Designing Correlation Indices with Bucketing and Composition

Hideaki Kimura
Computer Science Department
Brown University

Advised by Stanley B. Zdonik
May 15, 2008

Preface

This paper is submitted in fulfillment of the requirement for my Sc.M. degree in the department of Computer Science at Brown University. The work is done in a joint research project between Massachusetts Institute of Technology and Brown University. We submitted a paper [17] for the 34th VLDB conference as a result of the collaboration.

My contribution to the project is to improve the cost model, automatic designer and experimental implementation as a realistic system. I started from surveying cardinality estimators described in Section 9 and then introduced bucketing and composition described in Section 5. Bucketing and composition makes Correlation Index (CI) more practical but it expands the explored design space much larger. I made an efficient CI designer with bucketing and composition based on the cardinality estimator survey. I also implemented an experimental system for testing CI designs on PostgreSQL and improved the accuracy of the CI cost model by the results described in Section 6.

The collaboration in the project was a success and gave me an opportunity to develop my research skills thanks to everyone in the project; George Huo, Alexander Rasin, Samuel Madden and my advisor Stanley B. Zdonik.

ABSTRACT

In relational query processing, there are generally two choices for 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 will frequently perform sequential scans, unless they can satisfy nearly all of the queries posed to them by a single clustered index [6], or unless specialized data structures – like bitmap indices, materialized views, or cubes – can be used to answer queries directly.

We present 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. We show that CIs can be implemented as an "add-on" to an existing database via a simple query rewriting scheme. In order to predict when CIs will exhibit wins over alternative access methods, we develop an analytical cost model that is suitable for integration with existing query optimizers. We also develop algorithms that search for strong candidate CIs and that recommend ways to "bucket" CIs to reduce their space utilization. We compare CI performance against sequential scans and unclustered B+Tree indices in PostgreSQL. Our results on several different data sets validate the accuracy of our cost model and establish numerous cases where CIs accelerate lookup times dramatically over other access methods. We also show that standard B+Trees can benefit from correlations just as CIs do, and that CIs are typically much smaller than B+Trees (by up to three orders of magnitude), making it possible to maintain many of them in memory over a single table.

1. INTRODUCTION

Online analytics 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 individual value lookups.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 many sequential scans. Though one can play various tricks to optimize sequential scans (for example, the Netezza warehouse appliance relies on special disk controllers and massive shared-nothing parallelism), ultimately sequential scans become a performance bottleneck.

The main idea that this paper explores is to use correlations between the key attribute of a clustered index (the "clustered attribute") and other "unclustered attributes" in the table to allow databases to answer a wider range of queries efficiently with a single clustered index. [1] Suppose we wish to apply a predicate to find all of the tuples with a particular unclustered attribute value; normally, this would require using an unclustered index (or a sequential scan) to access a set of disk pages arbitrarily scattered throughout the file. However, if there are correlations between the unclustered attribute and the clustered attribute, all of the satisfying tuples for the query may occur with just a few values of the clustered attribute, requiring access to only a few sequential ranges of pages (for example, given a clustered index on US State

[1] Although by default most databases cluster tables on the primary key, there is no particular reason or requirement that a primary key clustering be selected unless it leads to good performance.

over a table of customer records, a secondary index looking for customers from Boston will mostly hit pages in Massachusetts.) In this highly correlated case, the cost of using an unclustered index is much less than we would expect if pages were randomly distributed.

Hence, the main purpose of this paper is to 1) Explore the frequency with which such correlations occur, and 2) To develop a cost model that can predict the benefit such correlations have when accessing a relation through an unclustered index with a correlated, clustered index on it. Cost models are important because they allow the query optimizer to decide when a secondary index will be superior to a sequential scan and allow the database administrator to determine which secondary indices are likely to provide good performance on account of substantial correlations.

We also propose a secondary index representation, called a *correlation index* (CI) that is more compact than a basic unclustered index (which is important as we would like to support many such indices per clustered index in the system.) Our simple representation is a mapping from each distinct *value* (not tuple, as in an unclustered index) u in the domain of an unclustered attribute A_u to pages in the clustered index that contain tuples co-occurring with u in some tuple in the database. Then, queries over A_u can be answered by looking up the co-occurring values of u in the clustered index to find potentially matching tuples. We develop algorithms that recommend the best set of CIs to create for a particular query workload and schema.

In this paper, we describe the design and implementation of a correlation-indexing system, with the following key contributions:

1. We develop a model of secondary index performance (in particular, our compact CI representation) that allows us to predict how effective it 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.
2. We present a system design that is very low in complexity, and that can be integrated with existing access methods and query optimizers with little effort.
3. We present an algorithm, the *CI Advisor* that searches for the best CIs, considering not just CIs over single attributes, but also composite keys. The advisor also recommends ways to *bucket* CI keys to reduce their storage overhead without decreasing performance.
4. We evaluate the effectiveness of CIs on several data sets, coming from TPC-H, eBay, and the Sloan Digital Sky Survey (SDSS). We show that CIs can outperform both unclustered B+Trees (without an appropriate correlated index) and sequential scans by an order of magnitude. We also show that we can achieve the benefits of clustering using standard secondary B+Trees with appropriately matched clustered indices. We compare the sizes of CIs to B+Trees, showing that CIs are usually several orders of magnitude smaller.

In the next section, we present our CI representation in more detail. Then, in Section 3 and Section 4, we discuss our correlation index cost model and algorithms for CI utility. We elaborate on the idea of bucketing and describe building CIs over composite attributes in Section 5. Finally, in Section 6, we evaluate CI performance on several real world data sets before summarizing related work (Section 7) and concluding (Section 8).

2. SYSTEM OPERATION

In this section, we describe how correlation indices (CIs) are structured, built, and used. CIs are a very simple mapping data structure that work much as any database index does.

From a database client's standpoint, CIs also work much like standard indices; they support customary update and query operations. For the database administrator, we provide the *CI Advisor* tool to identify pairs of attributes that are likely to be good candidates for a CI. For such pairs, he can issue a simple DDL command linking a given attribute – the CI key – to a clustered index. We describe the operation of the CI Advisor in more detail in Section 4.

2.1 Building and Maintaining CIs

Given that the user wants to build a CI over an attribute $T.A_u$ of a table T (we call this is the *CI Attribute*), with a clustered index on attribute $T.A_c$, the CI is simply a mapping of the form $u \rightarrow S_c$, where

1. u is a value in the domain of $T.A_u$, and
2. 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 → {NH,MA}," indicating that there is a city called Boston in both Massachusetts and New Hampshire.

The algorithm for building a CI is shown in Algorithm 1. The basic algorithm works as follows: once the administrator issues a DDL command to create a CI, the system scans the table to build the index mapping (line 2). As the system scans the table, it looks up the CI key value in the mapping and adds the clustered index key to the set of key values (line 5). The system tracks the number of times a particular pair of (*uncorrelated*, *correlated*) values occurs using a "co-occurrence" count, which is initialized to 1 (line 5) and incremented as needed (line 9).

The count of the number of times a particular correlated value occurs with each uncorrelated value in the table is needed to support deletions. When a tuple t is deleted, the CI looks up the mapping m_{A_u} for the uncorrelated attribute value and decrements the count c for the value $t.A_c$ of the correlated attribute. When c reaches 0, the value $t.A_c$ is removed from m_{A_u}.

The insertion algorithm is very similar to the algorithm for building the index. The main loop (2 in Algorithm 1) is simply repeated for each new tuple that is added. Updates can simply be treated as a delete and an insert.

```
   input  : Table T with attribute T.A_u and clustered index I over
            attribute T.A_c
   output: Correlation index C, a map from T.A_u values to co-occurring
            T.A_c values, along with co-occurrence count.
 1 C ← new Map(Value → Set)
 2 foreach tuple t ∈ T do
 3     m ← C.get(t.A_u)
 4     if (m.get(t.A_c) = null) then
           /* Add fact that t.A_c co-occurred with t.A_u to
              mapping for t.A_u, initializing co-occurrence
              count to 1                                    */
 5         m.put(t.A_c, 1)
 6     end
 7     else
           /* Increment co-occurrence count for t.A_c in
              mapping for t.A_u                             */
 8         cnt ← m.get(t.A_c)
 9         m.put(t.A_c, cnt + 1)
10     end
11 end
12 return C
```

Algorithm 1: CI Construction Algorithm

Since a CI is just a key-value mapping from each unclustered attribute value to the corresponding clustered attribute values, it can be physically stored using any map data structure. This is con-

venient because database systems provide B+Trees and Hash Indices that can be used for this purpose. In our implementation, we physically represent a CI using a B+Tree keyed by the unclustered attribute, with the set of clustered index keys and counts as the value for each record in the B+Tree. Whenever a tuple is inserted, deleted, or modified, the CI must be updated as discussed above. 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 the sizes of CIs for several real-world attributes in our experimental evaluation in Section 6, showing that they are generally much more compact than the equivalent unclustered B+Tree.

2.2 Using CIs

The API for performing lookups on the CI is straightforward; the CI implements a single procedure, `ci_lookup`($\{v_{u1} \ldots v_{uN}\}$). It takes as input a set of N values over the CI attribute and returns a list clustered attribute values.

Rather than modifying the internals of the database system or optimizer to use CIs, we use a query rewrite method where queries with predicates over the unclustered attribute of a CI are augmented with predicates over the clustered attributes. The clustered attribute values that predicates range over are determined by performing a CI lookup.

Specifically, given a range predicate p over a CI attribute, the query rewriter looks up all of the records in this range in the CI. It takes the union of all of the resulting clustered attribute value sets and generates a sorted list. It then adds predicates over these clustered attribute values to the query, and submits the query to the optimizer for execution. It is necessary to retain predicates over the CI attribute, since some values in the clustered index may not satisfy the unclustered predicates – for example, a scan of the states "MA" and "NH" to find records with city "Boston" will encounter many records from non-satisfying cities ("Cambridge," "Manchester," etc.)

The optimizer is free to choose the plan that it determines will result in the best performance – for example, if the predicates over the CI attribute are very selective, and there is an unclustered B+Tree on the CI attribute, it may choose to perform a lookup on this index rather than performing less selective lookups on the correlated index. We find in our experiments, however, that when there is substantial correlation between the CI attribute and the correlated attribute, these introduced predicates often allow the query executor to perform significantly faster.

Figure 1 illustrates an example CI and how it might affect the plan used by a query executor. Here, the user has a table with three attributes: *state*, *city*, and *salary*, with a clustered B+Tree on *state* (called BT in the figure). The administrator has created a CI on *city*. The CI (shown on the top left of the figure) 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 rewriter introduces predicates over the states "MA", "NH", and "OH" into the query according because those states have cities named "Boston" or "Springfield" according to the CI (bottom left of the figure.) The database system generates a physical query plan (shown on the bottom right of the figure) that performs an in-order lookup on the clustered B+Tree to find the pages containing records from these states (1, i, and j in the example). The tuples on these pages are fed to a selection operator, which just returns tuples with city equal to "Boston" or "Springfield."

2.3 Discussion

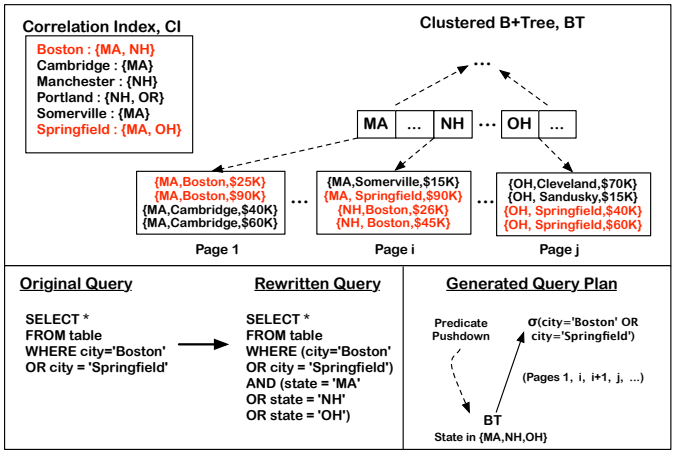

Figure 1: Diagram illustrating an example CI index and its use in a query plan.

In this section, we briefly review a few important points about the applicability and value of CIs.

2.3.1 *Cases when CIs are a Benefit*

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.

Correlation indices perform best when the number of tuples across all clustered values that an unclustered value maps to 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, the index will also be useful for answering queries over city name, since the number of tuples for each county is small as compared to the size of the database. Conversely, correlations with small-domain attributes may be less valuable (e.g., a clustered index on gender – even if highly correlated with some unclustered attribute – is unlikely to reduce access costs for most scans.)

2.3.2 *Frequency of Correlations*

Correlations that can be exploited by CIs arise naturally in many domains. In some cases, these are due to physical constraints (such as the fact that the ship date of a product must be after the order date, often within a day or two). In other cases, they arise because databases are storing hierarchical or near-hierarchical data. The geographic examples we have used illustrate this nicely: states contain cities, counties, zip codes and area codes and counties contain cities. Area codes and zip codes may overlap several counties or even states (for example, the 83110 zip code is in both Wyoming and Idaho), but are strongly correlated with geography. Clustering over multiple such geographic attributes and then building a CI on the other attributes (which is likely to be quite feasible as the CIs will be very compact) will offer good performance for many queries.

Hierarchies occur naturally in many other domains as well – for example, product catalogs in websites like eBay and Amazon often categorize items in a semi-hierarchical fashion. Items on eBay are placed in a six-level hierarchy, but a given item may categorized in several different places in the hierarchy. If a correlated index is built on any level of this categorization, CIs on the other

levels can be used to quickly look up items in certain categories or sub-categories. We show examples of queries over both product hierarchies and geographic data in our results in Section 6.

2.3.3 Relationship to Unclustered Indices

It is not necessary to use a CI to exploit correlations in many database systems as unclustered (dense, B+Tree) indices achieve similar gains with an appropriately selected correlated attribute. For example, in PostgreSQL, rather than using a CI and predicate introduction, one can build a secondary index on the unclustered attribute. When applying predicates over this attribute, rather than performing many random I/Os by following pointers from the unclustered index to the data pages, PostgreSQL uses the index to build a bitmap (with one bit per page) indicating the pages that contain records that match predicates. It then scans the heap file sequentially and reads only the pages whose bits are set in the bitmap. If the table is clustered on an attribute that is highly correlated with the unclustered index key, then this scan will access far fewer pages than if there was no such correlation. This results in the same benefit that CIs exploit, so PostgreSQL will perform about the same number of I/Os when using a CI or a unclustered index. (Because unclustered indices are larger, they may incur more I/Os performing lookups to generate the bitmap.) We show that the two approaches are comparable in our experiments.

To illustrate the benefit obtained by both CIs and unclustered indices, we visualized the distribution of page accesses when performing lookups of the form $A_u = v$ (for different A_u) via an unclustered index over A_u on the *lineitem* table from the TPC-H benchmark. We performed two different experiments: in one case, we had a clustered B+Tree on an attribute A_c (correlated with A_u), and in another case the data was clustered randomly. Figure 2 shows the distribution of page accesses throughout the *lineitem* table. Each long, narrow box represents the pages in *lineitem* numbered from 0 to n; a black vertical bar for a particular page indicates that it was read. The upper box of each pair shows pages read when the clustered index was present; the lower box shows pages read when the table was clustered randomly. For these attributes, a clustered index accesses substantially fewer pages, particularly in high-correlation cases (e.g., *shipdate* and *receiptdate*).

Besides the space savings offered by a CI, the major contribution of our work is to demonstrate a cost model and the design of the index advisor that help in selecting the best attribute (or set of attributes) on which a table should be clustered, which we focus on in Sections 3–5. First, we briefly describe how CIs can be bucketed.

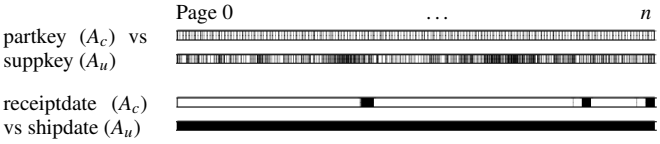

Figure 2: Visualization showing page access patterns when performing lookup via an unclustered B+Tree on A_u over the *lineitem* table in TPC-H with and without a clustered index on a correlated attribute A_c. Lookups were performed on three A_u values in each case and the accessed pages were recorded.

2.4 Bucketing CIs

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 are relatively small. However, for large attribute domains (such as with continuous attributes), 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 unclustered 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:

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

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

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

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 in the interest of brevity.

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). We evaluate the effects of bucketing in more detail in Section 5, and also discuss a sampling-based algorithm we have developed to search for a good bucketing.

3. MODEL

So far in this paper, we have given some intuition for situations where correlations give us benefits for unclustered index lookups; however, we have not yet provided a formal analysis for when this will be true. In this section, 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 CIs, full table scans, and unclustered B+Trees in situations with or without useful correlations.

3.1 Preliminaries

In the following discussion, we assume a table with clustered attribute A_c and secondary attribute A_u on which we query. Our model assumes that the table is stored as a clustered B+Tree sorted on A_c for fast sequential scans. Thus, to read all of the tuples corresponding to a clustered attribute value, we perform a lookup for the value in the clustered B+Tree index (which is independent of a CI) and read the relevant tuples sequentially.

We do not charge the B+Tree or CI access methods for reading index pages, because the caching of index pages is specific to the cache implementation (we show in Section 6 that CIs are often far smaller and more likely to fit in cache). We assume that the cache initially holds none of the data pages.

In Table 1, we summarize the statistics that we calculate over each relation. Additionally, in Table 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 Section 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 with traversing a B+Tree nor with filtering tuples in the CI, and the cost of the sequential

Table 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.
u_tups	Average number of tuples appearing with each A_u 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.
btree_height	Average height of a clustered B+Tree path, root to leaf.
n_lookups	Number of A_u values to look up in one query.

Table 2: Hardware parameters used by the cost model.

sequential_page_cost	Time to read one disk page sequentially.
Typical value:	.078 ms
disk_seek_cost	Time to seek to a random disk page and read it.
Typical value:	5.5 ms

scan is independent of the number of values we look up. We have observed these assumptions to be reasonable in our results in Section 6.

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 underestimated the true cost of a scan in a popular commercial database implementation, but the effects of fragmentation are unlikely to dominate the cost.

3.3 Cost of Correlation Index

Suppose that the CI has a set of A_u values to look up. For each A_u value, the CI visits *c_per_u* different clustered attribute values. We need to perform one clustered B+Tree lookup to reach each of these clustered attribute values, followed by a scan of all of the pages for that A_c value. We model the cost of the clustered B+Tree lookup to be proportional to the height of the B+Tree. In terms of the parameters we are given, the number of pages for a given A_c value is *c_tups*/*tups_per_page*. When the CI scans a large fraction of the file, its access pattern becomes gradually more like a full table scan – indeed, the CI pattern will never be more expensive than a sequential scan. Combining these expressions, the overall cost of a CI lookup is:

$$c\_pages = \frac{c\_tups}{tups\_per\_page}$$

$$cost_{ci} = \min((c\_per\_u)((disk\_seek\_cost)(btree\_height) + (sequential\_page\_cost)(c\_pages)),\ cost_{scan})$$

One simplification in our model is that we ignore the overlap between the sets of A_c keys associated with 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. This is a concern, for example, when evaluating a range predicate over an unclustered attribute that has linear correlation with the clustered attribute (order receipt dates will overlap heavily for a range of ship dates). Although we have chosen to omit this statistic in favor of simplicity, we describe in Section 6.2 a procedure that examines a series of bucketings to correct for the overestimation.

3.4 Cost of Unclustered B+Tree

So far, we have argued that correlations play a major role in determining the cost of unclustered index lookups. As a result, it makes sense to model the cost of unclustered B+Tree operations when we expect no meaningful correlations separately from the case when we are aware of correlations.

3.4.1 Uncorrelated Case

In the absence of meaningful correlations between the indexed attribute and the clustered attribute, we call the unclustered B+Tree a *lonely* index. We model every tuple that the lonely B+Tree reads as a disk seek. The reason why we charge the costs of expensive seeks here is that we have no expectation for any two tuples to fall on the same page, unless the fraction of the table scanned is large (when a sequential scan will do better). In a reasonable implementation of a B+Tree access method, such as PostgreSQL's bitmap scan, the relevant pages will be accessed in sorted order (see Section 6.1 for more details). As a result, B+Tree performance should level off to that of a sequential scan under low selectivity. We expect *u_tups* tuples to be associated to each A_u value, and the cost of *n_lookups* values is simply

$$cost_{uncorrelated} = \min((disk\_seek\_cost)(u\_tups)(n\_lookups),\ cost_{scan})$$

3.4.2 Correlated Case

In contrast to the expensive seeks per tuple that take place in the uncorrelated case, the unclustered B+Tree behaves much like a CI when we ensure useful correlations. The access pattern will involve large seeks between islands of densely grouped pages (as in Figure 2), and reads within each island will be largely sequential due to short seeks and disk prefetching. We therefore apply the same expression as we use for the CI to calculate the cost of a lookup on an unclustered B+Tree with meaningful correlations. Our results in Section 6.2 show that the costs of a CI and a B+Tree in this case are quite similar in practice.

3.5 Discussion

Based on the above analysis, we provide some intuition for situations where a CI might be more or less expensive than a sequential scan or unclustered B+Tree.

Sequential scan: The CI access pattern can be thought of as a subset of a sequential scan – that is, the CI always reads segments of a file in the same order as a sequential scan would, but it jumps over some stretches of the file. 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).

Unclustered B+Tree: The difference between the performance of CIs and secondary B+Trees is less straightforward to grasp. In the case when the database administrator is deciding whether to re-cluster an existing table that employs unclustered B+Trees, the model predicts that a CI with correlations will frequently perform much better than a lonely B+Tree. When *c_tups* is high, a lonely B+Tree may still do better at high selectivity queries because the CI will be forced to scan large ranges of irrelevant tuples. B+Tree lookups are more expensive, however, when there are many tuples for each A_u value – with no useful correlations, the access pattern involves many seeks per lookup.

If we look at a table that already exhibits strong correlations between the clustered attribute and unclustered attributes, the model explains that a CI and B+Tree will perform similarly. The underlying access patterns read pages in nearly the same way, differing only in that the CI will scan some ranges of irrelevant tuples where

the B+Tree seeks over them. In this case, the important consideration is that the CI is often orders of magnitude smaller than the corresponding B+Tree. Thus, we trade a small overhead in performance to be able to index over many more attributes and to hold the index structures in memory.

4. PREDICTION

There are several reasons why it is valuable to be able to determine whether a CI will be beneficial in query processing. First, a database administrator may need to understand whether creating CIs on a particular pair of attributes is likely to improve performance; although CIs are compact, creating them on every pair of attributes is not a good idea as the update and storage penalty will be severe. Second, the query rewriter needs to be able to estimate whether a given query should be rewritten or not; introducing additional predicates on the clustered attribute slows down the query compiler and introduces overhead into query execution and may cause the query optimizer to generate a non-optimal plan.

For these reasons, we have developed the *CI Advisor*, a tool that scans existing tables and calculates the statistics needed by the cost model. Given these statistics and measurements of underlying hardware properties, the CI Advisor can predict how much (or if) a given pair of attributes benefit from a CI; the database administrator can use these measurements to choose to build CIs and cluster tables on high-benefit attribute pairs 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 Section 6, 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 its statistics based on the current state of the database. Our approach for doing this is quite simple, and we use a sampling-based method to reduce the heavy costs of exact parameter calculation.

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. Similarly, the DBMS must be able to determine *btree_height*, the height of a clustered B+Tree index. 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 by the average width of a tuple.

Our model further relies on the *sequential_page_cost* and *disk_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.

4.1.1 Estimating *c_per_u*

The remaining *c_per_u* statistic, 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. The problem of estimating the *c_per_u* counts reduces to the problem of predicting the number of distinct values in a column over some partition of that column.

We first observe that it is unnecessary to calculate the distinct A_c value count for each A_u group explicitly, since the model only requires an average over all groups. Let us write the number of distinct values over a pair of attributes A_i and A_j as $D(A_i, A_j)$ and the number of distinct values over a single attribute as $D(A_i)$. Then, it is clear that we can also write *c_per_u* as $D(A_u, A_c)/D(A_u)$. In other words, instead of computing *c_per_u* values by averaging over an expensive grouping aggregate operation, we can alternatively calculate distinct counts for each single attribute and each attribute pair. Then, we simply divide the distinct counts as necessary to derive each *c_per_u* value.

Since computing precise distinct value counts can be expensive for large tables, it is natural to ask if we can achieve reasonable estimates via sampling. 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.

The basic problem of predicting the number of distinct values in a column has seen extensive treatment in both the database and statistics communities, where it is known as the problem of estimating the number of species (e.g. [1]). We have evaluated the Distinct Sampling (DS) algorithm by Gibbons [9], which achieves estimates that are far more accurate than purely sampling-based approaches at the cost of one full table scan.

To validate the effectiveness of DS in estimating *c_per_u* values, we developed a DS implementation in C++. We created one DS instance for each single attribute and attribute pair in a reduced TPC-H *lineitem* table with 18 million tuples (see Section 6.1 for more information about the *lineitem* table). We compared the DS estimates to the exact values and present accuracy results over four different DS space bounds: $B \in \{100, 1000, 10000, 100000\}$ tuples. These space bounds correspond to storing {.00056%, .0056%, .056%, .56%} of the full table, respectively, for each of 28 DS instances in a trial. As the space bound increases, we expect a more costly runtime but more accurate results. We found that, within PostgreSQL, computing *c_per_u* exactly using distinct count queries required over 1.5 hours for our 2.5GB *lineitem* table – far too expensive to be practical. Using DS improves the computation time to 6 minutes for $B = 100$ and 14 minutes for $B = 100,000$ – over an order of magnitude in improvement.

We also measured the accuracy of our sampling-based approach and found it to be highly effective. By choosing DS instances with a space bound that is merely .056% ($B = 10,000$) of the entire table, we are able to achieve an average of 10% error in a total of roughly 14MB of memory; for space bound of .56% of the table, error decreases to 3.5%, but memory usage grows to 110 MB. These results show that it is possible to calculate the model parameters efficiently by using DS. Furthermore, DS has the key property that the $D(A_i)$ and $D(A_i, A_j)$ value estimates can be maintained efficiently online in the presence of insertions. It is possible, therefore, for the DBMS to maintain up-to-date *c_per_u* estimates for use by the planner during query optimization.

5. BUCKETING AND COMPOSITES

As discussed earlier, CIs work best when a strong correlation exists between the indexed column and the clustered column. However, there are situations where a composition of two columns might have a much stronger correlation than either of the columns individually. For example, consider longitude and latitude. Each can cross many different zipcodes, but combined, (longitude, latitude) can determine a zip code uniquely. This is particularly useful because it works even if longitude and latitude are bucketed using a reasonable bucket size such as 1 second. A traditional (secondary) composite B+Tree over a large table cannot handle this well because, unless there are highly selective predicates on both

key columns, it will have to read a large fraction of the B+Tree pages. Since secondary indices are inherently dense (i.e., they store one entry for each row) scanning such an index requires significant I/O. Because bucketed CIs are so much smaller, their I/O requirements can be significantly less.

In this section, we describe how the CI Advisor finds such composite correlations from a vast number of possible column combinations and proposes promising column bucketings that keep the size of CI small without significantly degrading query performance. Our experimental results show that a well designed composite CI can be both faster than a composite B+Tree Index and up to three orders of magnitude smaller.

Before going into the details of the composite CI selection algorithm, we first describe how the CI advisor chooses possible bucketings for a single column that contains many values. We describe how to do this for both for the clustered attribute and the unclustered attribute (for which we build the CI). For unclustered attributes, the CI Advisor produces a set of candidate bucketings for every column, which are used by the composite CI selection algorithm.

5.1 Bucketing Many-valued Columns

As described in Section 2.4, bucketing can dramatically reduce the size of a CI; in particular, bucketing allows the CI Advisor to consider many-valued (even unique) columns when making CI recommendations. However, we must be careful when choosing bucketing granularity. Very large buckets may result in poor performance because of unnecessary reads of large blocks of the correlated attribute, while small buckets produce large data structures, increasing CI access cost (and preventing it from fitting in memory). In this section we describe how our CI Advisor algorithm finds the "ideal" bucketing granularity that strikes a balance between size and performance.

5.1.1 Clustered Column Bucketing

If the clustered column or columns are many-valued, the CI can become very large even in the presence of a strong correlation between the clustered column and the unclustered column, since each unclustered attribute value will map to many clustered values. This causes two problems: first, CI access becomes more expensive due to its size. Second, the rewritten query has an extremely long IN clause, containing one entry per selected row in the worst case. We found that this can cause a significant overhead in the PostgreSQL query optimizer and execution engine.

To bucket the clustered column, we add a new column to the table that represents the "bucket ID." All of the tuples with the same clustered attribute value will have the same bucket ID, and some consecutive clustered attribute values will also have the same bucket ID. The CI then records mappings from unclustered values to bucket IDs, rather than to values of the clustered attribute. To actually perform the bucketing, during its sequential scan of the table (to compute *c_per_u* statistics), the CI Advisor begins by assigning tuples to bucket $i = 1$. Once it has read b tuples, it reads the value v of the clustered attribute of the bth tuple. It continues assigning tuples to bucket i until the value of the clustered attribute is no longer v, at which point it starts assigning tuples to bucket $i + 1$ and increments i (this ensures that a particular clustered attribute value is not spread across multiple buckets). This process continues until all tuples have been assigned a bucket.

Observe that the primary effect of bucketing is to cause queries rewritten to use a CI to read a larger sequential range of the clustered attribute, increasing sequential I/O but not adding disk seeks. We have observed that the additional effects of this sequential I/O are not dramatic. To illustrate this, we "bucketized" the Sloan Digital Sky Survey (SDSS) dataset, containing 3GB of data about celestial imagery (see Section 6). We then measured the time to run the query *SX6*, which does a lookup on two values of the field *fieldId*, which is well correlated with the clustered attribute (*ObjID* in this case). We varied the bucketing of the clustered attribute. The results are shown in Table 3. We found that performance is relatively insensitive to the bucket size; a value of b such that about 10 pages of tuples map to each bucket appears to work well, taking only about 1 ms longer to read than no bucketing (bucket size = 1). The reason for this is that random I/O dominates the cost of sequential I/O.

Table 3: Clustered column bucketing granularity and I/O cost

Bucket Size [pages/bucket]	Physical Scans [pages]	IO Cost [ms]
1	96	15.34
5	105	15.925
10	110	16.25
15	135	17.875
20	140	18.2
40	160	19.5

5.1.2 Bucketing Unclustered Columns

Bucketing in unclustered columns has a larger effect on performance than bucketing in clustered columns because merging two consecutive values in the unclustered domain will potentially increase the amount of random I/O the system must perform (since it will have to look up additional, possibly non-consecutive values in the clustered attribute). The previous section showed that bucketing the clustered attribute only causes additional sequential I/O.

The CI Advisor performs bucketing by constructing equi-width histograms over unclustered attribute values based on random samples of attributes it is considering to build a bucketed CI over. These samples are randomly collected during the sequential pass of the table, yielding an optimum random sample as described in [20].

After collecting a random sample that fits in memory, the CI advisor builds histograms of several different bucket widths from the sample. Each of these histograms represents one possible bucketing scheme for the attribute under consideration. For a single-attribute CI, the *c_per_u* value for each bucketing can be computed directly from each histogram, by calculating the average number of clustered attribute values that appear in each bin of the histogram, as described in Section 4. Histograms with fewer, wider bins will have more clustered values per bin and a higher *c_per_u*, whereas histograms with more, narrower bins, will have lower *c_per_u* values. Of course, wider bins also result in a smaller *c_per_u*. In practice, we find that there is often a "natural" bucketing to the data that results in little increase in *c_per_u* while substantially reducing the index; we show this effect in our experiments in Section 6.

One question that remains is how we determine how many different bucketings to consider for each attribute. Our algorithm works by considering all bucketings that result in a number of values per bucket between a range *minv* and *maxv*, with bucket sizes scaling exponentially (in our experiments, we use powers of 2). For example, if a column has 1000 values, with $minv = 10$ and $maxv = 100$, the algorithm considers bucket widths of 16, 32, and 64 (since a bucket width of 8 yields less than 10 values per bucket, and a bucket width of 128 yields more than 100 values per bucket). This (admittedly heuristic) approach captures the intuition that a low-cardinality column should not be bucketed, and that we do not want to consider too many bucketings for many-valued columns in order to keep search costs down.

As another example, Table 4 below shows the output from bucketing on the SDSS data set. Here, the CI advisor outputs columns such as *mode* and *type*, which are few-valued, without bucketing. For the many-valued columns *fieldID* and *psfMag_g*, it recommends a range of bucketings that keep the number of values per bucket in the range between 10 and 20000.

In the case of building a composite CI, we do not directly compute *c_per_u* for each of the single-attribute histograms, but rather pass the possible binnings and the random sample we collected to the composite CI selection algorithm which tries to select a good multi-attribute CI. We describe this process next.

Table 4: Unclustered column bucketings considered for *SX6* query in the *SDSS* benchmark.

Column	Cardinality	Bucket Widths
mode	3	*none*
type	5	*none*
psfMag_g	196352	$2^4 \sim 2^{14}$
fieldID	251	*none* $\sim 2^4$

5.2 Composite CI Selection

The number of possible composite CI designs for a given table is very large because there are $\prod_{c=C_1}^{C_N} (Bucketing(c) + 1) - 1$ unique combinations of N columns and bucketings. Consider Table 4 again. Here, there are two options for *mode*: whether to include it or not. For *fieldID*, there are four options: to include it unbucketed, to include it with bucket width 4^1 or 4^2, or not to include it. Similar choices apply for the other attributes. Hence, in total, Table 4 implies $(2*2*7*4)-1 = 111$ different *candidate designs* for CIs from just 4 columns. Unlike the single column CI prediction, running Distinct Sampling for all candidate designs is impractical. Hence, we combined a set of known techniques from the database literature to achieve practical performance and reasonable accuracy to solve this problem.

5.2.1 Training Queries

A composite CI can help the query execution only when some (or, ideally, all) of its columns are used as predicates in the query. In other words, an interesting CI design for a query should contain some subset of the predicated columns. As each single query usually has a fairly small number of predicates, this limits the size of a CI design for that query to a much smaller space than all table columns.

For example, in our SDSS dataset, the DBA might provide the following set of sample queries (alternatively, such a training set can also be collected by monitoring popular queries at runtime):

Training Set

Query 1: SELECT ... FROM ... WHERE *ra* BETWEEN 170 AND 190 AND *dec* < 0 AND *mode* = 1

$\Longrightarrow$ { *ra*, *dec*, *mode* }

Query 2: SELECT ... FROM ... WHERE *fieldID* IN (...) AND *mode* = 1 AND *type* = 6 AND *psfMag_g* < 20

$\Longrightarrow$ { *fieldID*, *mode*, *type*, *psfMag_g* }

Query 3: ...

The goal of the Composite CI Designer is to output one or more recommended CI designs for each query with the expected speedup factors and CI size estimates. The DBA can then choose which CIs to create. As long as CIs are small, it is reasonable to expect that there will be several CIs on any given table.

5.2.2 Fast Cardinality Estimation

Our CI Advisor exhaustively tries possible CI designs for a given training set query. Because there are only a few predicated columns per query, and because we have selected a small number of possible bucketings per column, in practice the total number of designs that must be considered is not infeasible, but can still be quite large. To evaluate a design, the CI advisor must compute the *c_per_u* for the composite CI columns of each candidate composite CI.

We previously used Distinct Sampling (DS) in Section 4.1.1 for cardinality estimation, but DS cannot output accurate cardinality estimates unless it is able to scan the entirety of a relation. As we cannot keep the whole table in memory, computing a distinct sample requires substantial I/O. Thus, it would be very slow to perform many passes of DS to estimate the cardinalities of a large number of candidate composite CIs. One way to avoid this repeated I/O cost would be to scan the table only once and compute cardinality estimates for many candidate composite CIs simultaneously. However, as we verified in Section 4.1.1, in order to keep the error ratio small, DS requires substantial amounts of memory for each cardinality estimate, limiting the number of simultaneous cardinality estimates we can perform.

Hence, to recompute these estimates we use the Adaptive Estimator (AE) algorithm [5]. This allows us to quickly estimate the cardinality of each combination of columns and its respective bucketings. AE is a cardinality estimation algorithm based on random samples which does not require a full table scan. In our evaluation of different cardinality estimation algorithms that use random samples, AE gave the most stable results and provided reasonable accuracy on datasets with varying skew. It also runs very quickly when data samples are already in memory. See Section 9 for more detail.

We use AE as follows: as discussed earlier, the CI Advisor preserves random samples collected during the initial table scan and histogram collection. For each CI design under consideration, we can bucketize this random sample according to the current design. We can then compute *c_per_u* over this sample as in Section 4, using AE instead of distinct sampling.

In our experiments, the CI Advisor uses a sample size of 30,000 tuples, which we found to be sufficient for the datasets we experimented on. Using this sample, AE can compute cardinality and bucketing estimates in approximately 5 milliseconds per candidate design with reasonable accuracy. In our experiments, the CI Advisor can return candidate CI designs in under 20 seconds for a query with 4 or 5 predicated columns. Given that the CI Advisor is an offline algorithm, we believe this is pragmatic.

5.2.3 Ranking designs by size and *c_per_u*

As we described in Section 3, the *c_per_u* associated to a CI is a good indicator of the expected query runtime improvement. However, a large CI tends to be less useful, even if it has a very low *c_per_u*, because it requires too much disk space and thus provides no advantage over conventional B+Tree indices. Therefore simply recommending the CI with the lowest *c_per_u* is a poor idea. For this reason, we output CI designs grouped by size category – e.g. CIs less than 100KB, CIs less than 1MB, and so on.

Table 5 shows CIs designs grouped by size for our SDSS dataset, sorted by *c_per_u* rating within each group. This format allows the DBA to evaluate the trade-off between disk space and query improvement based on their requirements.

5.3 Discussion

There are two ways to implement a composite CI recommended by CI Advisor. One way is to construct a CI as described in Sec-

Table 5: Recommended composite CI designs grouped by size and sorted by *c_per_u*

CI smaller than 100KB

CI Design	*c_per_u*	Size
type, *fieldID*	12.4	62KB
mode, *fieldID*	18.4	61KB
fieldID	20.8	23KB
...	...	...

CI smaller than 500KB

CI Design	*c_per_u*	Size
psfMag_g(4^7), *mode*, *fieldID*	6.4	315KB
psfMag_g(4^7), *type*, *fieldID*	9.2	304KB
type, *mode*, *fieldID*	10.1	125KB
type, *fieldID*	12.4	62KB
...	...	...

tion 2 with composite values, which is what we did in our experiments. Another way is to *intersect* single column CIs. Intersecting several CIs can often return a bitmap result that is very similar to the composite CI result, but can sometimes cause a clustered attribute value to be read when when a page has at least one tuple that satisfies every predicate but has no tuples that satisfy all predicates. A purely composite CI would not make this mistake. The advantage of using CI intersection is that we can use the same CI in multiple composite predicate lookups, assuming similar bucketing granularity works across these multiple predicates. Thus we might be able to build one CI per column that is used in any of the queries, rather than building one CI (or even several) CIs per query.

As currently implemented, the CI Advisor has to be re-run from scratch to evaluate the expected benefit of a new clustered index. For this reason, finding a multi-column clustered index may be prohibitively expensive. Most of our experiments only use the best single column clustered attribute (although, in a few cases, we did manually identify well-performing composite clustered attributes). Some of our results suggest that a good clustered index design might be to cluster on some composition of columns where each column has strong correlation with a different group of predicated columns, since intuitively, we want a clustered index that is correlated with all (or most) predicated columns to make the most of CIs or any other secondary index structures. Exploring more efficient algorithms to search for the best combination of clustered attribute columns remains an area for future work.

6. EXPERIMENTAL EVALUATION

In this section, we present an experimental validation of our results. The primary goals of our experiments are to validate the accuracy of our analytical model, to establish that useful correlations are reasonably common in large data sets, and to explore optimal CI design using ideal bucketing based on underlying correlations.

6.1 Setup

We ran our tests on a single processor machine with 1G of RAM and a 320G 7200rpm SATA II disk. All experiments were run on PostgreSQL 8.3. We flushed memory caches between runs by using the Linux `/proc/sys/vm/drop_caches` mechanism and by restarting PostgreSQL for each trial. Note that whenever we compare our results to a B+Tree, we are using the standard PostgreSQL secondary index.

In order to perform multiple index probes for a single query, PostgreSQL implements the *bitmap scan* approach. When looking up a set of values $v_1, \ldots, v_n$ in a column with a B+Tree index, PostgreSQL first creates an in-memory bitmap whose bits represent each possible tuple offset in the heap file. For each resulting file offset identified by the B+Tree for each query value v_i, it sets the corresponding bit. After all index probes have been performed, PostgreSQL scans the bitmap in-order and reads the corresponding file locations. This approach effectively sorts the set of pages visited, minimizing the cost of disk seeks.

Hierarchical Data: The dataset that we use for the majority of our experiments is derived from eBay category descriptions that are freely available on the web [8]. The eBay data contain 24,000 categories arranged in a hierarchy of sub-categories with a maximum of 6 levels (e.g. antiques → architectural & garden → hardware → locks & keys).

We have populated this hierarchy with unique *ItemID*s. We chose 500 to 3000 *ItemID*s uniformly per category, resulting in a table with 43M rows (occupying 3.5GB on disk). Each category is assigned a unique key value as its Category ID (*CATID*), and the sub-categories for each *CATID* are represented using 6 string-valued fields – *CAT1* through *CAT6*. The median value for the price of each category was chosen uniformly between $0 and $1M. Individual prices within a category were generated using a Gaussian around that median with a standard deviation of $100. Thus, there exists a strong (but not exact) correlation between *Price* and *CATID*. The schema for this dataset is as follows:

ITEMS (CATID, CAT1, CAT2, CAT3, CAT4, CAT5, CAT6, ItemID, Price)

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 correlations. The table consists of approximately 18M rows of 136 bytes each, for a total table size of 2.5GB. The partial schema for this database follows:

LINEITEM (orderkey, partkey, suppkey, ..., shipdate, commitdate, receiptdate, ...)

SDSS Data: Our third source is an extended version of the desktop SDSS skyserver [11] dataset which originally contains 200,000 tuples. We extend it by copying the right ascension (*ra*) and declination (*dec*) windows 10 times in each dimension to produce a 100-fold increase in the dataset size (20M rows, 3GB). The partial schema for this database follows:

PhotoTag (objID, ra, dec, g, rho, ...)

6.2 Results

We now present the results of a variety of experiments. The goal of the experiments is to understand the relative trends in running times for correlated and uncorrelated B+Trees, sequential scans, and CIs as we vary the number of values that we look up and the degree of correlation present.

Experiment 1: Our goal in this experiment is to compare two different clustering schemes using queries over TPC-H data. In the first, the clustered key for the *lineitem* table is *receiptdate* which is correlated with *shipdate*. In the second, we cluster on the primary key of *lineitem* – (*orderkey*, *linenumber*) – which is not correlated with *shipdate*. The query used in this experiment is:

SELECT AVG(*extendedprice* * *discount*) FROM LINEITEM
WHERE *shipdate* IN <list of 1 to 100 random shipdates>

The list of shipdates in this query is chosen randomly from 2500 possible values. Thus, each point on the x-axis corresponds to a different query.

As the graph in Figure 3 shows, the correct choice for the clustered attribute can significantly improve the performance of the secondary B+Tree index. For the uncorrelated case the performance degrades rapidly, reaching the cost of a sequential scan for queries over 4 shipdates. This happens because the query on the uncorrelated attribute selects receiptdate values that are scattered (approximately 7000 per shipdate), so the B+Tree access pattern touches a large fraction of the *lineitem* table. PostgreSQL uses a bitmap index scan to process this query. We have observed the same behavior in other commercial database products as well. This graph also shows that our cost model (Section 3) can accurately predict the performance of unclustered B+Trees in the presence of correlations; we revisit the cost model accuracy in Experiment 3.

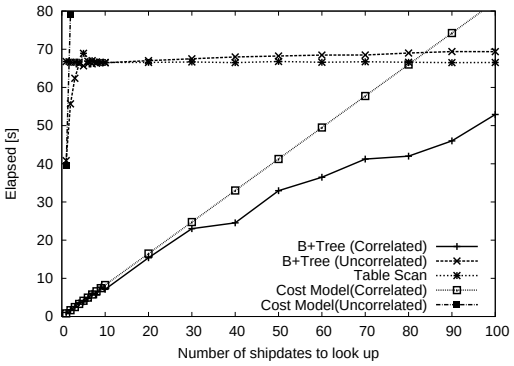

Figure 3: Performance of B+Tree index with a correlated clustered index *(shipdate)* and an uncorrelated clustered index *(orderkey, linenumber)*

Experiment 2: In our next experiment, we explore the performance implications of using a CI instead of a secondary B+Tree on the eBay hierarchical dataset clustered by *CATID*. We pick a bucket size of 4096 tuples per bucket for the *Price* attribute (and we explain this choice in Experiment 4). We use the following query with varying price ranges as indicated below.

SELECT COUNT(DISTINCT *CAT2*) FROM ITEMS
WHERE *Price* BETWEEN 1000 AND 1000+PriceRange

In Figure 4, we omit the points for the full table scan, which takes more than 100 seconds – both the CI and the secondary index outperform it. However, the CI performs somewhat worse than the secondary index. This is explained primarily by the increasing number of extraneous heap pages that the CI needs to read (which are avoided by the bitmap scan since they do not contain the desired unclustered attribute value), as well as the overhead associated to query rewriting. The observation is that even with a simple implementation scheme like query rewriting, the CI is competitive. At the same time, it is three orders of magnitude smaller (the CI is 0.9MB on disk, the secondary B+Tree is 860MB).

Experiment 3: In this experiment, we demonstrate that our cost model based on c_per_u captures actual query costs accurately. The dataset and clustered key used in this experiment are the same as in Experiment 2, but we use a different query that has a predicate on *CAT5*; in other words, we select over a particular subcategory in the fifth level of the eBay product hierarchy. We build a CI on *CAT5*, which is strongly correlated with *CATID*.

SELECT AVG(*Price*) FROM ITEMS WHERE *CAT5*=X

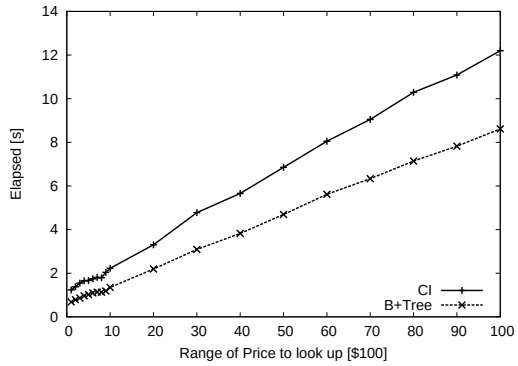

Figure 4: Performance of CI (Query Rewrite) and unclustered B+Tree index for queries over range of *Price*

Here, we tested different values chosen from the *CAT5* category that exhibited different c_per_u counts (ranging from 4 to 145). Our cost model predicts that the CI's performance is primarily determined by how many clustered attribute values the predicated unclustered value corresponds to. As Figure 5 shows, this cost model effectively captures the performance of a CI and PostgreSQL secondary B+Tree with various c_per_u. Also, we observe again that the CI mirrors the performance of the B+Tree quite well at 1/4000 of the size.

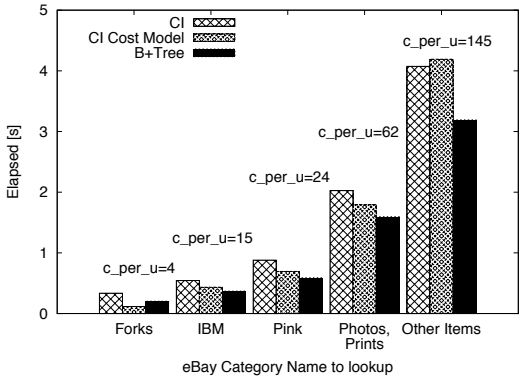

Figure 5: CI cost model based on c_per_u

To highlight the effect of the correlation, we tested the same query after replacing *CATID* with an uncorrelated clustered attribute. As Table 6 shows, the performance of both the B+Tree and CI deteriorates and approaches a table scan, confirming the result of Experiment 1. The c_per_u follows this spike, growing 1,000 times larger. This result shows that our cost estimation based on c_per_u is accurate, and that lower c_per_u values due to choosing correlated clustered attributes dramatically speeds up both secondary B+Trees and CIs.

Table 6: Performance degradation and c_per_u after replacing a correlated clustered attr. → uncorrelated clustered attr.

CAT5 value	c_per_u	B+Tree Runtime[s]	CI Runtime[s]
Forks	4 → 5653	0.2 → 35	0.3 → 62
IBM	15 → 15270	0.3 → 62	0.5 → 67
Pink	24 → 18398	0.5 → 65	0.8 → 67
Photos, Prints	62 → 21004	1.5 → 65	2.0 → 68

Experiment 4: Next, we explore how to optimize over bucketing schemes by balancing the performance of the target query and the size of CI. Since this query may be one of many running on the

system, choosing small CIs that fit in memory and are still effective allows us to create CIs tailored for many different queries and improve overall performance. We again use *CATID* as the clustered attribute, but instead of relying on one fixed bucket layout for the unclustered attribute, we vary the bucket size using the approach presented in Section 5. We run the following query:

SELECT COUNT(DISTINCT *CAT3*) FROM ITEMS
WHERE *Price* BETWEEN 1000 AND 1100

The selectivity of this predicate is 6617 rows out of 43M, or 0.000154. In order to understand how bucketing affects performance and the size of the CI, we vary the bucket size by powers of two. Therefore, a bucket level of 3 indicates that each bucket holds 2^3 unclustered attribute values.

Looking at Figure 6, we see that the CI's performance is nearly the same as that of the B+Tree up to a bucket level of about 13. With no bucketing, the size of the CI is 350MB, which is already smaller than the PostgreSQL secondary B+Tree (850MB). Observe that as we increase the bucket size, the CI size continues to decrease. It is notable that even a CI on a many-valued column like *Price* can become very compact after bucketing.

Figure 6 demonstrates a tradeoff between runtime and size. The lookup runtime grows rapidly after the CI hits a particular bucket size. The intuition behind this critical bucket size is the following: if there are two adjacent buckets in the CI that point to the same set of buckets in the clustered index, doubling the CI bucket size has no effect on *c_per_u*. The key bucket size in this example occurs at $2^{13} = 8192$, which is the number of *Price* values closest to the 6617 selected by the range predicate. Hence, we can see that there is an "ideal" choice for the bucket size that occurs at the knee of the curve.

In order to plot the CI cost model in Figure 6, we applied one refinement to the $cost_{ci}$ expression presented in Section 3.3. In that section, we made the simplifying assumption to disregard the overlap in sets of clustered (*CATID*) values associated to two unclustered (*Price*) values, which causes overestimates in some cases. The overestimate becomes particularly important in this example when evaluating bucket sizes less than 2^{12}. To see why, consider splitting a bucket of size 2^{12} into two buckets of size 2^{11} – the tuples in the smaller buckets will in fact point to no more *CATID* values than before. However, since each of the smaller buckets still points to the *same* values of *CATID*, the cost model double-counts the number of clustered pages to visit and overestimates the runtime by a factor of 2. To avoid overestimating, we observe that splitting a bucket in half can never increase the number of clustered pages visited. Thus, we predict that the actual runtime of a lookup at bucket level i is the minimum of the $cost_{ci}$ value calculated by the cost model and the runtime predicted at bucket level $i + 1$ (in other words, the cost is a non-decreasing function of the bucket level). The values on Figure 6 indicate that the refinement gives us an accurate representation of the actual lookup runtimes.

Experiment 5: For the final experiment, we use the SDSS dataset to demonstrate a situation where composite CIs have an advantage over single-attribute CIs and even B+Tree indices with a real-world dataset and query. The clustered attribute in this dataset is *objID*. The *objID* is correlated strongly with the pair *(ra, dec)*, but the correlation is weaker with each individual attribute. We use the following query, a variant of Q2 from the SDSS benchmark that identifies objects having blue and bright surfaces within some region. [2]

[2]In this query, *g* is a logarithmic measure of blueness and *rho* is a logarithmic measure of brightness.

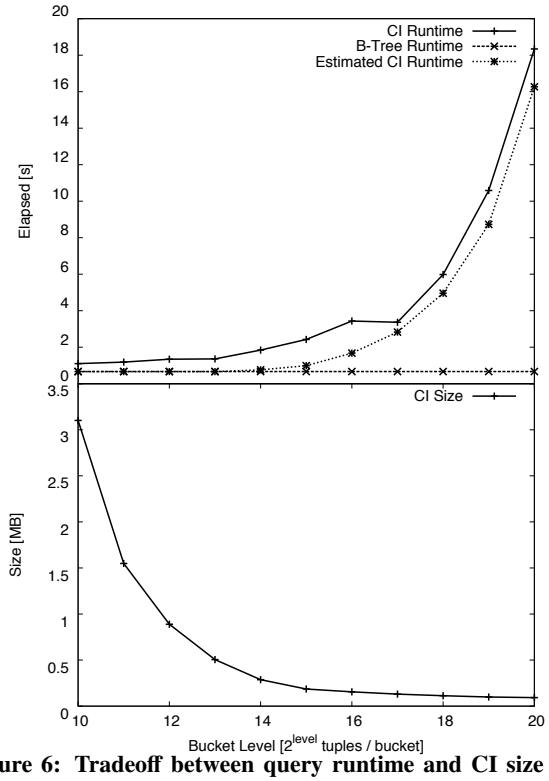

Figure 6: Tradeoff between query runtime and CI size as a function of bucket level. The query selects a range of *Price* values.

SELECT COUNT(*) FROM PhotoTag
WHERE *ra* BETWEEN 193.117 AND 194.517
AND *dec* BETWEEN 1.411 AND 1.555
AND *g* + *rho* BETWEEN 23 AND 25

We choose the columns and bucket sizes for the CI recommended by the CI Advisor. As we can see in Table 7, the composite CI performs much better than a single attribute CI when neither attribute predicts the clustered value but the composition of the attributes does. Both the CI on right ascension and the CI on declination perform worse than the B+Tree index on the pair. However, the CI on the pair of attributes actually performs even better than the B+Tree.

The reason for the composite CI win is that the B+Tree index performs poorly given multiple range predicates. The secondary index will only be used for the range on right ascension, which is the prefix of the compound key. The CI does not suffer from this problem since it is only 693KB and it can be scanned in its entirety in memory, inexpensively. The size of the secondary index on *(ra, dec)*, on the other hand, is 542MB.

Table 7: Single and composite CIs for an SDSS range query

Index	Bucketing	Runtime[s]	Size[MB]
CI(*ra*)	2^{12}	4.0	0.67
CI(*dec*)	2^{14}	1.7	0.936
CI(*ra*, *dec*)	2^{14}(*ra*) 2^{16}(*dec*)	0.21	0.693
B+Tree(*ra*, *dec*)	-	1.12	542

6.3 Summary

We compared the performance of secondary B+Tree lookups, CIs, and sequential scans in PostgreSQL on a variety of different data sets, showing that CIs and B+Trees can both exploit correlated clustered attributes, that our cost model is a good predictor

of performance, and that our CI Advisor can automatically select high performance multi-attribute CIs.

In Experiment 1, we showed that the same secondary B+Tree index structure exhibits dramatically different performance characteristics depending on the level of correlation with the clustered attribute. Using a TPC-H query that predicates on shipdates, we observed a speedup factor of up to 38 when comparing correlated and uncorrelated clustered attributes.

Having established that correlations can have a large impact on the performance of a secondary B+Tree, we compared a B+Tree to a bucketed CI in Experiment 2 on a hierarchical eBay dataset. By decreasing the selectivity of a range predicate over the unclustered attribute, we observed that the CI performs nearly as well as the B+Tree (within a factor of 1.4) while being three orders of magnitude smaller (merely 0.9MB). Thus, it becomes practical to create many of these CIs and to cache them in memory.

Next, we provided a case study for bucketing a many-valued attribute (*Price*) to improve the size of a CI from 350MB to just 0.9MB, while decreasing the performance of lookups by only a factor of 1.2. We illustrate this optimization problem along dimensions of size and runtime, and suggest a refinement to the cost model that predicts the actual runtimes accurately.

Finally, we demonstrated a case where multi-attribute CIs can be used to exploit correlations that are stronger than those available with pairs of any single attributes. As a result, we show in our experiment that a multi-attribute CI performs better than both a secondary B+Tree as well as single-attribute CIs, while remaining three orders of magnitude smaller than the B+Tree.

We also confirmed that our cost model does a reasonable job of predicting the actual performance. We also confirm that the cost model remains accurate for a range of degrees of correlation (as the c_per_u parameter varies).

7. RELATED WORK

One can view our work as an extension of certain optimization approaches from the field of semantic query optimization (SQO); there has been a long history of work in this area [14, 18, 22, 4]. The basic idea is to exploit various types of integrity constraints (often expressed as rules [23, 3, 15, 21, 2, 19, 16]) – 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. [7] describe as one of their optimizations *predicate introduction* (which was originally proposed by Chakravarthy et al [4] and is the same technique we use in rewriting queries), 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; in this case they use the logical constraints (as in Gryz etl al. [12], described below) to identify candidate predicates to insert.

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. [10] have looked extensively at discovering and utilizing "soft constraints" for semantic query optimization. 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 in the face of updates.

8. CONCLUSIONS

In this paper, we showed that is is possible to exploit correlations between attributes in database tables to provide substantially better performance from unclustered database indices than would otherwise be possible. Our techniques exploit correlations by transforming lookups on a CI or standard unclustered B+Tree on the unclustered attribute to lookups in the associated clustered index. 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 built to identify correlated attributes and recommend CIs and bucketings that will provide good performance.

Our experimental results over several different data sets validate the accuracy of our cost model and establish numerous cases where CIs dramatically accelerate lookup times over either unclustered B+Trees (without an appropriate clustered column) or sequential scans. We also showed that CIs are much smaller than conventional unclustered B+Trees, making it possible to maintain a large number of indices to speed up many queries. Based on these results, we conclude that CIs, coupled with our analytical model, have the potential to offer substantial performance improvements to a broad class of index-based queries.

9. APPENDIX:CARDINALITY ESTIMATOR

To choose optimal Cardinality Estimator, we conducted a survey into the literature, prototyped the algorithms and compared their accuracy and performance.

Cardinality Estimation based on Random Sampling is a well studied research area. The area is derived from two larger research areas; Random Sampling and Distinct Values Estimation. Olken's thesis [20], the long thesis more than 150 pages, includes a good overview of Random Sampling. As for Distinct Values Estimation, Bunge's technical report [1] gives a good overview, citing more than 100 papers.

9.1 Random Sampling

Sampling Method affects the accuracy of almost all cardinality estimator algorithms. Following is a list of random sampling methods [20].

1. Sampling with Replacement (WR)
   Sample n tuples, uniformly and independently from R. The sample is a bag (multiset) of tuples from R, as specific tuples could be sampled multiple times.
2. Sampling without Replacement (WoR)
   Sample n ***distinct*** tuples from R, where each successive sample is chosen uniformly from the set of tuples not already sampled. The sample is a set of n ***distinct*** tuples from R.
3. Independent Coin Flips (CF)
   For each tuple in R, choose it for the sample with probability

Table 8: Frequency statistics of distinct values

f_i	The number of attribute values that appear exactly$i$times in the sample.
n	The number of rows in the sample.
N	The total number of rows in the table.
d	The number of distinct values that appear in the sample.
n_j	The number of tuples in the sample with attribute value j.
D	The number of distinct values in the table; *the answer*.
$\overline{D}$	The estimated number of distinct values in the table.

P, independent of other tuples. The sample is a set of X ***distinct*** tuples from R, where X is a random variable with the binomial distribution B(n, P) and has expectation n, which is the desired number of sample rows.

4. Unweighted Sampling and Weighted Sampling
   A weighted sample has a non-uniform probability distribution to choose a tuple from R. If each tuple t has a specified weight w(t), any tuple t is chosen with probability proportional to w(t). Weighted Sampling is critical to some of queries that involve joins.

We used WoR with Unweighted Sampling because we assume only one large fact table.

9.2 Frequency Statistics

All the algorithms collect and use following information to estimate the number of distinct values. Above all, the frequency statistics f is a compact and useful tool used by many algorithms. An example algorithm based on the statistics is

$$\overline{D}_{Chao} = d + \frac{f_1^2}{2f_2}$$

It's easy to calculate this value if we have f. However, this diverges to infinity when $f_2 = 0$, which occurs frequently in various kinds of data.

9.3 HYBSKEW

HYBSKEW [13] is a hybrid estimator that determines algorithm based on two values ***u*** and ***x***.

1. If $u > x$, the data has high skew. ***Shlosser's Estimator*** is employed.
2. If $u \leq x$, the data has low skew. ***Smoothed-Jackknife Estimator*** is employed.

Shlosser's Estimator

$$\overline{D}_{Shloss} = d + \frac{f_1 \sum_{i=1}^{n}(1-q)^i f_i}{\sum_{i=1}^{n} i(1-q)^i f_i} (q \equiv \frac{n}{N})$$

Smoothed Jackknife Estimator

$$\overline{D}_{sjack} = \frac{d+N h_n(\overline{N}) g_{n-1}(\overline{N}) \overline{\gamma}^2(\overline{D_0})}{1-\frac{(N-\overline{N}-n+1)f_1}{nN}}$$
$$\overline{D_0} = \frac{d-\frac{f_1}{n}}{1-\frac{(N-n+1)f_1}{nN}}$$
$$\overline{N} = \frac{N}{\overline{D_0}}$$
$$h_n(\overline{N}) = \frac{\Gamma(N-\overline{N}+1)\Gamma(N-n+1)}{\Gamma(N-\overline{N}-n+1)\Gamma(N+1)}$$
$$g_n(\overline{N}) = \sum_{i=1}^{n} \frac{1}{N-\overline{N}-n+i}$$
$$\overline{\gamma}^2(\overline{D_0}) = \frac{(N-1)\overline{D_0}}{Nn(n-1)} \sum_{i=1}^{n} i(i-1)f_i + \frac{\overline{D_0}}{N} - 1$$

$h_n(\overline{N})$ is expanded as following to avoid overflow:

$$h_n(\overline{N}) = \frac{\Gamma(N-\overline{N}+1)\Gamma(N-n+1)}{\Gamma(N-\overline{N}-n+1)\Gamma(N+1)} = \frac{(N-\overline{N})(N-\overline{N}-1)\cdots(N-\overline{N}-n+1)}{N(N-1)\cdots(N-n+1)}$$
$$= \frac{N-\overline{N}}{N} \frac{N-\overline{N}-1}{N-1} \frac{}{\cdots} \frac{N-\overline{N}-n+1}{N-n+1}$$

Note that Smoothed Jackknife Estimator still can blow up if $d = 1$, that means there is only one distinct value in the sample. In this case, consequently $f_1 = 0$ because the only one distinct values appears a large number of times, not only once. Then,

$$\overline{D_0} = \frac{d-\frac{f_1}{n}}{1-\frac{(N-n+1)f_1}{nN}} = \frac{1-\frac{0}{n}}{1-\frac{(N-n+1)0}{nN}} = 1$$
$$\overline{N} = \frac{N}{D_0} = N$$

Consequently, $g_{n-1}(\overline{N}) = \sum_{i=1}^{n} \frac{1}{N-N-(n-1)+i} = \sum_{i=1}^{n} \frac{1}{i-n+1} = \infty (\because (n-1)-n+1 = 0)$ We used 1 as the answer in this case.

Degree of Skew: u

$$u = \sum_{\{j:n_j>0\}} \frac{(n_j - \overline{n})^2}{\overline{n}} (\overline{n} \equiv \frac{n}{d})$$

As shown above, u is a variance of the number of tuples with each attribute value. If this value is large, it means there are highly frequent values and very rare values. If this value is small, on the other hand, it means many distinct values appear in almost same frequency.

Calculating this value over all j is costing and requires storing all distinct values and their frequency. As we already have a good statistics of distinct values, f, the value is calculated as follows:

$$u = \sum_{\{j:n_j>0\}} \frac{(n_j - \overline{n})^2}{\overline{n}} = \frac{\sum_{\{j:n_j>0\}}(n_j - \overline{n})^2}{\overline{n}}$$

$$= \frac{\sum_i \sum_{j=1}^{fi}(i-\overline{n})^2}{\overline{n}} = \frac{\sum_i f_i(i-\overline{n})^2}{\overline{n}}$$

Chi Square Test for uniformity: x

For $k > 1$ and $0 < \alpha < 1$, let $x_{k-1,\alpha}$ be the unique real number such that if $\chi^2_{k-1} < x_{k-1,\alpha}$ is a random variable having $k-1$ degrees of freedom then $P\{\chi^2_{k-1} < x_{k-1,\alpha}\} = \alpha$. The original paper and papers citing it used 0.975 as α. We used it, too. In short, $x_{n,\alpha}$ is the inverse-chi-square distribution with freedom n and probability α , or $F^{-1}(x, \alpha)$.

9.4 Guaranteed Error Estimator (GEE)

GEE [5] is a cardinality estimator with analytic error guarantee.

$$\overline{D} = \sqrt{\frac{N}{n}} f_1 + \sum_{i=2}^{n} f_i$$

This algorithm is fast and never blows up.

However, this sometimes underestimates the number of distinct values, especially when all values are distinct. For example, Let $n = 30,000$, $r = 10,000$ and $f_1 = 10,000$. That means GEE takes 33% samples and all rows are distinct; so $f_{n>=2} = 0$.

$$\overline{D} = \sqrt{\frac{N}{n}} f_1 = \sqrt{\frac{30,000}{10,000}} 10,000 = 17,320$$

But, the answer is most likely 30,000; all different values. This error can happen when the data has low skew.

9.5 Hybrid Guaranteed Error Estimator (HYBGEE)

Like HYBSKEW, HYBGEE [5] is a hybrid estimator that determines algorithm based on two values ***u*** and ***x***.

1. If $u > x$, the data has high skew. ***GEE*** is employed.
2. If $u \leq x$, the data has low skew. ***Smoothed-Jackknife Estimator*** is employed.

The difference is that HYBGEE uses GEE instead of Shlosser's Estimator because GEE performs better than Shlosser's Estimator when the data has high skew.

9.6 Adaptive Estimator (AE)

AE [5] is a cardinality estimator designed to improve GEE in low-skew data and also to keep GEE's accuracy in high-skew data.

$$\overline{D} = d + m - f_1 - f_2$$

$$m - f_1 - f_2 = f_1 \frac{\sum_{i=3}^{r} e^{-i} f_i + m e^{-\frac{f_1+2f_2}{m}}}{\sum_{i=3}^{r} i e^{-i} f_i + (f_1 + 2f_2) e^{-\frac{f_1+2f_2}{m}}}$$

$$d \le \overline{D} \le n(SanityBoundary)$$

Static AE

If $f_1 = 0$, the right side is 0, then $m = f_2$. So, $\overline{D} = d$ Because the idea behind AE is that low frequency values contribute most, AE determines that no new distinct values will appear.

Simple AE

If $f_i = 0$ for all $i > 2$,

$$m - f_1 - f_2 = \frac{m f_1}{f_1 + 2f_2}$$
$$\therefore m = \frac{\frac{(f_1+f_2)(f_1+2f_2)}{2f_2}}{}$$
$$\therefore \overline{D} = d + \frac{(f_1+f_2)(f_1+2f_2)}{2f_2} - f_1 - f_2$$

When $f_2 = 0$, following the sanity boundary, $\overline{D} = n$.

Static AE and Simple AE are much faster than calculating Full AE equation below.

Full AE

Because the equation has both linear part and exponential part, there is no analytic solution. We calculated m by a numerical approach, the classic Newton-Raphson method which repeatedly applies $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$.

Let m_n be the value of m after n iterations and let $F(m)$ be the function to be calculated.

$$A = f_1 + f_2, B = \sum_{i=3}^{r} e^{-i} f_i, C = \sum_{i=3}^{r} i e^{-i} f_i, D = f_1 + 2f_2$$

$$m - A = f_1 \frac{B + m e^{-\frac{D}{m}}}{C + D e^{-\frac{D}{m}}} \Leftrightarrow (m - A)(C + D e^{-\frac{D}{m}}) - B f_1 - f_1 m e^{-\frac{D}{m}} = 0$$

Let the left side be $F(m)$.

$$F(m) = (m - A)(C + D e^{-\frac{D}{m}}) - B f_1 - f_1 m e^{-\frac{D}{m}}$$
$$F'(m) = C + D e^{-\frac{D}{m}} + (m - A)(D \frac{D}{m^2} e^{-\frac{D}{m}}) - f_1 (e^{-\frac{D}{m}} + m \frac{D}{m^2} e^{-\frac{D}{m}})$$
$$= C + D e^{-\frac{D}{m}} + (m - A)(\frac{D^2}{m^2} e^{-\frac{D}{m}}) - (1 + \frac{D}{m}) f_1 e^{-\frac{D}{m}}$$
$$= C + e^{-\frac{D}{m}} (D + (m - A)\frac{D^2}{m^2} - (1 + \frac{D}{m}) f_1)$$
$$= C + e^{-\frac{D}{m}} (D - f_1 + \frac{D(D - f_1)}{m} - \frac{A D^2}{m^2})$$

Because the equation has continuous nature, we used Simple AE to get the initial value m_0. We used 50 as maximum iteration times and observed that it's enough to converge to precision under 1. However, AE sometimes blows up when the derivation gets to 0. In that case, the sanity check gives n as the answer and in most cases it's correct; such situations occur especially when almost all values are distinct.

9.7 Accuracy Comparison

We conducted a series of experiments with synthesized data to observe each estimator's behavior in various skews. We used the generalized zipfian distribution for generating 1 million tuples with skew parameter from 0 to 4. The actual number of distinct values is configured to be around 10000, but fewer with higher skew value because 1 million tuples are not enough to generate 10000 distinct values with high skew.

Figure 7 to Figure 10 shows the part of the result with skew value 0 to 2. The x-axis is the ratio of tuples sampled from the table. The y-axis is the error ratio defined by:

$$ErrorRatio \equiv \frac{\left|\overline{D} - D\right|}{D}$$

Usually, the error ratio decreases as more tuples are sampled.

As we can see, the most precise algorithm varies on skew values. There is no single algorithm that has the best accuracy for all skew values. However, AE is the most stable and generally precise estimator that might be occasionally outperformed by other estimators but never produces a big error at least for generalized zipfian distribution with skews between 0 and 1.5 which covers most cases. This is why we decided to use AE in this paper.

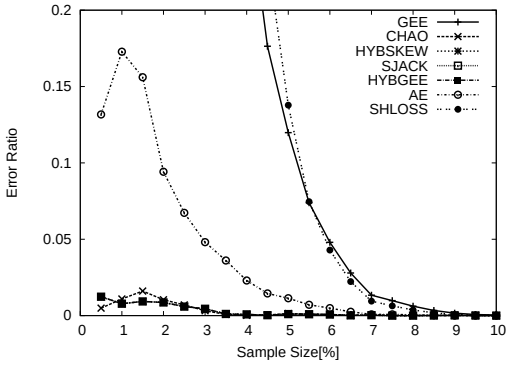

Figure 7: Accuracy Comparison with Zipfian skew=0.0

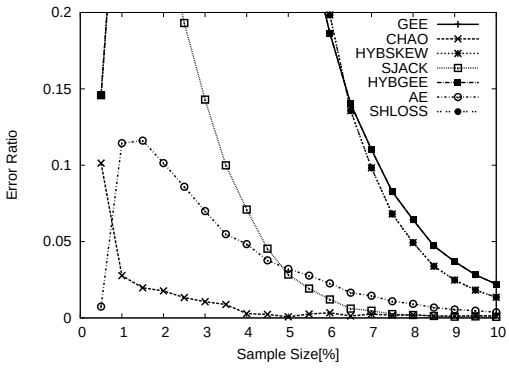

Figure 8: Accuracy Comparison with Zipfian skew=0.5

9.8 Overhead Comparison

We also conducted experiments to test overhead of each algorithm. Table 9 shows the total computation time of each algorithm in millisecond for SSBM lineorder table Scale 1 (600MB) and Scale 10 (6GB), which has 17 columns and 136 pairs of columns. We did 3 experiments for each cell and took average of them.

Because all algorithms do random access on data file, the size of data file doesn't affect. Instead, computation time is exactly proportional to the size of sample data. We concluded that though AE has to solve additional equations by newton method, the overhead is negligible.

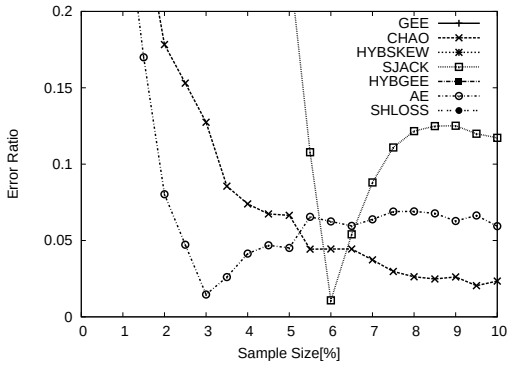

Figure 9: Accuracy Comparison with Zipfian skew=1.0

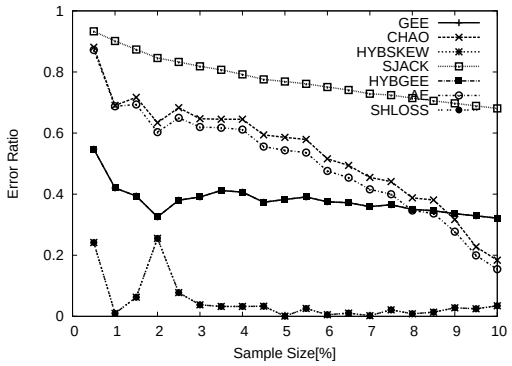

Figure 10: Accuracy Comparison with Zipfian skew=2.0

Table 9: Computation Time of Cardinality Estimator

Samples	File Size	GEE	CHAO	HYBSKEW	HYBGEE	AE
6178	600MB	422	372	425	421	356
6028	6GB	412	358	417	408	363
11600	600MB	818	684	820	824	692
11321	6GB	752	631	753	749	620
54922	600MB	3925	4132	4689	4967	4120
53592	6GB	3990	4420	4894	4191	4062

10. REFERENCES

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

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

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

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

[5] M. Charikar, S. Chaudhuri, R. Motwani, and V. R. Narasayya. Towards estimation error guarantees for distinct values. In *PODS*, 2000.

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

[7] Q. Cheng, J. Gryz, F. Koo, T. Y. C. Leung, L. Liu, X. Qian, and K. B. Schiefer. Implementation of two semantic query optimization techniques in the DB2 universal database. In *VLDB*, 1999.

[8] eBay Developer API. http://developer.ebay.com/products/trading, 2008.

[9] P. B. Gibbons. Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *VLDB*, 2001.

[10] P. Godfrey, J. Gryz, and C. Zuzarte. Exploiting constraint-like data characterizations in query optimization. In *SIGMOD*, 2001.

[11] J. Gray, D. Slutz, A. Szalay, A. Thakar, J. vandenBerg, P. Kunszt, and C. Stoughton. Data mining the sdss skyserver database, 2002.

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

[13] P. J. Haas, J. F. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *VLDB '95: Proceedings of the 21th International Conference on Very Large Data Bases*, pages 311–322, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[14] M. Hammer and S. Zdonik. Knowledge based query processing. In *VLDB*, 1980.

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

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

[17] G. Huo, H. Kimura, A. Rasin, S. Madden, and S. B. Zdonik. Correlation indices: Exploiting correlated attributes using secondary indices. In *VLDB*. ACM, 2008. In Submission.

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

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

[20] F. Olken and D. Rotem. Random sampling from databases - a survey, 1995.

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

[22] S. T. Shenoy and Z. M. Ozsoyoglu. A system for semantic query optimization. In *SIGMOD*, 1987.

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