Towards a Robust Query Optimizer: A Principled and Practical Approach

Brian Babcock*
Stanford University
babcock@cs.stanford.edu

Surajit Chaudhuri
Microsoft Research
surajitc@microsoft.com

ABSTRACT

Research on query optimization has focused almost exclusively on reducing query execution time, while important qualities such as consistency and predictability have largely been ignored, even though most database users consider these qualities to be at least as important as raw performance. In this paper, we explore how the query optimization process can be made more robust, focusing on the important subproblem of cardinality estimation. The robust cardinality estimation technique that we propose allows for a user- or application-specified trade-off between performance and predictability, and it captures multi-dimensional correlations while remaining space- and time-efficient.

1. INTRODUCTION

From a system management point of view, the *consistency* and *predictability* of a database management system are very important. This is particularly true since the DBMS is typically just one component of a larger system involving many application programs. Tuning and testing of the system as a whole is greatly simplified when its components behave predictably. Despite the importance of consistency and predictability, these qualities have received relatively little attention from database researchers and implementors, as compared to raw performance considerations. One goal of this paper is to argue that the oft-overlooked qualities that lead to maintainable systems should be formulated as important objectives for database systems. Our results develop this principle in the context of query planning: we illustrate how broader, system-level considerations such as predictability can be incorporated into the query optimization process to produce a more robust query optimizer.

The task of the query optimizer is to select a low-cost query plan, but only incomplete and imprecise information about query plan costs is available to the optimizer at query compilation time. The standard approach to query optimization is as follows: first, generate rough guesses as to the values of the relevant cost model parameters, using rules of thumb or extrapolating from any available statistics. Next, using the rough guesses as inputs, invoke a search algorithm to find the least costly plan. The search phase typically treats the estimated parameter values as though they were completely precise and accurate, rather than the coarse estimates that they actually are. Not surprisingly, query optimization has acquired the dubious repuatation of being something of a black art.

In this paper, we argue for an alternative approach to query optimization. Unlike the standard approach, which ignores the uncertainty about the values of important cost parameters, our approach uses probabilistic reasoning to acknowledge uncertainties in the query planning process in a principled manner. Consequently, our approach is capable of producing query plans that are more robust to estimation errors and changes in the runtime environment.

Our particular focus is on the cardinality estimation phase of query optimization. We propose a cardinality estimation procedure, based on Bayesian inference from precomputed random samples, that avoids the problematic attribute value independence assumption. Recognizing that the goals of predictability and performance may sometimes be at odds, our procedure selects the appropriate trade-off between the two goals based on user or application preferences, expressed at a system-wide and/or query-specific level. Our procedure is compatible with the architecture of existing query optimizers, allowing it to be easily integrated into a commercial database management system.

2. CARDINALITY ESTIMATION

Cardinality estimation is a central subproblem in query optimization. The time that a particular query plan takes to execute is crucially dependent on the sizes of the relations accessed in the query, both the base relations stored on disk and the temporary relations produced at intermediate stages in the query plan. Of course, the sizes of relations produced as intermediate results in a query plan can not generally be computed exactly without first executing the query plan, so in order to produce an estimate for the cost of a query plan, the query optimizer needs to rely on quickly-computed estimates of the sizes of intermediate relations. The problem of producing accurate size estimates for intermediate results is known as the cardinality estimation problem.

Typically, the sizes of base relations are known; the challenging part of cardinality estimation involves estimating the selectivities of the various selection conditions and join predicates in a query. For this reason, we will use the terms "cardinality estimation" and "selectivity estimation" mostly interchangeably[1].

Cardinality estimation is a well-studied problem with a rich literature. Early work on cardinality estimation is surveyed in [23], while [16] surveys more recent approaches. Various estimation techniques have been proposed, including histograms (e.g. [16, 18, 28]), sometimes compressed using discrete wavelet [24], fourier [29], or cosine [20] transformations; sampling (e.g. [14, 15, 22,

*Work performed while visiting Microsoft Research.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

[1]Some cases where cardinality estimation requires more than just a selectivity estimate are discussed in Section 3.5

31]); and parametric methods (e.g. [5]).

The *attribute value independence* (AVI) assumption is a heuristic used by practically all database systems to simplify the cardinality estimation problem. Under the AVI assumption, predicates on different attributes are assumed to be independent of each other. The AVI assumption makes it easy to estimate the selectivity of a conjunction of predicates on multiple attributes: simply take the product of the marginal selectivities of the individual predicates, e.g. $\Pr(A = a \wedge B = b) = \Pr(A = a) \cdot \Pr(B = b)$.

There is no real practical justification for the AVI assumption—it is basically an ad hoc measure that is used because it simplifies the cardinality estimation process considerably, rather than because it accurately models real data. In fact, the AVI assumption is frequently violated in practice, and faulty application of the AVI assumption is arguably the single biggest source of significant query optimizer errors [2, 11, 17, 27]. For a broad class of queries, the one-dimensional histograms used by most modern database systems are insufficient to adequately capture the necessary information about data distributions that would allow the query optimizer to choose satisfactory plans.

Because the shortcomings of the AVI assumption are widely recognized, numerous techniques for modeling correlated multidimensional distributions (e.g., multidimensional histograms [20, 24, 25, 27] and graphical models [9, 12]) have been proposed in the research literature. To the best of our knowledge, none of the abovementioned multidimensional summary techniques have yet been adopted in commercial DMBSs, in part due to their complexity and to the well-known "curse of dimensionality": the number of pairwise and higher-order interactions between attributes is much higher than the number of attributes, so it is only feasible to maintain statistics about a small fraction of these interactions.

A consequence of the curse of dimensionality is that, even when using multidimensional summary statistics, cardinality estimates often exhibit a high degree of imprecision and uncertainty. Despite this fact, most types of summary statistics conventionally used for cardinality estimation only provide a single-point estimate of cardinality without providing any information about the uncertainty of the estimate. This is unfortunate, because knowledge about the degree of uncertainty can be quite important in selecting the most appropriate query plan, as discussed next.

2.1 The Performance/Predictability Tradeoff

Frequently, the query optimizer has a choice of multiple query plans which differ in the degree to which their execution time depends on query selectivity. An example is the choice of the access method used to retrieve records from relation R that satisfy the predicate $(A = a) \wedge (B = b)$, where A and B are two indexed attributes of R. An *index intersection* plan that identifies the qualifying records based on the indexes and then retrieves just those records will perform well if the number of records to be retrieved is low. However, since the index intersection plan requires one random disk read per record, it fares poorly when the selectivity is high. The cost of a *sequential scan* plan, on the other hand, is essentially independent of the query selectivity.

When the query selectivity is uncertain, the decision to select a risky query plan (such as index intersection) that might turn out to be either blazingly fast or agonizingly slow, versus a steady but mediocre alternative (such as sequential scan), should depend on two factors: (1) What is the likelihood of a low selectivity vs. a high selectivity, based on the best available evidence, and (2) What is the risk tolerance of the current database application?

Whether a risky plan is sufficiently likely to be faster than a stabler plan to justify adopting the risky plan depends on the requirements of the database application; different scenarios call for different "standards of proof" for selecting a risky query plan. In some cases, the riskier plan might be preferrable whenever the evidence indicates that it would be more likely than not to be the fastest alternative, while in other situations, the stable plan might be the preferred choice unless the risky plan could be shown "beyond a reasonable doubt" to be faster. In short, there is a tradeoff between predictability and expected performance, and the optimal point in the tradeoff space will vary from one application to another.

For example, a user who is issuing a series of ad hoc, exploratory data analysis queries is likely to prefer that queries be answered as quickly as possible, while being willing to wait if a few of the queries turn out to run slowly. On the other hand, for an application that involves a series of short end user interactions repeated over time, consistent query execution times may be of paramount importance. Users develop expectations about application responsiveness through repeated interactions, and if those expectations are violated, the users are likely to become dissatisfied. A query that occasionally takes significantly longer than usual can lead to the perception of performance problems, even if the execution time is low on average.

2.2 Reasoning About Uncertainty

From the optimizer's point of view, the best case would be for the selectivity estimation process to produce not a point estimate of selectivity, but rather a *probability distribution* over possible selectivities. Such a probability distribution fully quantifies the estimation uncertainty, allowing the optimizer to intelligently select the appropriate query plan after taking into consideration the relative importance of predictability and performance for the current application.

The observation that information is lost when the probability distribution for a parameter (i.e. selectivity) is collapsed to its expected value has been made in previous papers [6, 7, 8, 10, 31]. These papers have advocated selecting the query plan that has *least expected cost*. Because query cost does not necessarily depend linearly on parameters such as selectivity, the plan with least expected cost need not be the same as the least-cost plan for the expected value of the parameter, which is what traditional query optimizers aim to produce. A major goal of the papers [6, 7, 10] is to leverage the additional information present in a probability distribution while keeping the basic structure of existing query optimizers.

The query optimizer in a modern commercial database system is a sophisticated and extremely complex software artifact, the product of many person-years of labor. Because the development of an industrial-strength query optimizer is so expensive, query optimizer modifications that can be easily incorporated into existing optimizers are greatly preferrable to modifications that require wholesale restructuring of the optimizer. Previous approaches that modeled uncertain parameters using probability distributions treated the existing query optimizer as a black box that is invoked multiple times as a subroutine, using different parameter values on each invocation [7, 10]. Such an approach is faithful to the goal of minimizing disruption, but it results in a blowup in optimization time by a factor equal to the number of subroutine invocations.

In this work, we adopt a different approach: after deriving a probability distribution for selectivity, the next step in cardinality estimation is to interpret the distribution in light of user preferences about the predictability vs. performance tradeoff to produce a single-value cardinality estimate, suitable for consumption by an existing query optimizer. Our solution (described in the next section) avoids the AVI assumption, does not suffer from the "curse of dimensionality" and has minimal impact on optimization time.

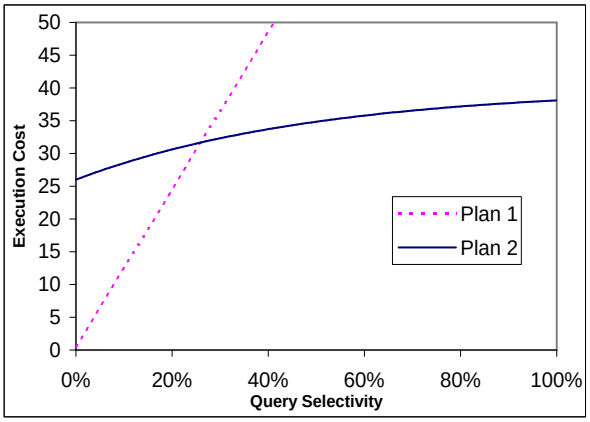


Figure 1: Execution Costs for Two Hypothetical Plans

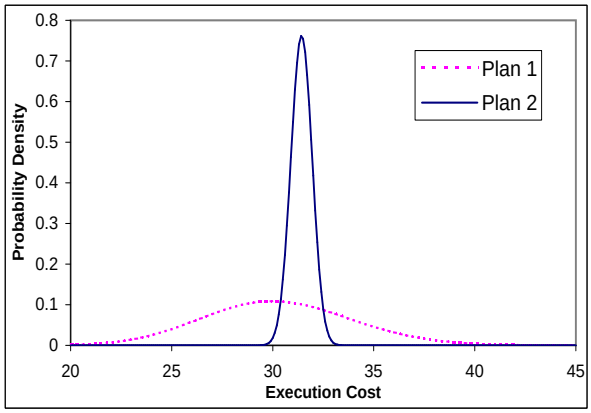


Figure 2: Probability Density Function for Execution Cost

3. MAKING UNCERTAINTY EXPLICIT

The preceding discussion motivates the desirability of quantifying estimation uncertainty and incorporating knowledge about this uncertainty in the query optimization process, but it leaves unanswered two questions: how can knowledge of the probability distribution for selectivity be used to improve the query optimizer, and how can such a probability distribution be estimated? These questions are addressed in the next three subsections. Section 3.1 demonstrates that taking advantage of knowledge about uncertainty need not require fundamental changes to the architecture of the query optimizer. Section 3.2 discusses the advantages of using precomputed uniform random samples as concise summaries of the data for the purposes of cardinality estimation. Finally, Section 3.3 shows how a random sample of the data can be used to describe the selectivity of a query predicate as a probability distribution, rather than merely providing a single-point estimate of the selectivity.

3.1 Incorporating the Probability Distribution

We will illustrate our technique for deriving a single-value estimate from the distribution of possible selectivities using an example. Suppose the query optimizer is trying to decide between two alternative query execution plans for some query Q. The two plans have different degrees of dependency on the (unknown) query selectivity. This situation is depicted graphically in Figure 1, which plots the execution cost of two hypothetical query plans, labeled "Plan 1" and "Plan 2", as a function of the query selectivity. As can be seen from the figure, Plan 1 is the lower-cost plan if the query selectivity is less than 26%, whereas Plan 2 is preferrable if the query selectivity is greater than 26%. Of course, the query optimizer cannot know in advance what is the exact selectivity of a given query, because it must rely on imprecise, quickly-computed selectivity estimates. We refer to a selectivity value (such as 26% in this example) where execution cost functions for two query plans cross, causing the optimal query plan to change, as a *crossover point*. In general, a query can have many crossover points.

Let us assume for a moment that the optimizer is able to derive probability distributions for the execution cost of each query plan for Q. The use of a distribution over possible cost values, rather than a single fixed cost, reflects the fact that the optimizer is uncertain about the query selectivity, an important cost estimation parameter. The probability distribution for an uncertain quantity Y can be represented by its probability density function $f(Y)$, which has the property that $\Pr[a \leq Y \leq b] = \int_a^b f(z)dz$. The probability density functions for the execution costs of the two alternative plans for query Q are shown in Figure 2. (Section 3.1.1 explains how Figure 2 is derived based on statistics collected by the query optimizer and knowledge of the execution cost functions for each query plan.) As can be seen from Figure 2, the optimizer's uncertainty about the actual selectivity has a much more pronounced impact on Plan 1's execution cost than on the cost of Plan 2. This is because the execution cost of Plan 1 is more dependent on query selectivity compared to the cost of Plan 2, as can be observed from Figure 1. Based on Figure 2, we can conclude that the cost of Plan 2 will almost certainly be between 30 and 33, whereas the cost of Plan 1 might be as low as 20 or as high as 40.

Whether Plan 1 or Plan 2 would be a better choice in this scenario depends on the relative importance that the database user or application places on *expected cost* vs. *predictability of cost*. If minimizing expected cost is the overriding concern, then Plan 1 should be selected. However, for users that are more risk-averse, Plan 2 may be preferrable, since the slightly higher expected cost of Plan 2 may be compensated for by its greater predictability, which reduces the risk that the query will take much longer than expected to execute. Essentially, the decision comes down to which part of the probability distribution is more important: should the focus be on the middle part of the distribution (i.e. the "typical" behavior), or the right-hand tail of the distribution (i.e. the "realistic worst-case" behavior)?

In our approach to cardinality estimation, the desired tradeoff between performance and predictability is expressed by means of an user- or application-specified parameter called the *confidence threshold*. The confidence threshold specifies what percentile value of the query execution cost probability distribution to look at when comparing alternative query plans, giving a way to condense the probability distribution to a single cost value. For example, when using a confidence threshold of 50%, query plans are ranked according to the median value of their cost distributions, meaning the estimated cost of Plan 1 would be 30.2 and the estimated cost of Plan 2 would be 31.5. Using a confidence threshold of 80%, on the other hand, would result in cost estimates of 33.5 for Plan 1 and 31.9 for Plan 2. This is because there is an 80% chance that the cost of Plan 1 will be 33.5 or less, and there is an 80% chance that the cost of Plan 2 will be 31.9 or less. The name "confidence threshold" reflects the fact that, when using a confidence threshold of $T\%$, cost estimates are assigned in such a way that the optimizer is $T\%$ confident that the actual cost of using a particular query plan will not exceed the optimizer's estimated cost for that plan. Increasing the confidence threshold causes the query optimizer to adopt a more conservative strategy, while decreasing the threshold makes

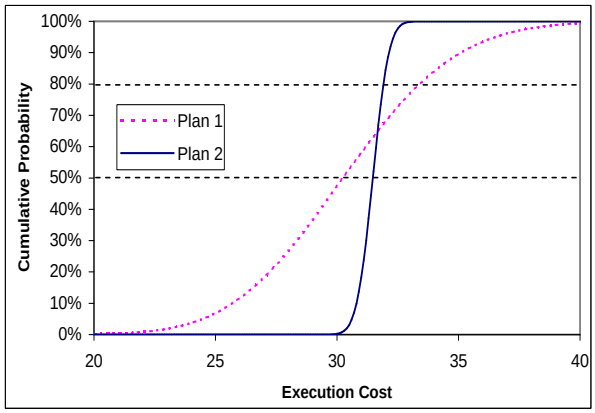


Figure 3: Cumulative Probability for Execution Cost

the optimizer behave more aggressively.

The confidence threshold can be alternatively (and equivalently) described in terms of the cumulative distribution function (cdf) for query execution cost. (The cdf for a probability distribution with density $f(Y)$ is defined as $cdf(Y) = \int_0^Y f(z)dz$.) If the confidence threshold is $T\%$, then the single-value cost estimate c for a given query plan is $c = cdf^{-1}(T\%)$. In other words, the cost estimate is derived by inverting the cdf for query execution cost, as illustrated graphically in Figure 3. The horizontal dashed lines in Figure 3 indicate the effect of confidence thresholds of 50% and 80%, and the points at which the cdf curve for each query plan crosses the dashed lines indicate the cost estimates that would be used for each query plan at each confidence threshold. As can be seen from Figure 3, Plan 1 would be preferred at confidence thresholds less than 65% whereas Plan 2 would be preferred at confidence thresholds greater than 65%.

3.1.1 *Derivation of Execution Cost Distribution*

The probability density function for the execution cost of a query plan can be derived from the following two pieces of information: (1) the probability density function $f(s)$ for the query selectivity s, which quantifies the imprecision inherent in selectivity estimation; and (2) the execution cost $c = g(s)$ for each potential query plan, expressed as a function of the selectivity s. From these, we can easily derive a probability density function $f^*(c)$ for the execution cost of each query plan, through a simple change of variable: $f^*(c) = f(g^{-1}(c))$.

Our technique for generating a probability density function for selectivity uses precomputed random samples and is the topic of Section 3.3. The selectivity distribution used in deriving Figure 2 was generated assuming that selectivity estimation was performed with the aid of a random sample of 200 tuples, out of which 50 tuples satisfied the query predicates.

The execution cost functions for each query plan are available in implicit form through the cost estimation module. However, usually the cost estimation module does not expose its cost functions in explicit, invertible functional form. Therefore, performing the change of variable to derive a probability distribution for execution cost from the probability distribution for selectivity could be an expensive task. Fortunately, under the assumption that query execution cost is a monotonically increasing function of selectivity, there is no need to actually derive the probability distribution for the execution cost of each query plan.[2] Instead, given a confidence threshold of $T\%$, it suffices to invert the cdf for selectivity to determine the selectivity value s' such that $cdf(s') = T\%$, and then invoke the cost estimation module once to compute $c' = g(s')$. This is how our selectivity estimation procedure is implemented. It can be shown that the resulting cost c' is the same as would be calculated by the more roundabout procedure of explicitly deriving the cdf for execution cost and then inverting it.

As a consequence, the changes necessary to incorporate our cardinality estimation procedure into a conventional database system can be entirely isolated within the cardinality estimation module. Other aspects of the query optimizer, such as plan enumeration, cost estimation, and search via dynamic programming, remain unchanged and need not be aware of probability distributions at all.

3.2 Selectivity Estimation via Sampling

The selectivity estimation technique discussed in this paper performs estimation using uniform random samples of the relations in the database. In contrast to previous sampling-based approaches, which estimate selectivity based on samples that are constructed *on the fly* at query execution time ([14, 15, 22]), the technique we describe uses *precomputed* random samples of a fixed size (a few hundred tuples). The main innovation in our work is a novel technique for *interpreting* a random sample that improves the robustness of the query optimization process.

Random sampling has four characteristics that set it apart from most selectivity estimation techniques:

1. It does not use the AVI assumption, and therefore it avoids degradation in estimation quality due to the propagation of estimation errors.
2. It avoids the "curse of dimensionality": the dimensionality of the data does not affect the accuracy of random sampling, and the space required to store the sample grows linearly with the number of attributes.
3. It is not restricted to equality and range predicates, but rather works for almost any type of query predicate, including arithmetic expressions, substring matches, etc.
4. It is simple to implement.

The random sampling procedure that we use has two phases, an offline precomputation phase and an estimation phase. The precomputation phase is analogous to histogram construction; it can be triggered manually through an `UPDATE STATISTICS` command in SQL or performed periodically whenever a sufficient number of database modifications have occured. The estimation phase takes place during query optimization: as the query optimizer explores the space of query plans via a series of transformations, the cardinality estimation module is invoked for each relational subexpression that is considered by the optimizer. In this paper, we discuss cardinality estimation for select-project-join (SPJ) expressions where all joins are foreign-key joins. Extending our techniques to work with the full generality of SQL is a direction for future work.

While using a uniform random sample to estimate the result cardinality of a single-table selection query is quite straightforward, the case of SPJ expressions involving joins is somewhat more complicated. The natural approach of creating independent samples of each relation and then evaluating the SPJ expression on the samples

[2]The assumption that query execution cost goes up as selectivity goes up may seem to be violated by certain negating operators such as set difference or antisemijoin. However, we really only require the weaker assumption that for each query operator, the estimated execution cost of the operator increases monotonically as the cardinality of its inputs increases (i.e. bigger inputs take longer to process, all other things being equal), which is a reasonable assumption for all standard relational operators.

does not work well (see [1]). Instead, we need to use the technique introduced in [1] of creating *join synopses* for each relation that has one or more foreign keys to other relations. Briefly, the join synopsis for relation R is constructed as follows:

1. Construct a uniform random sample of R using any of the known methods for sampling from databases [26].
2. For every relation S such that R has a foreign key to S, join the sample of R with the full relation S.
3. Repeat Step 2 recursively, i.e., for each relation S from Step 2, follow all its foreign keys, and so on.

We will assume acyclic join graphs, and therefore the join resulting from the above procedure is well-defined. The join synopsis for R consists of the the result of the join query thus defined. For any foreign-key join rooted at R, one can construct a uniform random sample of the join result by simply taking the appropriate projection of the join synopsis for R.

To summarize: In the precomputation phase, we construct join synopses for each relation in the database. During query optimization, the optimizer will request estimates for the cardinality of various relational expressions, which we assume are SPJ expressions with only foreign-key joins. For each such expression, we determine the root relation R (the one whose primary key is not involved in a join), and evaluate the expression on the join synopsis for R, counting how many tuples satisfy the expression. The number of satisfying tuples, and the overall number of tuples in the sample, give rise to a probability distribution for selectivity, as described in Section 3.3.

A major benefit of cardinality estimation using sampling is that the cardinality of each query expression can be directly estimated from a single sample, rather than by combining uncertain cardinality estimates for subexpressions. An example will illustrate this point. Consider the query $A \bowtie B \bowtie C$, possibly with some selection conditions on each of the relations A, B, and C. To optimize this query, the optimizer needs to estimate the selectivities of seven logical expressions: A, B, C, $A \bowtie B$, $A \bowtie C$, $B \bowtie C$, and $A \bowtie B \bowtie C$. When using histograms for cardinality estimation, estimates for the single-table expressions A, B, and C are computed directly, and estimates for the multi-table expressions are built up from the single-table estimates using the AVI assumption. When using sampling, the cardinality estimates for all seven expressions are computed directly from samples. Assume A has a foreign key to B which has a foreign key to C. Then the sample for A is used to estimate the selectivity of expressions A, $A \bowtie B$, $A \bowtie C$, and $A \bowtie B \bowtie C$; the sample for B is used for expressions B and $B \bowtie C$; and the sample for C is used for the expression C. When using histogram-based techniques, the errors introduced by the AVI assumption are exponentially magnified as they are propagated across subresults [17]. When using random sampling, by contrast, *no build-up of estimation errors occurs*, because the cardinality estimates for different subresults are computed independently from one another.

Our use of sampling as an estimation technique in this paper is due to the advantages mentioned above, and also because a probability distribution can be derived from a random sample in a principled manner using Bayes's rule, as discussed in the next subsection. We emphasize, however, that our decision to use sampling is orthogonal to the main contribution of this paper, which is our procedure for interpreting a probability distribution for selectivity in light of user preferences about performance vs. predictability, as described in Section 3.1. In principle, the same robust estimation procedure could be applied to a probability distribution generated using any cardinality estimation technique.

3.3 Deriving the Probability Distribution

Consider a database consisting of N tuples and a sample $S = s_1, s_2, \ldots, s_n$ consisting of n tuples chosen uniformly at random, with replacement, from the database. Suppose that P is a query predicate that is satisfied by pN tuples, i.e. a fraction p of the database. Let x_i denote the indicator variable that is equal to 1 if the ith sample tuple s_i satisfies the predicate P and 0 otherwise, and define X as the vector $\langle x_1, x_2, \ldots, x_n \rangle$. In the process of selectivity estimation, we observe X and attempt to infer the value of p, which is unknown to us. We will treat the unknown quantity p as a random variable and seek to determine the conditional probability distribution for p, given the observed data X. In other words, we seek the *conditional density function* $f(z|X)$, which we can integrate to determine the probability that the query selectivity falls in a particular range, given the observed data X: $Pr[(a \leq p \leq b)|X] = \int_a^b f(z|X)dz$. To calculate the conditional density, we can use Bayes's rule:

$$f(z|X) = \frac{\mathtt{Pr}[X|p = z]f(z)}{\int_0^1 \mathtt{Pr}[X|p = y]f(y)dy} \tag{1}$$

Notice that to calculate $f(z|X)$ using Bayes's rule, we need to know $f(z)$ (sometimes called the *prior probability*). If we have some prior knowledge about the query workload, we may be able to use that knowledge to estimate $f(z)$. When prior knowledge of the workload is lacking, as is often the case for database systems, a reasonable approach is to assume that all query selectivities are equally likely *a priori* and adopt the uniform prior distribution $f(z) = 1$ for $0 \leq z \leq 1$.

An alternative technique that can be applied in the absence of knowledge about the actual distribution of query selectivities is to choose a non-informative prior distribution based on Jeffreys's rule [19]. In the context of selectivity estimation from a random sample, the Jeffreys prior is the beta distribution with shape parameters $(\frac{1}{2}, \frac{1}{2})$, i.e. $f(z) \propto z^{-1/2}(1-z)^{-1/2}$ (see [3], page 315). Because the Jeffreys prior is the most widely accepted non-informative prior among statisticians [3, 21], for the rest of this paper, we will use the Jeffreys prior unless otherwise stated. In any case, the exact prior distribution chosen has little impact on selectivity estimation, as discussed in Section 3.4.

The terms from Equation (1) other than the prior probability $f(z)$ are straightforward to compute. Suppose that that k tuples from the sample satisfy the predicate P, i.e. $\sum_{i=1}^n x_i = k$. Because the sample tuples are selected independently and uniformly at random from a population of tuples in which a fraction p satisfy the query predicate and a fraction $1 - p$ do not, the variables x_i are independent identically distributed Bernoulli random variables and therefore $\mathtt{Pr}[X|p = z] = z^k(1 - z)^{n-k}$. The quantity in the denominator of (1), $\int_0^1 \mathtt{Pr}[X|p = y]f(y)dy$, is independent of z, so it can be treated as a normalizing constant.

Combining the expression for $f(z)$ with the expression for $\mathtt{Pr}[X|p = z]$ and normalizing yields the following formula for the probability density of p conditioned on X:

$$f(z|X) = \frac{z^{k-1/2}(1 - z)^{n-k-1/2}}{\int_0^1 y^{k-1/2}(y - z)^{n-k-1/2}dy} \tag{2}$$

This is just the beta distribution with shape parameters $(k + \frac{1}{2}, n - k + \frac{1}{2})$.

3.4 Summary of the Estimation Procedure

In the course of query optimization, the optimizer searches through many possible query plans in search of the optimal plan. During this search, the optimizer makes a number of subroutine calls to

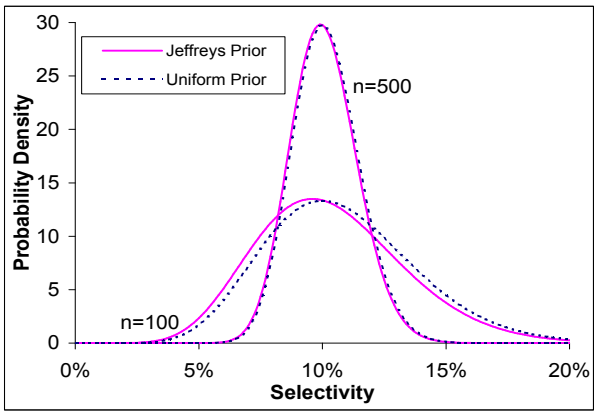


Figure 4: Sample Size Matters, Prior Doesn't

the cardinality estimation module to estimate the size of various intermediate query results. We modify the cardinality estimation module to estimate the selectivity of each query predicate using the following procedure:

1. Determine the appropriate precomputed random sample of the data to use, based on the relations involved in the query expression.
2. Evaluate the predicate on the sample and use Bayes's rule to infer a probability distribution for the actual selectivity given the selectivity observed for the sample.
3. Choose the proper confidence threshold $T\%$ based on user preferences and compute the selectivity $s = cdf^{-1}(T\%)$ using the inferred probability distribution.
4. Return s as the estimated selectivity of the predicate.

The following example illustrates the above estimation procedure and sheds light on the impact of the sample size and the choice of prior distribution on selectivity estimation. Suppose that 10 tuples from a 100-tuple sample satisfy the query predicate for some query Q. Equation (2) tells us that the probability density function for the query selectivity $f(z|X)$ is proportional to $z^{9.5}(1-z)^{89.5}$. If we had elected to use the uniform distribution as our prior distribution instead of the Jeffreys prior, the resulting probability density would be slightly different.

Figure 4 shows the density functions for the probability distributions resulting from the two different priors in this scenario. The same figure also illustrates the probability distributions that would result from using the uniform or Jeffreys prior to interpret a sample of 500 tuples, of which 50 satisfy the query predicate. As can be seen from Figure 4, the uniform and Jeffreys priors produce almost identical results. The size of the sample, however, does have a noticable impact on the probability distribution produced.

The selectivity estimate that is produced will depend on the choice of confidence threshold. If a confidence threshold of 20% is used, then the result of selectivity estimation will be 7.8%, because there is a 20% chance that the query selectivity is 7.8% or less. Confidence thresholds of 50% or 80% result in selectivity estimates of 10.1% and 12.8%, respectively. Note that while varying the confidence threshold changes the selectivity estimate that is produced, this does not necessarily translate into a different query plan being selected by the query optimizer. Whether a difference in estimated selectivity translates into a different choice of query plan depends on the location of the crossover points between different potentially optimal query plans.

3.5 Extensions to the Query Model

As mentioned earlier, in this paper, we assume queries are SPJ expressions with foreign-key joins, and we further assume that join synopses are available for all tables referenced in the queries. In this section, we briefly discuss how our techniques can be adjusted to allow these assumptions to be relaxed. A more detailed study of these extensions is deferred to future work.

No statistics available. In some cases, it is possible that full join synopses will not be constructed for all relations in a query. This could be because only a limited set of join synopses are constructed due to concerns about storage or maintenance overheads, or it could be due to queries involving non-foreign-key joins. There are several possibilities for handling queries for which the necessary samples are not available. First, it may be that a particular join synopsis is not available, but separate single-table samples are available for each of the tables being joined. In this case, traditional estimation techniques can be used: the selectivity of the predicates on each table can be estimated separately using the sample for that table, and the combined selectivity of all predicates can be estimated using the attribute value independence assumption and the containment assumption [30]. (Of course, to the extent that these assumptions are violated, this approach may lead to inaccurate estimates.) Second, it may be that no statistics whatsoever are available for a particular table or column referenced by a query predicate. In this case, many DBMSs use an arbitrary constant, or "magic number", as the selectivity estimate [30]. This same approach can be used with our technique, with the following possible extension: instead a single "magic number", a "magic distribution" can be used, which has the effect of varying the magic number depending on the confidence threshold.

In either case, estimation errors caused by questionable optimizer assumptions may occur, but the error will be confined to cardinality estimates for those subexpressions for which adequate samples are not available. Cardinality estimates for other subexpressions in the same query can continue to use our regular estimation technique.

Incorporating other operators. The result size for certain SQL operators, such as aggregation with `GROUP BY`, depends on the number of distinct combinations of attribute values for some set of grouping attributes. Estimating the number of distinct values in a relation from a random sample of that relation is a well-studied problem. Our cardinality estimation procedure can be extended to perform distinct value estimation by adapting known distinct-values estimation techniques (e.g. [13]).

4. RELATED WORK

In this section, we compare our approach with the most closely related prior research. (A broader overview of related work is found in Section 2.) Among previous research that we are aware of, the work by Chu, Halpern, and Gehrke [6] is most similar in spirit to this paper. Like this paper, the work of Chu et al. argues for taking a probabilistic view of uncertain cost parameters and also makes the point that optimizing solely for expected cost, without considering predictability, may not meet the needs of system users. The approach suggested in [6] is to optimize for a non-standard utility function (such as a linear combination of expected cost and estimated variance) that has a non-linear relation to query execution cost. However, as is pointed out in [6], standard dynamic programming search strategies break down when the utility of a query plan is not a linear function of the utility of subplans. Thus, in contrast to our approach, implementing the proposal of [6] would likely require significant changes to existing query optimizer technology.

Another difference between our work and [6] is that the discussion in [6] is at a rather high level, so many practical implementation details are not addressed. For example, [6] does not discuss how a probability distribution for selectivity can be obtained; it provides no guidance about how an appropriate utility function should be selected or specified by the user; and it provides no experimental evidence for the effectiveness of its proposed approach. In that sense, this paper can be viewed as a concrete and practical application of the philosophy espoused in [6].

The use of join synopses as data summaries was first proposed by Acharya, Gibbons, Poosala, and Ramaswamy [1] in the context of approximate query processing. Our work differs from [1] in several ways: our focus is query optimization rather than approximate answering of aggregate queries; we make predictions using a Bayesian approach rather than using the standard maximum likelihood estimator; and we aim to balance the twin objectives of minimizing the expected value and the variance of query execution time, instead of of trying to minimize relative error in estimation.

Seppi, Barnes, and Morris [31] take a Bayesian approach to parameter estimation from a random sample which is similar to our proposal in Section 3.3. However, the focus of [31] is the application of preposterior analysis to determine whether on-the-fly sampling to gather additional statistics is cost-effective. In contrast to [31], our work considers the trade-off between predictability and performance; we use precomputed samples rather than sampling on the fly;we do not assume a specific execution cost function; and we discuss how our techniques can be practically implemented and provide an experimental evaluation of their effectiveness.

5. ANALYSIS

In this section, we undertake an analytical exploration of our cardinality estimation procedure with the goal of understanding the effects that certain parameters of the estimation procedure have on query planning outcomes.

Our experiments with an actual database system, described in Section 6, consider queries of varying complexity; however, in this section, to keep the analysis simple, we focus on a simple single-table query based on the example query from Section 2.1. This query has two different optimal plans—index intersection or sequential scan—in two different selectivity ranges.

5.1 Analytical Model

Consider a single-table query Q running against a table with N rows. Suppose that either of two execution plans, P_1 and P_2, might be optimal for Q, depending on the query selectivity p. Assume a simple linear cost model, so that the execution time for query plan P_i has the form $v_i x + f_i$, where $x = pN$ is the number of tuples satisfying the query predicate, v_i is the incremental cost per tuple for plan P_i, and f_i is the fixed overhead, independent of query selectivity, for plan P_i. We will choose $N = 6,000,000$, $f_1 = 35$, $v_1 = 3.5 \times 10^{-6}$, $f_2 = 5$, and $v_2 = 3.5 \times 10^{-3}$. These values are chosen empirically to make plans P_1 and P_2 roughly resemble a sequential scan plan and an index intersection plan, respectively. The crossover point where plan P_1 becomes better than plan P_2 is at a selectivity of $p_c = \frac{f_1 - f_2}{v_2 N - v_1 N} \approx 0.14\%$.

Suppose selectivity estimation is performed using the technique described in Section 3, using a uniform random sample of n tuples, interpreted with a confidence threshold of $T\%$. Whether plan P_1 or P_2 will be selected depends on the composition of the random sample—specifically, it depends on how many of the tuples in the sample satisfy the query predicate. Suppose that query Q has selectivity p. This means that each tuple in the sample will satisfy the predicate with probability p and fail to satisfy the predicate with probability $1 - p$. Therefore, the number of tuples that satisfy the query predicate will be binomially distributed: the probability that k of the n tuples in the sample satisfy the predicate is $B(n,k) = \binom{n}{k} k^p (1-k)^{n-p}$.

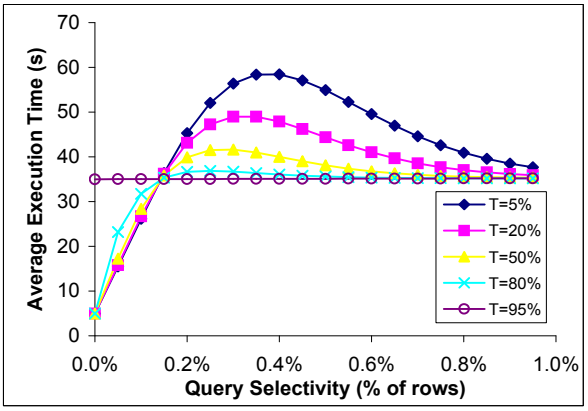


Figure 5: Effect of the Confidence Threshold

When k of n tuples satisfy the predicate, the cdf of the inferred probability distribution for selectivity is the cumulative beta distribution with shape parameters $(k + 1/2, n - k + 1/2)$. Evaluating the inverse of this function at the value $T\%$ gives the selectivity estimate that will be returned when k out of n sample tuples satisfy the query predicate and the confidence threshold is $T\%$. If this selectivity estimate is greater than p_c, then the optimizer will choose plan P_1, and otherwise it will choose plan P_2. In this way, for a fixed sample size n and confidence threshold $T\%$, we can determine which values of k correspond to plan P_1 and which to plan P_2. We know the probability that each value of k will occur, given the actual query selectivity p, so we can compute the chance that plan P_i will be selected by summing the probabilities of the values corresponding to plan P_i.

5.2 Analytical Results

Through our analysis, we seek to understand how varying the confidence threshold, sample size, and query selectivity impacts the query optimizer.

The most commonly used metric for evaluating cardinality estimation techniques in the research literature is relative error (i.e. the percentage difference between the actual and estimated cardinality). Although the relative error in cardinality estimates is a natural choice as an error metric, within the context of query optmization, a more appropriate metric exists. The best way to evaluate the effectiveness of various estimation techniques is to directly measure query optimization performance by comparing the running times of the query plans that are produced using each technique.

Since our goal is a robust query optimizer, we are interested not just in raw performance but also in the consistency and predictability of the database system. A reasonable metric for the predictability of the query execution engine is the variance in query execution times over a set of similar queries. For this reason, in evaluating cardinality estimation performance, we report not only the average query execution time, but also the variance in query execution time across the queries making up each experiment.

5.2.1 Confidence Threshold

Figure 5 shows the expected value of the query execution time for queries of different selectivities when the query plan is chosen using a random sample of 1000 tuples and the technique described

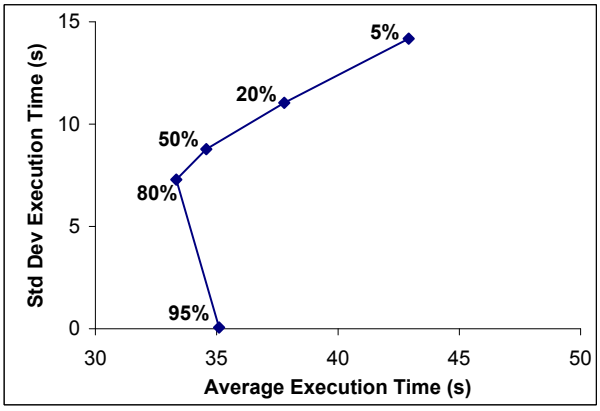


Figure 6: Performance vs. Predictability Trade-off

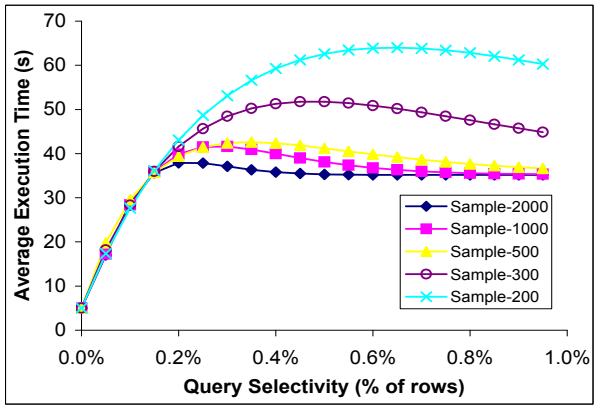


Figure 7: Effect of Sample Size

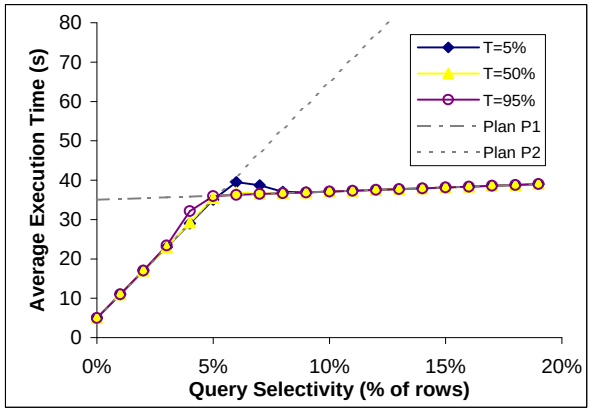


Figure 8: Crossover Point at Higher Selectivity

in Section 3. Each of the curves plotted in Figure 5 is a mixture of the curves for plan P_1 and plan P_2. If the query optimizer knew the actual query selectivity exactly, it would always choose plan P_2 whenever the selectivity p is less than the crossover point p_c and plan P_1 when $p > p_c \approx 0.14\%$. Since cardinality estimation must be done very quickly and cannot be exact, two kinds of errors can result: overestimating the selectivity can cause the optimizer to choose P_1 when P_2 would be better, and conversely underestimating the selectivity can cause the optimizer to choose P_2 when P_1 would be better.

As can be seen from Figure 5, higher values of the confidence threshold $T\%$ make the query optimizer more prone to overestimation, while lower values of the confidence threshold make the optimizer more prone to underestimation. The confidence threshold $T = 95\%$ is an extreme case: even when zero tuples out of a sample of 1000 satisfy the predicate, there is still at least a 5% chance that the query selectivity exceeds the crossover point p_c. Therefore, for the cost model we are using, the optimizer will never select the riskier plan p_1 when $T = 95\%$, because it can never be 95% sure that p_1 is better.

The query selectivities plotted in Figure 5 range from 0% to 1% at multiples of 0.05%. Figure 6 summarizes the performance of each setting of the confidence threshold by comparing the standard deviation of the query execution time with the average query execution time, under the assumption that any of the selectivities from Figure 5 is equally likely to occur. This figure gives us a condensed way to visualize the various performance vs. predictability tradeoffs that can be achieved by picking a particular confidence threshold and using it to optimize a set of queries with varying selectivities. As the figure shows, the higher the confidence threshold is set, the less variability occurs in the query execution time. This is because higher confidence thresholds favor plans whose running times are relatively independent of selectivity. A second observation suggested by Figure 6 is that moderate settings of the confidence threshold are better than extremely high or low settings at producing low expected execution times. Interestingly, the lowest average execution time occurs not when the confidence threshold is at the unbiased setting of 50%, but rather at the higher 80% level.

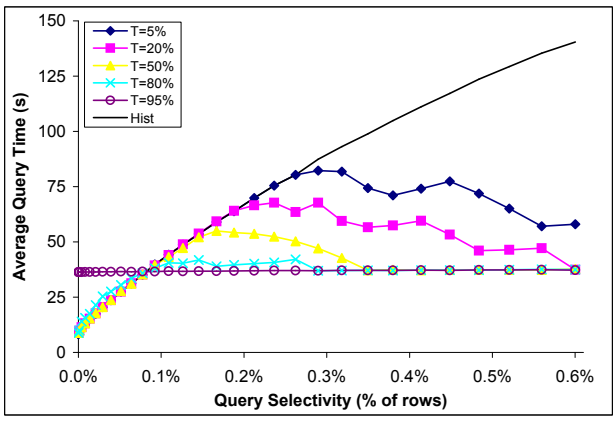
5.2.2 *Sample Size*

Figure 7, like Figure 5, shows the expected value of the query execution time for queries of different selectivities. In Figure 7, the confidence threshold is held constant at 50% while the sample size is varied. As one would expect, larger sample sizes lead to more precise cardinality estimates, and consequently lower query execution time. On the other hand, because the overhead of cardinality estimation is dependent on the size of the sample that is used, it is desirable for the sample size to be as small as possible. Figure 7 suggests that, at least for queries with crossover selectivities similar to the one in our analytical model, a sample size of 500 achieves a good tradeoff between these competing objectives. If the sample size is much smaller than 500 tuples, there is a significant decrease in query performance, while little additional benefit is gained by increasing the sample size beyond 500 tuples.

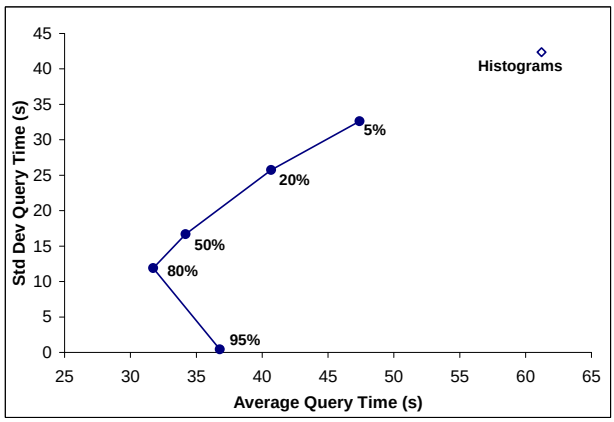
5.2.3 *Crossover Selectivity*

The analytical results reported thus far have all used the same cost model with a single crossover point at a rather low selectivity, $p_c \approx 0.14\%$. Figure 8 shows what happens if the cost model is perturbed so that the crossover point occurs at a significantly higher selectivity, $p_c' \approx 5.2\%$.

Figure 8 suggests that when the selectivity at which a crossover occurs is large enough, sampling-based estimation techniques work very well, regardless of the setting of the confidence threshold $T\%$. There are two reasons for this strong performance: first, when the crossover point occurs at a relatively large selectivity, there is less difference in the slope of the plots for query plans p_1 and p_2, meaning that the impact of making an incorrect choice is less. Second, larger selectivities are easier to estimate accurately, so the relative estimation error is less when the selectivity is higher. The combination of more accurate estimates and less penalty for mistakes causes cardinality estimation to be a fairly easy problem when query selec-



(a) Selectivity vs. Time



(b) Performance vs. Predicatability

Figure 9: Two-Predicate `lineitem` Query

tivity is high. Therefore, in our experimental evaluation, we concentrate on low-selectivity queries, since it is much more difficult to achieve good estimation performance on low-selectivity queries.

6. EXPERIMENTS

In order to try out the ideas proposed in this paper in an actual system, we modified the query optimizer of a commercial DBMS, Microsoft SQL Server, replacing the existing histogram-based cardinality estimation module with our proposed sample-based technique. We sought to answer several empirical questions:

- How does the estimation overhead of our proposed robust estimation procedure compare with traditional cardinality estimation techniques?
- Is the estimation procedure effective for complex queries that have many possible query plans?
- What guidelines are available to assist in the choice of the confidence threshold $T\%$?

6.1 Estimation Overhead

We chose to use 500-tuple samples for our experiments. This choice of sample size was motivated by two considerations: (1) the analysis from Section 5.2.2, and (2) the desire to achieve approximate parity with pre-existing histogram-based estimation modules, in terms of storage space for summary data and time spent during cardinality estimation. (We conducted one experiment in which we varied the sample size. See Section 6.2.4 for details.)

The unmodified version of the commercial database system that we used for our experiments normally uses histograms of approximately 250 buckets in size. Each histogram bucket stores an attribute value, along with counts of the number of records and distinct values in the bucket. In a random sample, only attribute values are stored—no counters are necessary. Therefore, if we assume 4-byte counters and 8-byte attribute values, then a 500-tuple of a relation uses the same amount of space as 250-bucket histograms on each attribute of the relation. In practice, it is often the case that histograms are built on only some of attributes of a relation [4]. (For example, attributes that are rarely used in selection predicates may not need histograms.) However, there is an analogous space-saving opportunity available to sampling-based estimation techniques: rarely-queried columns can be omitted from the random sample.

We found that the time spent in query optimization was about 30%–40% (on the order of tens of milliseconds) more when using our cardinality estimation technique than when using standard histograms. Because our implementation is a rough research prototype—it lacks even basic optimizations such as memoizing, and it retains all of the histogram bookkeeping code, even though the histograms are not used during cardinality estimation—we expect that an optimized implementation would have significantly less overhead.

6.2 Experimental Results

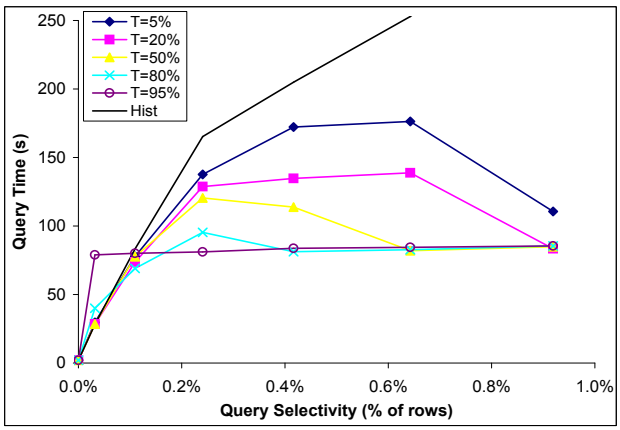
We applied our cardinality estimation technique to three different query scenarios: a single-table aggregation query with two selection predicates; a three-way join query; and a four-way "star join" between a fact table and three dimension tables, with selection predicates on each dimension table. The first two query scenarios used the data set from the TPC-H benchmark dataset [32] at scale factor 1 (approximately 1 GB of data), while the third scenario used a synthetic data set consisting of a star schema with a 10 million row fact table and three small dimension tables, each with 1000 rows. For query scenario #2, we modified the `part` table of TPC-H to introduce a correlated data distribution. Each query scenario used a fixed query template with one free parameter that could be varied to control the query selectivity by changing the degree of correlation between individual query predicates. The marginal selectivity of each individual predicate (i.e. the information tracked by histograms) remained constant regardless of the setting of the free parameter. In each scenario, we tried out five different settings for the confidence threshold (5%, 20%, 50%, 80%, and 95%). Because cardinality estimation performance can vary depending on the particular random choice of tuples for the samples, all of our experimental results are averaged over 12–20 different samples.

6.2.1 *Experiment 1: Single-Table Query*

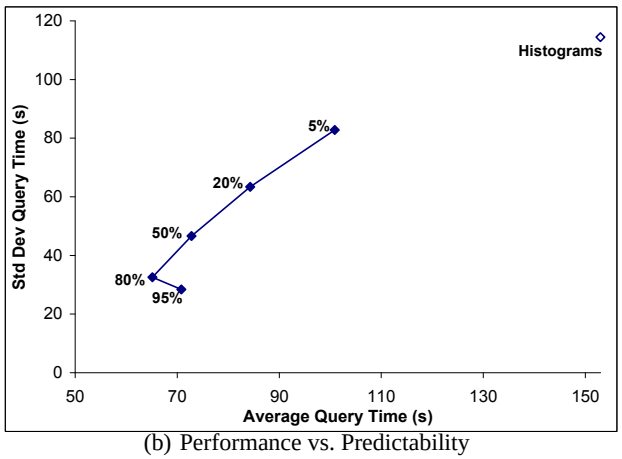
The template used for the first query scenario is as follows:

```
SELECT SUM(l_extendedprice)
FROM lineitem
WHERE l_shipdate BETWEEN
'07/01/97' AND '09/30/97'
AND l_receiptdate BETWEEN
('07/01/97'+?)  AND ('09/30/97'+?)
```

The value of "?" controls the degree of overlap between the set of rows satisfying the condition on `l_shipdate` and those satisfying the condition on `l_receiptdate`. We varied the degree of overlap so that the overall query selectivity was between 0% and 0.6%



(a) Selectivity vs. Time



(b) Performance vs. Predictability

Figure 10: Three-Table Join Query

of the 6 million rows from the `lineitem` table. The `lineitem` table was clustered by on its primary key, and prior to executing the queries for this scenario, we constructed two nonclustered indexes, one on `l_shipdate` and the other on `l_receiptdate`. Based on this physical design, the best query plan was either scanning the `lineitem` table, or else intersecting the two nonclustered indexes and then locating the qualifying records based on their RIDs, depending on the value of "?".

Figure 9(a) shows the average query execution time at each selectivity, where each data point is the average of 20 query executions, each based on a different 500-tuple random sample. The marked similarity between Figure 9(a) and Figure 5 can be explained by the fact that the cost model parameters used in the analysis underlying Figure 5 were based on the empirically observed performance of two of the possible query plans for this query scenario. The discontinuities in the higher selectivities for some of the lines plotted in Figure 9(a) is due to the fact that the experiments we averaged over a limited number of random samples. For comparison, the query performance when using the commercial DBMS's standard histogram-based estimation module is also plotted. The standard estimation module always selected the index intersection plan, which performed poorly at higher selectivities.

Figure 9(b) summarizes the performance of each setting of the confidence threshold by averaging across all the queries in this scenario. Each confidence threshold is represented by a point in a two-dimensional tradeoff space: average query execution time is measured on the horizontal axis, while standard deviation of query execution time is measured on the vertical axis. Figure 9(b) is quite similar to Figure 6, which is unsurprising given the close correspondence between the available plans in this query scenario and the query plans being modeled from Section 5. In Figure 9(b), just as in Figure 6, the variance in execution time decreases steadily as the confidence threshold increases. The lowest average execution time occurs at a confidence threshold of $T = 80\%$, closely followed by $T = 50\%$. Because of the close correlation between ship date and receipt date in TPC-H, standard histograms were significantly worse than our technique, both in terms of performance and predictability.

6.2.2 *Experiment 2: Three-Table Join*

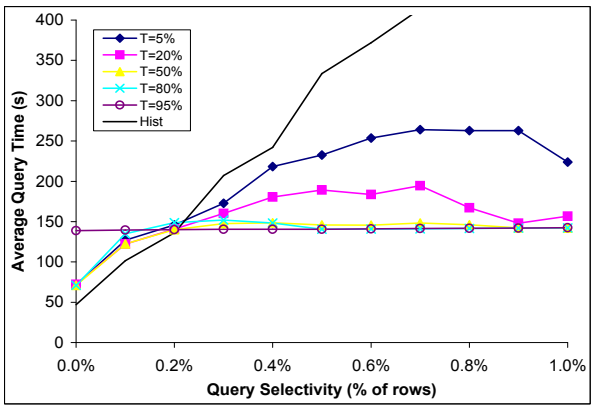
The second query scenario we explored also involved the TPC-H data set. In this query scenario, the query template consists of a natural join between the `lineitem` table, the `orders` table, and the `part` table. Besides the join predicates, there is an additional selection condition on the `part` table. The selectivity of this additional condition is the free parameter that is varied in this scenario. The physical design we selected had all relations clustered on their primary keys, with additional indexes on the foreign key columns.

In this query scenario, the optimal query plan has one of three different structures, depending on the number of rows from the `part` table that satisfy the predicate. At low selectivities, the best plan is to first join `part` to `lineitem` using indexed nested loops join, with `lineitem` as the inner relation, leveraging the index on the `l_partkey` column of `lineitem`, and then hash join with `orders`. When the selectivity gets a little higher, the best plan is a sequence of hash joins, first between `lineitem` and `part` and then with `orders`. When the query selectivity goes higher than approximately 10%, the optimal plan is to first merge join the two larger relations, `lineitem` and `orders`, and then hash join with `part`.

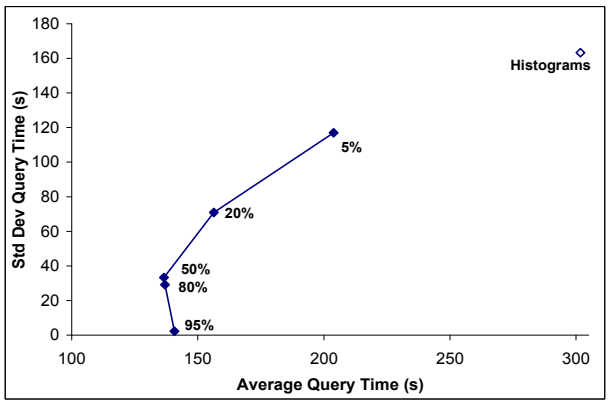
Although this query has two crossover points, due to space constraints, we will focus on the one that occurs at lower selectivity. (The results for the higher-selectivity crossover point are relatively uninteresting, because as was the case in Figure 8, query performance was more or less the same regardless of the confidence threshold that is chosen.) The first crossover point occurs at a selectivity between 0.1% and 0.2%. Figure 10 shows query execution performance in the general vicinity of this crossover point. Average execution time for each selectivity and confidence threshold is shown in Figure 10(a), and Figure 10(b) summarizes query performance for each setting of the confidence threshold. Each of subfigures is similar to its counterpart from the first experimental scenario, despite the fact that the types of queries involved in two scenarios are quite different. This suggests that the properties of our cardinality estimation procedure that were predicted analytically, and observed experimentally, for single-table queries have the potential to be applicable for a broader class of queries.

6.2.3 *Experiment 3: Four-Table Star Join*

The third and final query scenario used in our experiments models a typical data warehousing "star schema" query. We constructed a synthetic data set consisting of a 10-million-row fact table with some measure columns plus foreign keys to a number of dimension tables. We built nonclustered indexes on each foreign key column in the fact table. The query template consists of a four-way join between the fact table and three dimension tables, with aggregates computed on the measure columns of the fact table and filters applied to non-key columns in each of the dimension tables. Each



(a) Selectivity vs. Time



(b) Performance vs. Predictability

Figure 11: Four-Table Star Join Query

filter selected 10% of the rows of its dimension table. The distribution for the fact table rows was handcrafted so that by varying which rows were selected from each dimension table, any desired percentage of the fact rows between 0% and 10% could be made to join successfully. The standard histogram-based optimizer always estimated that 0.1% of the rows joined successfully because it relied on the independence assumption.

Depending on the number of fact rows participating in the join, one of two query plans was optimal in this scenario. When the number of fact rows was large, the best plan was a cascading series of hash joins. When the number of fact rows was small enough, then a sophisticated execution strategy involving semijoins was optimal: first, compute the semijoin of the fact table with each dimension table (relying on the indexed foreign key columns to retrieve the RIDs for the appropriate fact rows that join with each individual dimension table), then intersect the semijoin results, retrieve the qualifying records, and compute the aggregates. In addition to these two plans, we found that when using our cardinality estimation technique, the optimizer would sometimes select a third plan that was a hybrid of the other two. The hybrid plan used the semijoin strategy with two of the three dimension tables and a hash join to the third dimension table.

Figure 11 shows the impact that the choice of confidence threshold had when optimizing star join queries. Similar trends to the ones that were present in Figures 9 and 10 can again be spotted in Figure 11: low values of the confidence threshold give slightly better performance for queries with very low selectivity, but this achievement comes at the cost of weak performance on queries with higher selectivities. High values of the confidence threshold result in very consistent query performance across all selectivities, with the best average performance arising from thresholds of 50%–80%. The standard histogram-based estimation module failed to adjust to correlations in the data set and thus was not competitive with our approach.

6.2.4 *Experiment 4: Effect of Sample Size*

We conducted one experiment in which we used sample sizes other than the default size of 500 tuples. In this experiment, we used the single-table query scenario described in Section 6.2.1 with a fixed confidence threshold of 50% and varied the sample size from 50 to 2500 tuples, with the goal of understanding how query optimization performance degrades as the sample size is decreased. Figure 12 shows the effect of sample size on query performance. As can be seen from the figure, larger sample sizes resulted in both

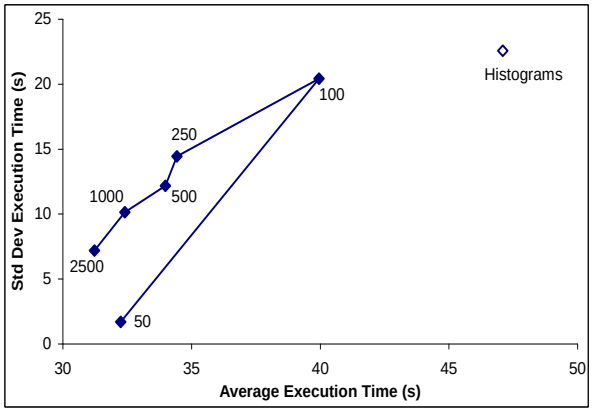


Figure 12: Effect of Sample Size

better average execution time and less variability in execution time.

The data point for 50-tuple samples represents an exception to the general trend. The probability distribution for query selectivity becomes more and more "spread out" as the sample size gets smaller, because the evidential value of a small sample is less and the uncertainty is larger. For a 50-tuple sample and a 50% confidence threshold, the uncertainty is sufficiently great that the predicted query selectivity is *always* greater than the crossover point at which sequential scan becomes optimal, even when zero tuples from the sample satisfy the query predicates. Thus the query optimizer always chooses a sequential scan plan when using a 50-tuple sample, resulting in very consistent query execution times, although the plan chosen is suboptimal when the actual selectivity is very low. This demonstrates a desirable "self-adjusting" feature of our estimation procedure: when a particularly risky query plan is only useful for a very limited range of selectivities, and the available statistics have insufficient resolution to adequately determine whether the query selectivity falls within that range, our estimation procedure will avoid that query plan in favor of safer alternatives.

6.2.5 *Experimental Conclusions*

The following recommendations summarize our conclusions from these experiments:

- A confidence threshold of 80% appears to be a good baseline level for general-purpose use. It achieves both good perfor-

mance (low average execution time) and good predictability (little variability in execution time).

- A confidence threshold of 95% leads to very stable query plans and few surprises. It is good for situations where predictability is the paramount concern.
- Confidence thresholds below 50% are rather speculative and are likely to be of limited applicability.

As future work, we plan to further refine and validate these conclusions through additional experimentation.

We envision the confidence threshold being set by database users and administrators in two ways: a system configuration parameter for robustness can be set to "conservative", "moderate", or "aggressive", corresponding to confidence thresholds of 95%, 80%, and 50% respectively. This setting will be used by default for all queries, except for queries that override the setting using a *query hint*—a special comment embedded in the SQL statement. Query hints are already used in commercial systems, e.g. to override the optimizer and force the use of a particular join method or access method.

7. CONCLUSION

The research results presented in this paper grew out of the question, "How can we increase the robustness of query optimizers?" By one definition, a robust query optimizer is one that generates plans that work reasonably well even when optimizer assumptions fail to hold. We have developed a novel cardinality estimation procedure that manages uncertainty in a principled way by reasoning probabilistically about selectivity. Because robustness sometimes comes at the cost of performance, users should be allowed to prioritize these competing objectives. To this end, our estimation technique incorporates into the query planning process user or application preferences about the predictability vs. performance tradeoff, explicity and succinctly expressed through the setting of a single parameter.

8. REFERENCES

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. 1999 SIGMOD Conf.*, pages 275–286, June 1999.

[2] G. Antoshenkov. Query processing in DEC Rdb: Major issues and future challenges. *Data Engineering Bulletin*, 16(4):41–50, 1993.

[3] J. M. Bernardo and A. F. M. Smith. *Bayesian Theory*. John Wiley, 1994.

[4] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. In *Proc. 2000 Intl. Conf. on Data Engineering*, pages 339–348, 2000.

[5] C.-M. Chen and N. Roussopoulos. Adaptive selectivity estimation using query feedback. In *Proc. 1994 SIGMOD*.

[6] F. Chu, J. Y. Halpern, and J. Gehrke. Least expected cost query optimization: What can we expect? In *Proc. 21st ACM PODS*, pages 293–302, 2002.

[7] F. Chu, J. Y. Halpern, and P. Seshadri. Least expected cost query optimization: An exercise in utility. In *Proc. 18th ACM PODS*, pages 138–147, 1999.

[8] R. L. Cole. A decision theoretic cost model for dynamic plans. *Data Engineering Bulletin*, 23(2):34–41, 2000.

[9] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In *Proc. 2001 SIGMOD*.

[10] D. Donjerkovic and R. Ramakrishnan. Probabilistic optimization of Top N queries. In *Proc. 1999 VLDB Conf.*, pages 411–422, 1999.

[11] C. Faloutsos and I. Kamel. Relaxing the uniformity and independence assumptions using the concept of fractal dimensions. *Journal of Computer and System Sciences*, 55(2):229–240, 1997.

[12] L. Getoor, B. Taskar, and D. Koller. Selectivity estimation using probabilistic models. In *Proc. 2001 SIGMOD Conf.*

[13] P. Haas, J. Naughton, P. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. 1995 VLDB Conf.*, pages 311–322.

[14] P. J. Haas, J. F. Naughton, S. Seshadri, and A. N. Swami. Selectivity and cost estimation for joins based on random sampling. *Journal of Computer and System Sciences*, 52(3):550–569, 1996.

[15] P. J. Haas and A. N. Swami. Sequential sampling procedures for query size estimation. In *Proc. 1992 SIGMOD Conf.*, pages 341–350, 1992.

[16] Y. E. Ioannidis. The history of histograms (abridged). In *Proc. 2003 VLDB Conf.*, pages 19–30, Sept. 2003.

[17] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. 1991 SIGMOD Conf.*, pages 268–277, May 1991.

[18] H. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. 1998 VLDB Conf.*, pages 275–286.

[19] H. Jeffreys. *Theory of Probability*. Clarendon Press, 1961.

[20] J.-H. Lee, D.-H. Kim, and C.-W. Chung. Multi-dimensional selectivity estimation using compressed histogram information. In *Proc. 1999 SIGMOD Conf.*, pages 205–214.

[21] P. M. Lee. *Bayesian Statistics: An Introduction*. Oxford University Press, 1989.

[22] R. J. Lipton, J. F. Naughton, and D. A. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. 1990 SIGMOD Conf.*, pages 1–11, 1990.

[23] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):192–221, 1988.

[24] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. 1998 SIGMOD Conf.*, pages 448–459, June 1998.

[25] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multi-dimensional queries. In *Proc. 1988 SIGMOD Conf.*, pages 28–36, 1988.

[26] F. Olken. *Random Sampling from Databases*. PhD thesis, University of California at Berkeley, 1993.

[27] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. 1997 VLDB Conf.*, pages 486–495, 1997.

[28] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proc. 1996 SIGMOD Conf.*, pages 294–305.

[29] K. Sarac, O. Egecioglu, and A. E. Abbadi. Iterated DFT based techniques for join size estimation. In *Proc. 1988 ACM Intl. Conf. on Information and Knowledge Management*.

[30] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, and T. Price. Access path selection in a relational database management system. In *Proc. 1979 SIGMOD Conf.*, pages 23–34, June 1979.

[31] 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.

[32] Transaction Processing Performance Council. TPC-H benchmark 2.0.0, July 2002. http://www.tpc.org.