# Towards Adaptive Costing of Database Access Methods

Ye Qin and Kenneth Salem
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
{yqin,kmsalem}@cs.uwaterloo.ca

Anil K Goel
Sybase iAnywhere
Waterloo, Ontario, Canada
anil.goel@sybase.com

## Abstract

*Most database query optimizers use cost models to identify good query execution plans. Inaccuracies in the cost models can cause query optimizers to select poor plans. In this paper, we consider the problem of accurately estimating the I/O costs of database access methods, such as index scans. We present some experimental results which show that existing analytical I/O cost models can be very inaccurate. We also present a simple analysis which shows that larger cost estimation errors can cause the query optimizer to make larger mistakes in plan selection. We propose the use of an adaptive black-box statistical cost estimation methodology to achieve better estimates.*

## 1. Introduction

Most database query optimizers use cost models to identify good query execution plans. Inaccuracies in the cost models can cause query optimizers to select poor plans. There are many potential sources of costing inaccuracies. Cost models rely on estimates of the selectivity of query predicates. If these estimates are poor, the resulting cost estimates are likely to be poor as well. Hence, there is a substantial body of work (e.g., [4, 5, 8, 10]) on problems related to selectivity estimation. However, even if selectivity is estimated accurately, obtaining accurate cost estimates can still be challenging.

In this paper, we focus on the problem of estimating the I/O costs of database access methods, such as index scans and table scans. Database systems generally use analytic models to estimate these costs. However, the true cost of an access method depends on factors that are difficult to capture in such models. Two such factors are the effects of clustering and data layout, and the impact of caching. In addition, these analytic models depend on parameters - such as the amount of memory available for a query or the costs of the underlying storage operations - that require calibration or are difficult to determine.

Is it important to have accurate I/O cost estimates? Reiss and Kanungo [12] studied the relationship between errors in calibrating storage cost model parameters (the costs of underlying storage operations) and the cost of the plan that was ultimately chosen by the query optimizer. They showed that, in some circumstances, such errors can have a significant effect on the optimality of the resulting plan.

In the first part of this paper, we consider this question primarily in the context of Postgres. By measuring the actual I/O costs of access methods and comparing them to cost model estimates, we present some empirical evidence of costing inaccuracies. Next, through an analysis of Postgres's analytic cost models for index scans and table scans, we characterize the potential impact of inaccurate cost estimates on the quality of the plan selected by the optimizer for a simple class of test queries. This analysis indicates that the order-of-magnitude cost estimation errors that we observed empirically can cause the optimizer to choose plans that are suboptimal by an order of magnitude.

In the second part of this paper, we present some initial steps towards improved costing of database access methods. We propose to adopt a statistical approach to cost estimation. This approach involves observing the actual costs of access methods as they are used in query plans, and then inferring cost models from these observations. By applying this procedure repeatedly, so that the current model is based on recent observations, we can produce cost models that adapt automatically to changes in the database system configuration or operating point. Similar approaches have been proposed for modeling the costs of complex XML query operators [14], for costing UDF executions [7], and for costing remote sub-queries in federated database systems [11]. However, we argue that the problem of cost estimation for database access methods presents some unique challenges. We have implemented, in Postgres, a prototype of a statistical cost modeling technique. We illustrate the issues by using the prototype to learn the costs of index scans.

## 2. The Importance of I/O Cost Estimation

In this section, we seek to answer two questions. First, how large are I/O cost estimation errors? Second, what is the impact of these errors on the query optimizer? If I/O costs are incorrectly estimated, will the optimizer choose a bad plan? How bad might it be?

To address these questions, we present a concrete analysis of a single case: the choice between an index scan and a table scan as a table access method in Postgres. We have used Postgres (Version 8.0.9) because it is open-source. This allows us to directly inspect the relevant analytic cost models. We have focused on index scans because estimating the cost of an index scan - especially for secondary, unclustered indexes - is difficult. Although the details of the cost model we present are specific to Postgres, every database system's cost model has its own sources of inaccuracy. Parameter calibration, in particular, is a problem faced by all models. In Section 2.4 we present a brief empirical study of index scan cost estimates in SQL Anywhere, a commercial database management system from Sybase iAnywhere [2]. This demonstrates that the kinds of cost estimation errors that we observed are not unique to Postgres.

### 2.1. I/O Cost Models for Table Scan and Index Scan

We begin with a brief description of the I/O cost models used for table scans and index scans in Postgres. Postgres uses two configurable server parameters to model the costs of individual I/O operations. $C_{seq}$ represents the cost of a sequential I/O, and $C_{rand}$ represents the cost of a random I/O operation. The cost of a table scan is estimated as $T_{tscan} = C_{seq}N_P$, where $N_p$ is the number of pages in the table being scanned.

Estimating the cost of an index scan is more challenging. Postgres uses a modified version of the cost model developed by Mackert and Lohman [9]. It first estimates the number of page I/Os that would be required if the index order were completely uncorrelated with respect to the clustering order of the associated table. This value, $N_{uncorr}$, is estimated using:

$$N_{uncorr} = \begin{cases} min(\frac{2sN_PN_T}{2N_P+sN_T}, N_P) & when\ N_P \leq b \\ \frac{2sN_PN_T}{2N_P+sN_T} & when\ sN_T \leq D \\ b + (sN_T - D)\frac{N_P-b}{N_P} & when\ sN_T > D \end{cases}$$

where $D = \frac{2N_Pb}{2N_P-b}$. In this expression, $s$ is the selectivity of the index scan, $N_P$ is the number of pages in the underlying table, $N_T$ is the number of tuples in the underlying table, and $b$ is Postgres' effective buffer cache size, another configurable server parameter. Next, it estimates the number of pages that would be required if the index were completely correlated with the physical order of the table: $N_{corr} = \lceil sN_P \rceil$. Finally, Postgres uses its estimate of the actual index correlation (denoted by $r$) to interpolate between the costs that would result at these extremes:

$$T_{iscan} = (1 - r^2)N_{uncorr}C_{rand} + r^2N_{corr}C_{seq} \quad (1)$$

Postgres's correlation estimates range from $r = 0$, indicating no correlation, to $r = 1$ indicating complete correlation. This model can be interpreted as $r^2N_{corr}$ pages read sequentially (at a cost of $C_{seq}$ per page) and $(1 - r^2)N_{uncorr}$ pages read randomly, so the estimated total number of page I/Os is given by the sum of these two numbers.

### 2.2. I/O Cost Estimation Errors

We ran a series of experiments intended to measure the accuracy with which Postgres estimates the costs of index scans under various conditions. Each index scan results in some number of page I/O requests, each of which imposes some load on the underlying storage system. The experiments presented here characterize errors in the estimation of *I/O counts*, i.e., the number of page I/O requests that occur during a scan. The optimizer may also make errors in estimating the per-page load (cost) imposed by these requests. Since the total cost of an index scan depends on both the number of page I/O requests and the costs of those requests, the estimation errors reported in our experiments may be magnified by any additional errors made in estimating per-page costs.

Our experiments used a single relation, $R$, with three attributes: $A$, $B$, and $C$. Attributes $A$ and $B$ are of type integer. Attribute $C$, of type varchar, is a padding attribute that is used to control the total size of each tuple. We define a clustered index on $A$, and an unclustered index on $B$.

We adopted a method used by Vander Zanden, Taylor, and Bitton [13] to populate $R$ with skewed data for which we can control the correlation between attributes A and B. Values for attribute $A$ are randomly generated in the range $[0, M]$ using a normal distribution with mean $\mu$ and standard deviation $\sigma$. Given a value $t.A$ for attribute $A$ of tuple $t$, we choose a value for $B$ using

$$t.B = t.A + U\left(\frac{-Mc}{2}, \frac{Mc}{2}\right)$$

where $c$ is a parameter that is used to control the correlation between $A$ and $B$ and $U(x, y)$ is a uniform random variable over the range $[x, y]$. If the resulting value $t.B$ is outside of the range $[0, M]$ we discard it and choose another. As the parameter $c$ increases from $c = 0$ to $c = 2$, $A$ and $B$ change from perfectly correlated to uncorrelated. In our experiments, we used $\mu = 10000$, $M = 2\mu = 20000$, and

$\sigma = 2000$. We generated 6000000 tuples for relation $R$, and generated values for attribute $C$ of length 154 bytes. The resulting relation occupied 112,548 8 Kbyte Postgres pages.

We ran a set of experiments, each of which consisted of the following steps:

1. Choose a value for the correlation parameter $c$ and create and populate relation $R$ plus the unclustered index on attribute $B$.
2. Choose a size, $b$, for the Postgres buffer cache and set the corresponding Postgres server parameter.
3. Run the Postgres ANALYZE command to ensure that Postgres has accurate statistics for table $R$.
4. Run a series of instances of the simple range query "`select count(*) from R where R.b ≤ :var`" using successively larger values of `var` in each instance. Restart Postgres after each query to ensure that each starts with an empty buffer cache.

Postgres has two possible plans for this query, one using a table scan of $R$ and one using an index scan on the unclustered index for $R.b$.[1] However, we forced Postgres to choose the index scan regardless of the costs of the two plans. For each query, we recorded $N_{est}$, Postgres's estimate of the number of page I/O operations required by the index scan. As was noted in Section 2.1, this estimate is given by $(1 - r^2)N_{uncorr} + r^2 N_{corr}$. However, Postgres does not report this estimate directly. It reports $T_{iscan}$, as defined by Equation 1. For the purposes of this experiment, we used $N_{est} = T_{iscan}/C_{rand}$, which is never greater than the actual estimate of $(1 - r^2)N_{uncorr} + r^2 N_{corr}$. We also recorded the actual number of page I/Os ($N_{meas}$) performed by Postgres when the scan was executed. We define the estimation error, $\epsilon_{io}$, as $\epsilon_{io} = N_{est}/N_{meas}$. Note that since Postgres always overestimates the actual number of I/O operations, and since the $N_{est}$ that we measured is always lower than Postgres's actual estimate, the estimation errors ($\epsilon_{io}$) that we report underestimate the true estimation errors.

Figure 1 shows the measured ($N_{meas}$) and estimated ($N_{est}$) I/O counts as a function of the selectivity of the test query when $c = 0.5$. Figure 2 shows the corresponding estimation errors, $\epsilon_{io}$. These figures show that Postgres's I/O estimates can be off by an order of magnitude. Furthermore, the amount of error depends on factors such as the index correlation and the buffer size. Figure 3 shows how estimation errors vary as a function of buffer size when the index correlation parameter($c$) equals 0.5 and $var = 5000$. We found that estimation errors initially grow very quickly as $b$ increases, before gradually declining.

[1] Postgres does not support index-only plans, so it has to retrieve tuples from relation $R$ to execute the test query.

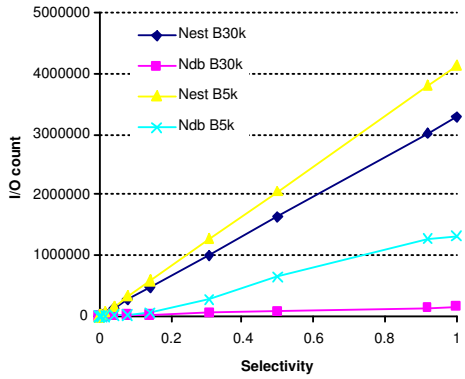


**Figure 1. Estimated and actual I/O counts vs. query selectivity, for buffer sizes $b = 5000$ and $b = 30000$. Correlation parameter $c = 0.5$.**

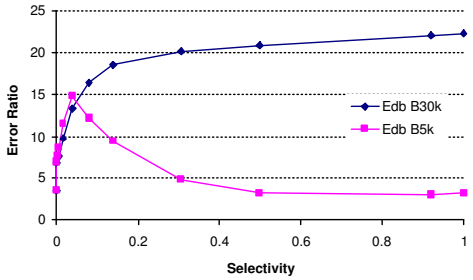


**Figure 2. Estimation error vs selectivity for buffer sizes $b = 5000$ and $b = 30000$. Correlation parameter $c = 0.5$**

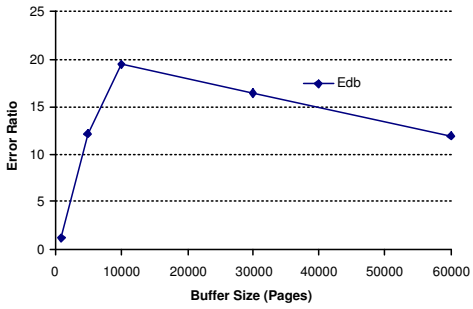


**Figure 3. Estimation error vs buffer size for parameters $c = 0.5$ and $var = 5000$**

Figure 4 illustrates the relationship between estimation errors and the index correlation when the Postgres buffer size ($b$) is equal to 5000 pages. It is observed that estimates were most accurate for indexes that were highly correlated ($c = 0$) or highly uncorrelated ($c = 1$), but were significantly worse for intermediate values of $c$. These estimation errors speak to the difficulty of accurately accounting for the effects of clustering and buffering in an analytic model.

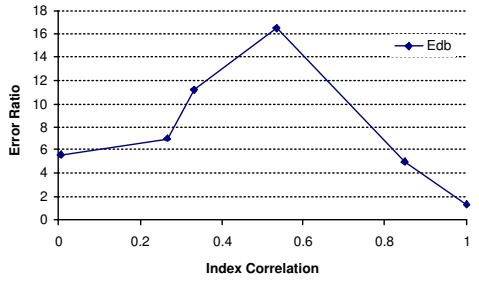


**Figure 4. Estimation error vs index correlation for buffer size $b = 5000$**

## 2.3. Optimizer Sensitivity

In the previous section, we showed that Postgres can make significant errors in estimating the costs of index scans. In this section, we address the follow-up question: can these cost estimation errors lead to errors in cost-based query optimization? Might Postgres choose a bad plan as a result of these estimation errors and, if so, how bad can that plan be?

We will not attempt to answer this question in its full generality. Instead, we will consider a much more specific problem. Suppose that we have a class of simple selection queries of the form used in Section 2.2, namely:

```
select count(*) from R
where R.b  ≤  :var
```

Suppose further that the query optimizer has two possible plans to choose from for this query: a table scan of $R$, or an index scan using an unclustered index on $R.b$. This is the case for Postgres. We would like to quantify the effect that I/O costing errors may have on the actual cost of executing this query. In particular, costing errors may cause the optimizer to choose the more expensive of the two plans.

To quantify this effect, we use the notion of *global relative cost*, as defined originally by Reiss and Kanungo [12]. Suppose that we have two cost models: the optimizer's cost model $M_{dbms}$ and an ideal, correct cost model $M_{ideal}$. Suppose that, under the actual cost model, the optimizer

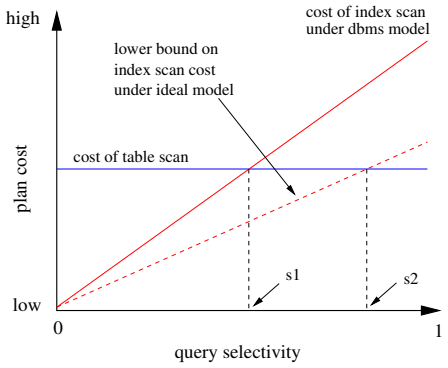


**Figure 5. Analysis of table scan and index scan costs under two cost models**

chooses plan $P_{dbms}$ for a given query, and that under $M_{ideal}$ the optimizer would choose plan $P_{ideal}$ for the same query. Note that, in general, $P_{ideal}$ may or may not differ from $P_{dbms}$. We define the global relative cost of $P_{dbms}$ as

$$\text{GRC}(P_{dbms}) = \frac{\text{Cost}(P_{dbms}, M_{ideal})}{\text{Cost}(P_{ideal}, M_{ideal})}$$

where $\text{Cost}(P, M)$ refers to the cost of plan $P$ according to cost model $M$. Thus, if $\text{GRC}(P_{dbms}) = k$, then the system is using a plan that is $k$ times as expensive as the plan that it would have chosen under the ideal cost model.

To simplify our analysis, we will consider the specific case in which actual cost model accurately estimates the cost of a table scan (which is not difficult to do) but overestimates the cost of index scans by a factor $\epsilon$. Specifically, if $P$ represents the index scan plan, then $\text{Cost}(P, M_{dbms}) \leq \epsilon\text{Cost}(P, M_{ideal})$. This scenario is illustrated in Figure 5, which shows $M_{dbms}$ and $M_{ideal}$ for index scan (as well as the cost model for table scans) as functions of the scan selectivity. At a query selectivity less than $s_1$, the index scan plan is chosen under both $M_{dbms}$ and $M_{ideal}$. Thus, there is no penalty for using $M_{dbms}$ and the global relative cost will be 1. Similarly, the table scan plan is chosen under both cost models for queries with selectivity greater than $s_2$. However, for queries with selectivity between $s_1$ and $s_2$, the optimizer will choose a table scan, although an index scan would have been cheaper. If we assume that the (unknown) ideal cost model for the index scan is monotonically non-decreasing with respect to the query selectivity, then the worst-case error will occur when the query selectivity is equal to $s_1$. Thus, we can write:

$$\text{GRC}(P_{dbms}) \leq \frac{T_{tscan}(s_1)}{(1/\epsilon)T_{iscan}(s_1)} = \frac{T_{iscan}(s_1)}{(1/\epsilon)T_{iscan}(s_1)} = \epsilon$$

That is, if the optimizer's cost model can overestimate the cost of an index scan by a factor $\epsilon$, then the cost of executing

a selection query may be a factor $\epsilon$ higher than it would have been under the ideal cost model.

## 2.4. I/O Cost Estimation Errors in SQL Anywhere

In Section 2.2 we showed that Postgres' query optimizer can make very inaccurate estimates of I/O counts for unclustered index scans. In this section we want to demonstrate that this problem is not unique to Postgres. To this end, we studied index scan cost estimates in the SQL Anywhere database management system (Version 10) [2]. SQL Anywhere's cost model is more sophisticated than that of Postgres. For example, it considers the current buffer residency ratio (BRR) of database objects when estimating I/O costs. The buffer resident ratio of a relation is defined as the ratio of the number of buffer resident pages for that relation to the total number of pages for that relation.

We measured cost estimation errors for SQL Anywhere using the methodology that was described in Section 2.2, with the exception that we also controlled the buffer resident ratio of queried relations. Experiments were performed under two different BRR conditions: $BRR$=0 and $BRR$=0.49. $BRR$=0 was achieved by flushing the DBMS buffer before running each test query. $BRR$=0.49 was achieved by first flushing the DBMS buffer and then running a buffer-warming range query against the target relation prior to running the test query. We took measures to ensure that accurate selectivity estimates were used by the optimizer in obtaining estimated I/O counts. These measures included building histograms with a large number of buckets and providing user hints. Thus, the cost estimation errors that we observed were due to inaccurate I/O count estimates, not inaccurate selectivity estimation. The estimated and actual I/O counts were obtained from SQL Anywhere's graphical query plans. We used a buffer cache size of 1 gigabyte for these experiments.

Figure 6 shows the estimated and actual I/O counts for two different buffer resident ratios, and Figure 7 illustrates the corresponding estimation errors. These two figures show that SQL Anywhere can experience I/O estimation error of the same magnitude as those we observed with Postgres. This is true despite the fact the SQL Anywhere is able to use its knowledge of the BRR in making cost estimates. This illustrates how difficult cost estimation can be: even if we know how much of a relation is cached in the buffer, it is difficult to estimate how effective the cache will be. On the other hand, we also note that when $BRR = 0.0$ and queries were very selective, SQL Anywhere's I/O count estimates were better than those produced by Postgres.

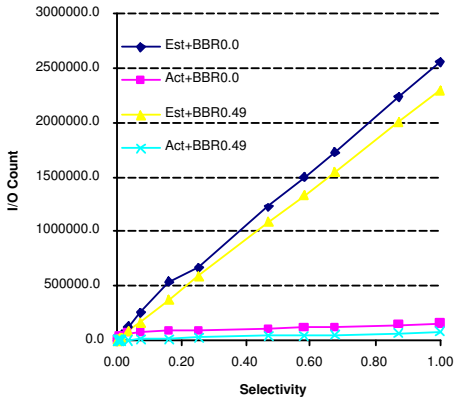


**Figure 6. Estimated and actual I/O counts for two different buffer resident ratios ($BRR$=0.0 and $BRR$=0.49)**

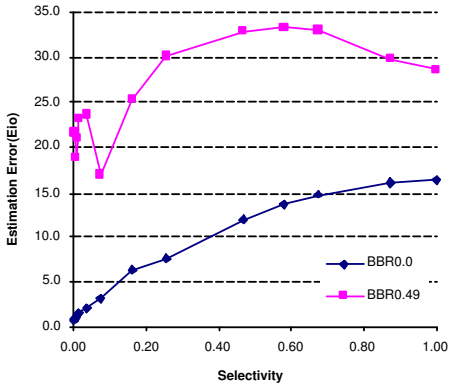


**Figure 7. Estimation error ($E_{io}$) for two different buffer resident ratios ($BRR$=0.0 and $BRR$=0.49)**

## 3. Black-Box I/O Cost Estimation

It is difficult to build and calibrate accurate analytic I/O cost models, and model inaccuracies can lead to poor plans. To address this problem, we are exploring the use of black-box statistical I/O cost models for database operators. The general idea is to measure the actual costs of database operators and then fit a model to these observations. More specifically, for each type of database access method (e.g., index scans) and each physical database object used by that method, we model the cost of accessing the object using the method. These models are parameterized in the same way as the corresponding access method. For example, the cost models for index scans are parameterized by the selectivity of the scans.

This general approach to cost estimation has been applied in a variety of settings in which it is difficult to develop good analytic models. Examples include estimating the costs of user-defined functions [7], estimating the costs of remote sub-queries in federated database systems [11], and estimating the costs of complex XML query operators [14]. The so-called "parade-of-runs" approach [1] relies on a synthetic calibration workload to produce a training data set that covers the entire input parameter space. Alternatively, calibration can be avoided in favor of an adaptive approach that measures the costs of operators (or functions or subqueries) that occur as a result of the normal system workload, and then uses these costs to build a model. A drawback of the adaptive approach is that there is no guarantee that the resulting training data will cover the entire parameter space. Hopefully, though, they are representative of the actual workload and thus cover the interesting part of the space. We adopt the latter approach, as it avoids the calibration step and allows for the possibility of models that will adapt to changes in workload and system configuration. Thus, for each database access method, this approach will over time produce a succession of models, $M_0, M_1, M_2, \ldots$, with each new model somehow incorporating more recent observations than its predecessors.

We are currently exploring the application of this idea to the problem of costing database access methods. One challenge in using this approach for costing database access methods is the problem of optimizer-induced "gaps" in the training data. These are gaps that occur because the query optimizer may have more than one access method to choose from, and the model learning mechanism can observe only those access methods that are actually selected for use by the query optimizer.

To illustrate this problem, we return to the scenario described in Section 2 in which the database includes the relation $R$ with an index on $R.B$ and is handling a workload consisting of simple range queries on $R.B$. Suppose that the initial cost model $M_0$ for index scans on $R.B$ is given by the upper (solid) index scan cost model in Figure 5. In this situation, the learning algorithm will *only* observe index scans with selectivity $s < s_1$, since the optimizer will choose an index scan rather than a table scan under that conditions. Thus, there is a "gap" in the training data for the index scan model, since the portion of the input parameter space for which $s > s_1$ will not be represented.

To address this problem, we first try to construct a cost model that is accurate in that portion of the selectivity space for which we have training data: $s < s_1$ in our example. Next, we extrapolate the model into the gaps, resulting in a complete model that covers the entire selectivity space. In our prototype, described in Section 4, we have considered two possible extrapolation techniques.

## 4. Prototype

We have implemented an adaptive, statistical learning mechanism in Postgres and applied it to the problem of learning cost models for index scans. When a query plan uses an index scan, our prototype records the selectivity of the scan and the actual number of page I/O operations that are performed by the scan. Periodically, a learning module is invoked. For each index, this module uses the observations that have accumulated since it was last invoked to learn the functional relationship between index scan selectivity ($s$) and I/O count ($N$). We will use $N_i(s)$ to represent the function that is learned by the $i$th invocation of the learning module. To estimate the cost of an index scan with selectivity $s$, we use $N_i(s)C_{rand}$. Thus, this procedure results in a series of cost models $M_0, M_1, \ldots, M_i, \ldots$, where model $M_i$ ($i \geq 1$) is based on the learned function $N_i$. Once $M_i$ has been produced by the learning module, it replaces model $M_{i-1}$ and is used by the query optimizer until the learning module is next invoked to produce model $M_{i+1}$. We use Postgres's built-in analytic index scan cost model as the initial model, $M_0$, for all indexes.

Note that by using $C_{rand}$ to translate the I/O count $N_i(s)$ into an I/O cost, we are overestimating the costs of I/O operations and thus introducing a bias against index scans in the cost based optimizer. For the purposes of our experiments this is not a problem. However, a more realistic implementation of this technique would need to incorporate a better means of estimating per-page I/O costs. The issue of estimating I/O counts rather than I/O costs is discussed further in Section 5.

To determine the I/O count function $N_i(s)$ from a set of observations, the learning module proceeds in two steps. Let $\hat{s}_i$ represent the largest scan selectivity observed for the target index during the $i$th iteration. The learning module first creates a model for the selectivity range $[0, \hat{s}_i]$, where we have training data. To produce this partial model, we have tried two regression methods: standard linear regres-

sion (LM) and the multivariate adaptive polynomial spline regression (POLYMARS) [6]. Next, it extrapolates this model into the range $(\hat{s}_i, 1]$. The result is a two-part model that covers the entire range of scan selectivities. To extend the model into $(\hat{s}_i, 1]$, we also tried two approaches. The first, which we call linear extension (LN), extends the $[0, \hat{s}_i]$ model linearly into $(\hat{s}_i, 1]$. The second, which we call mixed extension (MIX), uses a selectivity-weighted average of a linear extension and the original Postgres analytic I/O count estimates. Specifically, for $s > \hat{s}_i$, we use:

$$N_i(s) = N_0(s) - (N_0(\hat{s}_i) - N_i(\hat{s}_i))\frac{1-s}{1-\hat{s}_i}$$

where

$$N_0(s) = (1-r^2)N_{uncorr} + r^2 N_{corr}$$

as described in Section 2.1. Since the original Postgres model overestimates the costs of index scans, MIX extrapolation results in higher cost estimates than LN extrapolation. We ran a series of experiments to explore the behavior of the prototype. Our test database was a small (scale factor 0.1) TPC-H instance with skewed data generated using a technique developed by Chaudhuri and Narasayya [3]. The Postgres cache size was set to its default value (1000 pages). The query workload consisted of two simple target queries plus a "background" workload consisting of the TPC-H queries. The first target query was a simple range query on the orderkey attribute of the lineitem table. The second target was simple range query on the orderkey attribute of the orders table. The physical database design includes unclustered indexes on each of these attributes, so that both of the target queries are answerable using an index scan if the optimizer chooses to use one. Our goal in these experiments was to to learn cost models for index scans on the orderkey attributes of the lineitem and orders tables. Each experiment consisted of fifteen cost model iterations. On each iteration, we ran 100 consecutive batches of queries, with each batch consisting of a random permutation of the TPC-H queries plus the two target queries. Thus, each of our two target queries occurs 100 times on each iteration.

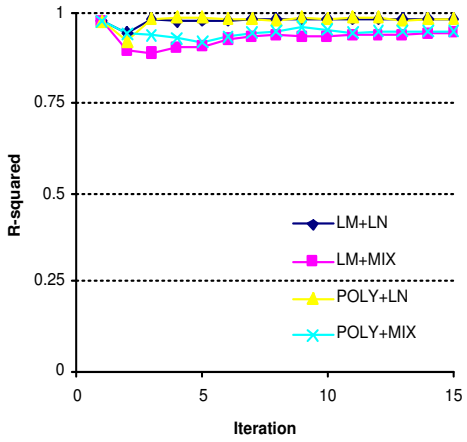


**Figure 8. Correlation coefficient ($R^2$) vs iteration number for the index on `lineitem`**

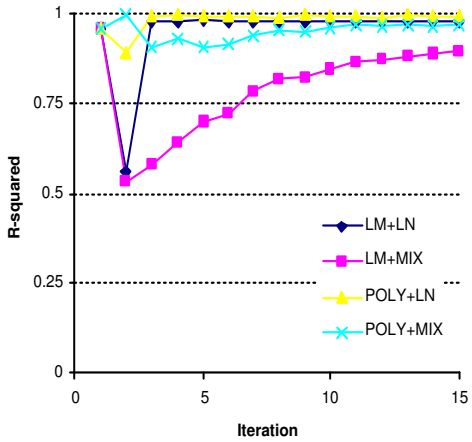


**Figure 9. Correlation coefficient ($R^2$) vs iteration number for the index on `orders`**

By combining a regression technique (LM or POLYMARS) with an extrapolation technique (LN or MIX), we obtain four potential cost model learning modules. Figures 8 and 9 show the correlation coefficient of the fitted models for the lineitem index and the orders index, respectively, in the range $[0, \hat{s}_i]$ as a function of the iteration number, for each of these techniques. These graphs show that both regression techniques resulted in good fits to the observed costs.

Of more interest are Figures 10 and 11, which show the maximum observed scan selectivity ($\hat{s}_i$) as a function of the iteration number, $i$. Under each cost model $M_i$, there exists a *crossover selectivity*, at which the modeled index scan becomes more expensive than a table scan. When $M_i$ is used, we expect to observe index scans with selectivities up to this crossover point. Thus, $\hat{s}_i$ will be a lower bound on the crossover selectivity for $M_i$.

In our test configuration, the true crossover selectivities for index scans on `lineitem` and `orders` are approximately $0.254$ and $0.3$, respectively. However, as shown in Figure 11, $\hat{s}_i$ peaked as high as $0.7$ when LN extrapolation was used in the case of `orders`. This indicates that

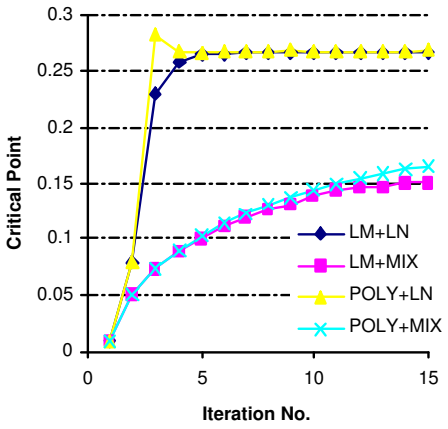


**Figure 10. Maximum observed scan selectivity ($\hat{s}_i$) vs iteration number for the index on `lineitem`**

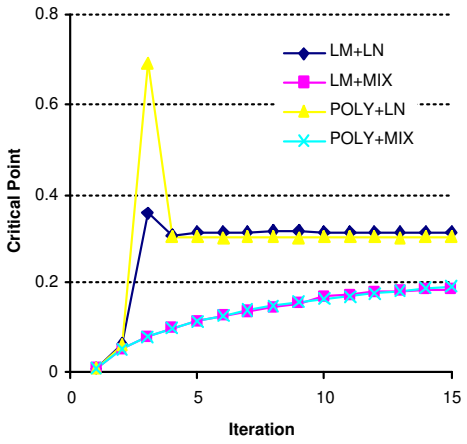


**Figure 11. Maximum observed scan selectivity ($\hat{s}_i$) vs iteration number for the index on `orders`**

LN underestimated costs when it extrapolated, resulting an a crossover selectivity much higher than the true crossover. This illustrates an interesting tradeoff. By extrapolating aggressively, using low cost estimates, the learning module will quickly be able to observe index scans - and hence build an accurate cost model - over a wide range of selectivities. However, the price is potentially poor query performance while those observations are collected. In our example, index scans with selectivity greater than 0.3 are slower than a table scan would have been. In contrast, MIX interpolation was much less aggressive. Under MIX, $\hat{s}_i$ moves slowly towards the actual crossover selectivity, but it avoids overshooting. Note that failure to find the true crossover selectivity quickly also results in performance penalties, since the optimizer may *fail* to choose an index scan when it should. However, these are the same kind of errors that were made under Postgres's original cost model $M_0$. Thus, a virtue of MIX is that it gradually improved on $M_0$ without using index scans inappropriately.

## 5. Conclusion and Future Work

In this paper, we showed that I/O cost models for database access methods can be very inaccurate, and that such inaccuracies can cause the query optimizer to choose suboptimal plans. For example, if the cost of an index scan is overestimated by a factor of ten (which we observed), the query optimizer may choose to use a table scan even though an index scan would have been ten times faster.

To obtain better cost estimates for database access methods, we propose to use an adaptive statistical approach that learns cost models from observations of the actual costs of access methods under normal operating conditions. We presented some preliminary results from a prototype implementation of this approach in Postgres. These results illustrate one of the challenges of this approach, namely the difficulty of building a complete cost model for a given access method using only observations of index scans that are actually selected by the query optimizer.

Currently, our prototype learns to predict I/O counts for index scans. The optimizer must then somehow translate these estimated counts into costs. One advantage of this approach is that I/O counts are insensitive to the load on the underlying storage system, and to the performance characteristics of that storage system. As a result, we expect our predictions to be robust with respect to changes in load or storage system configuration. Of course, the disadvantage is that we have not solved the problem of modeling the costs associated with requests made to the underlying storage system. An alternative approach would be to use the adaptive learning mechanism to directly learn I/O costs, rather than I/O counts. For example, instead of observing the I/O counts associated with index scans at particular se-

lectivities, we could observe the total time required by the scan. This would allow the learning module to directly learn the cost models $M_i$, rather than the I/O count models $N_i$. One challenge associated with this approach is that costs will fluctuate, perhaps rapidly, with the storage system load. The adaptive nature of our methodology would allow it to accommodate fluctuations that are not too fast. To account for more rapid changes, we may have to consider techniques such as addition of a load parameter as an input to our learned cost function. A second challenge is to ensure that the access "costs" that we learn are compatible with the rest of the optimizer's cost model. In particular, for throughput-based optimizers, it may be undesirable to include contention-induced latencies in the learned costs. We are currently exploring these and other related problems.

## 6. Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council of Canada.

## References

[1] J. Boulos and K. Ono. Cost estimation of user-defined methods in object-relational database systems. *ACM SIGMOD Record*, 28(3):22–28, 1999.

[2] I. T. Bowman, P. Bumbulis, D. Farrar, A. K. Goel, B. Lucier, A. Nica, G. N. Paulley, J. Smirnios, and M. Young-Lai. SQL Anywhere: A holistic approach to database self-management. In *Proc. Second International Workshop on Self-Managing Database Systems*, Apr. 2007.

[3] S. Chaudhuri and V. Narasayya. Program for TPC-D data generation with skew. Microsoft Corp. ftp://ftp.research.microsoft.com/users/viveknar/tpcdskew, 1999.

[4] Y. Ioannidis. The history of histograms(abriged). In *Proc. Int'l Conf. on Very Large Data Bases*, pages 19–30, 2003.

[5] Y. E. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 268 – 277, 1991.

[6] C. Kooper, S. Bose, and C. J. Stone. Polychotomous regression. *Journal of the American Statistical Association*, 92(437):117–127, 1997.

[7] B. S. Lee, L. Chen, J. Buzas, and V. Kannoth. Regression-based self-tuning modeling of smooth user-defined function costs for an object-relational database management system query optimizer. *The Computer Journal*, 47(6):673–693, 2004.

[8] R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 1–11, 1990.

[9] L. F. Mackert and G. M. Lohman. Index scans using a finite lru buffer: A validated i/o model. *ACM Transactions on Database Systems*, 14(3):401–424, Sept. 1989.

[10] M. V. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database systems. *ACM Computing Surveys*, 20(3):191–221, Sept. 1988.

[11] A. Rahal, Q. Zhu, and P.-A. Larson. Evolutionary techniques for updating query cost models in a dynamic multidatabase environment. *VLDB Journal*, 13(2):162–176, 2004.

[12] F. Reiss and T. Kanungo. A characterization of the sensitivity of query optimization to storage access cost parameters. In *Proc. ACM SIGMOD Int'l Conf. on Management of Data*, pages 385–396, 2003.

[13] B. T. V. Zanden, H. M. Taylor, and D. Bitton. Estimating block accesses when attributes are correlated. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 119–127, 1986.

[14] N. Zhang, P. J. Haas, V. Josifovski, G. M. Lohman, and C. Zhang. Statistical learning techniques for costing XML queries. In *Proc. Int'l Conf. on Very Large Data Bases*, pages 289–300, 2005.