically with respect to the cost formula. Consequently, the only difference between the $c + c\log c$ combinations arises from the sum of the costs of the two input plans. Consequently, the cost of checking these combinations is expected to be small compared to the cost of evaluating a cost formula. By considering all $j \in S$, we get j lists of c top plans. We then take the top c plans from the combined list. This extension can be easily implemented and is a relatively small and localized change to current optimizers.

Theorem 3.2: *Algorithm B computes the top c left-deep plans for each of the b choices of parameter values at αb times the cost of computing the single best left-deep plan for one specific setting of the parameters, for some small constant $\alpha > 0$.*

Once we have the top c plans for each of the b memory sizes, we can then again compute the expected cost of each of these cb plans, and choose the plan of least expected cost. As we showed in the previous section, the computation of expected costs of candidate plans is small compared to the cost of candidate generation. In this case, the number of candidates is increased by a factor of c, but we still expect

Algorithm B to be roughly b times as expensive as a single optimizer invocation.

While Algorithm B generates more candidates (and thus is more likely to end up with a good approximation to the LEC plan), it still does not necessarily end up with the LEC plan. As we now show, if we are willing to modify the basic query optimization algorithm further, we can produce the actual LEC plan.

3.4 Algorithm C: A Generic Algorithm for Computing the LEC Plan

We now provide a generic modification of the basic System R query optimizer that can directly compute the LEC plan, merging the candidate generation and costing phases. We assume inductively that we have associated with each node up to depth k in the dag the plan with least expected cost for computing the join associated with that node (as well as the expected cost itself). We further assume that with each node we have associated a probability distribution over the possible memory sizes. Intuitively, this is the probability distribution over the available memory when we reach that node during an actual execution of a query plan containing the node as a subplan. If we assume that memory size does not change during the course of executing the plan, and that join operations are not pipelined in the plan, then the distribution is the same at every node. We do away with this restriction in the Section 3.5.

Algorithm C

> Again, we proceed inductively down the dag. For each stored relation A_i, find an LEC access path for it. To compute the plan with least expected cost for a node at depth $k+1$ labeled S, consider each $j \in S$ and let $S_j = S - \{j\}$. The expected total cost of S is the expected cost of computing S_j (which, by assumption, we already have in hand) added to the expected cost of joining S_j and A_j, which we can compute given the probability distribution of available memory. If we consider a probability distribution over b different memory sizes, this computation requires b evaluations of the cost formula for the join algorithm. We retain the plan for S with the least expected total cost, discarding all the other candidates.

Theorem 3.3: *Algorithm C gives us the LEC left-deep plan.*

Proof: The proof is a straightforward adaptation of the argument for the correctness of the basic System R algorithm, using the fact that expectation distributes over addition. Suppose S is a nonempty subset of $\{1,\ldots,n\}$. Let P_S denote a left-deep plan for computing $\bowtie_{i\in S} A_i$. We can conceptually think of P_S as consisting of a choice of $j \in S$, a plan P_S^R for accessing A_j, and, if $|S| > 1$, a plan P_S^L for computing $B_j = \bowtie_{i\in S_j} A_i$ and a plan $\mathsf{P}_S^{\bowtie}$ for computing $B_j \bowtie A_j$. If $|S| > 1$ then

$$\mathbb{C}(\mathsf{P}_S, \mathbf{v}) = \mathbb{C}(\mathsf{P}_S^L, \mathbf{v}) + \mathbb{C}(\mathsf{P}_S^R, \mathbf{v}) + \mathbb{C}(\mathsf{P}_S^{\bowtie}, \mathbf{v}).$$

It follows that

$$\mathbf{E}_{\mathbb{C}}(\mathsf{P}_S) = \mathbf{E}_{\mathbb{C}}(\mathsf{P}_S^L) + \mathbf{E}_{\mathbb{C}}(\mathsf{P}_S^R) + \mathbf{E}_{\mathbb{C}}(\mathsf{P}_S^{\bowtie}).$$

Let $\widehat{\mathsf{P}}_S$ be the plan that Algorithm C outputs for S. We want to show that $\mathbf{E}_{\mathbb{C}}(\widehat{\mathsf{P}}_S) \le \mathbf{E}_{\mathbb{C}}(\mathsf{P}_S)$ for all P_S. We proceed by induction on $|S|$.

For the base case we have $|S| = 1$. Suppose $S = \{i\}$. Then $\mathbf{E}_{\mathbb{C}}(\widehat{\mathsf{P}}_S) \le \mathbf{E}_{\mathbb{C}}(\mathsf{P}_S)$, since Algorithm C will choose an LEC access path for A_i. Now assume that the claim holds for all S with $|S| = k$. Let S be a subset of $\{1,\ldots,n\}$ with $k+1$ elements. For all $j \in S$, let $\mathsf{P}(j)$ be a plan such that $\mathsf{P}(j)^L = \widehat{\mathsf{P}}_{S_j}$, $\mathsf{P}(j)^R = \widehat{\mathsf{P}}_{\{j\}}$, and $\mathsf{P}(j)^{\bowtie}$ is an LEC method to compute the join. It is clear from the description of Algorithm C that $\widehat{\mathsf{P}}_S \in \{\mathsf{P}(j) : j \in S\}$ and that $\mathbf{E}_{\mathbb{C}}(\widehat{\mathsf{P}}_S) = \min\{\mathsf{P}(j) : j \in S\}$. Suppose P_S is an arbitrary left-deep plan to compute $\bowtie_{i\in S} A_i$. Suppose P_S computes $\bowtie_{i\in S_j} A_i$ first. Thus P_S^L computes $\bowtie_{i\in S_j} A_i$. By the induction hypothesis and the definition of $\mathsf{P}(j)$, we see that $\mathbf{E}_{\mathbb{C}}(\mathsf{P}_S^L) \ge \mathbf{E}_{\mathbb{C}}(\widehat{\mathsf{P}}_{S_j}) = \mathbf{E}_{\mathbb{C}}(\mathsf{P}(j)^L)$. Furthermore, $\mathbf{E}_{\mathbb{C}}(\mathsf{P}_S^R) \ge \mathbf{E}_{\mathbb{C}}(\widehat{\mathsf{P}}_{\{j\}}) = \mathbf{E}_{\mathbb{C}}(\mathsf{P}(j)^R)$, since $\widehat{\mathsf{P}}_{\{j\}}$ is an LEC access path. Finally, $\mathbf{E}_{\mathbb{C}}(\mathsf{P}_S^{\bowtie}) \ge \mathbf{E}_{\mathbb{C}}(\mathsf{P}(j)^{\bowtie})$, since $\mathsf{P}(j)^{\bowtie}$ is an LEC join method. Thus $\mathbf{E}_{\mathbb{C}}(\mathsf{P}_S) \ge \mathbf{E}_{\mathbb{C}}(\mathsf{P}(j)) \ge \mathbf{E}_{\mathbb{C}}(\widehat{\mathsf{P}}_S)$ as required. ∎

If we assume that memory does not change as we execute the plan, then the cost of the computation is b times the cost of the standard computation using a single memory size. Again, Algorithm C works both at compile-time and at start-up time. And again, we can combine these ideas with those of parametric query optimization, precomputing the LEC plans under various assumptions about the probability distributions and storing them for use at start-up time.

3.5 Dealing with Change During Execution

So far we have assumed that the amount of available memory stays constant throughout the execution of the plan. This may not be so reasonable if the query takes a long time (on the order of minutes or more). During the execution, concurrent new queries may start while old queries may finish. If we assume that available memory is mainly determined by the number of queries being run concurrently, then it may well change during the execution.

To deal with dynamic memory changes, we assume a probability measure over the possible sequences of memory sizes. We then must evaluate each candidate plan with respect to this sequence and determine its expected cost. To keep the analysis from becoming too unwieldy, we assume that plan execution takes place in phases, each corresponding to a join in the plan. We assume that memory does not change during the execution of a phase, but can change between phases. If we compute a join over n relations, there are $n-1$ phases. Thus, we need to consider possible run-time environments corresponding to each sequence of memory sizes of length $n-1$. We need to use a probability distribution over all such memory size sequences.

Where does this distribution come from? Perhaps the most natural assumption is to assume that we have some distribution over the initial memory sizes, and that there is a transition probability describing how likely memory is to change by m units, for each value of m. For simplicity, we assume that this transition probability depends only on the current memory usage, not on the time. This is a reasonable assumption for 24×7 systems in stable operational mode.

Under these simplifying assumptions, we can apply Algorithm C presented in the previous subsection to calculate the LEC plan even with dynamic memory. We simply associate the initial distribution with the root of the dag, and use the transition probabilities to compute the distribution associated with each node in the tree. We can then apply the

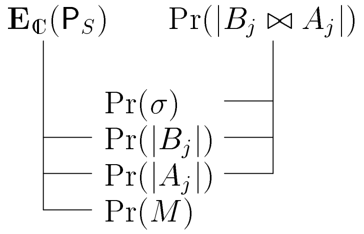

Figure 1: Distributions Needed at Each Node and What Depends on Them

algorithm without change, since Algorithm C does not rely on the fact that $\mathbf{v}$ consists of a single memory size. Note, however, that the complexity of Algorithm C is clearly affected, since the parameter space is potentially much bigger (since if there are b_M memory values we consider, there are ${b_M}^{n-1}$ sequences of length $n-1$).

Theorem 3.4: *Given the simplifying assumptions above, Algorithm C returns the LEC left-deep plan even in the presence of dynamically varying parameters.*

Proof: While $\mathbf{v}$ now consists of a sequence of memory sizes, the proof of Theorem 3.3 works here also, since that proof did not rely on the fact that $\mathbf{v}$ is a single memory size in any way. ∎

3.6 Dealing with Multiple Parameters

We have focused on only one parameter so far (i.e., the amount of available memory). In practice, we typically have to deal with a number of parameters. In this section, we consider the effect of multiple parameters on our algorithms (specifically, available memory and the selectivities of all the query predicates). Selectivities, in particular, are notoriously uncertain. We believe that by representing the uncertainty by a probability distribution and computing the LEC plan, we can ameliorate some of the difficulty. Note that the ideas of [SBM93] for deciding when to sample may also be usefully applied here. We focus on the static case in this section. We can apply the idea from Section 3.5 to deal with the dynamic case.

When the amount of available memory is the only uncertainty, we need only a single probability distribution at every node in the dag to allow us to compute the LEC plan. When there is more than one uncertain parameter, we may need to expand that to a joint distribution, whose size may grow exponentially as the number of parameters. Since independence often holds in practice or is the default assumption of existing query optimizers, we assume for the rest of this section that all parameters of interest are independent. This simplifying assumption means that we can carry a separate distribution for each parameter and avoid the exponential blow-up in the description of the joint distribution. If are some dependencies between the variables, but not too many, we can still describe the distribution succinctly using a Bayesian network [Pea88]. We believe that the techniques that we present here will also be applicable to that case.

Note that the standard formulas for computing the cost of a join plan typically take three parameters: the sizes of the two relations being joined and the amount of available memory. Thus, to compute the expected cost of a join method applied to a particular pair of relations, we need just three distributions. This means that even though we may have many more parameters to deal with, if we apply Algorithm C to the multi-parameter setting, at each node only three distributions are required to compute the LEC plan for that node. To be able to apply this idea inductively at every node in the dag, we also need to compute the distribution of the size of the result of the join, since the parent node of the current node (should there be any) needs that as the distribution of one of its input size. In order to compute the distribution of the size of the result, we need to have a distribution of the selectivity of the join predicates. Thus at each node, we need exactly four distributions—the three distributions for computing expected cost plus the distribution for the selectivity of the join predicate—no matter how many distributions we start with. Figure 1 shows the distributions we carry at each node and the quantity that depends on them.

Here is the generic modified algorithm. As in Section 3.4, the algorithm works on the dag.

Algorithm D

> To compute the LEC plan for a node at depth $k+1$ labeled S, consider each $j \in S$ and let $S_j = S - \{j\}$. Let $B_j = \bowtie_{i \in S_j} A_i$. We assume inductively that we have the LEC plan for S_j as well as the distribution of $|B_j|$. We also assume we have the best way to access A_j and the distribution of $|A_j|$ after any initial selection. We evaluate the cost formula for each triple of possible values of M, $|B_j|$ and $|A_j|$, and use that to compute the expected cost of calculating $S_j \bowtie A_j$ for each method. Taking b_X to denote the number of buckets for random variable X, this shows that we need $b_M b_{|B_j|} b_{|A_j|}$ evaluations to compute the expected cost of a particular method for computing $S_j \bowtie A_j$. We can compute the expected cost for each choice of j, and retain the plan of least expected cost. The total cost of this naive computation is $b_M \sum_{j \in S} b_{|B_j|} b_{|A_j|}$ for each join algorithm considered.
>
> To compute the distribution for the size of the result, we fix a j and compute, for each triple (a, b, σ) of possible values of $|A_j|$, $|B_j|$, and selectivity σ the probability that the join has size $ab\sigma$. (Since we are assuming independence, this is just the product of $\Pr(|A_j| = a)$, $\Pr(|B_j| = b)$, and the probability that the selectivity is σ.) From this we can fill in the distribution for the size of the result.

Thus, we require $O(b_{|B_j|} b_{|A_j|} b_\sigma)$ operations to compute the distribution. Since the size of the result is independent of the choice of j, we need to do this computation for only one j; we can choose the j for which $b_{|B_j|} b_{|A_j|}$ is minimal. In practice, we expect that we will have $b_{|A_j|} = b_{|B_j|}$ for all j, so the choice will not matter.

To summarize, the generic method given above takes $\sum_{j \in S} b_M b_{|B_j|} b_{|A_j|}$ evaluations of the cost formula and takes $O(b_{|B_j|} b_{|A_j|} b_\sigma)$ operations to compute the distribution for $|B_j \bowtie A_j|$. Can we do better if we have more knowledge of the cost formula? Most join methods used in practice have relatively simple cost formulas. We pick sort-merge join and page nested-loop join as examples, and demonstrate in the next two subsections that we can indeed do much better.

3.6.1 The Case of Sort-Merge Join

Let $L = \max\{|A|, |B|\}$; the cost of sort-merge join for $A \bowtie B$ is:[2]

$$\mathbb{C}(\mathsf{SM}, \mathbf{v}) = \begin{cases} 2 \cdot (|A| + |B|) & \text{if } M > \sqrt{L} \\ 4 \cdot (|A| + |B|) & \text{if } \sqrt[3]{L} < M \leq \sqrt{L} \\ 6 \cdot (|A| + |B|) & \text{if } M \leq \sqrt[3]{L} \end{cases}$$

Note that

$$\begin{aligned} \mathbf{E}_{\mathbb{C}}(\mathsf{SM}) \;=\; & \mathbf{E}_{\mathbb{C}}(\mathsf{SM} : |A| \leq |B|) \Pr(|A| \leq |B|) + \\ & \mathbf{E}_{\mathbb{C}}(\mathsf{SM} : |A| > |B|) \Pr(|A| > |B|). \end{aligned}$$

We now show how the first term can be computed efficiently. The second term can be computed analogously; we leave the details to the reader. Let $F_b = \mathbf{E}(|A| : |A| \leq b) + b$. Let $Val(X)$ denote the representatives from the b_X buckets for variable X. We can rewrite the first summand as follows:[3]

$$\sum_{b \in Val(|B|)} \Pr(|B| = b) \left(\begin{array}{ll} 2F_b \Pr(M > \sqrt{b}) & + \\ 4F_b \Pr(\sqrt[3]{b} < M \leq \sqrt{b}) & + \\ 6F_b \Pr(M \leq \sqrt[3]{b}) & \end{array} \right). \quad (1)$$

We now describe how to compute $\mathbf{E}_{\mathbb{C}}(\mathsf{SM} : |A| \leq |B|)$ and $\Pr(|A| \leq |B|)$ in time $O(b_M + b_{|A|} + b_{|B|})$. First we compute $\Pr(M \geq m)$ for each $m \in Val(M)$. We can easily compute all $Val(M)$ probabilities in time $O(b_M)$. We store these values in a table. By table lookup, we can then compute $\Pr(M > b)$, $\Pr(\sqrt[3]{b} < M \leq \sqrt{b})$, and $\Pr(M \leq \sqrt[3]{b})$, for each value of b, in constant time.

Next we compute $\Pr(|A| \leq b)$ for each $b \in Val(|B|)$ and store these values in a table. Since $\Pr(|A| \leq b') = \Pr(|A| \leq b) + \Pr(b < |A| \leq b')$, for $b < b'$, we can compute all of these probabilities in time $O(b_{|A|} + b_{|B|})$ (because we need only go through each set of buckets once).

Then we compute $\mathbf{E}(|A| : |A| \leq b)$ for each $b \in Val(|B|)$ and store these values. Again, this can be done in time $O(b_{|A|} + b_{|B|})$, since $\mathbf{E}(|A| : |A| \leq b') = \mathbf{E}(|A| : |A| \leq b) \Pr(|A| \leq b) + \mathbf{E}(|A| : b < |A| \leq b') \Pr(b < |A| \leq b')$ for $b < b'$ (and so we only need to go through each set of buckets once).

We can now compute $\mathbf{E}_{\mathbb{C}}(\mathsf{SM} : |A| \leq |B|)$ using (1). Note we need only constant time to compute each summand, since for each $b \in Val(|B|)$ (since F_b can be computed by adding b to $\mathbf{E}(|A| : |A| \leq b)$, which we already have, and it takes only constant time to compute the probabilities involving M, since they have been stored. Thus, we can compute the expectation in total time $O(b_M + b_{|A|} + b_{|B|})$ (including the time it take to compute all the values we have stored in the tables).

Finally, we need to compute $\Pr(|A| \leq |B|)$. This can be done in time $O(b_{|B|})$, since

$$\Pr(|A| \leq |B|) = \sum_{b \in Val(|B|} \Pr(|A| \leq b) \Pr(|B| = b),$$

and we have already computed $\Pr(|A| \leq b)$.

[2] Our formulas consider I/O costs only and are based on the analysis presented in [Sha86], simplified to three cases. Commercial database systems use more complicated formulas, usually represented in the form of complex code. These are sometimes the result of aiming for too much accuracy when modeling the algorithm, despite the fact that the parameters used to instantiate the model are inaccurate. We speculate that a return to simple formulas in combination with LEC optimization may result in more reliable query optimizers.

[3] We write $X = x$ as an abbreviation for the statement "X takes on a value in the bucket whose representative is x".

The whole computation takes time $O(b_M + b_{|A|} + b_{|B|})$, so the algorithm we informally described is linear in the total number of buckets. Note that this algorithm is (asymptotically) optimal, since we must at least look at the entries in the individual distributions.

We need to carry out this computation for every node in the dag. (The A and B above become B_j and A_j, using our earlier notation.) Note that some of the cost can be amortized over the nodes. We need to do the computation of $\Pr(M \geq m)$ for $m \in Val(M)$ and and $\Pr(A_j \leq a)$ for $a \in Val(A_j)$, $j = 1, \ldots, n$ only once, since these probability distributions do not change over the course of the execution. If we precompute these values, then the amount of work at each node is only $O(b_{|B_j|})$ (for sort-merge).

3.6.2 The Case of Nested-Loop Join

As another example, we look at the nested-loop join method. Let $S = \min\{|A|, |B|\}$. The cost formula for $A \bowtie B$ using nested-loop join is:

$$\mathbb{C}(\mathsf{NL}, \mathbf{v}) = \begin{cases} |A| + |B| & \text{if } M \geq S + 2 \\ |A| + |A| \cdot |B| & \text{if } M < S + 2 \end{cases}$$

As in the previous section, we split $\mathbf{E}_{\mathbb{C}}(\mathsf{NL})$ into two terms: $\mathbf{E}_{\mathbb{C}}(\mathsf{NL} : |A| \leq |B|) \Pr(|A| \leq |B|)$ and $\mathbf{E}_{\mathbb{C}}(\mathsf{NL} : |A| > |B|) \Pr(|A| > |B|)$. Again we focus on the first term and leave the second term to the reader. Let $G_a = \mathbf{E}(|B| : a \leq |B|)$, for $a \in Val(|A|)$. Note that the first term can be rewritten as follows:

$$\sum_{a \in Val(|A|)} \Pr(|A| = a) \left(\begin{array}{l} (a + G_a) \Pr(M \geq a + 2) + \\ (a + aG_a) \Pr(M < a + 2) \end{array} \right). \quad (2)$$

As before, we can do the computation in time $O(b_M + b_{|A|} + b_{|B|})$. The procedure is very similar to that for sorted-merge join, so we just sketch the details here.

We again compute $\Pr(M \geq m)$ for each $m \in Val(M)$. This takes $O(b_M)$ steps and enables us to compute $\Pr(M \geq a + 2)$ in a constant number of steps. Next we compute $\Pr(a \leq |B|)$ for each $a \in Val(|A|)$. As before, we can compute all $Val(|A|)$ probabilities in time $O(b_{|A|} + b_{|B|})$. Then we compute $\mathbf{E}(|B| : a \leq |B|)$ for each $a \in Val(|A|)$. Arguments similar to those used in the sort-merge case show that this can be done in time $O(b_{|A|} + b_{|B|})$. We can then compute $\mathbf{E}_{\mathbb{C}}(\mathsf{NL} : |A| \leq |B|)$ using (2). Again each summand only requires constant time, since we already have G_a and we can determine the probabilities involving M in a constant number of steps. Finally, we can compute $\Pr(|A| \leq |B|)$ in time $O(b_{|A|})$, just as in the case of sort-merge.

The whole process again takes time $O(b_M + b_{|A|} + b_{|B|})$, so the algorithm is linear in the (total) number of buckets.

As before, we only considered a single node in the dag. If we precompute $\Pr(M \geq m)$ and $\Pr(B \geq b)$, then the amount of work at each node for nested-loop is $O(b_{|A|})$.

3.6.3 The Distribution of the Result Size

We showed that the expected cost of specific join methods can be computed in time linear to the number of buckets. However, recall that we also need to compute the distribution for the size of the result at each node. This takes time $O(b_{|A|} b_{|B|} b_\sigma)$. Can we do better? Again, for specific distributions, it may be possible. However, we can say more. Suppose for uniformity we decide to have b buckets at every node for each variable. If we have b buckets for each of the variables $|A|$, $|B|$, and σ, then we can get as many as b^3 buckets for the size of the join $A \bowtie B$. To maintain b buckets, we would have to "rebucket" after computing the probability. Instead of rebucketing after doing the computation, we can rebucket each of $|A|$, $|B|$, and σ so that they have $\sqrt[3]{b}$ buckets. Then the whole computation takes time $O(b)$, as desired, and we have b buckets for $|A \bowtie B|$. More generally, if we rebucket each of $|A|$, $|B|$, and σ so that they have $\sqrt[3]{b_{|A|}}$, $\sqrt[3]{b_{|B|}}$, and $\sqrt[3]{b_\sigma}$ buckets, respectively, then we can carry out the computation in time $O(\sqrt[3]{b_{|A|}b_{|B|}b_\sigma}+b_{|A|}+b_{|B|}+b_\sigma) = O(b_{|A|}+b_{|B|}+b_\sigma)$ steps.

3.7 Strategies for Partitioning the Parameter Space

As we have seen, the complexity of all our algorithms to computing or approximating the LEC plan depends on partitioning the parameter space into a number of buckets. A large number of buckets gives a closer approximation to the true probability distribution, leading to a better estimate of the LEC plan. On the other hand, a smaller number of buckets makes the optimization process less expensive. (As we mentioned earlier, the algorithm with one bucket reduces to the standard System R algorithm.) For specific examples (such as Example 1.1), choosing the buckets is straightforward, as a function of the join algorithms being considered and the sizes of the relations being joined. We do not yet have a general mechanism for choosing buckets optimally and efficiently. However, we have insights that should help us explore this issue in future work.

Consider our first two algorithms (Sections 3.2 and 3.3) which used the System R algorithm on a number of parameter settings to generate candidate plans, and then evaluated each of these candidates to find the one of least expected cost. The partitioned parameter distributions are used in two ways: the first is to generate candidate plans (by computing the best plan or c best plans for each parameter value considered) and the second is in computing the actual expected cost of the candidates generated. Different, but related, issues arise in each case. When generating candidates, we are basically interested in determining the region of parameter space in which to search for good candidates. We can partition it coarsely at first, and then generate more candidates in the region of parameter space that seems most likely to contain the best candidates. When computing costs, recall that our goal is to find the candidate of least expected cost. We do not always need an extremely accurate estimate of the cost to do this. We expect to be able to associate a degree of accuracy with each particular partitioning—that is, a guarantee that the estimated expected cost of a plan using this partitioning is within a certain degree of the true expected cost. We may be able to use coarse bucketing to eliminate many plans and then use a more refined bucketing to decide among the remaining few plans.

This insight applies even more to our third algorithm (Section 3.4), which actually computes the LEC plan. If there are j algorithms being compared at a given node in the dag, the expected cost of only one of them (the algorithm of least cost) needs to be computed accurately, since the other plans are pruned. With respect to the pruned plans, we simply need to be satisfied that their expected costs are higher than the chosen plan. Consequently, we can start with a coarse bucketing strategy to do the pruning, and then refine the buckets as necessary.

Moreover, note that we do not have to use the same partitioning strategy at every node. We should use the partition appropriate to the strategies being considered at that node. The cost formulas of the common join algorithms are very simple, at least with respect to a parameter like available memory. As we saw, for fixed relation sizes, the cost for a sort-merge join has one of three possible values, depending on the relationship between the memory and the size of the larger relation; similarly, the cost of a nested-loop join has only one of two possible values. Consequently, if we are considering a sort-merge join (resp., nested-loop join) for fixed relation sizes, we need deal with only three (resp., two) buckets for memory sizes. In general, we expect that we will be able to use features of the cost formulas to reduce the number of buckets needed *on a per-algorithm per-node basis.*

Another way to approach bucketing is to realize that, ultimately, we want to compute the expected cost. Fix a plan P. Note that we can express $\mathbf{E}_{\mathbb{C}}(\mathsf{P})$ as follows:

$$\mathbf{E}_{\mathbb{C}}(\mathsf{P}) = \sum_{c=0}^{\infty} c \Pr(\mathbb{C}(\mathsf{P},\mathbf{v}) = c).$$

For a fixed c, values of $\mathbf{v}$ that yields $\mathbb{C}(\mathsf{P},\mathbf{v}) = c$ are called a *level set* of $\mathbb{C}(\mathsf{P},\mathbf{v})$. If the cost of P has relatively few level sets, then it may be wise to bucket the parameter space with these level sets in mind. Suppose $\mathbb{C}(\mathsf{P},\mathbf{v})$ has ℓ level sets. In principle, we can compute $\mathbf{E}_{\mathbb{C}}(\mathsf{P})$ with ℓ evaluations of the cost function, ℓ multiplications, and $\ell-1$ additions.[4] We can do this if we have the probabilities for each level set. In general the buckets will not correspond to level sets and we may evaluate the cost function many times only to get the same answer each time. So if we bucket the joint distribution by using the level sets (instead of bucket each parameter separately), we can minimize the computation involved in computing the expected cost. The cost function may have many level sets. If we are willing to settle for an approximate answer, we can bucket the range of the function, thereby coalescing some of the level sets. One problem with this approach is that the probability of each level set may not be easy to determine.

4 Concluding Remarks

This paper presents the simple idea of searching for query execution plans with the least expected cost. To the best of our knowledge, this is a new approach to query optimization, departing from the established approach used for the past two decades. Our approach can be viewed as a generalization of the traditional approach in the sense that the traditional approach is essentially our approach restricted to one bucket (in the static case). Although this paper proposes the approach, we are aware that many details need to be worked out. To make our task easier, we have made simplifying assumptions. We now revisit some of these assumptions.

- Our presentation of the System R query optimization algorithm is rather simplistic. The major issue we do not consider is parallelism, which can play a role in two ways (either through bushy join trees or through pipeline parallelism). In both cases, there is an interaction between the parallel components in terms of memory used. While we have ignored this issue, current query optimizers do model it, and we believe the

[4] This is also a lower bound, if we want the exact expected cost (and if we treat the cost function as a black box).

same techniques can be applied to LEC optimization as well.

- In dealing with changes during the execution of the plan, we made the simplifying assumption that no change occurs during any one join "phase". This is clearly an approximation of reality. Further, pipelined joins should be treated together as a single phase while other algorithms (like a sort-merge join) may involve multiple phases. While we have certainly made simplifying assumptions here, we note that our approach at least allows us to tackle a problem not addressed by other works in this area.
- When we considered multiple parameters, we assumed that the parameters were independent. This may not always be a reasonable assumption in practice. It would be of interest to see to what extent we could extend our techniques to situations were there are some dependencies between the variables.

Although there is clearly work that needs to be done before we can use LEC query optimization in production database systems, we believe that it is an approach well worth exploring. We are currently prototyping the algorithm of Section 3.4 to test its benefits against realistic queries and execution environments. Such a prototype will also be useful to investigate the impact of bucket choice (see Section 3.7) on the quality of LEC plans.

References

[Ant93] Gennady Antonshenkov. Dynamic Query Optimization in Rdb/VMS. In *Proceedings of the Ninth IEEE Conference on Data Engineering, Taipei, Taiwan*, pages 538–547, 1993.

[GC94] Goetz Graefe and Richard Cole. Optimization of Dynamic Query Evaluation Plans. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis, MN*, 1994.

[IK90] Y.E. Ioannidis and Y.C. Kang. Randomized Algorithms for Optimizing Large Join Queries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 312–321, 1990.

[Ill94] Illustra Information Technologies, Inc, 1111 Broadway, Suite 2000, Oakland, CA 94607. *Illustra User's Guide*, June 1994.

[INSS92] Y. Ioannidis, R. Ng, K. Shim, and T.K. Sellis. Parametric Query Optimization. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB), Vancouver, Canada*, pages 103–114, 1992.

[KD98] Navin Kabra and David DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Plans. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 106–117, 1998.

[LNSS93] Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science*, pages 195–226, 1993.

[Loh98] Guy Lohman, 1998. Personal communication with Praveen Seshadri.

[Pea88] J. Pearl. *Probabilistic Reasoning in Intelligent Systems.* Morgan Kaufmann, San Francisco, Calif., 1988.

[PHH92] Hamid Pirahesh, Joseph Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, 1992.

[PIHS96] V. Poosala, Y.E. Ioannidis, P.J. Haas, and E.J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 294–305, 1996.

[Res87] M. D. Resnik. *Choices: An Introduction to Decision Theory.* University of Minnesota Press, Minneapolis, 1987.

[SAC+79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD '79 International Conference on Management of Data*, pages 23–34, 1979.

[SBM93] K. D. Seppi, J. W. Barnes, and C. N. Morris. A bayesian approach to database query optimization. *ORSA Journal on Computing*, 5(4):410–419, 1993.

[Sha86] Leonard Shapiro. Join Processing in Database Systems with Large Main Memories. *ACM Transactions on Database Systems*, 11(3):239–264, September 1986.

[Swa89] Arun Swami. Optimization of Large Join Queries: Combining Heuristic and Combinatorial Techniques. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 367–376, 1989.

[UFA98] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 130–141, Seatle, WA, June 1998.