Computer Sciences Department

RDBMS Index Support for Sparse Data Sets

Jennifer Beckmann
Eric Chu
Jeffrey Naughton

Technical Report #1566

July 2006

UNIVERSITY OF WISCONSIN MADISON

RDBMS Index Support for Sparse Data Sets

Jennifer L. Beckmann Eric Chu Jeffrey F. Naughton
{jbeckmann, ericc, naughton}@cs.wisc.edu

University of Wisconsin-Madison
1210 W. Dayton St.
Madison, WI 53706 USA

Abstract

Maintenance costs and storage overheads incurred by indexes often limit the number of indexes created per table in an RDBMS. For sparse data, where a table may have hundreds of attributes, indexing only a few attributes means that a vanishingly small percentage of attributes will have indexes, which unfortunately means that a table scan is the only evaluation plan for almost all selection queries on that table. This paper demonstrates that sparsity of the data actually enables index support for most, if not all, attributes in the data. Our approach leverages "sparse indexes," which are partial indexes that store only non-*null* values. Sparse indexes incur low maintenance costs and storage overheads because most values in a sparse table are *null*. Properties of the data lead us to two other contributions toward index support for sparse data: we show that sparse indexes benefit greatly from building all indexes in one-pass of the data; and we identify that multi-column sparse indexes are preferable as covering indexes when attributes in the data are correlated. We qualitatively evaluate our approaches with synthetic and real-world data to show that our suggestions significantly outperform traditional indexing approaches designed for dense data.

Proceedings of the 32nd VLDB Conference, Seoul, Korea, 2006

1 Introduction

"Sparse" data sets arise in applications that allow objects to have values assigned to any subset of attributes. These data sets there often have hundreds (or thousands) of attributes, but each object has values assigned to only a handful of the total attributes. As examples, for e-commerce data sets where companies define idiosyncratic attributes for products, Agrawal [1] cites a source with over 5,000 attributes, and we encountered a data set with over 2,000 attributes where most of the objects have values assigned only to five attributes. In another example, Pyle [9] describes a demographic data set set with nearly 700 attributes, half of which are null in 98% of the objects. Sparse data is usually queried over as a single table in an RDBMS because the objects often have values defined for similar sets of attributes and the queries search over all objects. Since the common wisdom is that it is expensive to build many indexes on a table, the large number of attributes in the data gives the impression that it is impossible to provide good index coverage and efficiently support queries over the data.

In this paper we demonstrate that for sparse data it is possible to index many more attributes than one might think. The key observation that makes it possible falls directly from the definition of sparse data—there are many more *null* values in the data than non-*null* values. We leverage sparse indexes to show that indexes can be built on many (if not all) sparse attributes in a data set. Unlike a *full* index that maps both non-*null* and *null* values to object identifiers (ids), a *sparse* index maps only the non-*null* values to ids. Sparse indexes are supported by at least one commercial RDBMS, but to our knowledge their application to sparse data has never been evaluated in the published literature. We show that a sparse index over a sparse attribute is much smaller than its full index counterpart, while retaining most of the capabilities of the full version. Sparse indexes incur a much lower maintenance cost than full indexes, because any operation that changes data in a table, such as inserting, deleting, and updating a tuple, will only affect indexes

on attributes that have non-*null* data.

The unique properties of sparse data enable us to make three other observations that aid in providing index support for this type of challenging data set. First, we observe that sparse indexes are rather small and are fast to build for sparse data. The traditional approach to building indexes scans a table for each index. Multiple scans, though not ideal for dense data, are usually acceptable for dense data because they have few indexes and the bulk of the time is spent in building the index. Since sparse indexes are so small and sparse data requires hundreds of indexes, multiple scans are highly inefficient for building the indexes. Thus, we demonstrate that minimizing passes while building indexes is effective and essential when indexing sparse data.

Second, the sparsity of the data allows better indexing support for queries over text data. We conjecture that the "IS NOT NULL" predicate is particularly useful when querying sparse data. This type of predicate selects rows of a table that are non-*null* for an attribute, regardless of the actual value for the attribute. Such a query over a sparse data set will return only a fraction of the total number of rows (usually less than 10%; the common rule of thumb for choosing an index access path [11]). Inverted indexes are usually used to index sparse text data and are not capable of efficiently answering an 'IS NOT NULL" predicate. Therefore, in addition to an inverted index, we propose to create for a text attribute an "is-defined" index, which records the rows with non-*null* values for the attributes. It complements an inverted index to provide support for *null*-oriented queries over text attributes.

Our third observation identifies that attributes within sparse data can be related, and we extend our study to multi-column indexes. Sparse data can contain certain pairs of attributes that appear non-*null* together in rows, and we call these pairs correlated attributes. In contrast, uncorrelated attribute pairs seldom appear non-*null* together within objects. Our analysis compares the query performance of two-column indexes with single-column indexes. Single-column index plans use set-based operations, such as intersection, to evaluate complex predicates. Multi-column indexes simply lookup the matching rows. The set operations are more costly to execute than using a composite index, regardless of correlation, but uncorrelated attributes result in larger multi-column indexes and have higher evaluation costs in some cases. Covering indexes for correlated attributes are especially effective for projection queries that match the index.

Finally, we experiment with our techniques using a commercial database. Our experiments include results from a synthetic data set and an analysis of a real-world e-commerce data set, which validate our main contributions that:

- It is feasible to index many, if not all, attributes in a sparse data set.
- Sparse indexes over sparse data are small and fast to build, thus one-pass index construction has a large benefit for creating many such indexes.
- "Is-defined" indexes are important for supporting "IS NOT NULL" predicates over text attributes in sparse data.
- Multi-column indexes have more benefit for attributes that appear non-*null* together and are less effective for attributes that seldom appear non-*null* together.

We present a motivating example from e-commerce in the next section and single-column indexing techniques in Section 3. Multi-column indexes and their implications are presented in Section 4. We demonstrate the viability of our techniques with synthetic and real-world data in Section 5. We review related work in Section 6 and conclude in Section 7.

2 Motivating Example

A typical example of a sparse data set is an e-commerce product catalog. Such catalogs allow users to search one source for product specifications from many manufacturers. For example, the electronics catalog CNET [6] has product information for categories such as mp3 players, cell phones, computers, and so forth. The product specification for an Apple iPod lists the *audio-formats* as *Mp3, ACC, Apple Lossless*, and so forth along with 55 other attributes with values. Many e-commerce product catalogs, including CNET, are sparse [3] because one schema is not appropriate for all products. In fact, the data can be sparse even within the same product category.

E-commerce product catalogs are difficult to manage, because products can have a non-*null* value for any attribute. One might consider partitioning the attributes into separate tables in order to get dense subsets of the data. However, even when a set of products has non-*null* values for a shared set of attributes, the individual products may define non-*null* values for attributes outside the shared set of attributes. Thus, it is difficult to cleanly partition the attributes into separate tables.

Figure 1 shows a product specification table that we will use as an example throughout this paper. The table has dense attributes appearing first followed by the sparse ones. The rows are designated by product type, which are contained in the category column. Our study concerns the sparse data of the table.

Relational-style querying of sparse attributes in catalogs can be a powerful product discovery tool. Users can search across vendors and categories to find a wide variety of products. We consider the performance of queries over the sparse attributes and, in doing so, define the following types of queries that a database should answer efficiently:

Product Table

pid	Category	Name	Mftr	Audio Formats	Recharge Time (h)	Radio	Prvdr	Talk Time (min)	RAM (MB)	...
1	Mp3 player	Zen	Creative	Mp3, WMA, ...	2.5	FM				
2	Mp3 player	iPod	Apple	Mp3, ACC,...	4					
3	Mp3 player	Sansa	SanDisk	Mp3, DRM, ...		FM				
4	Cell Phone	CDM-8940	Audiovox	Mp3			Verizon	240		
5	Cell Phone	6030	Nokia			FM	Cingular	180		
6	Desktop	XPS 200	Dell						512	
7	DVD Player	DVR-533H-S	Pioneer	Mp3, WMA, ...						

Dense Attributes — Sparse Attributes

Figure 1: **An e-commerce electronics product table.**

T1 *Which products support the mp3 audio format?*

T2 *Which products support the mp3 audio format and have at least 4 hours of recharge time?*

T3 *Which products support the mp3 audio format and do not have a radio tuner (radio is null)?*

3 Sparse Indexes

We contend that it is feasible to build indexes on many sparse attributes of a sparse data set. Our proposal may seem impractical at first thought because a sparse table typically has hundreds or even thousands of attributes. Indexing so many attributes in a single table is largely frowned upon in practice for two main reasons: the large storage overheads and the high maintenance costs that indexes incur. Although these reasons are valid for indexes over a dense data set, in this and the following sections, we show that, for a sparse data set, we can keep these overheads surprisingly low by using sparse indexes.

In this paper, we assume that queries are over a data stored in a horizontal schema. A horizontal schema is by far the most familiar way to view and query data and our example product table in Figure 1 is in this format. Past research has explored the storage representations for sparse data. Agrawal et al. [1] considered horizontal and vertical approaches and concluded, among the alternatives available to them at the time, that vertical storage was best. Since that paper was written, improvements in *null* storage in commercial RDBMSs have moved the tradeoff between horizontal and vertical formats so that the vertical format is no longer "uniformly" better than the horizontal format. This observation is especially true for queries that project more than a handful of attributes, such as our three query types outlined in the previous section. Furthermore, with storage studied by Beckmann et al. [4], the horizontal schema outperforms the vertical approach for sparse data sets.

The rest of this section begins with a definition of sparse indexes. Our discussion classifies attributes into text-based and non-text-based scalar attributes, and focuses on the implications for sparse B-tree and inverted indexes.

3.1 Definition

A *sparse index* over an attribute A_n is an index that includes only the non-*null* values in A_n. In comparison, a *full index* covers both *null* and non-*null* values. Sparse indexes are important in our discussion because they have a small index size and low index maintenance costs for sparse data. A full index would include all rows of a table regardless of *null* or non-*null* value. The *null* values in full indexes require the index to be maintained for each tuple insert or delete. In contrast, a sparse index only needs updated when the attribute in the index is non-*null*.

A partial index [14] is an index that contains a proper subset of table rows that satisfy a conditional expression, called the predicate of the index. Partial indexes can be used to avoid indexing common values in a table and a sparse index can be defined as a partial index. The partial index definition of a sparse index a column An in a relation H is

```
CREATE INDEX An_sparse_index ON H(An)
       WHERE An is not NULL
```

The index definition disallows any row that is *null* for the attribute. One way to think about a sparse index is that it takes a sparse column of the larger table and creates an index over a logical projection view that contains no *nulls*.

Although sparse indexes can be defined as partial indexes, using generic partial indexes may not be the most efficient implementation for a sparse index. A partial index implementation requires a system to add index maintenance and query optimization techniques for arbitrary conditional expressions. For maintenance tasks, such as a tuple insert, the system must evaluate each expression to determine which indexes need updates. If one has hundreds of partial indexes, then there are hundreds of predicate evaluations. The query optimizer must also match the conditional expressions to arbitrary query predicates to choose an appropriate access path.

Sparse indexes simplify the index maintenance process of partial indexes, because they consider the specific predicate of whether an attribute is *null* or not. A lookup of the non-*null* attributes of a tuple determines which indexes need maintenance. Sparse indexes also simplify the index selection tasks for optimizers, be-

cause the indexes apply to any predicate except the IS NULL predicate.

3.2 Scalar Attributes

By scalar attributes, we mean non-text attributes such as integers, doubles, dates, and possibly atomic character strings. The Recharge-time, Radio, RAM, Talk-time, and Provider attributes in the example product catalog are scalar attributes. Scalar attributes can either use hash-based or tree-based indexes. Sparse indexes apply to both types of indexes, but we address B-tree indexes only in this paper.

A sparse B-tree index retains most capabilities of a full B-tree index. Both can be used as an access path for a query with a non-*null* predicate over a column. Consider the query

```
Q1: SELECT * FROM Products WHERE Recharge-time = 4
```

If the value 4 occurs with the Recharge-time attribute, then the sparse index will contain the value and a query plan can use the index to retrieve the matching rows.

One difference between sparse indexes and full indexes is that sparse indexes cannot directly answer "is-*null*" queries, which are queries that contain predicates of the form "attr IS NULL." For instance, query Q2 requests products that do not have a radio, but have 4 hours of Recharge-time.

```
Q2: SELECT * FROM Products
    WHERE Recharge-time = 4 AND Radio is null
```

If there are indexes on each of these attributes, then an index-only query plan will find the set of rows that contain Recharge-time = 4 and use set difference to remove the rows that contain a non-*null* value for Radio. The query demonstrates that systems need to support index set difference and use it in query optimization.

3.3 Text Attributes

Text types may appear in sparse data sets in the form of lists or short descriptions, such as a comment attribute. The attribute Audio Formats in Figure 1 is a text attribute that lists the digital audio formats that the product plays or records. Databases treat text as a bag (multiset) of tokens and allow queries that search for tokens in the text. Inverted indexes help to support fast retrieval of rows that contain a query term. After tokenizing the text data for an attribute, an inverted index stores each token and the row ids where the token appears. Inverted indexes do not store *null* values because a *null* value does not contain any tokens.

One limitation of inverted indexes for sparse data is that the indexes can not answer is-*null* and is-not-*null* queries efficiently. Consider the following query over a text-attribute:

```
Q3: SELECT * FROM Products
    WHERE Audio-Format is not null
```

An inverted index over text does not support the query. The execution engine, if forced to comply, would have to scan the entire posting file in the index. The reason is that to find all rows that are not *null*, a system would scan the inverted list of each token in the lexicon to discover all of the ids in the index.

Therefore, in addition to an inverted index, we propose to create for a text attribute an "is-defined" index, which records the ids of records that have non-*null* values in the attribute. An is-defined index is suitable for a sparse attribute because its size reflects the small portion of defined data. It complements an inverted index to provide support for *null*-oriented queries over text attributes. A non-text attribute, however, does not need an is-defined index because a sparse B-tree index already captures the same information.

Algorithm 1 One-pass Index Build

```
INPUT: Table T with schema S(c_1, ..., c_n)
OUTPUT: Indexes c_1index, ..., c_nindex
for all tuple (t_1, ..., t_n) in T do
  if t_i is not null then
    INSERT (t_i, id) INTO c_iindex
  end if
end for
for all index c_i in S do
  BUILD INDEX c_iindex
end for
```

3.4 Creation and Bulk-loading

Indexes can be built over a data set in two primary ways: bulk-loading or individual multiple inserts. In general, using individual inserts is an expensive operation that leads to inefficient storage. Therefore, many database systems support bulk-loading, which creates an index over the rows of a table in a single operation. A key question during bulk loading is whether a database should create indexes on a table by scanning the data set once for each index or use one scan to create many indexes. In general, if a table requires multiple non-clustered indexes, such as what we are suggesting, a database system will scan the table once for bulk-loading each non-clustered index [8]. This approach clearly presents a problem to creating hundreds of indexes on a table.

Scanning a table as few times as possible for bulk-loading multiple sparse indexes provides much better scalability. Algorithm 1 outlines a technique to create multiple indexes in one pass of the base table. In the one pass over the table, temporary tables store non-*null* data entries for indexes. In a second pass over the temporary tables, the algorithm builds the indexes from the temporary tables. The single pass algorithm is limited in the number of indexes it can build only by the number of temporary output tables it can use. Thus, the scalability of the algorithm depends on the number of columns in the data and not the on the number of rows in the data. The efficiency of the second phase depends on the size of the c_i index, as we discuss in the next paragraph.

The one-pass algorithm greatly reduces the time spent on IO during bulk-loading. Although this technique applies to both sparse and full indexes, the saving is especially significant for sparse indexes because after scanning for the data entries, the cost of creating a sparse index is much lower than that of a dense index. The reason is that the amount of time to build an index is proportional to the number of data entries in the index. A full index has the same number of entries as the number of rows in the table, whereas a sparse index has the same number of entries as the number of non-*null* values, which is about 1% of the number of rows. The cost of building a sparse index on a sparse attribute is then roughly $\frac{1}{100}$ the cost of building a full index. As a result, the saving by the one-pass algorithm is a much larger fraction of the total indexing time for a sparse index than it is for a full index.

4 Multi-column Sparse Indexes

Section 2 explains that sparse data cannot be neatly partitioned into a set of dense tables. The reason is that rows in the table can define a non-*null* value in any attribute and grouping rows together increases the number of attributes for the group. In the e-commerce example in Figure 1, the cell phone products have common attributes `Provider` and `Talk-time`; however, the rows also include special-case attributes `Radio` and `Audio-formats`.

Although one cannot partition the data into a set of tables that are dense, the attributes in the data may have some correlation. In fact, the attributes in the e-commerce example have some structure. The attributes `Provider` and `Talk-time` are defined for all cell phones, but not for any other product types. The structure in sparse data possibly results from someone suggesting a schema for product type and most products of that type following the suggestion. The data becomes sparse and unstructured when a suggested attribute is not appropriate for a product (a Mp3 Player that uses disposable batteries does not have a recharge time), or the schema needs an extension to handle unusual product features (a cell phone that plays mp3s). Such ad hoc schemas results in attributes that are strongly correlated with attributes of the same kind of products, but those attributes are largely uncorrelated with the rest of the attributes in the schema.

The relationships among sparse attributes suggest that composite indexes may provide better performance than single-column indexes while still allowing reasonable maintenance costs. For instance, the following type of query would benefit from a composite index over the attributes `Provider` and `Talk-Time`.

T4 *What is the (non-null) talk time for the products that use Verizon as a Provider?*

A query plan for T4 uses the covering composite index to select products with Verizon and project the `Talk-time` attribute. The same query using a single-column index plan uses the `Provider` index to select products with Verizon and joins the selected rows with the index on `Talk-Time`. Although the index-join plan might be more efficient than a table scan, the composite index will have the best performance for this query.

The benefits of composite indexes over single-column indexes, however, depend largely on the distribution of *null* values within the columns being indexed. For example, a composite index over the attributes `Provider` and `RAM` in Figure 1 provides very little advantage over single-column indexes because `Provider` and `RAM` are never non-*null* together.

4.1 Definition

We extend the definition of a sparse index to include multi-column indexes by using a partial index whose predicate selects rows that have at least one non-*null* value in any of the index keys. Note that composite indexes are restricted to scalar attributes because text attributes have multi-set semantics and composite indexes with text attributes make little sense. The partial index definition of a sparse index over the scalar columns `An, ..., Am` in a relation `H` is

```
CREATE INDEX sparse_index ON H(An, ..., Am)
    WHERE NOT (An is null AND ... AND Am is null)
```

The index predicate disallows any row that contains *null* values for all attributes in the index key. Similarly to single-column sparse indexes, a composite sparse index has the functionality of its full index counterpart for all queries except for those predicates with `IS NULL`. Composite indexes are different from single-column sparse indexes, in that *null* values may appear in the index when a search key attribute is non-*null* while another search key attribute is *null*.

The number of *nulls* in an index will depend on the distribution of non-*null* values among the attributes in the index key. By distribution we mean the location of the non-*null* values in the rows of the relation. Consider the attributes in the e-commerce product table in Figure 1. A two-column composite index over any two attributes will result in different *null* distributions in the resulting index. For example, an index over `Provider` and `Talk-time` attributes will not have any *nulls*, whereas an index over `Provider` and `RAM` will have three *nulls*—a *null* in all rows that are in the index.

Different non-*null* distributions can result in four basic patterns of *nulls* and non-*nulls* in a composite sparse index. Figure 2 illustrates the patterns by indicating non-*null* values in black and *null* values in white. The first illustrations shows a containment pattern, where the `B` attribute, when non-*null*, appears with a non-*null* `A` value. In the e-commerce example, the `Audio-format` and `Recharge-Time` attributes have the containment property. The correlated pattern is like the `Provider` and `Talk-time` attributes

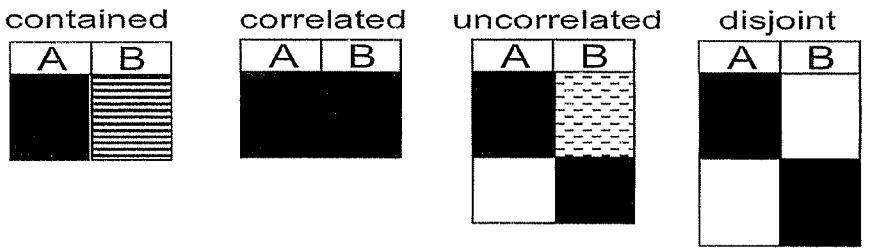


Figure 2: **Four possible distributions in composite sparse indexes.**

where whenever one attribute is non-*null* the other is always also non-*null*. An uncorrelated pattern occurs when one attribute is usually *null* with the other, but sometimes is non-*null* (`Provider` and `Radio`). Finally, the disjoint property happens when neither attribute is non-*null* together, like `Provider` and `RAM`.

In real-world sparse data, the distributions of *nulls* within indexes usually falls between the two extremes of correlated and disjoint. The amount of correlation between two attributes has an affect on the number of rows in a composite index. Consider two attributes that each have n number of non-*null* values. A truly correlated distribution will index n rows and a fully disjoint distribution will index $2n$ rows. An uncorrelated pair will fall between these two extremes. A distribution that is more correlated will have fewer indexed rows and, consequently, fewer index maintenance tasks and faster scan performance if the index is used as a covering index.

5 Experimental Evaluation

In this section, we report upon experiments we ran on a commercial database system. The database ran on Windows 2003 Server with a 2.4 GHz Intel Pentium processor, two physical disks and 1GB of physical memory. Data was stored on a raw disk partition. Logs were stored on the other physical disk, with logging set to minimal. The buffer pool size was set to 64MB and the maximum amount of memory allowed during index creation was 1024KB. We ran our experiments with both a hot and a cold buffer pool. In cases where a warm buffer pool did not provide any additional insight, we only report on the cold buffer pool performance. We report all experiments with a 95% confidence interval over at least 5 runs of the operation.

5.1 Data Sets

Our experiments focus on index performance on synthetic data. Simulated data helps us to isolate performance to the issues of sparsity and control the selectivity of query predicates. We model our synthetic data on properties of a real world data set that we collected from the Internet company, CNET. Throughout our experiments, we discuss the practical implications of our results in the context of our CNET data, which we describe next.

5.1.1 Real Data

CNET Networks, Inc. is a company that provides a commercial e-commerce website with detailed product information for software, computer systems, and other technologies. With permission from CNET, we collected all of the product specs from the catalog as of March 2005 [6]. The catalog contains 233,304 products and 142,567 have product specifications that define a subset of 2854 attributes. A majority of the attributes are very sparse and are undefined in more than 99% of the products with specifications. The average number of attributes in a product is eleven and the mode is five. More details on the data collection and statistics of the dataset can be found in [3].

5.1.2 Synthetic Data

We use synthetic data in two ways: the first is in experiments with index maintenance and the other is in experiments with selection predicates. Both evaluations use an initial table with 136K rows and on average five non-*null* values distributed over 204 integer attributes per row. The size of the table on disk is 118 MB.

For building and maintenance evaluation, synthetic data needs to show: (1) how the number of sparse indexes on a table scales to large numbers of indexes, and (2) how the performance of inserts are affected by the density of a row in the data. The first evaluation uses this table in an experiment that builds indexes over the table and in another experiment measures insert time for rows with 5 non-*null* values per row. The second evaluation considers insert performance into the table for rows with various non-*null* density. The position of the values in the row are random.

Our evaluation of selection predicates requires synthetic data that has characteristics described in Section 4. Twelve columns in the table have varying overlap between them. We created one attribute as the primary attribute with 2000 non-*null* values that are randomly distributed throughout the range of `ids` in the table. The primary attribute has cardinality one. The other 11 columns overlap with the `ids` of the primary attribute from 0% to 100% in increments of 10%. Thus, the attribute with 0% overlap has a disjoint pattern and the one with 100% overlap has a correlated pattern. The cardinality of the overlapping columns is five.

5.2 Implementation

The system that we use in our experiments stores *null* values in its indexes. The advantage of using a system that supports full indexes is that it provides a fair evaluation of full indexes. One commercial system all indexes are 'sparse' in that they do not store *null* values, but using this system would require us to fake full indexes with some "special" value designated as *null* in the data. This approach, however favors sparse indexes and may unfairly disadvantage the full index approach, because systems have special handling for *null* values, such as ensuring the value sorts high in the index.

Our approach emulates sparse indexes within the system, as described in Section 5.2.1, and puts the sparse index at a slight disadvantage to the full index. Since our conclusion is that sparse indexes are superior to traditional indexes for our workloads, the disadvantage only strengthens our conclusions.

5.2.1 Indirect Indexes

We stored a sparse data set in a wide table, which we refer to as the "base table." The schema for the integer data is

```
H(id     INTEGER NOT NULL,
  A1     INTEGER,
  A2     INTEGER,
  ...
  A204   INTEGER,
  PRIMARY KEY(id))
```

For the database that we used, the primary key constraint on `id` clusters the table in a B-tree organized file sorted by `id`. Therefore, B-tree indexes on the rest of the columns are non-clustered. A non-clustered B-tree index over a column in `H` stores the values of the column as (column value, primary key id) pairs. A data lookup with this index finds the matching `(value, id)` pairs in the index, extracts the `ids`, and then retrieves the tuples by the `ids` with the primary key clustered index on `H`.

We use an indirect method to create sparse indexes and define an "index-table" for each attribute that we want to index. An index-table acts as a sparse index by storing the projection on the non-*null* values of a column along with the corresponding `ids` from the base relation `H`. The index-tables have the schema

```
A1_Index(value  INTEGER NOT NULL,
         id     INTEGER NOT NULL
         PRIMARY KEY(value, id))
```

Each index-table has a clustered B-tree index on `(value, id)`.

The database does not recognize index-tables as sparse indexes. Therefore, we force the database to use these tables as an access method to the base relation, by using the index-tables in the queries instead. For instance, we would replace the query Q2 with the following query

```
SELECT H.*
FROM ((SELECT id FROM Recharge-time_Index
       WHERE value = 4)
       EXCEPT
       (SELECT id FROM Radio_Index
       WHERE value is not null)) I, H
WHERE  H.id = I.id
```

We use SQL set-based operators INTERSECT and EXCEPT to construct index-based plans. For the query above, an execution engine first differences the index-tables to find the matching `ids` for the predicate. Next, a join between the result of the intersection and the base table fetches the matching tuples from the base table. Note that for the query optimizer to use the correct plan, we need to impose unique and foreign key constraints on the `id` column of the index-tables so that the index-tables refer back to the base table.

Our indirect approach of creating sparse indexes as tables suffers some disadvantages in comparison to the native implementation of full indexes. First, our indirect implementation incurs a higher maintenance cost, which includes the cost of maintaining the base table and the cost of maintaining the index-tables separately. Second, index lookups to the base table require a join from the index-table to the base table. However, as we discussed in the introduction to this section and revisit in Section 5.4, even with these disadvantages, our indirect implementation of sparse indexes often performs the same or better than the native full B-tree indexes on queries over sparse data.

In order to gauge the performance disadvantages of the indirect approach, we also implemented full indexes through the use of index-tables. The implementation of indirect full indexes is exactly the same as sparse indexes described above, except that the index-table stores all `ids` and allows *nulls* in the `value` column.

Finally, a fill factor for an index indicates the percentage of each page in the index that is reserved for future data insertions [10]. The fill factor affects the number of page splits that happen during inserts into the index. All of the indexes in our experiments have an 80% fill factor, which leaves 20% of each leaf-level page for future insertions. This fill factor insures that our experiments have minimal effects from index page splits.

5.3 Creation and Maintenance

In this section, we consider the performance of creating many indexes on our synthetic data and the performance of inserting tuples. We also ran experiments for deleting tuples, but the result does not add any insights beyond what we observe for tuple insert.

5.3.1 Bulk Load

Figure 3 compares the time the system takes to create native full B-tree indexes, indirect full B-tree indexes, and indirect sparse B-tree indexes, with multiple scans over the base table (i.e., once for each index). The graph varies the number of indexes on the table from

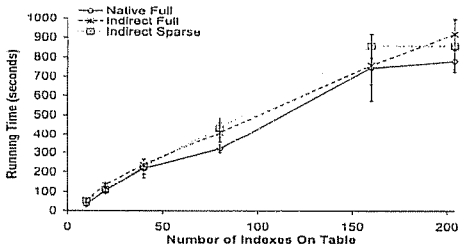


Figure 3: **Performance of creating 204 indexes in multiple passes.**

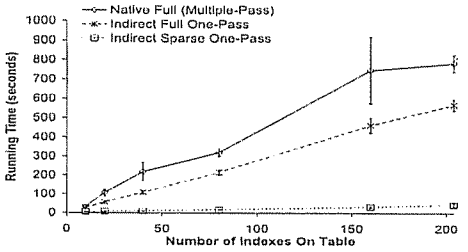


Figure 4: **Performance of bulk loading indexes compared to the native implementation that uses multiple passes.**

10 to 204 columns. The graph shows that indirect indexes take more time to create than the native indexes. However, the error bars overlap for most of the graph, so we can only conclude that the creation time with multiple scans of the base table is about the same for all index types. The reason is that scanning the table for each index dominates the cost of the index creation time.

Section 3.4 introduces a one-pass algorithm for creating all indexes over a table. Because we could not modify the database to build indexes in one pass, we implemented our algorithm in an external JDBC program. The program scans the base table, creates 204 separate flat files in the file system for the 204 indexes, then loads those files into the index-tables, and finally builds the clustered index over the index-tables.

In Figure 4, we compare the costs of bulk-loading the following: native full indexes in multiple passes, indirect full indexes in one pass, and indirect sparse indexes in one pass. Our results show that creating the indirect full indexes in one pass is about 1.5 times faster than creating native full indexes in multiple passes. Moreover, creating the indirect sparse indexes is between 6 and 20 times faster than creating the native full indexes in multiple passes. The results show that even though our implementation for the one-pass bulk-loading is crude, it is still much faster than the native implementation of scanning the base table once for each index (and would be even faster had the system implemented the algorithm itself).

Comparing the time to create indirect full indexes in one pass and the time to create indirect sparse in-

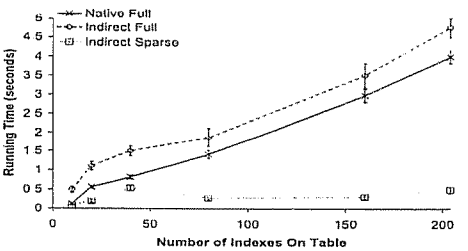


Figure 5: **Performance for inserting one tuple with 5 non-*null* values into a table with various number of indexes.**

dexes in one pass, we see that a single scan over the data does not impact the total cost of creating full indexes as much. The reason is that, after scanning the base table, the full indexes are still expensive to create because the number of data entries in a full index is the same as the number of rows in the data set. During the one-pass algorithm, the system spends 70% of the time loading and building full indexes on average. In contrast, sparse indexes are much smaller because they have only as many data entries as non-*null* values. Excluding the time spent on scanning the base table, the time to load and build all sparse indexes is about 17 times less than the time to load and build all full indexes on average. In conclusion, our one-pass algorithm drastically cuts the cost of creating sparse indexes, while providing only limited improvement on creating full indexes.

5.3.2 Inserts

Figure 5 shows the average cost of inserting one tuple into the integer data set with a varying number of indexes on a cold buffer pool. The insert cost is higher for the indirect full indexes than the native indexes because the indirect approach requires a join between the index-table and the base table, as discussed in Section 5.2. The insert costs for both the indirect and the native indexes are linear in the number of indexes on the table. Indirect indexes have a slightly higher cost because of the extra overhead of issuing SQL insert statements for each index-table. For sparse indexes, the insert cost per tuple depends more on the number of non-*null* values in an inserted row, than on the number of indexes on the table.

The maintenance cost of having many indexes on a table may vary depending on the density of a row inserted into the table. Figure 6 shows the cost to insert values into the table with 204 indexes with a different number of non-*null* values in the inserted tuple. The graph shows that the full index approach is constant as the number of values in the tuple increases. Sparse indexes have linearly increasing cost as the number of values in the tuple increases. When a row has 204 values, the cost to insert is the same as a full index.

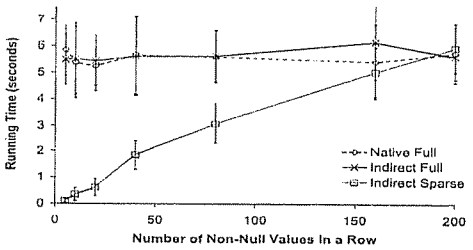


Figure 6: **Performance for inserting one tuple into a table with 204 indexes. The graph various number of non-*null* values in the inserted tuple.**

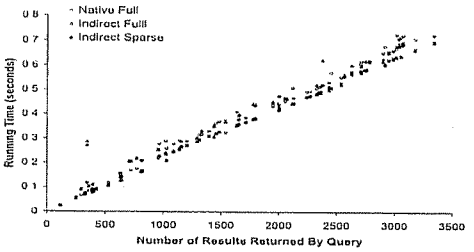


Figure 7: **Performance of queries with one simple predicate.**

Number Non-*null* in Row	Percentage of Tuples
0 to 5	33%
6 to 10	38%
11 to 20	15%
21 to 96	14%

Table 1: **The distribution for the number of non-*null* values in a row for CNET.**

5.3.3 Implications for Real World Data

In the CNET data, the distribution of the number of non-*null* values in a row is skewed to a low number of attributes. The average number of non-*null* attributes in a row is eleven, but the mode is five. Table 1 lists the distribution of the number of non-*null* values in a row. The table indicates that 71% of all rows have fewer than eleven non-*null* attributes and that only 14% have more than 21 attributes that are non-*null* in a row.

5.4 One-attribute Queries

Figure 7 shows the behavior of our indirect indexes compared to native indexes for a query that has one simple predicate, such as Q1, on a warm buffer pool. The figure indicates that our indexing method has performance similar to the native indexes, even though we perform an explicit join with the base table. Also, for the same queries with a cold buffer pool, the performance between the full and the native indexes are statistically insignificant (as confirmed by a two tailed paired t-test). In the following, we concentrate on the performance of sparse indexes alone on the data set and do not consider full indexes any further.

5.5 Two-attribute Queries

In this section we evaluate the performance of the indexes for queries over two attributes. In our evaluation, we use the eleven columns that have varying overlap with a primary column, as discussed in Section 5.1. The queries use P to denote the primary column and A_n to represent one of the other overlapping columns. In Section 5.5.1, we present the performance of using single-column indexes as an access path to the underlying base table. Section 5.5.2 considers the performance of single-column and multi-column queries where the indexes *cover* the query, meaning that all the columns that are necessary to the query are in the one two-column index or in two separate single-column indexes.

5.5.1 Single-Column Index Query Plans

In this section, we consider plans that use single-column indexes to answer queries and show that attribute correlation plays a roll in plan selection. We use the following queries to evaluate index access path plans:

```
Q4 (T2): SELECT * FROM H
         WHERE P = 3 AND A_n = 3

Q5 (T3): SELECT * FROM H
         WHERE P = 3 AND A_n is null
```

Consider these queries in the context of overlapping attributes. For Q4, when the overlap is high, or the attributes are correlated, the query returns many results. Query Q5 is has the opposite relationship as Q4, when the attributes are correlated there are few results returned. It is important to remember the relationship between the query result size and attribute correlation because the result size also affects performance.

Single-column indexes have two basic query plans for each of the queries. The first plan type, which we call index-fetch, evaluates the simple-predicate that selects the fewest rows in the conjunction by using the corresponding column index. Next, the plan fetches the matching rows from the table and evaluates the other simple-predicate. In our experiments, query Q4 uses the index on A_n because it returns the fewest rows. Query Q5 uses the index on P, because the sparse A_n index cannot be used. The second plan type, index-sets, uses the indexes on both attributes to select matching row ids for each simple-predicate. The two sets of row ids are then intersected (for Q4) or set-differenced (for Q5) to discover the matching rows in the table. In the following analysis, we do not consider plans that scan the underlying table, because the time to scan is far greater than using an index to lookup the tuples (it is a little under 25,000 ms to scan).

Q4 Performance. Figure 8 shows the performance for the two single-column index plans and the one two-column plan for query Q4. The Index-fetch plan remains constant across the graph be-

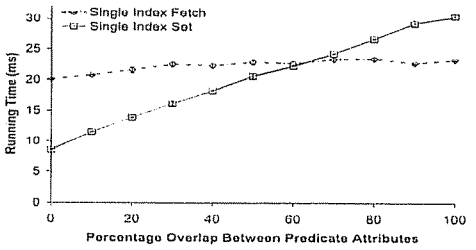


Figure 8: **Query Q4 performance for single-column index plans with varying overlap between the attributes in the query.**

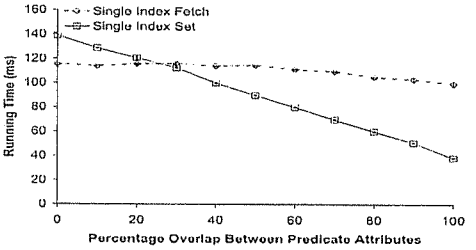


Figure 9: **Query Q5 performance for single-column index plans with varying overlap between the attributes in the query.**

cause it uses one index to fetch rows that match the simple-predicate A_n = 3. The simple-predicate always fetches the same number of rows, thus the time is relatively constant (it increases slightly because the query returns more results). The Index-set plan increases in running time as the overlap increases because number of results increases. The Index-set plan has better performance compared to the Index-fetch plan until 60% overlap when the time to intersect the indexes is higher than the time to fetch the rows.

Q5 Performance. Figure 9 shows the performance for the two single-column index plans and the one two-column plan for query Q5. Recall that the Index-set query uses set difference over two single-column indexes to evaluate the query. The figure shows that the query has nearly opposite performance as compared to query Q4, because the simple-predicate is looking for *null* values for the second attribute. More overlap between the two columns means fewer *null* values and fewer results. The conclusion from this experiment is that set-difference is important for single-column index queries that use an "IS NULL" predicate. If the system does not support set-difference for index plans, it will choose Index-fetch, which has far worse performance for attributes that have high overlap.

The results show that there is a tradeoff in choosing single-index plans for queries over sparse attributes. For correlated attributes, index-fetch is better for predicates over non-*null* values and set-difference is better for "IS NULL" predicates. For uncorrelated attributes, set-intersection is better for predicates over non-*null* values and index-fetch is better for "IS NULL" predicates.

Finally, since our focus was on single-column performance, we will note that two-column indexes can also be used to answer the queries. Two-column performance is always better than both single-column index plans. The two-column indexes can evaluate the predicate in one search of the index and identify the matching rows quicker than an single-column set-based operations and it retrieves fewer rows (or the same number of rows for 100% overlap) than the single-column fetch plan. Single-column index plans are still important, however, because it is infeasible to build indexes on all combinations of columns.

5.5.2 Queries Covered by Indexes

A query is *covered* by indexes when one or more indexes contain all of the attributes in the query. A covering query plan can be more efficient compared to plans that access the base table using an index fetch or table scan. The reason that covering plans can be faster is that indexes are usually smaller than tables, so index scans usually require less memory and I/O. Also, the indexes are sorted by attribute value and queries that order the result set, use group by, or join with another table can run without sorting.

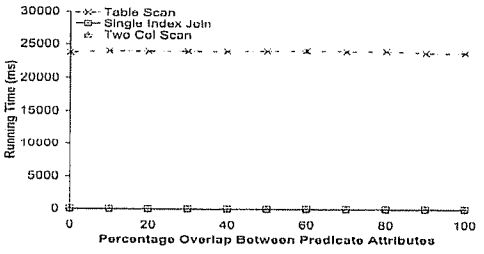
Covering indexes have long been used to evaluate projections covered by indexes. In dense data sets, the performance gain over table scans arises because the index omits some columns, hence the index is smaller. In sparse data sets, the effect is even more pronounced, because a sparse index omits rows (those with all *nulls* in the covered attributes) in addition to omitting columns.

We use the following query:

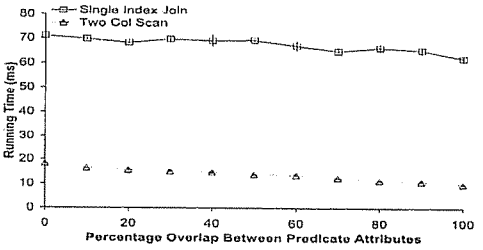
```
Q6      : SELECT P, A_n FROM H
          WHERE P is not null OR A_n is not null
```

Query Q6 projects two columns whenever either column is non-*null* in a row. Notice that the query exactly matches a two-column index over the attributes P and A_n. For two-column indexes, there are two possible orderings of columns to consider: the P attribute can be first with A_n second, or the P attribute second and A_n first. The performance of query Q6 does not depend on the order of the attributes in the index. Single-column indexes over the two separate attributes also cover the query by using a query plan that joins the indexes on id using a full outer join.

Q6 Performance. Figure 10 shows the performance for query plans that answer query Q6. The graph in in Figure 10(a) compares the performance of the base table scan plan, the two-column covering index plan, and the single-column covering index plan. The scan plan is extremely expensive because it scans all of the data in the table to project out only two attributes. The index plans are over 1300 times faster than the scan plan.



(a) Table Scan Plan is Slow



(b) Index Covering Plans

Figure 10: **Performance for Q6 plans with varying overlap between the attributes in the query.**

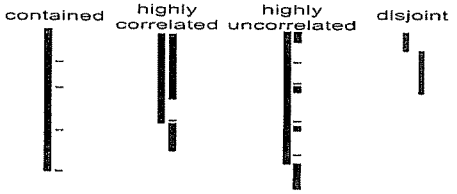


Figure 11: **Two-column indexes for the real world CNET data set.**

Since scan plans are so expensive, we concentrate on the performance of queries that use covering indexes. Figure 10(b) takes a closer look at the index-based plans. The two-column index plan is always faster than the single-index join plan, because the single-column plan has to compute the join to return the result and the two-column plan just has to scan the index. The figure shows that as the overlap between the attributes increases and the attributes become correlated, the running time of both index plans decreases. The reason is that the size of the two-column index depends on the correlation between the two single columns.

Correlation influences the performance of the plans. For disjoint attributes, the two-column index is 3.9 times faster than the single-column plan. For correlated attributes, the two-column index is 6.4 times faster than the single-column plan. The two-column plan benefits more from correlation, because the index is smaller. The single-column plan cannot benefit as much from correlation, because even though the correlation affects the result size, the index join always processes the same number of rows.

5.5.3 Implications for Real World Data

The CNET data has a little over four million possible attribute pairs and 150,121 pairs have some overlap. Less than 1.7% (2608) of the overlapping pairs have overlap greater than 80% and only 0.3% (438) overlap completely and are entirely correlated. Figure 11 shows two-column index patterns for four pairs of attributes in the CNET data. The patterns are similar to those presented in Section 4. It is much more likely that attribute pairs in the CNET data are like the last two distributions, disjoint or highly uncorrelated.

Multi-column indexes are promising for correlated attributes because they can provide better query performance compared to single-column indexes. The indexes are very useful as covering indexes for queries that operate on specific sets of attributes, but as Section 5.5.2 shows, query plans that scan a covering index have the best performance when the attributes in the index are correlated.

The number of attributes in a sparse data set, however, leads to many possible attribute combinations for composite indexes. One challenge with real world sparse data is determining the attributes to index with multi-column indexes, and the order the attributes should be included index. Certainly one would want to choose attributes that are commonly queried together, but one also has to manage the tradeoff between adding attributes into the index and possibly adding attributes that are uncorrelated and increasing the *null* values in the index.

5.6 Text Performance

We also performed index creation, maintenance, and query experiments for text data. All of our conclusions and observations from experiments with scalar integer data did not change for text data. Our experiments confirmed that an "is-defined" index is essential for good performance for "`IS NOT NULL`" queries over text, because a table scan is the only option without the index and scanning the base table is slow.

6 Related Work

Stonebraker introduced partial indexes in [14] and that work was extended by Seshadri et al. [12]. Seshadri proposed the use of generalized partial indexes to increase index coverage for dense data. The work demonstrated that systems should not index values that occur frequently in the data. This suggestion is similar to our proposal to index all non-*null* values of sparse data. However, generalized partial indexes requires special techniques for index selection and for query optimizers to choose these indexes as access paths. Sparse indexes act similarly to full indexes and, thus, do not require special support for any non-*null* value. Oracle [2] implements indexes that do not store *null* values, however to our knowledge there

is no published literature evaluating the performance of the indexes on sparse data sets.

Agrawal et al. explored the tradeoffs of horizontal and vertical storage using then-current commercial technology for queries over a horizontal view of sparse data [1], but did not consider the issue of indexing sparse data stored horizontally. Beckmann et al. [4] evaluated an alternative record format for sparse data that reduces the overheads for horizontal storage and has overall better performance than the vertical representation. Our work extends the study of horizontal schema by considering the issues of indexing data stored in the representation.

Index maintenance and building times have been explored in the context of on-line data processing [8, 13]. Much of this work focused on minimizing the effects of locking and latching for maintaining indexes, but none of the work evaluated build times or maintenance costs for sparse indexes.

Index-based query optimization has been considered in the context of dense data [11]. Mohan et al. used indexes to evaluate arbitrary selection predicates and discussed how to choose indexes based on the set of eligible indexes over the data [7]. Chaudhuri et al. considered efficient algorithms for factoring complex predicates in index-based plans [5]. With accurate knowledge of the distributions, many of the factorization techniques also apply to sparse data sets.

7 Conclusions

Relational database systems are increasingly facing the demands of applications with sparse data sets. The selectivity of the attributes in sparse data motivates indexes over all attributes. In this paper, we showed that sparse indexes, which only index the non-*null* values of an attribute, are the best approach for indexing all attributes of sparse data. In fact, they are so efficient that it is practical to index far more sparse attributes in a data set than one would think. Compared to full indexes, which include both non-*null* and *null* values of a data set, sparse indexes incur much lower maintenance costs on sparse data. Our evaluation shows that the maintenance costs of full indexes depends on the number of indexes built on the table, whereas maintenance costs for sparse indexes depends on the number of non-*null* values in a row, which for sparse data is only a handful of the total number of attributes.

In order to efficiently support queries over sparse data, we propose the use of index differencing for queries that use is-*null* predicates and the addition of an "is-defined" index for is-not-*null* queries over text data. These techniques significantly facilitate queries that contain *null* predicates, which are an important class of queries for sparse data.

Finally, we showed that the data distribution of sparse attributes impacts the effectiveness of index-based query plans for conjunctive queries. Specifically, for single-column indexes, uncorrelated attributes benefit more from index plans that use set-based intersection, whereas correlated attributes should use a single-index to fetch rows. The results for single-column indexes suggests that information about the distribution of *nulls* between attributes will help systems make informed decisions for query plans. We also demonstrate that two-column indexes can perform better than single-column indexes for all data distributions and can be especially beneficial as covering indexes.

Although keeping multi-column statistics on all pairs of hundreds of attributes is unrealistic, it may be beneficial to keep statistics for only specific groups of attributes. The reason is that sparse data sets, such as e-commerce data, usually follow schema guidelines that suggest to users the sets of attributes that should be defined together for objects. An area for future work includes using these schema guidelines to aid index selection and query optimization.

References

[1] R. Agrawal, A. Somani, and Y. Xu. Storage and querying of e-commerce data. In *VLDB*, pages 149–158, 2001.

[2] R. Baylis. *Oracle Database Administrator's Guide, 10g Release 1 (10.1)*, 2003.

[3] J. Beckmann. The CNET E-Commerce Specifications Data Set. `http://www.cs.wisc.edu/~jbeckham/TR/cnet.pdf`, June 2005.

[4] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. In *ICDE Conference*, 2006.

[5] S. Chaudhuri, P. Ganesan, and S. Sarawagi. Factorizing Complex Predicates in Queries to Exploit Indexes. In *SIGMOD Conference*, pages 361–372, 2003.

[6] CNET Networks, Inc. CNET Product Directory. `http://shopper.cnet.com`.

[7] C. Mohan, D. J. Haderle, Y. Wang, and J. M. Cheng. Single Table Access Using Multiple Indexes: Optimization, Execution, and Concurrency Control Techniques. In *EDBT*, pages 29–43, 1990.

[8] C. Mohan and I. Narang. Algorithms for Creating Indexes for Very Large Tables Without Quiescing Updates. In *SIGMOD Conference*, pages 361–370, 1992.

[9] D. Pyle. *Data preparation for data mining*. Morgan Kaufmann Publishers Inc., 1999.

[10] R. Ramakrishnan and J. Gehrke. *Database Managment Systems, 3rd Ed.* McGraw-Hill Higher Education, 2002.

[11] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access Path Selection in a Relational Database Management System. In P. A. Bernstein, editor, *SIGMOD Conference*, pages 23–34. ACM, 1979.

[12] P. Seshadri and A. N. Swami. Generalized Partial Indexes. In *ICDE*, pages 420–427, 1995.

[13] V. Srinivasan and M. J. Carey. Performance of on-line index construction algorithms. In *EDBT*, pages 293–309, 1992.

[14] M. Stonebraker. The Case for Partial Indexes. *SIGMOD Record*, 18(4):4–11, 1989.