Estimating Page Fetches for Index Scans with Finite LRU Buffers

Arun Swami
IBM Research Division
Almaden Research Center
650 Harry Road, San Jose, CA 95120-6099
arun@almaden.ibm.com

K. Bernhard Schiefer
IBM Toronto Laboratory
1150 Eglinton Ave. East
Toronto, Ontario, Canada M3C 1H7
schiefer@vnet.ibm.com

Abstract

We describe an algorithm for estimating the number of page fetches for a partial or complete scan of a B-tree index. The algorithm obtains estimates for the number of page fetches for an index scan when given the number of tuples selected and the number of LRU buffers currently available. The algorithm has an initial phase that is performed exactly once before any estimates are calculated. This initial phase, involving LRU buffer modeling, requires a scan of all the index entries and calculates the number of page fetches for different buffer sizes. An approximate empirical model is obtained from this data. Subsequently, an inexpensive estimation procedure is called by the query optimizer whenever it needs an estimate of the page fetches for the index scan. This procedure utilizes the empirical model obtained in the initial phase.

1 Introduction

In a relational database management system, the query optimizer determines the query execution plan that will be used to retrieve the data requested. This plan consist of a set of primitive operations, e.g., join, a sequence in which the operations will be performed, e.g., join order, specific methods of performing the operations, e.g., sort-merge join, and access methods to obtain records from the base relations, e.g., index scan. In a cost-based query optimizer both estimates of I/O and CPU resource consumption are used to select the most efficient query execution plan.

In this paper we will examine methods of estimating I/O requirements for full and partial index scans that require data page access. We begin with Section 2 which provides an overview of the problem of estimating page fetches. In Section 3 we survey known work on the index scan problem. We describe the new algorithm, called **EPFIS**, in Section 4. We used the data from a customer database and synthetic data to compare the error behavior of Algorithm **EPFIS** with other algorithms in Section 5. We summarize the paper in Section 6.

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

2 Background

A table may have one or more B-tree indexes defined on it. Predicates on the index columns can be used to restrict the records that are retrieved. Index scans with such predicates are called *partial* index scans as opposed to *full* index scans. In a partial index scan, the optimizer estimates the *selectivity*, i.e., the fraction of records that are expected to be retrieved in the index scan. Methods for estimating the selectivity are well known [6] and are not discussed here.

For an access plan involving an index scan, the optimizer estimates the number of data page fetches. The number of page fetches from disk may depend on the number of buffer pool slots available to hold the pages fetched. As in most relational database systems, the buffer pool is assumed to be managed using the *L*east *R*ecently *U*sed (LRU) algorithm.

Term	Notation
No. of pages in buffer pool	$\mathcal{B}$
No. of pages in table	$\mathcal{T}$
No. of records in table	N
No. of distinct values in index	$\mathcal{I}$
No. of pages accessed in scan on index	$\mathcal{A}$
No. of pages fetched in scan on index	$\mathcal{F}$
Selectivity of start and stop conditions	σ
Selectivity of index sargable predicates	$\mathcal{S}$
Clustering factor	$\mathcal{C}$ or CR

Table 1: Notation

In Table 1, we list some of the notation that we use in this paper. The number of pages in a table is denoted by $\mathcal{T}$. For a table scan, the number of page fetches is exactly $\mathcal{T}$ because each page is accessed exactly once. Note that the number of page fetches is independent of the value of the buffer pool size, which is denoted by $\mathcal{B}$.

The number of data pages *accessed* during the scan of an index is denoted by $\mathcal{A}$. A data page is accessed if at least one record on the page is examined during the scan. The value of $\mathcal{A}$ depends on the number of records retrieved in the index scan and the placement of the retrieved records in the pages of the table. It does *not* depend on $\mathcal{B}$. Finally, the number of data pages *fetched* while scanning the index is denoted by $\mathcal{F}$. The value of $\mathcal{F}$ depends on $\mathcal{A}$ and may depend on $\mathcal{B}$ as shown below.

The placement of the retrieved records in the pages of the table determines how clustered the index is. An index is called a *clustered* index if the records are stored in the table in the order of the index column. When the records are retrieved in the order of values of the index column, no page is accessed more than once. Hence, $\mathcal{F} \equiv \mathcal{A}$ *independent* of $\mathcal{B}$. If this is a partial scan, $\mathcal{A} < \mathcal{T} \Longrightarrow \mathcal{F} < \mathcal{T}$.

An index is said to be *unclustered* if the records in the table are not stored exactly in the order of the index column. When the records are retrieved in the order of values of the index column, a page may be accessed more than once. The page may be replaced in the buffer pool between two accesses to the page due to other page fetches in the interim.

Clearly, one significant parameter that affects $\mathcal{F}$ is the value of $\mathcal{B}$. As $\mathcal{B}$ increases, the buffer may be able to compensate for any lack of order in the page reference pattern. When $\mathcal{B}$ approaches $\mathcal{A}$, disorganization in the key sequence of records becomes irrelevant. Similarly, as the buffer becomes smaller, even a slightly unclustered index will have to redo many page fetches, since the accessed pages will already have been discarded by some previous reference. In the worst case, each new record will require an additional page fetch. Bounds can therefore be placed on $\mathcal{F}$ as follows: $\mathcal{A} \leq \mathcal{F} \leq N$, where N denotes the number of records fetched.

Some indexes are greatly affected by even a small change in the size of the buffer pool. The degree to which the changes occur depends on the amount of disorganization that exists. If there are multiple records on a page and the index is highly unclustered, an index scan may result in multiple accesses to pages. Hence, $\mathcal{F}$ can be large if $\mathcal{B}$ is sufficiently small compared to $\mathcal{A}$.

In Figure 1, we show how the number of page fetches for a full index scan varies with the buffer size. The curves are shown for indexes over columns CMAC.BRAN, CMAC.CEDT, INAP.APLD, INAP.MALD and INAP.UWID in the Great-West Life benchmark database [8]. These curves are called *FPF* (for **F**ull index scan **P**age **F**etch) curves. $\mathcal{B}$ is expressed as a fraction of the number of pages in the table $\mathcal{T}$. $\mathcal{F}$ is expressed in multiples of $\mathcal{T}$. For a full index scan, the minimum value of $\mathcal{F}$ is $\mathcal{T}$ (in Figure 1, this would correspond to a value of 1). We see that the value of $\mathcal{F}$ can be quite sensitive to the buffer size $\mathcal{B}$ available.

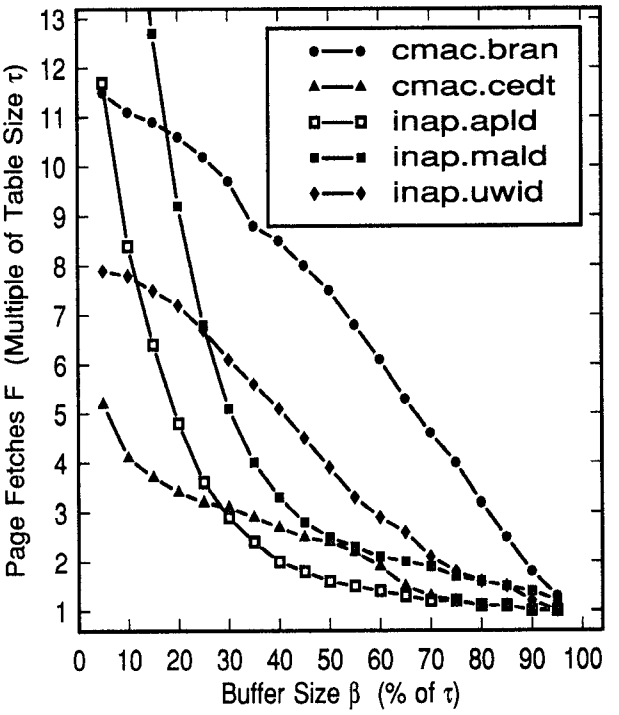

Figure 1: Data Page Fetches for a Full Index Scan: Sensitivity to Buffer Size

Hence, in order to choose a good access plan involving an index, it is crucial to accurately estimate the number of page fetches $\mathcal{F}$. The index scan may be a partial scan or a full scan. In this paper we describe an algorithm called **EPFIS** (**E**stimation of **P**age **F**etches in **I**ndex **S**cans). Algorithm **EPFIS** is given an estimate of the number of records being retrieved by the index scan and the buffer size $\mathcal{B}$ available for the index scan. Let us discuss the problem of choosing an access method in more detail. We wish to access a table and retrieve either all the records or some subset of the records in the table. If a subset is desired, one or more predicates are given that determine the records to be retrieved.

Let an index be defined on columns a and b with a as the major column. *Starting* and *stopping conditions* can be used to limit the range of the index scan. Examples of starting conditions are $a > 50$ and $a >= 25$. Examples of stopping conditions are $a < 75$ and $a <= 100$. Starting and stopping conditions can be combined, e.g., $40 <= a$ AND $a < 60$. Let the selectivity of the starting and stopping conditions be denoted by σ, i.e.,

the fraction of records satisfying the predicates.

We can have other predicates on the index columns that do not define a contiguous range of values and hence do not restrict the range of the index that needs to be scanned. We refer to these predicates as *index sargable* predicates. For example, the predicate $b = 5$, where b is not the major column of the index, is an index-sargable predicate. Let the selectivity of the index sargable predicates be denoted by S.

An index on a table is said to be *relevant* if any of the following conditions are satisfied:

1. One or more of the predicates can be used to form starting and/or stopping conditions on the index.
2. Records retrieved using the index would be in the desired sort order.

The number of basic access plans to be considered is the number of relevant indexes plus one (for the table scan). We are assuming that there is no RID-list sort, union or intersection before the data records are fetched.

In order to choose between the different access plans, the optimizer has to determine the costs of the different access plans. A major component of the cost of an access plan is the number of page fetches from secondary storage (disk) that are required under the plan. For an access plan involving an index scan, the optimizer has to estimate the number of page fetches.

The number of records remaining after applying starting and stopping conditions and index sargable predicates forms an upper bound on the number of pages fetched. Using the independence assumption, the number of qualifying records is given by $N \times \sigma \times S$.

3 Previous Work

Others have studied the estimation of page fetches when the records are selected at random (with or without replacement) [1, 9, 11]. In addition to the random selection assumption, it is usually assumed that the records are placed randomly on the pages. Attempts have been made to relax these assumptions [2], but these models usually require assuming some probability distributions on the data. Further, an infinite buffer is usually assumed whereas, in practice, the finite size of the buffer can have a large effect on the number of page fetches. Elsewhere, in [12], the number of block accesses when attributes are correlated is estimated using an occupancy model. The effect of clustering on efficient assembly of complex objects is studied in [3].

Mackert and Lohman [5] consider the effect of finite buffers managed using the LRU replacement policy. They propose an algorithm (we label it as Algorithm **ML**) for estimating page fetches for unclustered index scans. We also study three other algorithms that are based on calculating a quantity called "cluster ratio" which is an indication of how clustered the index is. These algorithms are labeled **DC**, **SD**, and **OT**[1]. We describe these algorithms in more detail below.

The number of records in the table is denoted by N. Let the index being scanned have $\mathcal{I}$ distinct key values.

3.1 Algorithm ML

In [5], an iterative formula is developed to estimate the number of page fetches to fetch x key values. The basic idea is to have a moving window of a single buffer size and use it to extrapolate probabilistically to any buffer size. Let $R = N/\mathcal{T}$ and let $D = N/\mathcal{I}$. Then the number of pages fetched from disk for retrieving all tuples that match x key values is estimated by:

$$\begin{cases} \mathcal{T}(1-q^x) & \text{if } x \le n \\ \mathcal{T}(1-q^n) + (x-n)\mathcal{T}pq^n & \text{if } n < x \le \mathcal{I} \end{cases}$$

where

$$q = \begin{cases} \left(1-\frac{1}{\mathcal{T}}\right)^D & \text{if } D \le R \\ \left(1-\frac{1}{\mathcal{I}}\right)^R & \text{if } D > R \end{cases}$$

$$p = 1 - q$$

$$n \equiv \max\{j \in \{0, 1, \cdots, \mathcal{I}\} \mid \mathcal{T}(1-q^j) \le \mathcal{B}\}.$$

Approximations to this formula for easy computation were also derived. We embodied the iterative formula in an algorithm we term Algorithm **ML** for use in the experimental comparisons.

3.2 Algorithm DC

A cluster ratio CR is calculated as follows. A "cluster" counter CC is initialized to zero. The index entries are scanned in key sequence order. CC is incremented by one if the first page containing the records of the next key value is the same or a higher page than the last page containing the records of the previous key value. Then,

$$\mathrm{CR} = \min\left(1, \left(\frac{CC}{\mathcal{I}} + \min\left(0.4, 5\ln\left(\frac{\mathcal{T}}{\mathcal{I}}\right)\right)\right)\right)$$

and the number of page fetches is calculated as

$$\sigma(\mathcal{T} + (1 - \mathrm{CR})(N - \mathcal{T}))$$

3.3 Algorithm SD

A cluster ratio CR is calculated as follows. The index entries are scanned in key sequence order. The number of page fetches with a buffer pool of one page is obtained (denote this by J). Then,

$$\mathrm{CR} = \frac{N - J}{N - \mathcal{T}}$$

[1] These algorithms have been abstracted from the internal algorithms of existing database products.

which essentially measures the number of "jumps" from one page to another above the minimum number of jumps needed (this is one measure of how the index is unclustered). Let

$$U = \sigma \times \mathcal{I}(\mathcal{T}(1-(1-1/\mathcal{T})^{\mathcal{T}/\mathcal{I}})$$

Here Cardenas's formula [1] is used to estimate the number of pages fetched for random location of tuples on pages.

$$V = \begin{cases} \min(U, \mathcal{T}) & \text{if } \mathcal{T} < \mathcal{B} \\ U & \text{otherwise} \end{cases}$$

The number of page fetches is calculated as

$$CR \times \mathcal{T} \times \sigma + (1 - CR)V$$

where the weighting captures the intuition behind the cluster factor CR. If the index is perfectly clustered (CR = 1), exactly $\sigma\mathcal{T}$ pages are fetched. Otherwise, according to the degree of clustering, up to V pages may be fetched.

3.4 Algorithm OT

The index entries are scanned in key sequence order. The number of page fetches with a buffer pool of three pages is obtained (denote this by J). The "cluster ratio" CR is calculated as follows:

$$CR = \frac{N + \mathcal{T} - J}{N}$$

This is an alternative calculation of CR using a slightly different definition of jumps. Then the number of page fetches is calculated as

$$\sigma(\mathcal{T} + (1 - CR)(N - \mathcal{T}))$$

If the index is perfectly clustered (CR = 1), exactly $\sigma\mathcal{T}$ pages are fetched. Otherwise, according to the degree of clustering, up to N pages may be fetched.

The very first attempts at modeling page fetches assumed that an index was either perfectly clustered ($\mathcal{F} = \mathcal{T}$) or perfectly unclustered ($\mathcal{F} = N$). Later work (as described above) relied on obtaining and using a single statistic (cluster factor) derived from the page fetch data. The use of a single statistic is based on probabilistic models that are not often valid for real data. Based on the page fetch data of the index, we obtain a function that maps from the number of records touched and the buffer size to the number of pages fetched. Our model which is based on empirical data enables us to predict page fetches much more accurately.

4 Algorithm EPFIS

Algorithm **EPFIS** has two components. The first component, Subprogram **LRU-Fit**, is run as part of the statistical collection routines in the database. These routines are called periodically to calculate some parameter values that can be stored in the catalog. Some of these parameter values are then used by the second component, subprogram **Est-IO**. Subprogram **Est-IO** is called by the optimizer during query compilation. It provides an estimate of page fetches for an index scan. We describe the two subprograms in more detail below.

4.1 Subprogram LRU-Fit

Subprogram **LRU-Fit** first determines the range of buffer pool sizes that need to be modeled. It then obtains a table of *FPF* (for **F**ull index scan **P**age **F**etch) data in the range of interest. **LRU-Fit** approximates the *FPF* curve using some number of line segments. Subprogram **Est-IO** uses the approximated *FPF* curve in the estimation of page fetches.

Determining Modeling Range

We need to know the range of buffer pool sizes ($\mathcal{B}$) that are likely to be encountered during optimization. The optimizer will need accurate estimates of $\mathcal{F}$ for these values of $\mathcal{B}$. If desired, the range of $\mathcal{B}$ can be specified by the database administrator (DBA). If the range is not specified, **LRU-Fit** chooses the range of $\mathcal{B}$ values to model as follows. The minimum value of $\mathcal{B}$ (denoted by $\mathcal{B}_{\min}$) is taken to be $\max(0.01 \times \mathcal{T}, \mathcal{B}_{\text{sml}})$, where $\mathcal{B}_{\text{sml}}$ is the smallest buffer pool size modeled. $\mathcal{B}_{\text{sml}}$ is chosen to avoid the large effects on page fetches due to too small a buffer size. In our experiments, we set $\mathcal{B}_{\text{sml}} = 12$. The maximum value of $\mathcal{B}$ (denoted by $\mathcal{B}_{\max}$) is taken to be $\mathcal{T}$, the size of the table in pages.

LRU-Fit will next determine the number of page fetches for a full index scan at selected $\mathcal{B}$ values in the range determined above. These $\mathcal{B}$ values are denoted by $\mathcal{B}_1, \mathcal{B}_2, \cdots, \mathcal{B}_k$, with $\mathcal{B}_1 = \mathcal{B}_{\min}$ and $\mathcal{B}_k = \mathcal{B}_{\max}$. The values $\mathcal{B}_2, \cdots, \mathcal{B}_{k-1}$ are equally spaced and are obtained using the following heuristic formula [2]:

$$\mathcal{B}_{i+1} = \mathcal{B}_i + 2 \times \sqrt{\mathcal{B}_{\max} - \mathcal{B}_{\min}}, \quad 1 \le i < k$$

This allows an increased number of buffer size values to be modeled for larger ranges but the increase is slower than the increase in the range size. If the buffer pool size falls outside of the range ($\mathcal{B}_{\min} \cdots \mathcal{B}_{\max}$), extrapolation is used to generate page fetch estimates.

[2] Goetz Graefe suggests $\mathcal{B}_i = \mathcal{B}_{\min}\left(\sqrt[k]{\mathcal{B}_{\max}/\mathcal{B}_{\min}}\right)^i$

Obtaining the *FPF* Data

A full scan of all the index entries produces the sequence of page numbers as stored in the index. A scan of the index for index statistics collection has exactly these characteristics. We can simulate a buffer pool of size $\mathcal{B}$ and use the sequence of page accesses in order to determine the number of page fetches that would result from a full index scan. In order to simultaneously perform this simulation for a number of buffer pool sizes without maintaining that many buffer pools, the *stack* property of the LRU algorithm [7] is used to do the simulation using a buffer pool of the largest size. The computational expense due to linear search of the buffer pool is avoided by using hash tables of buffer pages. **LRU-Fit** uses these techniques in order to generate the table of *FPF* data, consisting of $(\mathcal{B}_i, \mathcal{F}_i)$ pairs, where $\mathcal{F}_i$ is the number of page fetches for the full index scan corresponding to buffer size $\mathcal{B}_i$. For examples of *FPF* curves, see Figure 1 in Section 2.

Let the number of page fetches for a buffer size of $\mathcal{B}_{\min}$ pages be denoted by $\mathcal{F}_{\min}$. In the single pass over the trace, **LRU-Fit** also determines the value of $\mathcal{F}_{\min}$. This value is used to calculate the value of the "clustering factor" (denoted by $\mathcal{C}$) as follows. Then,

$$\mathcal{C} = \frac{N - \mathcal{F}_{\min}}{N - \mathcal{T}}$$

using a similar intuition as for Algorithm **SD** in Section 3. Here, $\mathcal{C}$ is a measure of how "clustered" the index is and will be in the range $[0, 1]$. If $\mathcal{C} \approx 0$, the index is very unclustered and records are located at random on pages. The "degree" of clustering tends to increase as $\mathcal{C} \to 1$.

Approximating the *FPF* Curve

We wish to use the *FPF* curve subsequently for estimation. To reduce the amount of data that needs to be stored, we approximate the *FPF* curve. Any approximation method that permits sufficiently accurate approximation, e.g., polynomial curve fitting, could be used. We use the simple but adequate method of approximating the *FPF* curve using line segments. The line segment information is captured by storing the coordinates of the end-points of the line segments. This coordinate information can be stored in a system catalog entry associated with the index for later use by **Est-IO**.

Clearly, the larger the number of line segments, the more accurate the approximation. However, for each additional line segment, an additional pair of values needs to be stored in the catalog. If space usage in the catalog structure is of concern, it is desirable to keep the minimum number of line segments that result in acceptable errors in page fetch estimation. We performed a large number of experiments on different indexes to study the sensitivity of the estimation errors to the number of approximating line segments. The experiments show that the estimation errors do not change very much when the number of line segments is greater than five. Hence, we use six line segments to approximate the *FPF* curves.

4.2 Subprogram Est-IO

As discussed in Section 2, the optimizer often needs to choose between scanning a table or scanning one of its relevant indexes. For a relevant index, the optimizer determines the applicable predicates. Currently, the database administrator specifies the buffer size available for the scan. In order to estimate the number of page fetches required for the scan on an index, the optimizer calls subprogram **Est-IO**.

Subprogram **Est-IO** uses the approximation to the *FPF* curve obtained by **LRU-Fit**. Let the number of page fetches resulting from a *full* scan on an index, given a buffer size of $\mathcal{B}$, be denoted by $\mathrm{PF}_{\mathcal{B}}$. **Est-IO** first determines which line segment contains the mapping value for the value $\mathcal{B}$. It then uses the equation for the line segment to calculate the corresponding number of page fetches. This is the value of $\mathrm{PF}_{\mathcal{B}}$.

Subprogram **Est-IO** then scales down the value of $\mathrm{PF}_{\mathcal{B}}$ by σ in order to obtain the number of page fetches corresponding to the scan on the index (possibly restricted by starting and stopping conditions). Thus, it estimates the number of page fetches by $\sigma \times \mathrm{PF}_{\mathcal{B}}$.

Correcting for Small Selectivity σ

Experiments show that the above method tends to significantly underestimate the number of page fetches when the following conditions hold together:

1. The selectivity σ is small.
2. $\phi = max(1, \mathcal{B}/\mathcal{T})$ is significantly greater than σ.
3. The index is not very clustered. This is reflected by a value of $\mathcal{C}$ not close to 1.

Underestimation occurs because we are scaling down from full scans to partial scans using a linear scaling factor. Now, if σ is large enough this is not a problem because behavior in the large still holds. However if σ is small enough, the caching that helps larger scans (when buffer size is large enough) does not take effect for the partial scan. Hence, the number of page fetches is larger than the linear scaling would predict.

An indicator variable $\mathcal{V}$ is used as follows: if $\phi \geq 3\sigma$, then $\mathcal{V}$ is 1, otherwise it is 0. Then, a new estimate for the number of page fetches is given by

$$\begin{aligned} &(\sigma \times \mathrm{PF}_{\mathcal{B}}) + \\ &\mathcal{V} \times \min(1, \phi/(6\sigma)) \times \\ &(1 - \mathcal{C}) \times (\mathcal{T} \times (1 - (1 - 1/\mathcal{T})^{\sigma N})) \end{aligned} \qquad (1)$$

Here, we are using the heuristic correction term

$$\min(1, \phi/(6\sigma)) \times (1 - \mathcal{C}) \times (\mathcal{T} \times (1 - (1 - 1/\mathcal{T})^{\sigma N}))$$

when $\sigma << 1/3$ and $\sigma << \mathcal{B}/\mathcal{T}$. The factor $(\mathcal{T} \times (1 - (1 - 1/\mathcal{T})^{\sigma N}))$ is known in the literature as Cardenas's formula [1]. Cardenas derived the formula under the assumption that the σN records are randomly selected from N records with replacement. The more unclustered an index is, the more likely it is that a partial index scan looks like a random selection. If the index is very clustered, i.e., $\mathcal{C}$ is close to 1, the second factor $(1 - \mathcal{C})$ will be small. Hence, the factor $(1 - \mathcal{C})$ is a measure of how unclustered the index is and it is used to reduce Cardenas's estimate. We observed that the term $\sigma \times \mathrm{PF}_{\mathcal{B}}$ tends to underestimate page fetches primarily when ϕ is large compared to σ. We take this into account by using the factor $(1, \phi/(6\sigma))$ to reduce the contribution of the second term.

Effect of Index Sargable Predicates

The index sargable predicates are applied to the index column values inspected during the (partial) index scan. Those records that qualify are then fetched. Hence, such predicates can have the effect of reducing the number of pages fetched. We use a simple urn model to estimate the effect of index sargable predicates. We first need to estimate the number of pages *referenced* after applying the starting and stopping conditions.

If the index is highly clustered ($\mathcal{C} \approx 1$), the number of pages referenced is close to $(\sigma\mathcal{T})$. If it is highly unclustered, the number of pages referenced is close to $(\min(\sigma N, \mathcal{T}))$. Using a simple linear model, we estimate the number of pages referenced after applying the starting and stopping conditions to be

$$Q = \mathcal{C}\sigma\mathcal{T} + (1 - \mathcal{C})\min(\mathcal{T}, \sigma N)$$

We use k to denote the number of qualifying records after index sargable predicates are applied.

$$k = S\sigma N$$

We can view the process as that of assigning k balls to Q urns that are initially empty. Using Cardenas's formula [1], we get that the expected number of non-empty urns as:

$$Q \times (1 - (1 - 1/Q)^k)$$

The factor by which the number of pages referenced is reduced is

$$\frac{Q \times (1 - (1 - 1/Q)^k)}{Q} = (1 - (1 - 1/Q)^k)$$

The number of page fetches is estimated to be reduced proportionately. Using Equation 1, the estimated number of page fetches ($\mathcal{F}$) taking index sargable predicates into account is:

$$\begin{aligned} \mathcal{F} \;=\; & (1 - (1 - 1/Q)^k) \times \\ & ((\sigma \times \mathrm{PF}_{\mathcal{B}}) + \\ & \mathcal{V} \times \min(1, \phi/(6\sigma)) \times \\ & (1 - \mathcal{C}) \times (\mathcal{T} \times (1 - (1 - 1/\mathcal{T})^{\sigma N}))) \end{aligned}$$

4.3 Complete Algorithm

Algorithm **EPFIS** consists of the following steps:

1. Determine the modeling range if not specified by the database administrator.
2. At statistics collection time, use LRU buffer pool modeling on the sequence of index page accesses to obtain the page fetches for different buffer pool sizes in the modeling range.
3. Approximate the page fetch curve by a small number of line segments.
4. At query compilation time, use the line segment approximation to determine the number of page fetches for a full index scan. The buffer size is specified by the optimizer.
5. Scale down the full scan page fetches by the selectivity of the starting and stopping conditions.
6. If necessary, use the heuristic correction described above.
7. Account for the effect of index sargable predicates on the number of page fetches.

5 Experimental Evaluation of Estimation Algorithms

We performed a number of experiments using both actual customer data and synthetic data to compare the accuracy of the estimates obtained by the algorithms described in Section 3 and Algorithm **EPFIS**. A partial scan is described by specifying the starting and stopping key values. Probability distributions were specified for partial scans as follows.

A scan is said to be *small* if it accesses only a small range of the table. It is said to be *large* otherwise. A small scan is modeled as follows. A random number, say, r, is generated between 0 and 0.2. A starting key value (say, k_1) is picked at random so that at least rN records have key values $\geq k_1$. The stopping key value (say, k_2) is found such that $k_2 \geq k_1$ and the number of records with key values in the range $[k_1, k_2]$ is $\geq rN$. We can thus generate a large number of small scans for experiments by choosing appropriate random numbers. Similarly, a large scan is modeled by generating the random number r to be between 0.2 and 1.

The algorithms do not exhibit uniform error behavior with respect to scan sizes. Hence, a mixture of scans was used for comparing the algorithms. For each data set we generated 200 random scans. The chance of picking a small scan was equal to that of picking a large scan.

We performed a number of other experiments where different mixes of scans were used. We ran experiments involving only small scans, only large scans, and only full scans. We also ran experiments where different ratios of small and large scans were used. In all these experiments the results were very similar to the results presented in Sections 5.1 and 5.2. A general trend was that the algorithms other than Algorithm **EPFIS** performed worse as the scan size was made larger.

For a given buffer size, we computed the error metric for an algorithm as follows. For any scan i ($1 \leq i \leq 200$), let the estimate obtained by the algorithm be denoted by e_i. Let the actual number of pages fetched be denoted by a_i. Then, the error metric is:

$$\sum_{1 \leq i \leq 200} (e_i - a_i) / \sum_{1 \leq i \leq 200} a_i$$

This error metric can be thought of as the relative error over the aggregate of all the scans. We choose not to use the mean of the individual relative error values as the error metric. The reason is that for small scans, the relative error values can be large but the absolute error values are usually small. For the optimizer it is the absolute difference that is important. Thus, we need to compare the absolute error values. The denominator in the error metric is identical for all the algorithms. The denominator is a normalizing factor so that the error metric that can be expressed as a meaningful fraction.

We computed the errors, i.e., the values for the above error metric, for buffer sizes in increments of 5% of the table size in pages ($\mathcal{T}$). The smallest buffer size checked was set to $\max(300, 0.05\mathcal{T})$ and the largest buffer size checked was $0.9\mathcal{T}$. In the graphs we show later, the buffer size is expressed as a percentage of $\mathcal{T}$ on the X-axis. The percentage error is shown on the Y-axis.

5.1 Customer Data

We compared the accuracy of the different estimation procedures on data from a customer database, the Great-West Life database [8]. We refer to this database as the **GWL** database. We selected 8 columns (labeled CMAC.BRAN, CMAC.CEDT, CAGD.CMAN, CAGD.POLN, INAP.APLD, INAP.MALD, INAP.UWID and PLON.CLID) from four tables (labeled CMAC, CAGD, INAP and PLON) in this database. All these columns had indexes defined on them. The tables had differing number of records and records per page (see Table 2). We selected columns showing a range of values with respect to column cardinalities (number of distinct values) and degrees of clustering as measured by the value of $\mathcal{C}$ (see Table 3).

Table	No. of Pages	No. of Records/Page
CMAC	774	20
CAGD	1093	104
INAP	1945	76
PLON	4857	123

Table 2: **GWL** Database Tables

Column	Col Card	$\mathcal{C}$(%)
CMAC.BRAN	131	43.3
CMAC.CEDT	2829	64.6
CAGD.CMAN	6155	35.3
CAGD.POLN	110074	99.6
INAP.APLD	729	79.4
INAP.MALD	517	64.3
INAP.UWID	60	90.8
PLON.CLID	437654	23.6

Table 3: **GWL** Database Columns

In Figures 2 through 9 we show the error behavior of Algorithm **EPFIS** and the algorithms described in Section 3.

We see that Algorithm **EPFIS** exhibits much less error in its estimates than the other algorithms. **EPFIS** dominates the other algorithms for all the indexes. The maximum error for **EPFIS** never exceeds 20%. In addition, **EPFIS** is very stable exhibiting low errors over the entire range of buffer sizes. Note that, in Figure 5, all the algorithms except for **SD** have essentially identical errors. In some of the figures, e.g., Figure 8, Algorithm **DC** exhibits very high errors and has only a few data points shown.

The other algorithms perform much more poorly than **EPFIS**. The maximum errors for the other algorithms are as follows: **SD** (1889.7%), **OT** (2046.2%), **DC** (2876.4%), and **ML** (97.8%). We see that, except for Algorithm **ML**, in the worst case, the error can be orders of magnitude higher than the error for Algorithm **EPFIS**. Except for Algorithm **ML**, none of the other algorithms exhibit stability in errors over the entire range of buffer sizes.

5.2 Synthetic Data

We wished to compare the algorithms on a larger number of data sets. It is useful to complement the customer data with synthetic data in order to test the

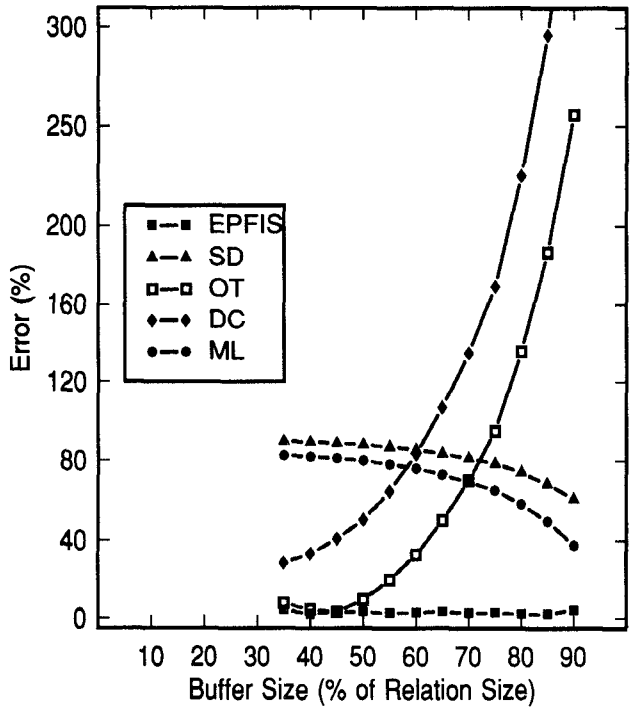

Figure 2: Error Behavior for CMAC.BRAN

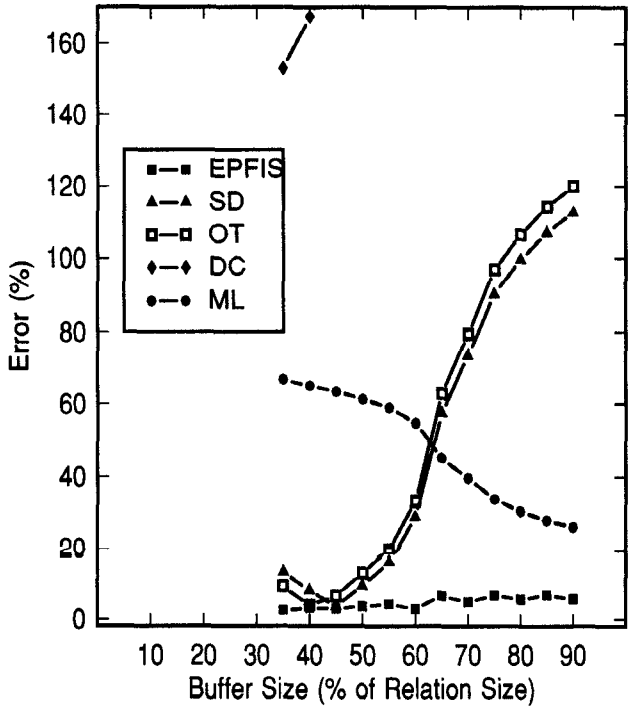

Figure 3: Error Behavior for CMAC.CEDT

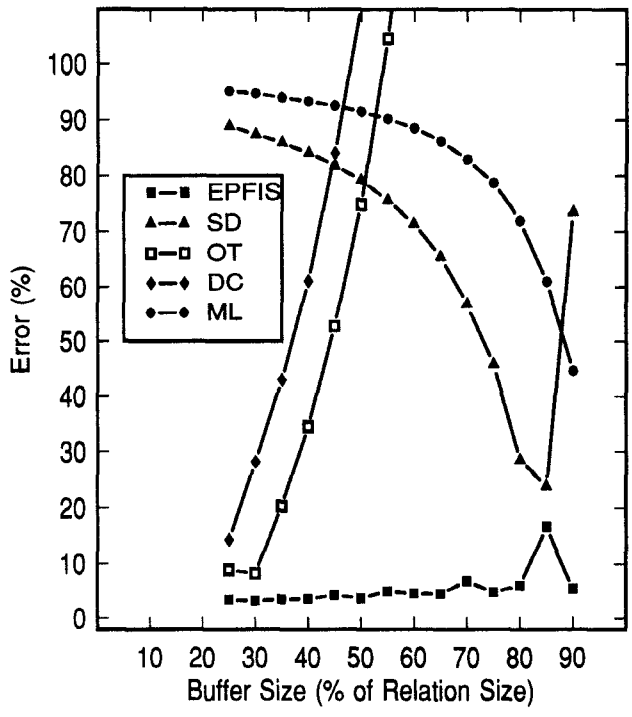

Figure 4: Error Behavior for CAGD.CMAN

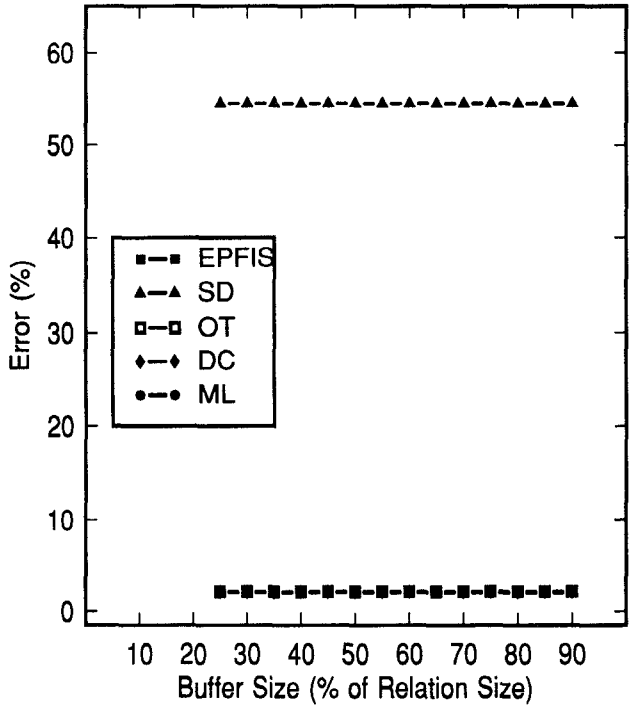

Figure 5: Error Behavior for CAGD.POLN

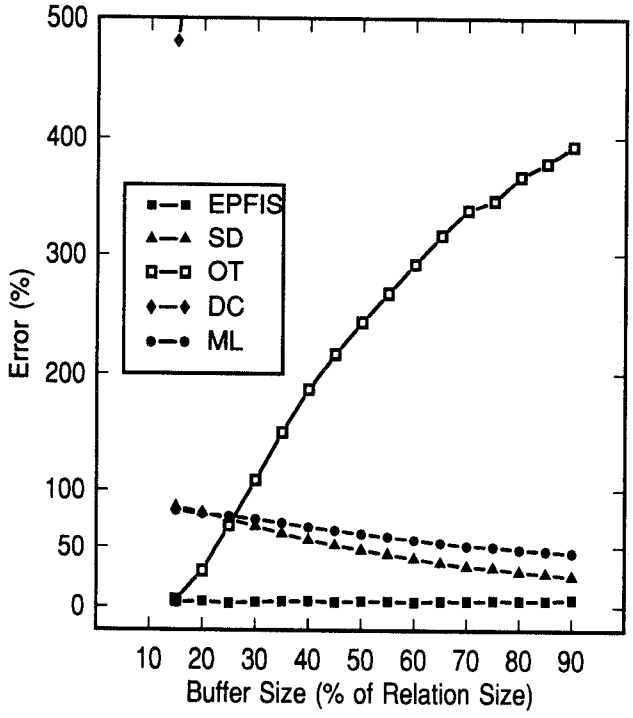

Figure 6: Error Behavior for INAP.APLD

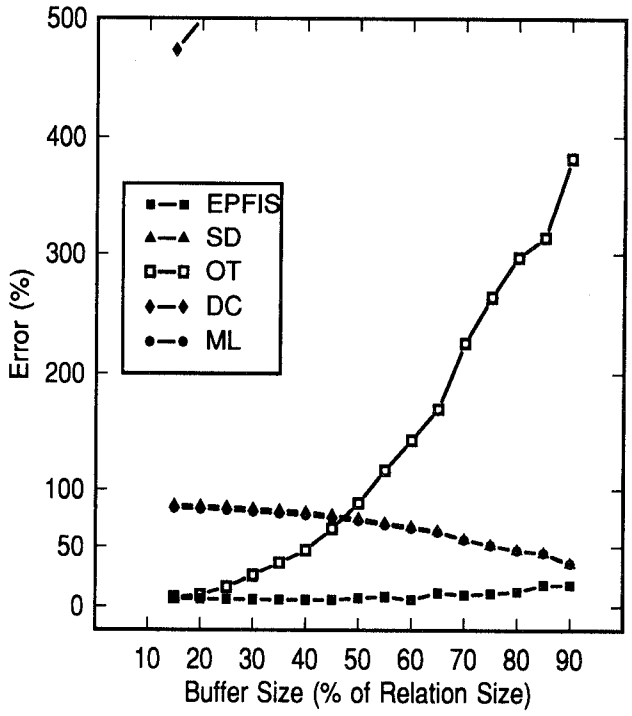

Figure 8: Error Behavior for INAP.UWID

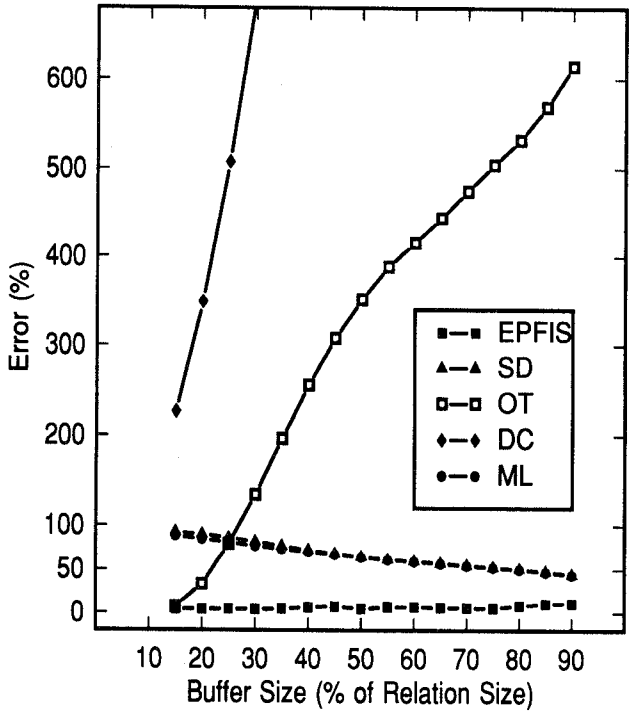

Figure 7: Error Behavior for INAP.MALD

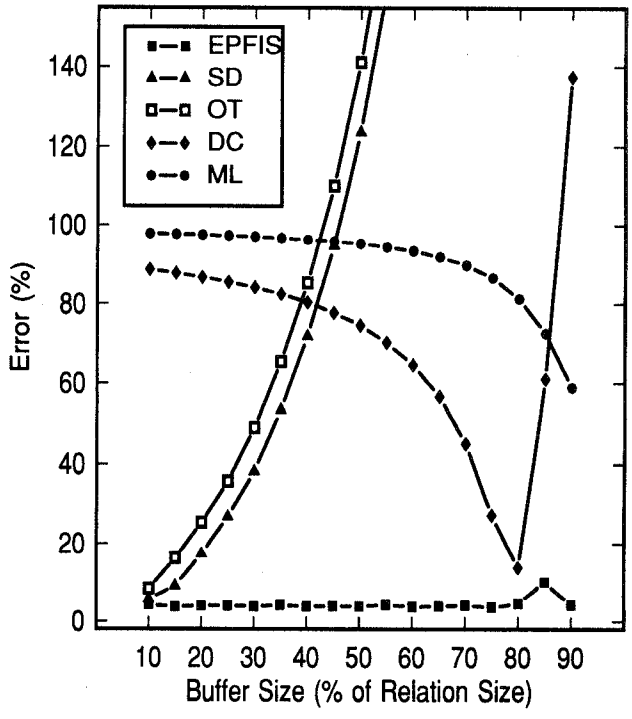

Figure 9: Error Behavior for PLON.CLID

algorithms with patterns of clustering not necessarily present in the available customer data. We generated data with differing degrees of clustering between the index entries and data records as described below.

The data sets were characterized by the following parameters. The range of values considered for each parameter is given in parentheses.

- number of records $N(10^6)$
- number of distinct values $\mathcal{I}(10^4)$
- number of records per page $\mathcal{R}(20, 40, 80)$
- generalized Zipf distribution of distinct values with parameter θ (0, 0.86)
- window size parameter $\mathcal{K}(0, 0.05, 0.10, 0.20, 0.50, 1)$

Zipf-like distributions have often been observed when the distribution of column values is skewed. Knuth in [4], page 398, describes a generalized Zipf distribution with a parameter θ that can be used to model distributions such as the uniform distribution ($\theta = 0$) or the "80-20" distribution ($\theta = 0.86$). We use this distribution and values for θ to model skew in the distribution of duplicates for distinct values.

To model correlation between frequent values, we modify a scheme described in [10]. The distinct values are processed in the order of their values. For each distinct value, its corresponding records are assigned to pages as follows. A window of pages is available and the records are assigned randomly in this window of pages. The smaller the window, the greater the degree of clustering. The window size is given by $\lceil \mathcal{KT} \rceil$. Thus, we model degrees of clustering ranging from high ($\mathcal{K} = 0, 0.05$) to low or none ($\mathcal{K} = 0.5, 1$).

When a page is full in the window, the next page not in the window is added to the window. The initial window is $[1, \mathcal{KT}]$. A small amount of noise in the assignment is permitted as follows. A record is assigned outside the window with a certain probability given by a noise factor. In our experiments, the noise factor was set to 5%, i.e., 95% of the time the records are assigned within the window.

For brevity, we will only show the experimental results for $\mathcal{R} = 40$ and $\mathcal{K} = 0.05, 0.50, 1$. The results for other values of these parameters are similar. The algorithms are compared in terms of their estimation errors in Figures 10 through 15.

Each figure corresponds to one combination of distribution parameter ($\theta = 0, 0.86$) and clustering parameter ($\mathcal{K} = 0.05, 0.50, 1$). In a few graphs, algorithms **OT** and **DC** do not appear, for example, Figure 10. This is because the algorithms exhibit errors that exceed the maximum error shown in the graphs ($\approx 100\%$).

We see that Algorithm **EPFIS** exhibits much less error in its estimates than the other algorithms. For

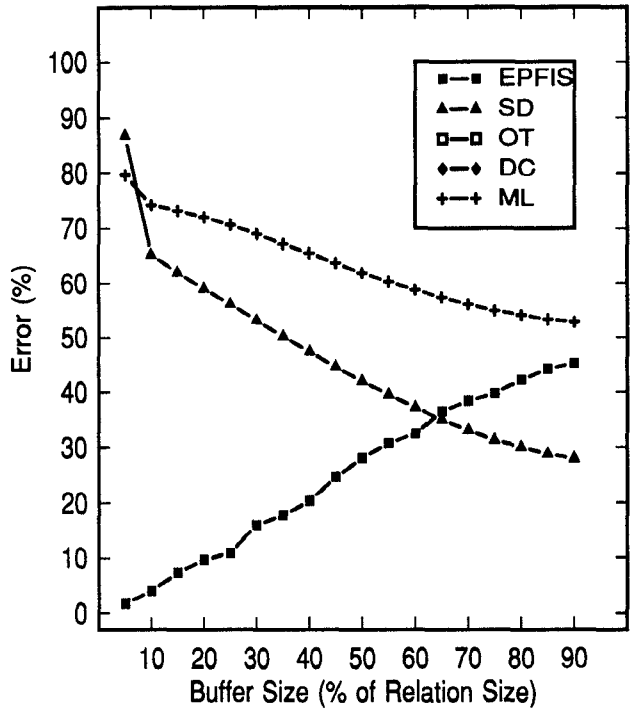

Figure 10: Error Behavior for $\theta = 0$, $\mathcal{K} = 0.05$

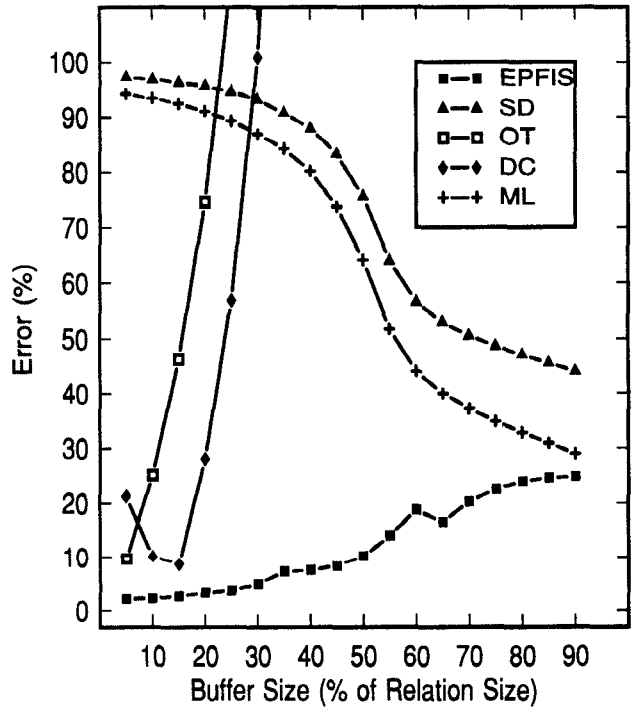

Figure 11: Error Behavior for $\theta = 0$, $\mathcal{K} = 0.50$

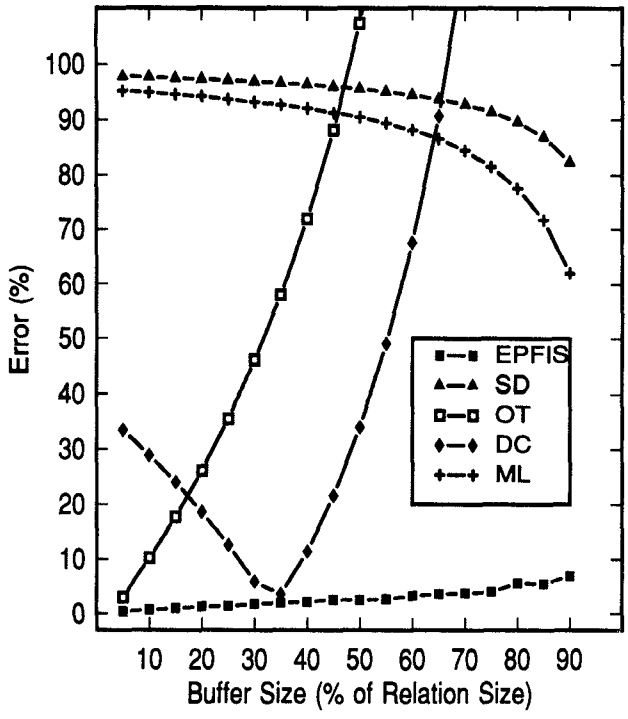

Figure 12: Error Behavior for $\theta = 0$, $\mathcal{K} = 1.0$

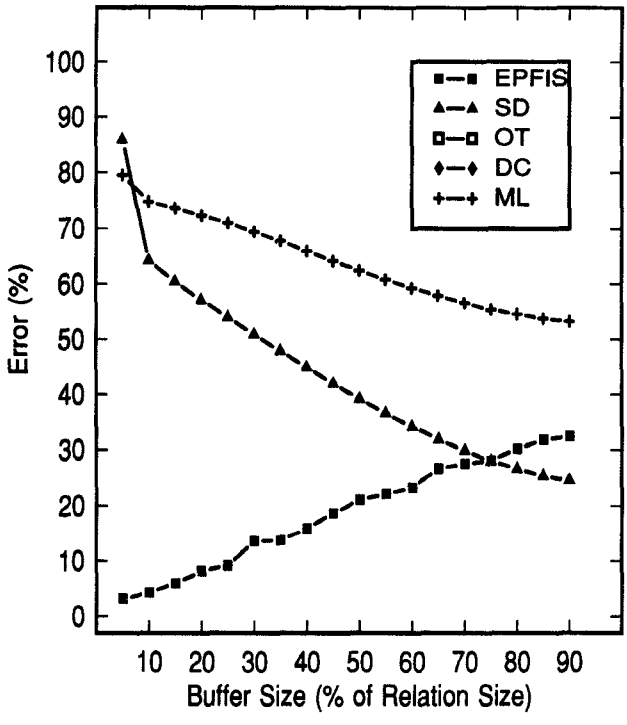

Figure 13: Error Behavior for $\theta = 0.86$, $\mathcal{K} = 0.05$

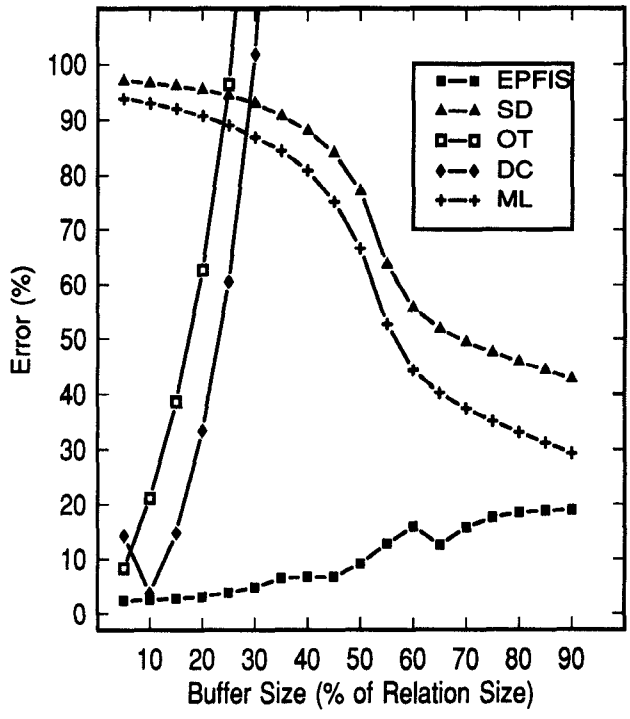

Figure 14: Error Behavior for $\theta = 0.86$, $\mathcal{K} = 0.50$

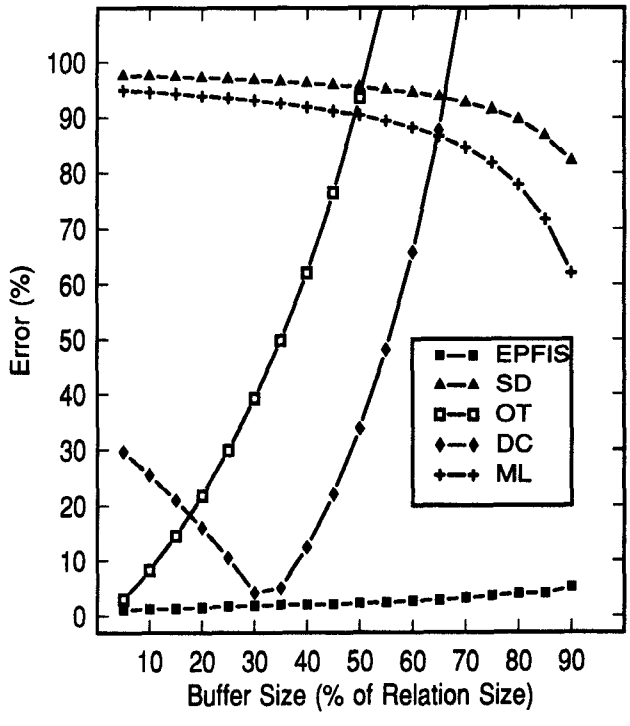

Figure 15: Error Behavior for $\theta = 0.86$, $\mathcal{K} = 1.0$

almost all the data sets, **EPFIS** dominates the other algorithms. No other algorithm dominates **EPFIS** for even a single data set. The maximum error for **EPFIS** is 48%. Also, **EPFIS** is very stable exhibiting low errors over the entire range of buffer sizes.

The other algorithms perform much worse than **EPFIS**. The maximum errors for the other algorithms are as follows: **SD** (97.6%), **OT** (2453.1%), **DC** (1994.8%), and **ML** (94.9%). We see that, except for Algorithms **SD** and **ML**, in the worst case, the error can be orders of magnitude higher than the error for Algorithm **EPFIS**. Except for Algorithms **SD** and **ML**, none of the other algorithms exhibit stability in errors over the entire range of buffer sizes.

6 Summary

In this paper we have described a new algorithm called **EPFIS** for estimating the number of page fetches for an index scan with a finite LRU buffer. Algorithm **EPFIS** performs LRU simulations on the index entries once. The data gathered as a result of these simulations is processed, summarized, and stored in the system catalogs. Subsequently, given the number of buffer pages available and the number of records being fetched, Algorithm **EPFIS** returns an estimate of the number of pages that will be fetched using the summary data in the catalog. Previous work was based on probabilistic models that are often not valid for real data. Our model which is based on empirical data enables us to predict page fetches much more accurately.

We compared Algorithm **EPFIS** to a number of other algorithms that are currently in use or have been proposed. Algorithm **EPFIS** dominates the other algorithms and exhibits good stability over the entire range of buffer sizes. The other algorithms exhibit much higher errors that can be orders of magnitude higher than the error of **EPFIS**. Algorithm **EPFIS** is not difficult to implement. During query optimization, the estimation procedure only involves computing a simple formula. The LRU simulations can be performed once while statistics are being gathered for other purposes. Hence, we believe that Algorithm **EPFIS** is the algorithm of choice.

Future work would consider the impact of some or all of the following: indexes with sorted RIDs for a given key value, use of multiple indexes, use of RID-list operations, index ANDing and ORing, intra-query contention, and multi-user contention.

Acknowledgements

We thank Goetz Graefe, Guy Lohman, John Lumby, Sheila Richardson, and Lori Strain for their comments.

References

[1] A. F. Cardenas. Analysis and Performance of Inverted Database Structures. *Communications of the ACM*, 18(5):253–263, May 1975.

[2] S. Christodoulakis. Estimating Block Selectivities. *Information Systems*, 9(1):69–79, 1984.

[3] T. Keller, G. Graefe, and D. Maier. Efficient Assembly of Complex Objects. *ACM-SIGMOD Record*, 20(3):148–157, September 1991.

[4] D. E. Knuth. *The Art of Computer Programming, Vol 3: Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[5] 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, September 1989.

[6] M. V. Mannino, P. Chu, and T. Sager. Statistical Profile Estimation in Database Systems. *ACM Computing Surveys*, 20(3):191–221, September 1988.

[7] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation Techniques for Storage Hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[8] G. C. Steindel and H. G. Madison. A Benchmark Comparison of DB2 and the DBC/1012. In *CMG '87, International Conference on Management and Performance Evaluation of Computer Systems*, pages 360–369, Orlando, FL, 1987. The Computer Measurement Group, Inc.

[9] S. J. Waters. Hit Ratios. *Computer Journal*, 19:21–24, 1976.

[10] J. Wolf, D. Dias, and P. Yu. An Effective Algorithm for Parallelizing Sort Merge Joins in the Presence of Data Skew. In *Proceedings of the 2nd International Symposium on Databases in Parallel and Distributed Systems*, pages 103–115. IEEE Computer Society, 1990.

[11] S. B. Yao. Approximating Block Accesses in Database Organizations. *Communications of the ACM*, 20(4):260–261, April 1977.

[12] B. T. Vander Zanden, H. M. Taylor, and D. Bitton. Estimating Block Accesses When Attributes Are Correlated. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, pages 119–127, Kyoto, Japan, August 1986. Morgan Kaufman.