Correlation Indices: a New Access Method to Exploit Correlated Attributes

by

George Huo

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in
Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2007
[June 2007]

Author .... [illegible] ....
Electrical Engineering and Computer Science
May 25, 2007

Certified by .... [illegible] ....
Samuel Madden
Assistant Professor
Thesis Supervisor

Accepted by .... [illegible] ....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Correlation Indices: a New Access Method to Exploit Correlated Attributes

by

George Huo

Submitted to the Department of Electrical Engineering and Computer Science
on May 25, 2007,
in partial fulfillment of the requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

In relational query processing, one generally chooses between two classes of access paths when performing a predicate lookup for which no clustered index is available. One option is to use an unclustered index. Another is to perform a complete sequential scan of the table. Online analytical processing (OLAP) workloads often do not benefit from the availability of unclustered indices; the cost of random disk I/O becomes prohibitive for all but the most selective queries. Unfortunately, this means that data warehouses and other OLAP systems frequently perform sequential scans, unless they can satisfy nearly all of the queries posed to them by a single clustered index [7], or unless they have available specialized data structures – like bitmap indices, materialized views, or cubes – to answer queries directly.

This thesis presents a new index data structure called a correlation index (CI) that enables OLAP databases to answer a wider range of queries from a single clustered index or sorted file. The CI exploits correlations between the key attribute of a clustered index and other unclustered attributes in the table. In order to predict when CIs will exhibit wins over alternative access methods, the thesis describes an analytical cost model that is suitable for integration with existing query optimizers. An implementation compares CI performance against sequential scans and unclustered B+Tree indices in the popular Berkeley DB [22] library.

Experimental results over three different data sets validate the accuracy of the cost model and establish numerous cases where CIs accelerate lookup times by 5 to 20 times over both unclustered B+Trees and sequential scans. The strong experimental results suggest that CIs offer practical and substantial benefits in a variety of useful query scenarios.

Thesis Supervisor: Samuel Madden
Title: Assistant Professor

Acknowledgments

I am indebted to my advisor, Sam Madden, for all of his invaluable support throughout my graduate experience. The ideas in this thesis are primarily due to the discussions that we've had over the year. The process of learning how to conduct research, to teach, to think on a graduate level can go wrong in many ways. Sam has provided just the right balance of guidance and freedom while always keeping his students' best interests foremost, and I have grown all the more because of it.

I'd like to thank Adam Marcus for always being available to bounce ideas around with, from daily programming details to our SUL-Tree research. Having the opportunity to work with Adam taught me a great deal about how to collaborate effectively.

I thank anonymous referees who have provided many useful comments that I've integrated into portions of the thesis.

Finally, I thank my family. I could not have made it without their support and sacrifice.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Database management systems (DBMSs) have become a central component of nearly all large software applications. One of the basic services provided by a DBMS is the ability to look up the records that satisfy a set of conditions in a user-specified predicate efficiently. Efficient lookups are fundamental to database systems not only because they support direct user queries, but also because they underlie many other database operations, such as aggregates and joins.

To implement such lookups efficiently, database systems employ *indices* over the columns in a table. An index is a data structure that is overlaid upon the table in order to be able to locate desired portions without reading the entire table. Traditionally, the B+Tree is the most common indexing structure in relational databases. Since an index represents some organization of the data, it is associated with the notion of a particular ordering of the records. We say that an index is *clustered* if the data are arranged on disk physically according to the order that the index represents. Unclustered indices lack many of the beneficial performance characteristics of clustered indices, especially as the size of the database table grows large. In particular, traditional unclustered indices are useful to look up specific values that occur infrequently, but queries that access more than a small fraction of the table become impractically slow.

1.1 Motivation for a new index structure

Online analytical processing (OLAP) workloads often do not benefit from the availability of unclustered indices. This is because queries in OLAP databases usually involve aggregation over regions of very large tables, instead of highly selective value lookups.[1] The overhead of disk seeks to fetch the data pages pointed to by the leaf records in unclustered B+Trees will be higher than the cost to scan the table if even a small fraction of tuples in a table are accessed by a query. Clustered indices (and sorted files in some warehouse systems) perform better – beating sequential scans for many queries of even relatively low selectivity – but most database systems limit users to a single clustered index or sort order per table.

Unfortunately, this means that data warehouses and other OLAP systems will frequently perform sequential scans, unless almost all of the queries posed to them can be satisfied by a single clustered index [7], or unless specialized data structures – like bitmap indices, materialized views, or cubes – can be used to answer queries directly. In fact, Netezza, a popular appliance-based data warehouse, uses no indexing, relying exclusively on sequential scans because it is hard to pick one index that will perform well for a range of warehouse queries.[2] Although there are many attempts to optimize sequential scans (Netezza relies on special disk controllers and massive shared-nothing parallelism), ultimately any full table scan can be quite slow. An index structure that leverages a particular table clustering to provide efficient lookups for more than one column, therefore, may yield benefits wherever systems currently do no better than sequential scans.

[1]We adopt the convention that a highly selective query returns fewer tuples.
[2]Netezza whitepapers [18] say "no indexing required."

1.2 Overview of the correlation index

This thesis introduces a new index data structure called a *correlation index* (CI) that allows OLAP databases to answer a wider range of queries from a single clustered index. The idea is to exploit correlations between the key attribute of a clustered index or sorted file (the "clustered attribute") and other "unclustered attributes" in the table. The simplest form of a correlation index is just a table that maps from each value u in the domain of an unclustered attribute A_u to values of the clustered attribute A_c that co-occur with u in some tuple in the database. Then, queries over A_u can be answered by looking up the A_c values co-occurring with u in the clustered index to find potentially matching tuples.[3]

If there is a high degree of correlation, then each u will co-occur with only a few clustered attribute values, such that answering queries over the unclustered attribute requires only marginally more lookups than would be required had a clustered index over A_u been available. Because these lookups are performed directly on the clustered index, the matching tuples can be read directly from the index in sorted order in a very efficient manner (e.g., without performing a disk seek for each additional tuple, as an unclustered index requires).

Obviously, this technique will not work for all attribute pairs. If there is no correlation between the clustered attribute and A_u, each value in A_u will map to many values in the index, and the efficiency of the approach will be quite low. Hence, one of the challenges of using correlation indices is determining when they will be effective and when they will be costly. This thesis presents an algorithm for determining the expected effectiveness of a correlated index based on simple statistics.

[3]This is more general than a standard multicolumn index with the clustered attribute as the lead index column, since the user is not required to specify a predicate over the clustered attribute in the query.

1.3 Background and related work

In this section, we survey past work in semantic query optimization (SQO), soft constraints, and the bitmap scan access method related to the ideas presented in the thesis. We also provide background on sampling-based approaches to distinct value estimation, a technique we exploit in § 4.1.2 to calculate statistics for our cost model.

1.3.1 Semantic query optimization

One can view our work as an extension of certain optimization approaches from the field of semantic query optimization; there has been a long history of work in this area [5,13,16,21]. The basic idea is to exploit various types of integrity constraints (often expressed as rules [3,4,14,15,17,20,23]) – either specified by the user or derived from the database – to eliminate redundant expressions in the query or to find more selective access paths during query optimization.

Past work in this area has studied several problems that bear some resemblance to correlation indices. Cheng et al. [8] describe as one of their optimizations *predicate introduction* (which was originally proposed by Chakravarthy et al [5]), in which the SQO injects new predicates in the `WHERE` clause of a query based on constraints that it can infer about relevant table attributes. Predicate introduction has traditionally focused on two approaches: index introduction and scan reduction. In the former case, a predicate is introduced naming a new attribute that may have an index available, which may then open up new possibilities for the query optimizer. In scan reduction, the introduction of range predicates can reduce the number of tuples that qualify for a join. These SQO techniques traditionally adopt a two-phase approach, where queries that are logically equivalent are first generated by query rewriting [8] and then submitted to the downstream query optimizer, from which the least expensive plan can be chosen. Thus, the effects of predicate introduction can be quite similar to those achieved by the CI for some clustered indices.

Gryz et al. [12] propose a technique for deriving what are called "check constraints," which are basically linear correlations between attributes with error bounds (e.g., "salary = age * 1008 +/- 20000") and show that these relationships exist in data like TPC-H. They also look at a "partitioning" technique for string-valued attributes that finds cases where when an attribute X takes on a particular value v, some other attribute Y has a bounded range $[a \ldots b]$. They show that these correlations can subsequently be exploited using predicate introduction over the detected constraint rules. Our approach generalizes Gryz et al.'s results in the context of indexing, because it can capture these relationships as well as non-linear relationships (such as the fact that city names are correlated with states, even though one city may occur in many states).

Godfrey et al. [11] have looked extensively at discovering and utilizing "soft constraints" for semantic query optimization. They classify these integrity constraints as *absolute soft constraints,* which hold with no exceptions in the current state of a database, and *statistical soft constraints,* which can have some degree of violation. They explain that such constraints can be used in query rewrite, query plan parameterization, and cardinality estimation in the optimizer's cost model for tighter guesses on selectivity factors.

However, the fact that their soft constraints capture only logical relationships between table attributes means that they must keep track of when the constraint no longer holds to invalidate the constraint or add violations to a special table that has to be unioned into the result of the query. They must account during every table update for the fact that the next change may invalidate a particular soft constraint. CIs need not worry about this issue, because they do not explicitly represent logical constraints; rather, representing sets of co-occurring values makes CI maintenance simple over updates.

1.3.2 Unclustered index bitmap scans

Another indexing technique that is similar to correlation indices is the use of a bitmap scan to guide lookups over an unclustered index. One of the important reasons why CIs are able to achieve good performance is that they perform a subset of an in-order table scan on disk, avoiding unnecessary random I/O. We can make unclustered B+Trees behave in this way as well, using the following modification.

Every value that we probe in the B+Tree returns a set of tuple or page offsets on disk. We map each of these offsets to a bitmap, and we union the bitmaps across multiple B+Tree probes to represent all of the disk offsets that we need to visit during one query. Such bitmaps can be generated on demand in memory. We can then scan the bitmap once to determine the sequence of offsets to visit, which will naturally occur in order. For example, the PostgreSQL database system has implemented a bitmap scan access method in versions since 8.1. [4]

Although the unclustered index bitmap scan makes no explicit mention of correlations, the scheme achieves good performance precisely when there are correlations between the lookup key and the clustered key – otherwise, the in-order schedule of resulting lookup pages still incurs random I/O costs and performs no better than a plain unclustered index. In this sense, the bitmap scan is a variation of the CI with different tradeoffs.

On one hand, a B+Tree identifies every matching tuple explicitly, so the optimizer has more information to work with when choosing a query plan. On the other hand, CIs are often far smaller than unclustered B+Trees. One of the main motivations for developing the CI is to enable using many more indices at once than with unclustered B+Trees. Although we make the common modeling assumption in § 3.1 that indices fit in memory, this is less likely when dealing with many B+Trees simultaneously; as fewer levels of the search trees fit in

[4]See discussion at `http://archives.postgresql.org/pgsql-performance/2005-12/msg00623.php`

memory, their performance will worsen sharply. Furthermore, we can tune the size of a CI freely using bucketing as we discuss in § 2.3, whereas a B+Tree's size is determined by the number of tuples it represents.

We present experimental results in § 5.2.1 to demonstrate our claim that bitmap scans perform well with a lookup column that is correlated to the clustered index, but that they degrade to a plain unclustered B+Tree when there are no useful correlations. Additionally, Chapter 5 describes the on-disk size of unclustered B+Trees versus CIs for each of our experiments, and we confirm that the CI is substantially smaller in every case.

1.3.3 Distinct value estimation

In § 4.1.2, we consider sampling-based approaches to measuring database statistics used by our cost model, including distinct value counts over single attributes and pairs of attributes. The basic problem of predicting the number of distinct values in a column has seen extensive treatment in both the database community and the statistics community, where it is known as the problem of estimating the number of species (e.g. [2]). All of the estimators proposed in the literature struggled with poor performance on input distributions with either high skew or low skew until a paper by Charikar et al. [6] proved a strong negative result that estimating the distinct count of a column within some small constant error requires reading nearly the entire table. The result is particularly decisive because it places few restrictions on the behaviour of the estimator, allowing any variety of random sampling and adaptive procedures.

To overcome this limitation of purely sampling-based approaches, Gibbons [10] proposed the Distinct Sampling algorithm that achieves far more accurate results at the cost of one full table scan. While other work claimed that a small random sample is the only practical way to compute statistics over very large tables, Gibbons argues that one pass algorithms with fast incremental maintenance are also scalable [10]. The one-pass approach to estimating distinct counts was introduced by Flajolet and Martin [9], who proposed hashing distinct values to a bit vector with specially tailored probabilities.

Since our approach to estimating statistics in § 4.1.2 relies only on distinct counts over attributes and pairs of attributes, it can also benefit from alternative work in the area. For example, Yu et al. [24] look at estimating the number of distinct values across a set of attributes using histogram information maintained within IBM DB2, instead of making additional passes over the data. We expect that future advances in the area will make our method even more practical and effective.

1.4 Contributions

In summary, correlation indices provide the potential to exploit correlations between attributes to allow clustered indices to be used as an access method for multiple attributes. This thesis describes the design and implementation of a correlation-indexing system, with the following key contributions:

- A description of the correlation index structure as well as algorithms for creating and querying it.

- A model of correlation index performance that allows one to predict how effective a CI will be compared to traditional database access methods (sequential scans and unclustered B+Trees). We show that this model is a good match for real world performance.

- A system design that is very low in complexity, one that can be integrated with existing access methods and query optimizers with little effort.

- An evaluation of the effectiveness of CIs on several data sets, including TPC-H data and data from the US Census Bureau. We show that CIs can outperform both unclustered indices and sequential scans by an order of magnitude on a variety of useful pairs of correlated attributes, and that for real world data sets a quarter of attribute pairs can benefit from CIs by more than a factor of two.

1.5 Organization of thesis

The remainder of this thesis is organized as follows.

- Chapter 2 overviews the operation of the system in more detail, including how correlation indices are implemented and used.
- Chapter 3 describes an analytical model for estimating the cost to perform a lookup in a particular correlation index, and it describes how to compare against the expected costs of an unclustered index and a sequential scan.
- Chapter 4 describes a *CI Advisor* tool that collects statistics about expected CI benefit and actually evaluates the cost model to predict the benefit of creating a given CI. The chapter also explores a practical sampling-based approach to gathering and maintaining system statistics.
- Chapter 5 illustrates the benefits of CIs over several real-world workloads, and looks at their costs and overheads.
- Finally, Chapter 6 summarizes our contributions and concludes.

Chapter 2

System Operation

In this chapter, we describe the operation of correlation indices: how we identify candidate CI attribute pairs, as well as how we physically store and maintain CIs.

From a database client's standpoint, CIs work much like standard indices; they support customary update and query operations. For the database administrator, we provide a *CI Advisor* tool to identify pairs of attributes that are likely to be good candidates for a CI. He can use this information to decide what clustered indices to build, as well as the associated CIs to build. By issuing a simple DDL command, he can add a CI linking a given attribute – the CI key – to a clustered index. We describe the operation of the CI Advisor and how it computes the expected effectiveness of a CI in Chapter 4.

2.1 Building correlation indices

Suppose that a user wants to build a CI over an attribute $T.A_u$ of a table T, with a clustered index on attribute $T.A_c$. The CI is simply a mapping of the form $u \rightarrow S_c$, where u is a value in the domain of $T.A_u$ and S_c is a set of values in the domain of $T.A_c$ such that there exists a tuple $t \in T$ of the form $(t.A_u = u,\ t.A_c = c, \ldots)\ \forall c \in S_c$. For example, if there is a clustered index on "product.state," a CI on "product.city" might contain the entry "Boston $\rightarrow$ {NH,MA}." The algorithm for building a CI is shown in Algorithm 1. Once the administrator issues a DDL command to create a CI, the system scans the table to build the CI (line 2). As the system scans the table, it looks up the CI key value in the mapping and adds the clustered index key value to the key value set (line 3). A similar algorithm can be used for on-line insertions. Deletions, if needed, require the maintenance of an additional count of the number of co-occurrences with each value in the value set.[1]

```
input  : Table $T$ with attribute $A_u$ and clustered index over attribute $A_c$
output: Correlation index $C$, a map from $A_u$ values to co-occurring $A_c$ values
1 $C \leftarrow$ new Map(Value $\rightarrow$ Set)
2 foreach tuple $t \in T$ do
      /* Record the fact that $t.A_c$ co-occurred with $t.A_u$ in the
         mapping for $t.A_u$                                          */
3     $C$.get($t.A_u$).add($t.A_c$)
4 end
5 return $C$
```

Algorithm 1: Correlation Index Construction Algorithm.

We physically represent the CI as a B+Tree keyed by the CI key, with the set of clustered index keys as the value for each record in the B+Tree. Whenever a tuple is inserted, deleted, or modified, the CI must be updated, as with a standard index. Because the CI is relatively compact (containing one key for each value in the domain of the CI attribute, which in our experiments occupy 1–50 MB for databases of up to 5 GB), we expect that it will generally reside in memory, although since it is a B+tree the database system is easily able to spill it to disk. We report on the sizes of the CI for several real-world attributes in our experimental evaluation in Chapter 5.

The clustered index over the table may be implemented either as a clustered B+Tree or as a simple sorted file. The CI benefits from the ability to scan the range of the table associated to a sort key efficiently. In a typical B+Tree implementation, while one can scan a series of matching tuples quickly one page at a time, it is still necessary to perform random disk seeks between pages.

[1]It is not necessary for correctness to support deletions in CIs – in the worst case, they will result in scanning unnecessary values from the clustered index.

On the other hand, a sorted file can be stored contiguously on disk in order to provide optimal scan performance, but it does not support real-time updates as a B+Tree does. A static sorted file may be practical in many OLAP environments, where updates are infrequent and can be batched until the sorted file is regenerated periodically. One possible trade-off is to use a clustered B+Tree implementation backed by a packed memory array [1], which provides a bound of $O(1 + (\log^2 N)/B)$ amortized memory transfers per update to a table with $\Theta(N)$ elements and a page size of B, and reduces sequential scan performance by a factor of 2 at worst.

2.2 Using correlation indices

The API for the CI access method is straightforward; the CI implements a single procedure with the same interface as traditional access methods. It takes as input a set of values over the unclustered attribute associated to the CI, and it returns a list of tuples matching the desired set.

```
TupleList ci_lookup(Set unclustered_attr_values);
```

When a query arrives that contains a predicate over an attribute for which a CI is available, the database system will use the CI if the query optimizer determines that it will be beneficial to do so. We did not implement a complete query optimizer for our experimental evaluation, but we provide a cost model that we intend to be used by the query optimizer in Chapter 3. Using the cost model, the optimizer should pick a CI access path if it estimates that the access cost and selectivity will be less than some other access path for the same table. The optimizer should not pick a CI access path if another access path will provide an "interesting order" [19] that avoids sorts or reduces the cost of later operations in the plan outweighing the savings of using the CI.

Using a correlation index is straightforward. The basic lookup algorithm is illustrated in Algorithm 2. Given a range predicate p over a CI attribute, the query executor looks up all of the records in the range in the CI (lines 2–4). It takes the union of all of the resulting clustered attribute value sets (line 3) and generates a sorted list of the clustered attribute values that need to be scanned (line 6). The executor looks up each of the matching values in the clustered index, in order (line 8). For each tuple that is returned, the executor checks to see if the tuple satisfies p (line 9), and if so, adds the tuple to the output set (line 10). This final predicate check is necessary because some values in the clustered index may not satisfy the predicate – for example, a scan of the states "MA" and "NH" to find records with city "Boston" will encounter many record from other cities ("Cambridge," "Somerville," "Manchester," etc.) that do not satisfy the predicate.

```
   input : Correlation index $C$ over attribute $A_u$
           Clustered index $I$ on attribute $A_c$
           Range $R$ over $A_u$
   output: List of tuples in $R$
 1 $indexKeys \leftarrow$ new Set()
 2 foreach value $r \in R$ do
 3     $indexKeys \leftarrow indexKeys \cup C$.get($r$)
 4 end
 5 $tups \leftarrow$ new TupleList()
 6 $sortedKeys \leftarrow indexKeys$.sortedList()
 7 foreach $k \in sortedKeys$ do
       /* I.lookupTuples returns tuples in I that have value $k$   */
 8     foreach $t \in I$.lookupTuples($k$) do
 9         if $t.a \in R$ then
10             $tups$.append($t$)
11         end
12     end
13 end
14 return $tups$
```

Algorithm 2: Correlation Index Lookup Algorithm.

Figure 2-1 illustrates an example CI. Here, the user has a table with three attributes: state, city, and salary, with a clustered B+Tree on state. The administrator has created a CI on city. The CI maintains a correspondence between each city name and the set of states it appears in. When a query with a restriction to the cities "Boston" and "Springfield" arrives, the system gener-

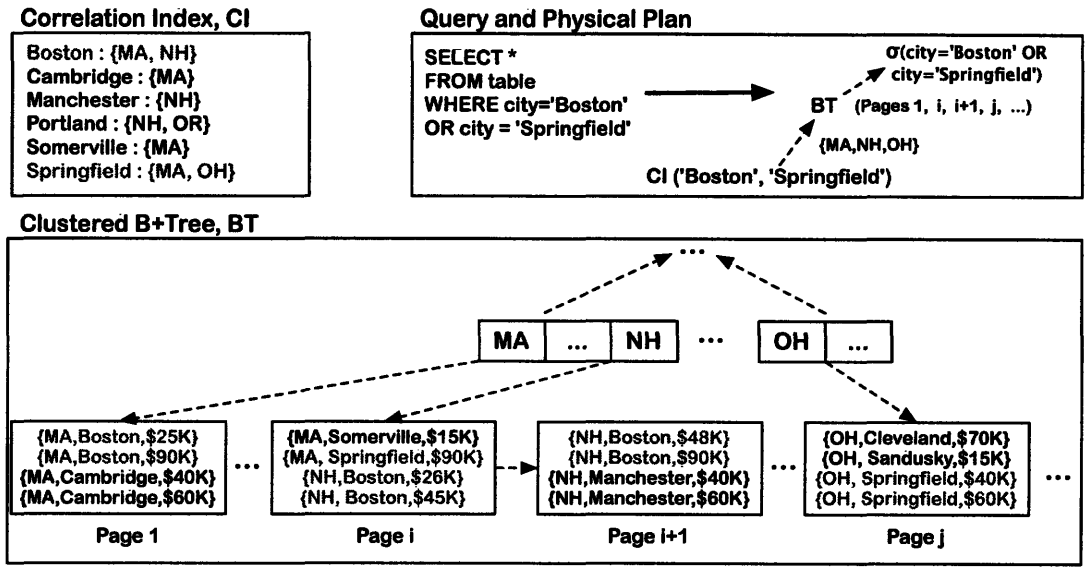


Figure 2-1: Diagram illustrating an example CI and its use in a query plan.

ates a physical query plan that uses the CI to identify the states that contain "Boston" and "Springfield" ("MA," "NH," and "OH"), and performs an in-order lookup on the clustered B+Tree to find the pages containing records from these states (1, 2, and 3 in our example). The tuples on these pages are fed to a selection operator, which returns only those tuples whose city matches "Boston" or "Springfield."

2.3 Bucketing correlation indices

The basic CI approach described in the previous section works well for attributes where the number of distinct values in the CI attribute or the clustered attribute is relatively small. However, for large attribute domains, the size of the CI can grow quite unwieldy (in the worst case having one entry for each tuple in the table). Keeping a CI small is important to keep the overhead of performing lookups low.

We can reduce the size of a CI by "bucketing" ranges of the CI attribute together into a single value. We can compress ranges of the clustered attribute stored in the CI similarly. A basic approach to bucketing is straightforward.

For example, suppose we build a CI on the attribute *temperature* and we have a clustered index on the attribute *humidity* (these attributes are often correlated – with lower temperatures bringing lower humidities).

Suppose the unbucketed CI looks as follows:

$$\begin{aligned}
\{12.3^oC\} &\rightarrow \{17.5\%, 18.3\%\} \\
\{12.7^o\} &\rightarrow \{18.9\%, 20.1\%\} \\
\{14.4^oC\} &\rightarrow \{20.7\%.22.0\%\} \\
\{14.9^oC\} &\rightarrow \{21.3\%, 22.2\%\} \\
\{17.8^oC\} &\rightarrow \{25.6\%, 25.9\%\}
\end{aligned}$$

We can bucket into 1^oC or 1% intervals via truncation as follows:

$$\begin{aligned}
&\{12 - 13^oC\} \rightarrow \{17 - 18\%, 18 - 19\%, 20 - 21\%\} \\
&\{14 - 15^oC\} \rightarrow \{20 - 21\%, 21 - 22\%, 22 - 23\%\} \\
&\{17 - 18^oC\} \rightarrow \{25 - 26\%\}
\end{aligned}$$

Note that we only need to store the lower bounds of the intervals in the bucketed example above. We omit a detailed algorithm for performing this truncation, but we observe that the bucketing scheme could be as simple as taking an integer floor or as complex as arbitrary user-defined stored procedures.

The effect of this truncation is to decrease the size of the CI while decreasing its effectiveness, since now each CI attribute value maps to a larger range of clustered index values (requiring us to scan a larger range of the clustered index for each CI lookup). In this thesis, we consider simple fixed-width binning schemes such as the one shown in the example above. [2] In § 5.2.3, we present

[2]Note that a poor bucketing can destroy correlations, since two CI values with very different clustered attribute sets may be placed into the same bucket. In this scenario, the CI must scan a large range of the clustered attribute space whenever we look up either CI attribute value. Hence, it is attractive for future work to look at adaptive binning techniques that merge together CI attribute buckets only when there is significant overlap in their clustered attribute sets.

experimental results to illustrate the effectiveness of the simple approach in one of our data sets.

2.4 Discussion

CIs capture the correlation between the CI attribute and the clustered attribute. If two attributes are highly correlated, each value of the CI attribute will co-occur in a tuple with only a few values in the clustered attribute, whereas if they are poorly correlated, the CI attribute will co-occur with many clustered attribute values.

Intuitively, correlated indices will be most effective when:

- There is a strict functional dependency between the unclustered attribute and the clustered attribute (that is, there is a many to one relationship, such that the unclustered attribute perfectly predicts the clustered attribute value). For example, in the United States, a zip code is a perfect predictor of a state. “Soft” functional dependencies should also perform well; for example, though a city name does not perfectly predict a state (since there is a Boston, MA and a Boston, NH), city is often sufficient to predict state.

- The number of tuples across all clustered values to which an unclustered value maps is small. For example, in a nationwide database, city name is a good predictor of county, and there are many cities and counties. If a clustered index on county exists, it is likely that the index will also be useful for answering queries over city name, since the number of tuples for each county will not be large compared to the size of the database.

  As a counter-example, consider the TPC-H benchmark attribute “returndate,” which indicates when a product was returned. It is perfectly correlated with the attribute “returncode” that indicates whether or not a product has been returned (returndate is null when no return has happened, and non-null when it has). However, this correlation may not be useful, since the domain of returncode is so small that a very large fraction of the database has to be scanned whenever a clustered index on returncode is used.

It is also instructive to to compare the performance of CIs to unclustered B+Trees on the CI attribute. Clearly, using the correlation index will cause the system to read more tuples than it would read using an unclustered index. However, the CI can be much less expensive than an unclustered B+tree because every tuple that is retrieved in an unclustered index requires a random disk I/O, whereas the CI scans all tuples with a given value from the clustered index, which keeps tuples with the same value together on the same disk page(s). We formalize this intuition in Chapter 3 where we model and compare the performance of unclustered B+Trees and CIs. [3]

[3]It is interesting to note that it is possible to use an unclustered index in conjunction with a clustered index in much the same way we use a CI. This is true since if there is a correlation between the unclustered attribute and the clustered attribute, lookups of a particular value in the unclustered index will tend to be grouped onto a few pages of the clustered index. One way to view a CI is as a more compact representation of of an unclustered index used in this way.

Chapter 3

Cost Model

So far in the thesis, we have given some intuition for when a CI might be less expensive than an unclustered index or a sequential scan; however, we have not provided a formal analysis for when this will be true. In this chapter, we describe an analytical cost model that we use to compare the absolute costs of the different access methods over an unclustered attribute. In particular, we examine an unclustered B+Tree index, a full table scan, and a CI over a given pair of attributes. Our goal is to compute the expected runtime of a lookup given parameters characterizing the underlying hardware and basic statistics extracted from the data.

3.1 Preliminaries

In the following discussion, we assume a table with attributes A_c and A_u. The field A_c serves as the clustered attribute for the table, and the field A_u is the unclustered attribute upon which we query. Our model assumes that the table is stored as a heap file sorted on A_c for fast sequential scans. Thus, to read all of the tuples corresponding to a clustered attribute value, we look up the file offset corresponding to the value and perform a disk seek. We then read the relevant tuples sequentially.

We assume that the database system is in a steady state where the desired

Table 3.1: Table statistics used by the cost model.

tups_per_page	Number of tuples that fit on one page.
c_tups	Average number of tuples appearing with each A_c value.
c_per_u	Average number of distinct A_c values for each A_u value.
total_tups	Total number of tuples in the table.
n_lookups	Number of A_u values to look up in one query.

Table 3.2: Hardware parameters used by the cost model.

sequential_page_cost	Time to read one disk page sequentially.
Typical value:	.065 ms
random_seek_cost	Time to seek to a random disk page and read it.
Typical value:	4.55 ms

queries have not recently executed. We do not charge the B+Tree or CI access methods for reading index pages, since the upper levels of an active index are likely to be cached in a warm system. We assume that the cache initially holds none of the table pages. After a page has been accessed once, we do not charge an operator to read it a second time.

In Table 3.1, we summarize the statistics that we calculate over each table. Additionally, in Table 3.2, we describe the hardware parameters we use, along with typical values measured on our experimental platform. Most model parameters are straightforward, and we describe in Chapter 4 how to measure them automatically.

We assume that all of the access methods are disk-bound. We do not model the CPU costs associated to traversing a B+Tree nor filtering tuples in the CI, and the cost of the sequential scan is independent of the number of values we look up. Through CPU profiling of the implementation that we present in Chapter 5, we have validated that our assumption is reasonable.

3.2 Cost of sequential scan

The sequential scan operator is the simplest access method to model. Given our model parameters, the number of pages in a table is $total\_tups/tups\_per\_page$.

The cost of scanning a table is then

$$cost_{scan} = (sequential\_page\_cost)\left(\frac{total\_tups}{tups\_per\_page}\right)$$

We note here that our model is oblivious to external factors such as disk fragmentation. We found that this may not be true, for example, with sequential scans over tuples stored in Berkeley DB B+Tree files, which tend to be highly fragmented. This factor is entirely implementation-specific, so we do not attempt to model it and we control for it in our experiments by defragmenting each file before a query begins.

3.3 Cost of unclustered B+Tree

In our model, every tuple read via an unclustered B+Tree index causes a random seek. Given that we expect *u_tups* tuples to be associated to each A_u value, the basic cost of looking up *n_lookups* values using a B+Tree is simply

$$cost_{btree} = (random\_seek\_cost)(u\_tups)(n\_lookups)$$

However, we must be careful about charging too much when the same pages are likely to be accessed more than once.

In addition to modeling the behavior of random disk accesses, it is important to model the effect of the database system's buffer pool on access costs. We have chosen not to adopt a complicated caching model, which would obscure the primary effects that we wish to compare between unclustered B+Trees and CIs. Instead, we propose the following simple model that captures the first-order effects of the buffer pool.

Suppose at the beginning of a query that we have a buffer pool of unlimited size that has cached none of the heap pages storing table data. The first time we request any heap page, we incur the cost of one disk read; subsequently, requests for that page are free. Furthermore, assume that the B+Tree samples pages uniformly at random from the heap with replacement.

Theorem 3.3.1. *Under our caching model, the expected number of cache hits after p requests in a file with n pages is*

$$1 + p - n + \frac{(n-1)^{p+1}}{n^p} \tag{3.1}$$

Proof. Let H_i be an indicator variable that is 1 if request i hits the cache, and let C_i be a random variable representing the number of pages in the cache immediately before request i. Then, the expected number of cache hits after p requests is $\sum_{i=1}^{p} H_i$. Manipulating this expression, we have:

$$\begin{aligned}
E\left[\sum_{i=1}^{p} H_i\right] &= \sum_{i=1}^{p} E[H_i] \\
&= \sum_{i=1}^{p} Pr(H_i = 1) \\
&= \sum_{i=1}^{p}\sum_{x=1}^{n} Pr(H_i = 1 | C_i = x) Pr(C_i = x) \\
&= \sum_{i=1}^{p}\sum_{x=1}^{n} \frac{x}{n} Pr(C_i = x) \\
&= \frac{1}{n}\sum_{i=1}^{p}\sum_{x=1}^{n} x Pr(C_i = x) \\
&= \frac{1}{n}\sum_{i=1}^{p} E[C_i]
\end{aligned}$$

Now, let us consider $E[C_i]$, the expected number of pages in the cache at request i. We wish to define a recurrence by relating the number of cached pages at step i to the number of cached pages at step $i-1$. Let us denote the page requested at step i by R_i. The number of pages in the cache will increase by one after a request iff that request misses the cache. Using this fact, by our model we have that

$$\begin{aligned}
E[C_i] &= E[C_{i-1}] + P(R_i \text{ misses the cache}) \\
&= E[C_{i-1}] + 1 - P(R_i \text{ hits the cache}) \\
&= E[C_{i-1}] + 1 - \frac{E[C_{i-1}]}{n} \\
&= 1 + \left(\frac{n-1}{n}\right) E[C_{i-1}]
\end{aligned}$$

This recurrence has the straightforward solution

$$E[C_i] = -n\left(\left(\frac{n-1}{n}\right)^i - 1\right)$$

Substituting this expression back into our original calculation, we conclude that

$$\begin{aligned}
E\left[\sum_{i=1}^{p} H_i\right] &= \frac{1}{n}\sum_{i=1}^{p} E[C_i] \\
&= \frac{1}{n}\sum_{i=1}^{p} -n\left(\left(\frac{n-1}{n}\right)^i - 1\right) \\
&= \sum_{i=1}^{p} 1 - \left(\frac{n-1}{n}\right)^i \\
&= p - \sum_{i=1}^{p}\left(\frac{n-1}{n}\right)^i \\
&= 1 + p - n\left(1 - \left(\frac{n-1}{n}\right)^{p+1}\right) \\
&= 1 + p - n + \frac{(n-1)^{p+1}}{n^p}
\end{aligned}$$

□

Finally, we incorporate the caching model into our expected B+Tree cost.

Instead of performing a random seek for each result tuple, we now perform one random seek only for each predicted disk read:

$$disk\_reads = (u\_tups)(n\_lookups) - cache\_hits$$

$$cost_{btree} = (random\_seek\_cost)(disk\_reads)$$

We have assumed that each tuple read via an unclustered B+Tree index causes a random seek – for example, if values were inserted in the B+tree in some secondary sort order. If the database system has more information about the distribution of values across pages (e.g. via histogram statistics in the system catalog), it can take the statistics into account by varying the *random_seek_cost* for B+Tree operations.

3.4 Cost of correlation index

Having developed a general caching model in the previous section, we now describe the expected cost of a correlation index lookup. We will see that we can apply our previous caching result almost directly, with one simple modification for the CI access pattern.

Suppose as usual that the CI has a set of A_u values to look up. For each A_u value, we must visit *c_per_u* different clustered attribute values. We will need to perform one random seek to reach each of these clustered attribute values, followed by a scan of all of the pages for that A_c value. In terms of the parameters we are given, the number of pages for a given A_c value is $c\_tups / tups\_per\_page$.

Now, we must take into account cache hits as before. The key difference while modeling the CI is that our former assumption of an access pattern sampling pages uniformly at random is false – each clustered attribute that we visit results in the CI reading a contiguous segment of pages (a "superpage"), and thus these reads are highly correlated.

To modify our assumption, suppose we know that there are $c\_pages = c\_tups/tups\_per\_page$ pages associated to each clustered attribute value. Then, there are now only $total\_tups/tups\_per\_page/c\_pages$ superpages in the file, and we can apply our caching model to the superpages.

Summarizing our ideas, the expected number superpages that we visit is determined by the following expressions.

$$\begin{aligned} c\_pages &= \frac{c\_tups}{tups\_per\_page} \\ n\_superpages &= (n\_lookups)(c\_per\_u) - cache\_hits \end{aligned}$$

Combining these expressions, the overall cost of a CI lookup is

$$\begin{aligned} cost_{ci} = (n\_superpages)(&random\_seek\_cost \\ &+ (sequential\_page\_cost)(c\_pages)) \end{aligned}$$

Similar to our assumption that each secondary B+Tree tuple causes a random seek, we choose not to model the overlap between the sets of A_c keys associated to two particular A_u values. In other words, if one A_u value maps to n different A_c values on average, then it is not true in general that two A_u values map to $2n$ different A_c values. Our model may overestimate the number of A_c values involved, and thus the cost of CI.

We have chosen to omit this statistic in the interest of a simpler model, but we observe here that the desired overlap can easily be measured from the table by sampling pairs of uncorrelated attribute values and computing the average overlap between the resulting sets of A_c values.

3.5 Discussion

We have presented a series of expressions that estimate the cost of lookups using a sequential scan, an unclustered B+Tree, and a CI. While the expressions are fairly simple, they involve disparate sets of parameters that make comparison inconvenient. For this reason, we now provide some intuition for situations where a CI might be more or less expensive than a sequential scan or B+Tree.

Sequential scan: The CI access pattern can be thought of as a subset of a sequential scan – that is, the CI will always read segments of a file in the same order as a sequential scan would, but it will jump over some stretches of the file. The CI reads fewer tuples than the sequential scan, but it still incurs some cost from disk seeks. Thus, there exists a trade-off in our model between reducing the number of tuples that the CI reads and the number of extra seeks that it performs.

In general, the CI will beat a sequential scan when the selectivity is high, and it is reasonable to expect the performance of CI to degrade to that of the sequential scan when the selectivity becomes low (indeed, the CI access pattern becomes more and more like a sequential scan). However, it is worth noting that it is possible for the CI to be more expensive than a sequential scan by a constant factor for low selectivities, because there is a noticeable overhead associated with seeking (even if the seeks are in-order and do not skip large distances in the file).

Unclustered B+Tree: The difference between the performance of CIs and B+Trees is less straightforward to grasp. There are essentially two trade-offs that may apply to a particular query situation.

First, suppose that *c_tups* is low; that is, the average number of tuples for each clustered attribute value is small. This suggests that the number of irrelevant tuples that a CI reads will not be too large, because the CI will not fall into reading long segments of the file with only a few matching tuples.

In this case, the primary trade-off occurs between the *u_tups* and *c_per_u* parameters. If *u_tups* is high, then the B+Tree will incur a higher cost by performing a random seek for each A_u value. On the other hand, if *c_per_u* is high, then the CI will need to visit a large number of clustered attribute locations in the file (each of which also costs a random seek).

Now, suppose that *c_tups* is high, so that each A_c clustered attribute value that the CI must visit is very expensive. In this case, it is important for the CI not to have to visit very many A_c values, or rather that the correlation between A_u and A_c values be high.

Given that *c_tups* is high, the scenario most favorable to the CI occurs when a given A_u value maps to a single A_c value, but where there are many tuples scattered throughout that A_c superpage matching A_u. The CI needs merely to scan the single superpage, while the B+Tree must perform numerous random probes within that superpage.

The scenario least favorable to CI occurs when there are only a handful of tuples in the file matching the A_u value, but where each of the tuples co-occurs with a different clustered attribute value. Now, the CI must scan and discard a large number of irrelevant tuples while the B+Tree can seek to the desired pages directly.

Chapter 4

Prediction

The introduction of a new access method to the database system complicates the job of the database administrator. Judging when a correlation index will be beneficial for a given pair of attributes is difficult to do by hand. Indeed, the administrator would need to weigh his estimation of the domain sizes of each attribute against the overall size of the table and the cost of random disk I/O.

Fortunately, given the analytical model that we have developed, we can fully automate the process of predicting when CIs will be useful. In our implementation, we built a *CI Advisor* tool that scans existing database tables and calculates the statistics needed by the cost model. Given these statistics and measurements of underlying hardware properties, the CI Advisor can predict accurately each pair of attributes that would benefit from a CI. Presented with this information, the database administrator need only choose pairs from the list that the application is likely to query.

Since the CI Advisor computes sufficient statistics to evaluate the cost model for any set of parameters, our implementation is capable of generating plots of the expected query performance over each of the three access methods. In Chapter 5, we present the plots predicted by the CI Advisor alongside our empirical results. Our results suggest that the CI Advisor produces accurate estimates.

4.1 Parameter collection

In order to form predictions based on the cost model, the CI Advisor must refresh the statistics listed in Figure 3.1 based on the current state of the database. These include *c_tups* and *c_per_u*, which are based on counts of the number of distinct values in the database. We first present a naïve procedure to gather the statistics. The approach is straightforward to understand, and it is possible to compute the relevant aggregates using standard SQL queries within the database system itself. Such aggregate queries may however be impractically expensive, however, and we present a sampling-based approach in § 4.1.2 to reduce the overhead of parameter collection by an order of magnitude or more. We present experimental results in § 5.3 to contrast the effectiveness of the different approaches.

The parameter *total_tups* is simply a count of the number of tuples in the table, which we expect the DBMS to maintain already as a routine statistic. The parameter *c_tups* can also be computed as *total_tups* divided by the number of distinct values in A_c, which is also routinely maintained in the system catalog. Furthermore, the average number of *tups_per_page* can be determined easily by dividing the size of a database page on disk by the average width of a tuple.

Our model further relies on the *sequential_page_cost* and *random_seek_cost* – parameters that are characteristics of the underlying disk. Instead of depending on the user to supply these values, our implementation measures them directly by creating large files on the target filesystem and reading them via sequential and random access patterns. Since the hard disk parameters associated with a given table file can change over time (due to external factors such as disk fragmentation), we recommend that these parameters be refreshed periodically to reflect the current state of the file.

4.1.1 Computing the *c_per_u* parameter exactly

The *c_per_u* parameter, the average number of distinct clustered key values for each lookup key value, is the chief statistic that captures degrees of correlation within our model. Unfortunately, it is also by far the most expensive statistic to compute exactly.

For the purposes of the CI Advisor, we must compute one *c_per_u* value for each ordered pair of fields in the table. To compute the set of *c_per_u* parameters for each pair, we perform the queries listed in Figure 4-1. For each attribute G in the table T, we issue one SQL statement that groups by G. In the inner query of the statement, the C_1 through C_n terms produce the distinct counts across every other attribute of T within each group of G. Then, the outer query takes the average number of distinct counts across all groups of G for each attribute. The final output values are the average counts of distinct A_i values for each G value. [1]

Note that each count(DISTINCT ...) term must be processed using a separate sort or hash operation. Thus, the number of disk output buffers in general across all *c_per_u* pairs will be quadratic in the number of columns, and the performance is likely to be impractical in real applications.

```
foreach attribute $A \in T$:
          select  avg(counts.$C_1$), ..., avg(counts.$C_n$)
            from  (select count(DISTINCT $A_1$) AS $C_1$,
                  ...,
                  count(DISTINCT $A_n$) AS $C_n$
                  from $T$
                  group by $A$) as counts
```

Figure 4-1: Algorithm to compute the model parameter *c_per_u* for all pairs of attributes in table T.

[1] In terms of *c_per_u* parameters, the group-by attribute G corresponds to the unclustered attribute A_u and the distinct count attributes A_i correspond to the clustered attribute A_c.

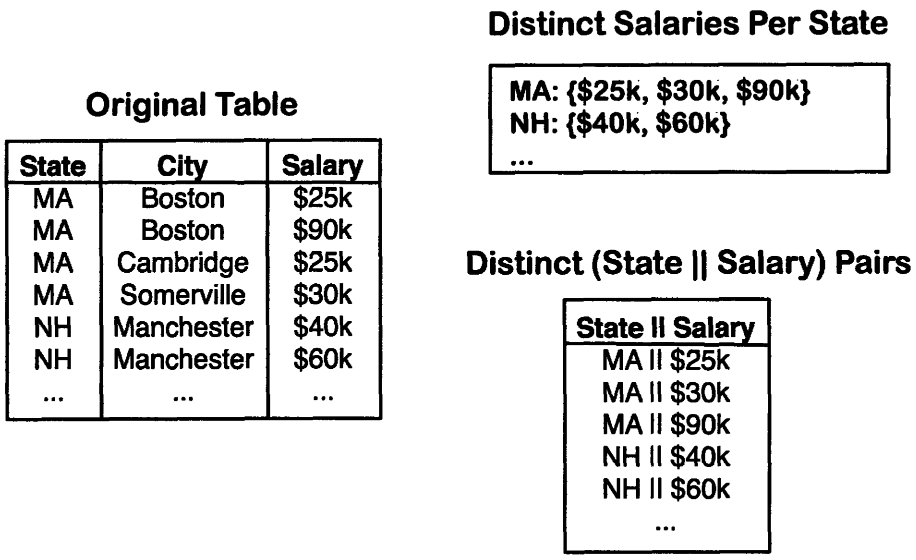
4.1.2 Approximating the *c_per_u* parameter

Our initial approach to computing exact *c_per_u* statistics is expensive. Previous works in semantic query optimization tools, which need to gather similar statistics, have proposed analogous schemes. For example, Gryz et al. [12] suggest applying a linear regression statistic model over all comparable pairs of fields in each table, which they can achieve only by a similarly brute-force approach.

Since intensive aggregates are expensive for very large databases, it is natural to ask if we can achieve reasonable estimates for our desired statistics via a sampling-based approach over our tables. If we can reduce the cost of determining the distinct counts of each field as well as the *c_per_u* counts, then we can reduce the execution time of the CI Advisor and improve the staleness of the cost model used for query optimization.

Before we turn to approximation, we observe that the naïve approach computed the distinct count over A_c values for *each* A_u group, and subsequently took the average of the per-group distinct counts. Since we desire only the average value as the output, computing the distinct count for each group individually is in fact unnecessarily expensive. We now present a simplification for the exact algorithm.

Consider the example of calculating the average number of distinct salaries per state in Figure 4-2. As opposed to counting the number of distinct salaries directly (shown above), we could alternatively enumerate the distinct pairwise values (shown below) and divide by the number of groups. In this case, there are five distinct (*state* || *salary*) pairs across two state groups, so the *c_per_u* count is 2.5. Stating our observation more generally, let us write the number of distinct values over a pair of attributes A_i and A_j within a set of tuples T as $D_T(A_i, A_j)$ and the number of distinct values over a single attribute as $D_T(A_i)$.



Original Table

State	City	Salary
MA	Boston	\$25k
MA	Boston	\$90k
MA	Cambridge	\$25k
MA	Somerville	\$30k
NH	Manchester	\$40k
NH	Manchester	\$60k
...	...	...

Distinct Salaries Per State

MA: {\$25k, \$30k, \$90k}
NH: {\$40k, \$60k}
...

Distinct (State || Salary) Pairs

State ll Salary
MA ll \$25k
MA ll \$30k
MA ll \$90k
NH ll \$40k
NH ll \$60k
...

Figure 4-2: An illustration of alternative ways to calculate *c_per_u* statistics: either by averaging the distinct *salary* count per *state* group, or by counting the number of distinct (*state* || *salary*) pairs.

Then, it is clear that

$$\frac{\sum_{a \in \text{groups of} A_u} D_a(A_c)}{\text{number of } A_u \text{ groups}} = \frac{D_T(A_u, A_c)}{D_T(A_u)}$$

In other words, instead of computing each *c_per_u* value explicitly using an average over an expensive grouping aggregate operation, we can alternatively calculate distinct counts for each single attribute and each attribute pairs. Then, we simply divide the distinct counts as necessary to derive each *c_per_u* value.

Now, we describe how to apply a sampling-based approach to improve the performance of our scheme for approximate answers. In § 1.3.3, we discussed the Distinct Sampling algorithm presented by Gibbons. Of particular interest to our problem, Distinct Sampling can provide highly accurate estimates of distinct counts (experimentally, often within 10%) over numerous table attributes using relatively small, constant space bounds and a single table scan. We refer the reader to [10] for a full presentation of the algorithm.

Not only does our revised *c_per_u* calculation based on distinct counts reduce complexity, the Distinct Sampling algorithm is well-suited to computing distinct value estimates over the entire set of target attributes in one pass. Thus, in order to approximate distinct count statistics efficiently, we create one Distinct Sampling instance for each target attribute and attribute pair. To derive the actual *c_per_u* values used by the model, we calculate $D_T(A_u, A_c)/D_T(A_u)$ as necessary.

Our revised procedure for computing *c_per_u* estimates achieves a high degree of accuracy while reducing the calculation runtime by over an order of magnitude, which we demonstrate experimentally in § 5.3. While the number of Distinct Sampling instances is still quadratic in the number of columns, each instance requires only a small, constant amount of state and the entire operation is likely to fit in memory. Furthermore, it has the key property that the $D(A_i)$ and $D(A_i, A_j)$ values supplied by each Distinct Sampling instance can be maintained efficiently online in the presence of insertions. It is now possible, therefore, for the DBMS to maintain up-to-date *c_per_u* estimates for use by the planner during query optimization, instead of relying on one-time statistics that are computed periodically and become stale.

Chapter 5

Experimental evaluation

In this section, we present an experimental validation of our results. We have developed an analytical model and an accompanying tool to identify cases where CIs are useful; however, we have not yet shown that such cases routinely exist. Additionally, we would like to argue that CIs are likely to be useful in a large fraction of pairs of attributes across a given table. The goals of our experiments are thus threefold:

1. to validate the accuracy of our analytical model

2. to establish that useful correlations are reasonably common in large data sets

3. to demonstrate that there exist cases where CIs win over both B+Trees and sequential scan and to measure the extent of that benefit.

5.1 Implementation

To conduct our experiments, we have completed an implementation of correlation indices that is suitable for comparison against Berkeley DB B+Trees and full table scans. We have chosen Berkeley DB as a low-overhead B+Tree library to avoid conflating unrelated system effects in a full DBMS.

For our experimental platform, we ran our tests on Emulab[1] machines with 64-bit Intel Xeon 3 GHz processors and 2 GB RAM running Linux 2.6.16. The machines each had single Seagate Ultra320 SCSI hard disks that spin at 10K RPM. We developed code in C++, linking against Berkeley DB version 4.4.20. In order to optimize the performance of the B+Tree, we employ the bulk-access API where appropriate and we supply a custom key comparator. In our experiments, we use a Berkeley DB cache size of 512 MB.[2]

To implement the clustered index over a table required by CIs, we have chosen to represent tables as sorted files on the clustered attribute. In this way, the pages of a file corresponding to each clustered attribute value are contiguous on disk. Our experiments are thus able to measure the performance of sequential access across a given clustered attribute value directly. We have confirmed that sequential access performance is as fast as reading the file from the disk directly, suggesting that the CPU overhead associated to processing tuples is insignificant. We decided that sorted files were a reasonable implementation of a clustered index because they are common in OLAP environments. Although they are expensive to maintain, updates are not common in such environments.

5.1.1 Experimental setup

We have chosen to experiment with three data sets exhibiting different characteristics: a purely synthetic table for model validation, the TPC-H lineitem table, and a table adapted from the US Census Gazetteer. We also study the degree of correlations in a fourth data set from a real business that includes geographic and demographic information about customers, but we do not present query tests on it because the scale of the data set is too small.

We begin each experiment by pre-loading the complete table into both the Berkeley DB B+Tree format and our custom sorted file format. While loading

[1]`http://www.emulab.net`

[2]Although our machines have more than 512 MB of memory, Berkeley DB double-buffers against the Linux filesystem cache. A Berkeley DB cache of 512 MB appeared to achieve optimal performance across the cache sizes that we tested.

the sorted file, we also build the physical correlation index that is stored on disk. We found that the loading times were costly for both file formats – they typically dominated the individual query times – which we expect for OLAP environments. For each of the test cases, we present the size of the on-disk CI data structure as well as the size of the unclustered B+Tree over the same attribute.

After the files have been loaded, we have available the set of distinct values in the uncorrelated attribute. We choose a subset of these values according to the SQL query that we present, and we begin timing once we have invoked the access method on the query values. In each experiment, we vary the number of unclustered attribute values that we look up per query. The values are chosen uniformly at random from the domain of the attribute, except where we have indicated a range predicate. For each trial, we measure the (wall clock) time elapsed from the time the input is specified to the time the final output tuple is returned.

We now give brief background describing each data set.

Synthetic data

For our first data source, we present a purely synthetic workload. We chose the values of the *pickles* and *factories* columns so that we could vary the degree of correlation and measure its effects on performance. In our experiments, we present results over one choice of parameters with the following characteristics:

- We create tuples that involve 50 distinct factories, altogether producing 5000 different types of pickles.
- Each factory produces 500 different types of pickles.
- A given pickle can be purchased from any of 5 different factories.

The table consists of approximately 36 million tuples of 136 bytes each, for a total table size of approximately 5.0 GB. To generate the table, we begin by instantiating 50 factories and 5000 pickles and assigning each pickle to 5 different factories at random. We then iterate over the factories; for each factory, we output 720000 tuples by randomly selecting one of the pickles assigned to that factory for each tuple. Since there are relatively few factories and pickles for the total number of tuples in the table, the degree of correlation between *pickles* and *factories* is high.

TPC-H data

For our second data source, we chose the *lineitem* table from the TPC-H benchmark, which represents a business-oriented log of orders, parts, and suppliers. There are 16 attributes in total in which we looked for correlation. While TPC-H is a widely recognized benchmark for OLAP applications, it is worth noting that the data are generated by a tool such that all correlations are manufactured intentionally.

In Figure 5-1, we present a graphical representation of the fraction of attribute pairs for which our model predicts a CI to win over both an unclustered B+Tree and sequential scan in this dataset. We have calculated the ratio of CI query time to the minimum of B+Tree and sequential scan query time, assuming one random query value on the unclustered attribute. To make the results more easy to visualize, we have truncated the ratios for which we predict a CI to lose (that is, ratios that we depict as 1.0 are in fact cases where CI loses by a potentially large ratio). The plot suggests that we will see a win in roughly half of the 240 attribute pairs, and a win by a factor of 2 about 20% of the time. Furthermore, we will see a win of about an order of magnitude for 8 attribute pairs.

The table consists of approximately 18 million tuples of 136 bytes each, for a total table size of approximately 2.5 GB.

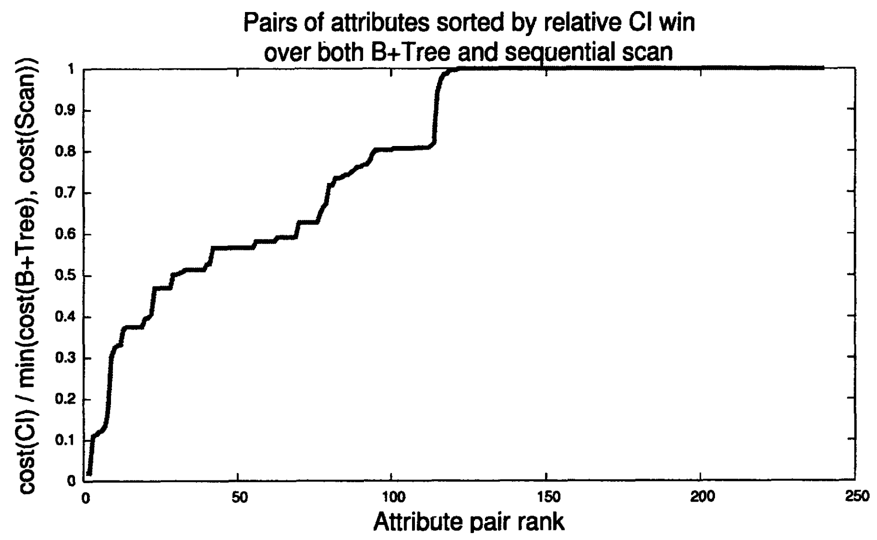


Figure 5-1: List of *lineitem* attribute pairs sorted by expected win for a CI. Smaller values represent more substantial wins. The plot suggests that we will see a win in roughly half of the 240 attribute pairs.

Census data

We intend for the census data to represent a real-world data set with geographical correlations and realistic skew effects. We began with the US Census Gazetteer Zips file[3] that contains fields for each state and zip code in the United States. Additionally, the original data include the population, longitude, and latitude for each zip code.

In order to scale the data set to a reasonable size, we expanded the zips table to include effectively one tuple for every person in the United States. We replicated the tuple representing each zip code as many times as the population specified, replacing the population field with a unique identifier field. In order to distribute zip codes evenly per state, we randomize the order of tuples within each state.

The resulting table consists of approximately 250 million tuples of 16 bytes each, for a total table size of approximately 3.8 GB.

[3]Available via `http://www.census.gov/tiger/tms/gazetteer/`

Company data

Additionally, we present results from running our model over a small (proprietary) data set obtained from a local company that describes geographic and demographic information about customers. We chose this table because it includes over 40 attributes that are representative of a variety of different aspects found in typical business data sets. Because the table fits easily into memory, it is not meaningful to run actual experiments over these data and we do not present results on these data in the next section. However, the subset of the data that we have does show a substantial degree of useful correlations, further substantiating our belief that CIs are beneficial in many similar real data sets. The results from applying our model are shown in Figure 5-2. This plot is organized in the same style as Figure 5-1. The results suggest that we will again see a win in over half of the 1980 attribute pairs, and a win by a factor of 2 about 25% of the time. In other words, a CI will win over alternative access methods more often than not. Furthermore, we may see a win of about an order of magnitude for approximately 35 attribute pairs.

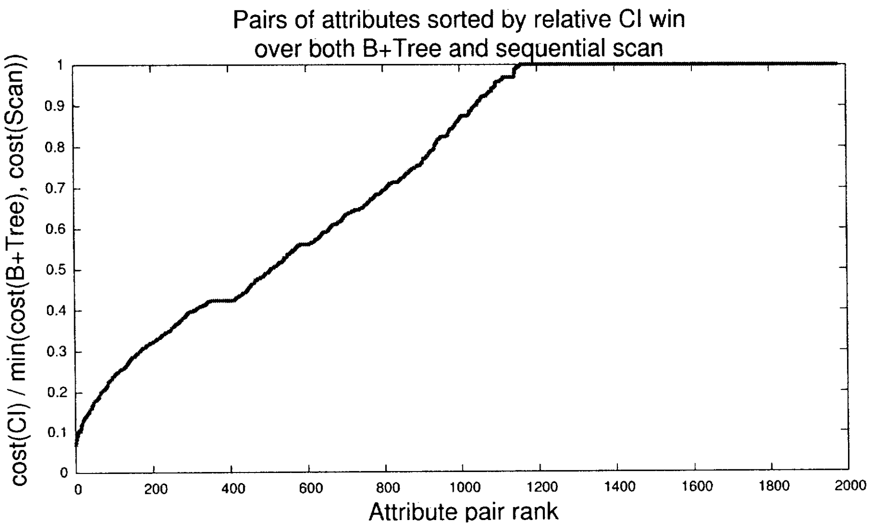


Figure 5-2: List of *company* attribute pairs sorted by expected win for a CI. Smaller values represent more substantial wins. The plot again suggests that we will see a win in roughly half of the 1980 attribute pairs.

5.2 Indexing results

We now present the results of a variety of experiments. The goal of each experiment is to understand the relative trends in running times for each of the unclustered B+Tree, sequential scan, and CI access methods as we vary the number of values that we look up.

In the subsequent sections, we describe the equivalent SQL query that each experiment implements. We additionally present two plots: on the upper plot, we show our actual measurements, and on the lower plot, we present the times predicted by our analytical model. One can confirm visually that the analytical model matches the measurements quite well in each case.

5.2.1 Synthetic results

In Figure 5-3, we show the results of looking up increasingly large sets of *pickles* in a table clustered on the *factory* attribute. The size of the pickle set that we request ranges between 1 and 10 on the horizontal axis, and we show the elapsed time on the vertical axis. For this experiment, we present five lines: one for sequential scan, one for the unclustered B+Tree, one for the CI, and two measurements over PostgreSQL bitmap scans (described in § 1.3.2). For the first bitmap scan result, we have clustered the table on the *factory* attribute and we expect the performance to be competitive with that of the CI. For the second bitmap scan result, we have clustered the table on a random ID field so that there is no correlation between *pickles* and the sort key; we expect the performance to be similar to that of an unclustered B+Tree. Since the runtime of the sequential scan is roughly constant in each test, it will appear as a horizontal line.

The purpose of our synthetic experiment is to validate the accuracy of our model. The lower plot in Figure 5-3 includes three lines predicting performance for the unclustered B+Tree, sequential scan, and CI. Indeed, we can see by comparing against the experimental measurements that the model predicts the actual performance quite well. Since the data are generated using a pseudo-random number generator and the model depicts the average case, we observe slight inaccuracies in the model predictions compared to our measurements. However, it is clear that our predictions are sufficient for a query optimizer to be able to make correct decisions.

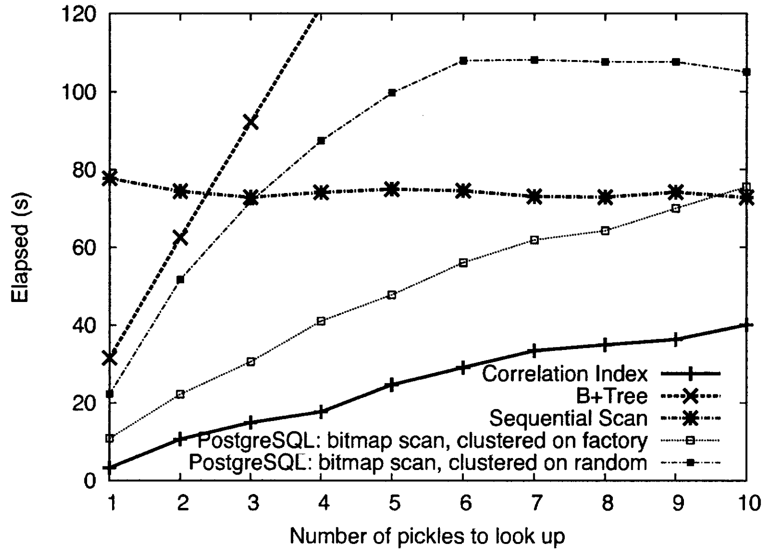
Although each *pickle* maps to 5 different *factories* at random, there are overlaps in the sets of factories between two pickles. This explains why looking up 10 different pickles results in scanning roughly half of the table. It is interesting to note that the CI outperforms the B+Tree with even a single *pickle* lookup (3.24s versus 31.6s). The model explains that this happens because each *pickle* appears in 7200 tuples, generating many random B+Tree disk seeks, while the CI scans of 5 *factories* more efficiently.

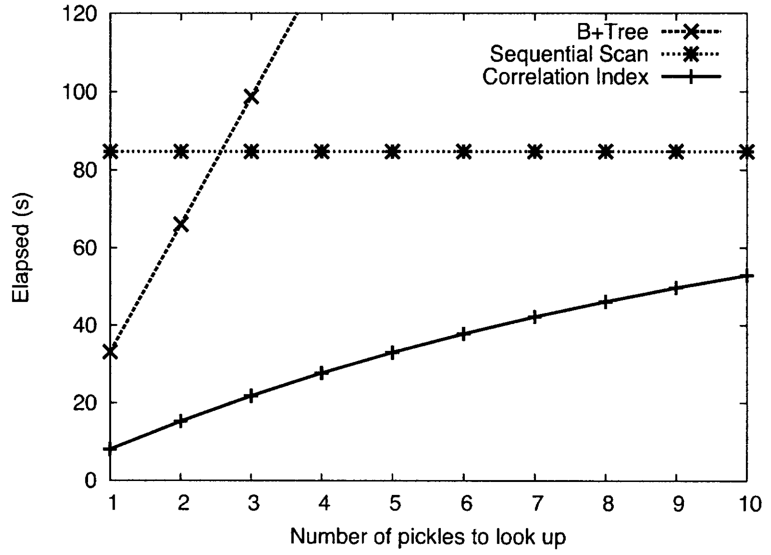
The PostgreSQL bitmap scan also behaves according to our expectations. When we cluster the table on *factory*, there is correlation between the lookup attribute and the clustered attribute, so the bitmap scan behaves similarly to the CI (since we are comparing our targetted implementation against a complete DBMS, we expect higher overhead from the DBMS). However, when we randomize the table, there are no useful correlations and the bitmap scan approximates the plain unclustered B+Tree. Note that the bitmap scan performance asymptotes to that of a sequential scan (at roughly 105s), since the bitmap scan access pattern is a subset of the sequential scan.

Recall to perform each query, we must first look up the set of clustered attribute keys to visit from the correlation index B+Tree structure. During our experiments, we logged the amount of time spent in this lookup phase compared to the time spent actually reading tuples from disk. Since the number of correlation index lookups is equal to the size of the query set, which is small, it is unsurprising we found the fraction of the total time spent in the lookup phase is insignificant. For the synthetic data, the fraction of time spent was between 1% and 3% of the total query time.

```
CLUSTER  synthetic ON factory;

 SELECT  *
   FROM  synthetic
  WHERE  pickle IN (p1, ..., pn);
```





Index type	**Size on disk (MB)**
Unclustered B+Tree	758.26
Correlation index	0.18359

Figure 5-3: Time elapsed for queries over the *pickle* (unclustered) and *factory* (clustered) attributes in the *synthetic* table: experiment (above) vs. model (below). We also show the size of each index type on disk.

5.2.2 TPC-H results

We first present our results for a pair of attributes where the CI wins over both the sequential scan and the unclustered B+Tree by over a factor of 20. We have clustered *lineitem* on the *receiptdate* attribute, and we perform lookups on the *shipdate* attribute. The query we have chosen computes the average sale price for a set of items shipped on any of set of interesting dates (for example, holidays). The results are shown in Figure 5-4.

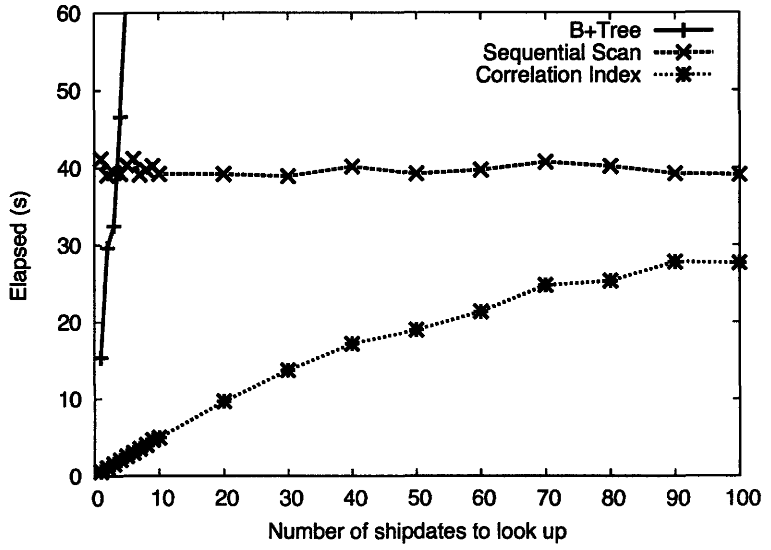
This pair of attributes is expensive for the B+Tree because each *shipdate* corresponds to approximately 7,000 tuples, resulting in a large number of random disk seeks for each query value. We can see the B+Tree curve in the results increasing sharply on the left, confirming our expectations. On the other hand, there is moderate correlation between *shipdates* and *receiptdates*, which results in strong CI performance.

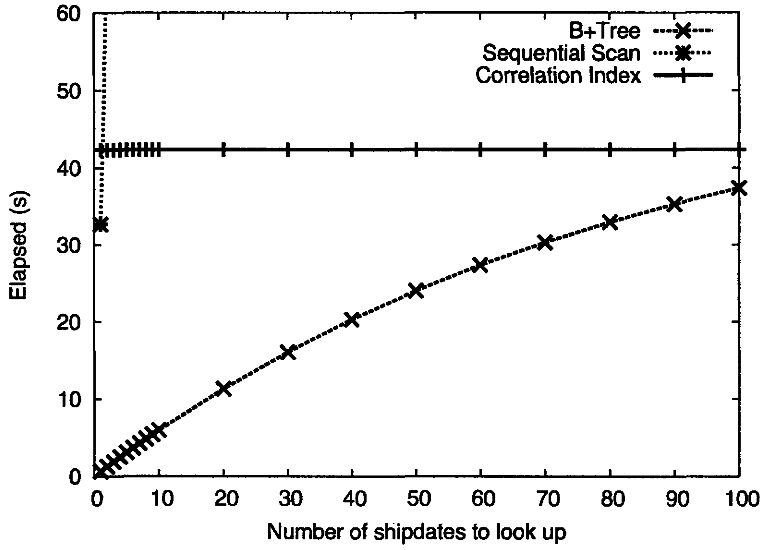
Next, in Figure 5-5 we present an interesting variation on the query above. The table that we load is identical; we have clustered *lineitem* on the *receiptdate* attribute. Instead of choosing the set of *shipdates* randomly, however, we apply a range predicate to choose a contiguous set of dates. The modified query is a realistic example of an aggregation over a range of dates for which we would like to make a business decision. We present only the plot of measurements, because the model prediction is identical to that in Figure 5-4.

The performance of sequential scan and the B+Tree in this experiment are roughly the same as that in the previous experiment, but the CI performance has improved by over an order of magnitude. It is easy to understand why by considering the semantic relationship between *shipdates* and *receiptdates*; specifically, an order is likely to be received in a small number of days after it has been shipped, and the attributes have a nearly linear relationship. As a result, a contiguous set of *shipdates* will map to a strongly overlapping set of (nearly contiguous) *receiptdates*. We therefore improve the CI access pattern dramatically, as it needs only to visit a small number of *receiptdate* ranges,

```
CLUSTER  lineitem ON receiptdate;

 SELECT  AVG(extendedprice * discount)
   FROM  lineitem
  WHERE  shipdate IN (date1, ..., daten)
```





Index type	**Size on disk (MB)**
Unclustered B+Tree	460.63
Correlation index	0.35156

Figure 5-4: Time elapsed for queries over *shipdate* (unclustered) and *receiptdate* (clustered) attributes in the *lineitem* table: experiment (above) vs. model (below). We also show the size of each index type on disk.

```
CLUSTER  lineitem ON receiptdate;

 SELECT  AVG(extendedprice * discount)
   FROM  lineitem
  WHERE  shipdate >= date_1
    AND  shipdate <= date_n;
```

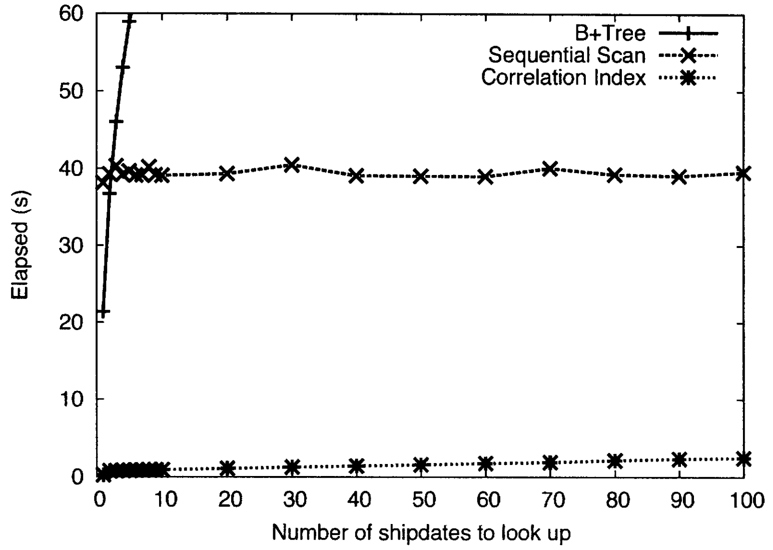


Figure 5-5: Time elapsed for queries over *shipdate* (unclustered) and *receiptdate* (clustered) attributes in the *lineitem* table, now using a range predicate. See Figure 5-4 for the model prediction.

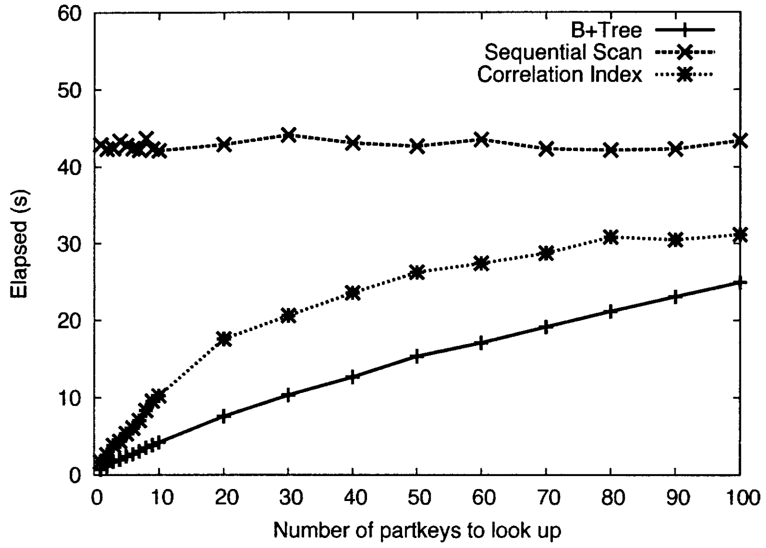
finding more matching *shipdates* within each range. As we have previously discussed in § 3.4, our analytical model becomes a poor predictor for the performance of the modified query, because it does not attempt to capture the relevant overlap.

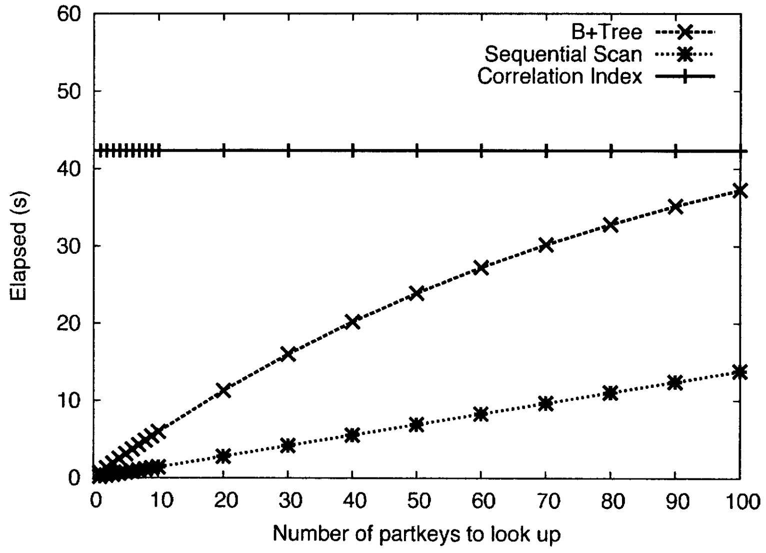
Now, we present results for a pair of attributes that is a variation on the earlier example in Figure 5-6 where the CI loses to the B+Tree in query time by a roughly consistent offset of 10 seconds. In the new experiment, the *lineitem* table remains clustered on the *receiptdate* attribute, but we now perform lookups on the *partkey* attribute. We can understand why the CI results in slower queries using the following statistics:

- partkeys identify a very small set of tuples (each partkey matches 30 tuples)
- the correlation between parts and the receiptdate is moderate (there are

```
CLUSTER   lineitem ON receiptdate;

 SELECT   AVG(extendedprice * discount)
   FROM   lineitem
  WHERE   partkey IN (p1, p2, ...);
```





Index type	Size on disk (MB)
Unclustered B+Tree	456.42
Correlation index	82.371

Figure 5-6: Time elapsed for queries over *partkey* (unclustered) and *receiptdate* (clustered) attributes in the *lineitem* table: experiment (above) vs. model (below). We also show the size of each index type on disk.

30 receiptdates for each part)

- the number of tuples for each receiptdate is large (there are 7000 tuples for each receiptdate)

The latter two statistics determining the performance of CI have not changed substantially compared to the earlier experiment, and thus the CI performance is roughly the same as in Figure 5-4. The B+Tree lookups are now substantially more efficient than before, because each value lookup results in only 30 random seeks. On the other hand, the CI must perform a random seek to each of 30 *receiptdates*, and furthermore it must read a total of over 200,000 tuples to compute a result set of merely 30 tuples.

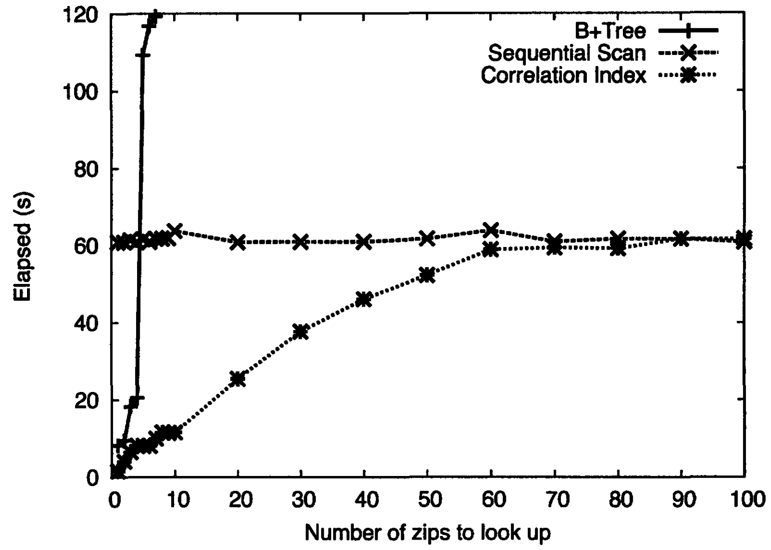
In each of our experiments over the *lineitem* table, our model does a reasonably good job of predicting the costs of each access method. In particular, the model captures the fact that the CI runtimes tend downward slightly from a straight line. The reason for this curvature is due to the caching of pages that are read repeatedly. However, the model overestimates the cost of B+Tree lookups, which is most likely due to optimizations in Berkeley DB that we do not model.

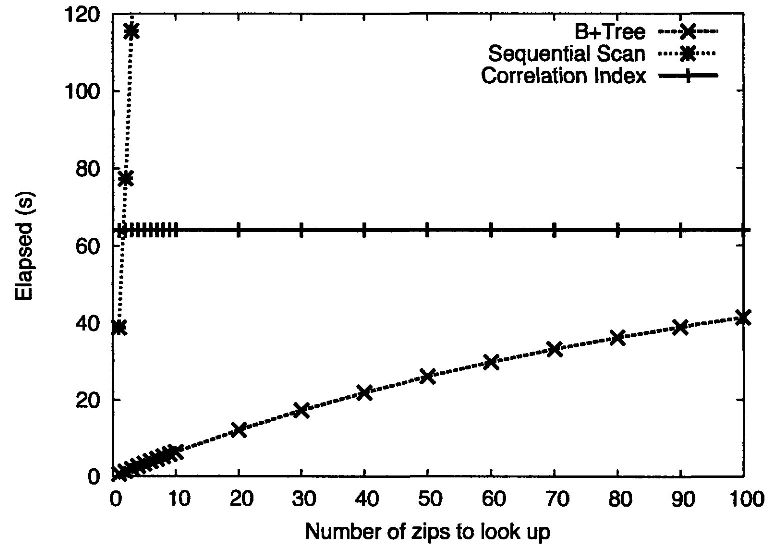
5.2.3 Census results

To conclude our presentation of experimental query results, we examine query performance over the *zip* and *longitude* attributes of the *census* table. In the original data, the zip code is a five-digit US mailing code defining a small geographic area and the longitude is a float value that specifies a geographic offset. Since *longitude* is essentially a unique identifier, by clustering over it we would expect the performance of lookups over zip codes to be nearly identical using CIs and unclustered B+Trees. This is because the CI will need to perform one random seek and read roughly one page for each result tuple (ignoring overlap), which is exactly the same as the B+Tree access pattern in our model.

```
CLUSTER   census ON integral_longitude;

 SELECT   count(person_id)
   FROM   census
  WHERE   zip IN (z1, ..., zn);
```





Index type	**Size on disk (MB)**
Unclustered B+Tree	4966.5
Correlation index	1.1328

Figure 5-7: Time elapsed for queries over *zip* (unclustered) and *longitude* (clustered) attributes in the *census* table: experiment (above) vs. model (below). The *longitude* values have been bucketed into integral bins from the original census data. We also show the size of each index type on disk.

Instead, to improve the performance of the CI, we have decided to apply a simple bucketing scheme to the *longitude* field. Here, we simply truncate each *longitude* to its integer value. As a result, there are 96 integral *longitude* values across the United States, and there are approximately 300 zip codes per longitude on average. The reason why we now expect CI to have better performance is that the CI can scan all of the tuples within an integral longitude sequentially, and the number of matching tuples within each longitude will be reasonably high. The results in Figure 5-7 confirm our expectations.

5.3 Parameter calculation results

In § 4.1.2, we presented both exact and approximate approaches to computing the values of the model parameter *c_per_u*. We conjectured that the sampling-based approach would provide sufficiently accurate results at a small fraction of the cost of computing the exact results. In this section, we validate our claims by evaluating the procedures over a subset of 8 numeric attributes (corresponding to 56 different *c_per_u* values) from the *lineitem* table. We perform our experiments on the Emulab machines described previously, by calculating distinct counts over a 2.5GB *lineitem* table with 18 million rows.

To compute *c_per_u* values directly, we first ran the query listed in Figure 4-1 on the PostgreSQL 8.2 database system. The results from this trial are labeled `postgresql_grouping` in Figure 5-8. We then used PostgreSQL to compute distinct counts over single attributes and pairs, in the trial labeled `postgresql_counts`. Both methods provide us with exact results.

Next, to evaluate our sampling-based approach, we developed a Distinct Sampling implementation in C++. The full Distinct Sampling algorithm returns a synopsis of tuples sampled from the full table that is suitable for subsequent predicate evaluation. Since the synopsis is irrelevant to our system – we care only about the distinct count estimate – we have simplified our implementation to accept only a single parameter, the space bound B on the number of tuples we can store at any time.

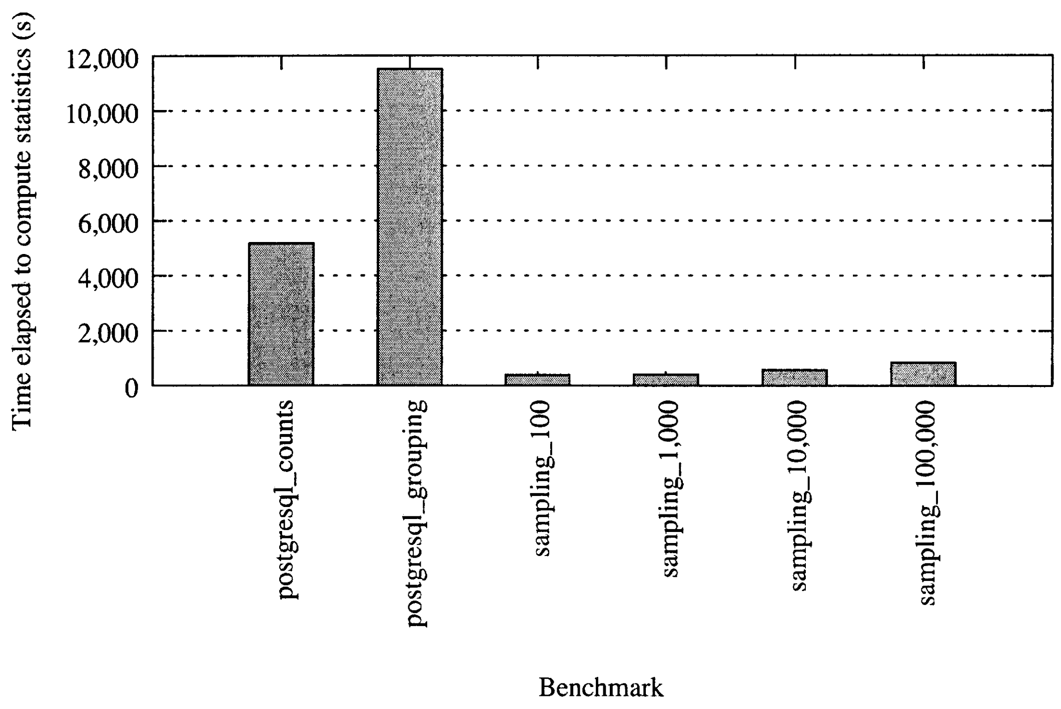


Figure 5-8: Time elapsed for six approaches that calculate *c_per_u* values: exact calculations in PostgreSQL and approximate calculations using Distinct Sampling with four different space bounds.

We created one Distinct Sampling instance for each single attribute and attribute pair in the reduced *lineitem* table, and we then performed a single table scan. We present results over four different space bounds: $B \in \{100, 1000, 10000, 100000\}$. These space bounds correspond to storing $\{.00056\%, .0056\%, .056\%, .56\%\}$ of the full table, respectively, for each of 28 Distinct Sampling instances in a trial. As the space bound increases, we expect a more costly runtime but more accurate results. Each trial with space bound B is labeled `sampling_`B in Figure 5-8.

Our results are summarized in Figure 5-8, which plots the runtime of each procedure, and Table 5.1, where we show the accuracy of each sampling trial. The *ratio error* is defined as $\max(\hat{D}/D, D/\hat{D})$, where D is the true distinct count and $\hat{D}$ is the estimate. Thus, values closer to 1.0 are better. Within PostgreSQL, our results show that computing *c_per_u* exactly using grouping aggregates re-

Table 5.1: Accuracy of Distinct Sampling lineitem estimates over four different space bounds. Ratio error values closer to 1.0 are better.

Space bound	**Memory usage (KB)**	**Average ratio error of $D(A_i)$ and $D(A_i, A_j)$ estimates**
100	2012	2.579
1000	3280	2.103
10000	14444	1.100
100000	109724	1.035

quires over 3 hours for our 2.5G lineitem table – far too expensive to be practical. Using distinct count calculations instead of grouping aggregates reduces the computation time by nearly half. Furthermore, sampling improves the computation time from 2-3 hours to 6-14 minutes – over an order of magnitude in improvement.

The accuracy results in Table 5.1 suggest that our sampling-based approach is highly effective. By choosing Distinct Sampling instances with a space bound that is merely .056% of the entire table, we are able to achieve an average of 10% error in a total of roughly 14MB of memory. Increasing the space bound to .56% of the table improves our average error to 3.5%, but it also increases the memory usage heavily to roughly 110MB. We conclude that it is possible to calculate the model parameters efficiently by using Distinct Sampling.

5.4 Summary of findings

In this chapter, we have presented experimental results over a variety of data with different characteristics and degrees of correlation. Our implementation establishes that CIs win in real query situations, often by a factor of two and sometimes by a factor of ten or more. Furthermore, the model predicts the true performance quite well in most situations, and we have identified areas where the model is inaccurate.

Furthermore, we have presented an evaluation of the Distinct Sampling algorithm implementing our sampling-based approach to computing model statistics. Our evaluation shows that sampling is an efficient and accurate way to compute the distinct count statistics required by our model.

Chapter 6

Conclusion

This thesis presents a new index data structure called a correlation index that enables OLAP databases to answer a wider range of queries from a single clustered index or sorted file. CIs exploit correlations between the key attribute of a clustered index and other unclustered attributes in the table. In order to predict when CIs will exhibit wins over alternative access methods, we developed an analytical cost model that is suitable for integration with existing query optimizers. Additionally, we described the *CI Advisor* tool that we use to identify pairs of attributes that are suitable for CIs. We described both exact and sampling-based approximate approaches to measuring the statistics required by our model. We compared the performance of an implementation of CIs against sequential scans and unclustered B+Tree indices in BerkeleyDB, and we furthermore evaluated the efficiency and accuracy of our approaches to gathering statistics.

We showed experimental results over three different data sets that validate the accuracy of our cost model and establish numerous cases where CIs accelerate lookup times by 5 to 20 times over either unclustered B+Trees or sequential scans. Furthermore, based on predictions from our model, we showed that in both TPC-H as well as a data set obtained from a local company that we will obtain gains of a factor of 2 or more in 25% of attribute pairs. Based on these results, we conclude that CIs, coupled with our analytical model, have the potential to improve substantially the performance of a broad class of index-based queries.

Bibliography

[1] Michael A. Bender and Haodong Hu. An adaptive packed-memory array. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 20–29, New York, NY, USA, 2006. ACM Press.

[2] J. Bunge and M. Fitzpatrick. Estimating the number of species: A review. *Journal of the American Statistical Association*, 88:364–373, 1993.

[3] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. In *TODS*, volume 19, 1994.

[4] S. Ceri and J. Widom. Deriving production rules for constraint maintenance. In *VLDB*, 1990.

[5] U. Chakravarthy, J. Grant, and L. Tanca. Automatic generation of production rules for integrity maintenance. *TODS*, 15(2), 1990.

[6] Moses Charikar, Surajit Chaudhuri, Rajeev Motwani, and Vivek R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, 2000.

[7] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *SIGMOD Record*, 26(1), 1997.

[8] Qi Cheng, Jarek Gryz, Fred Koo, T. Y. Cliff Leung, Linqi Liu, Xiaoyan Qian, and K. Bernhard Schiefer. Implementation of two semantic query optimization techniques in the DB2 universal database. In *VLDB*, 1999.

[9] Philippe Flajolet and G. Nigel Martin. Probabilistic counting algorithms for data base applications. *J. Comput. Syst. Sci.*, 31(2):182–209, 1985.

[10] Phillip B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB '01: Proceedings of the 27th International Conference on Very Large Data Bases*, pages 541–550, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[11] Parke Godfrey, Jarek Gryz, and Calisto Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *SIGMOD*, 2001.

[12] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and application of check constraints in DB2. In *ICDE*, 2001.

[13] Michael Hammer and Stan Zdonik. Knowledge based query processing. In *VLDB*, 1980.

[14] J. Han, Y. Cal, and N. Cercone. Knowledge discovery in database: An attribute-oriented approach. In *VLDB*, 1992.

[15] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Efficient discovery of function and approximate dependencies using partitions. In *ICDE*, 1998.

[16] J. King. Quist: A system for semantic query optimization in relational databases. In *VLDB*, 1981.

[17] H. Mannila and K-J. Raiha. Algorithms for inferring functional dependencies from relations. *TKDE*, 12(1), 94.

[18] Netezza Corporation. Data Warehouseing without the Complexitiy: How the Simplicity of the NPS Data Warehouse Appliance Lowers TCO. White Paper, 2006.

[19] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational

database management system. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.

[20] S. Shekhar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization: A data-driven approach. volume 5.

[21] Sreekumar T. Shenoy and Z. Meral Ozsoyoglu. A system for semantic query optimization. In *SIGMOD*, 1987.

[22] Sleepycat Software, Inc. Berkeley DB. White Paper, 1999.

[23] C. T. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *TKDE*, 1(3), 1989.

[24] Xiaohui Yu, Calisto Zuzarte, and Kenneth C. Sevcik. Towards estimating the number of distinct value combinations for a set of attributes. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 656–663, New York, NY, USA, 2005. ACM Press.