The Case for a Wide-Table Approach to Manage Sparse Relational Data Sets

Eric Chu
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI USA
+1-608-628-6941
ericc@cs.wsic.edu

Jennifer Beckmann*
Microsoft Corporation
One Microsoft Way
Redmond, WA, USA
+1-425-421-7754
jennifer.beckmann@microsoft.com

Jeffrey Naughton
Computer Sciences Department
University of Wisconsin-Madison
Madison, WI USA
+1-608-262-8737
naughton@cs.wisc.edu

ABSTRACT
A "sparse" data set typically has hundreds or even thousands of attributes, but most objects have non-null values for only a small number of these attributes. A popular view about sparse data is that it arises merely as the result of poor schema design. In this paper, we argue that rather than being the result of inept schema design, storing a sparse data set in a single table is the right way to proceed. However, for this to be the case, RDBMSs must provide sparse data management facilities that go beyond the previously studied requirement of storing such data sets efficiently. In particular, an RDBMS must 1) enable users to effectively build ad hoc queries over a very large number of attributes, and 2) support efficient evaluation of these queries over a wide, sparse table. We propose techniques that provide these capabilities, and argue that the single-table approach is a necessary component of self-managing database systems because it frees users from a tedious and potentially ineffective schema-design phase when managing sparse data sets.

Categories and Subject Descriptors
H.2 [DATABASE MANAGEMENT]: Miscellaneous

General Terms
Design, Management

Keywords
sparse data, relational, wide table

1. INTRODUCTION
Suppose an online retail store has a product catalog that has thousands of attributes, and that most objects in the catalog have non-null values for only a small number of these attributes. How should we manage this and other similar, "sparse" data sets in a relational database management system (RDBMS)? As sparse data arises from many sources, including e-commerce hubs [3, 10], distributed systems [19], and even data extraction systems [2], providing efficient RDBMS support for sparse data has become increasingly important.

Unfortunately, although a number of approaches are widely used to handle sparse data, none of them is very satisfying with the current technology. Perhaps the most straightforward approach is to store all the products in a horizontal table (i.e., each column represents a distinct attribute and each row represents an object). For a diverse set of products, this approach results in a very wide and sparse table that wastes a lot of space and makes scans highly inefficient. An alternative is to use vertical tables [3], which eliminates nulls but generally suffers from complex queries and poor query performance. Yet another approach is to store a few "dense" attributes, which are attributes that most rows define, in a horizontal table, then relegates the rest of the attribute-value pairs to a large, catch-all text object. Though nulls are again eliminated, the attributes in the text object cannot be used as normal attributes in SQL queries. A more extreme approach gives up on RDBMS technology and resorts to a semi-structured model such as XML, but using a different data model introduces potential data integration and management issues. Lastly, a popular view (especially in the research community) on sparse data is that it is a false problem, because with proper schema design, database administrators (DBAs) can decompose a sparse data set into a reasonable number of smaller, denser tables. However, this multi-table approach places a heavy burden on schema designers, may not always be possible, and presents problems with respect to data fragmentation and schema evolution. In this paper, we argue that RDBMSs can be augmented to manage sparse data sets efficiently, and in such an augmented RDBMS, storing a sparse data set in a single table is a good approach.

At first glance, it may seem like we are advocating the Universal Relation [15], but we are actually addressing a completely different problem. The Universal Relation was a user-interface proposal in which a wide virtual schema covers all the physical tables in the database. In entity-relationship terminology, it puts all entities and relationships in the same logical table. The main challenge in evaluating queries over the Universal Relation is

* Work done when the author was a student at the University of Wisconsin-Madison.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'07, June 11–14, 2007, Beijing, China.
Copyright 2007 ACM 978-1-59593-686-8/07/0006...$5.00.

to translate the queries into semantically equivalent queries over the actual underlying schema. In contrast, we are discussing the approach of physically storing a large number of entities that belong to the same entity set (but may have very different subsets of non-null attributes in the schema for that entity set) in the same table. For example, we would store all products in an online catalog in a wide table, but we would not put products, customers, and the "purchased_by" relations in the same physical table. The challenges here are storing the data efficiently, enabling users to effectively build queries over a large number of attributes, and evaluating these queries over the wide, sparse table efficiently.

Recent work has shown that a sparse data set can be stored in a horizontal table with good storage efficiency by employing interpreted storage [6]. However, just storing the data in a space-efficient manner is not sufficient – in particular, two problems still arise: (1) how should users build queries over a table with thousands of attributes, and (2) how can a system efficiently evaluate these queries over such a wide, sparse table? In the following, we give an overview of the techniques we use to address these problems.

Query Construction: Writing ad hoc SQL queries over sparse data is challenging because there are so many attributes to choose from (a drop-down menu of thousands of attributes is unlikely to be helpful). Keyword search is a natural alternative because users do not need to specify attribute names, but its imprecise semantics compared to structured queries can be a problem. For example, replacing a structured query "$A_i = V_k$" with the keyword query "find all objects that contain V_k" will have a low precision in the result if many objects contain V_k in attributes other than A_i. We show empirically that low precision is not a problem on at least two real-world sparse data sets, and discuss the properties of sparse data sets that explain this result.

Although promising, keyword search is still problematic when the keywords appear in many attributes, and is inapplicable when users require more-advanced queries such as range queries and aggregates. Therefore, in addition to keyword search, RDBMSs should support "fuzzy" attributes in both keyword and SQL queries. Specifically, users make a guess about an attribute name they want, and the system tries to find the best match for that guess. This feature is promising because it can be considered as a tractable instance of schema matching; however, it is only useful to users who have some idea about the target attributes. When users cannot exploit fuzzy attributes, we present to them a "hidden schema" that we automatically infer from the data set, so that users can more easily choose the right attributes for writing SQL queries. We can also build a browsing-based interface based on the hidden schema and let users browse the data set itself.

Query Evaluation: Wide, sparse tables pose challenges to query evaluation not found in queries over narrower, denser tables. For example, scans must process hundreds or thousands of attributes in addition to those specified in the query. Furthermore, for SQL queries, scans seem to be the dominant query evaluation option because with thousands of attributes, unless one builds thousands of B-tree indexes, the probability of having an index for a query on a randomly chosen attribute is very low. Fortunately, sparse data sets present new opportunities as well as new challenges, and we exploit these opportunities with two query evaluation techniques. The first is building sparse B-tree indexes, which minimize the number of non-null values. On a sparse data set, they incur much lower maintenance costs and storage overheads than their full index counterparts. The second technique is building materialized views or covering indexes over subsets of attributes automatically detected by the system as correlated. On a sparse data set, this technique achieves both horizontal and vertical partitions, so that a query that scans a partition benefits from processing many fewer rows and columns than scanning the base table.

In brief, the contributions of this paper are as follows:

- We discuss how storing a sparse data set in multiple tables can cause problems, and how using a single table can avoid these problems, without incurring the overheads of nulls if the underlying system uses the interpreted storage format. (Section 2)
- We show how well-studied techniques such as keyword search and schema matching can be exploited to assist in query construction over sparse data. (Section 3)
- We demonstrate that sparse B-tree indexes incur low storage and maintenance costs over sparse data sets. Therefore, we can efficiently build and maintain a large number of them on a wide, sparse table. (Section 4)
- We show how to exploit correlations of attributes by employing clustering algorithms to discover a "hidden schema," which suggests useful configurations for views and indexes to expedite query evaluation, and provides a logical organization to assist users in query construction. (Section 5)

Also, Section 6 presents experimental results; Section 7 discusses related work; Section 8 concludes the paper.

Finally, we note that augmenting an RDBMS to better handle wide, sparse tables follows the current trend of shifting work from the users of the system to the system itself. By recognizing and exploiting the unique properties of sparse data, we free users from the troublesome task of designing elaborate schemas, yet still allow them to conveniently enjoy the benefits provided by an RDBMS.

2. A WIDE, SPARSE TABLE

2.1 The Problems of Storing Sparse Data in Multiple Tables

We begin with a brief diversion from our sparse data handling techniques to discuss why attempting to decompose a sparse data set into narrower, denser tables, even if users are willing to spend the effort, is not a panacea for sparse data management. We also review the interpreted storage format, as it or some similar technique is necessary for the efficient storage of sparse data.

As mentioned in the introduction, two characteristics distinguish sparse data sets from the conventional dense data sets: 1) a sparse data set has a large number of attributes; and 2) most objects typically have non-null values for only a small number of attributes. For example, Pyle described a brokerage firm that has nearly 700 attributes, half of which are null in 98% of the objects [17]. Agrawal cited an e-commerce marketplace that has nearly 5000 attributes, most of which are null for most objects [3]. We also encountered an e-commerce data set that has over 2,000 attributes, with most rows having non-null values for only about 5 attributes [10].

Designing a schema that partitions a data set with thousands of attributes into narrower, denser tables is laborious and troublesome. More importantly, for tables to be dense, the distribution of the non-null values must conform, for the most part, to this multi-table schema. In Section 5, we will see that sometimes we can indeed identify relatively dense partitions in a sparse data set. However, when the non-null distribution is unknown or irregular, storing the data set in multiple tables can have the following problems:

Significant sparseness: The distribution of the non-null values can be so irregular that any multi-table schema with a reasonable number of tables would still contain significant sparseness (of course, we can always eliminate sparseness completely by storing each tuple in a separate table, but this option is clearly infeasible). This problem can also occur when the non-null distribution is unknown at schema design time. In this case, it is difficult to determine a multi-table schema that is effective for reducing sparseness.

Complications for queries and updates: In a relational sparse data set, each object is represented as a tuple in a row, with its object identifier (oid) as a key. When the attributes of the data set are partitioned into multiple tables, a tuple can have non-null values for attributes stored in multiple tables, and so becomes physically *fragmented*. To reconstruct the tuple, a join on the tuple's oid is required; however, the query writer must know and specify which tables need to be joined, which is a daunting task because tuples can be fragmented across different sets of tables.

An alternative is to require each tuple to be stored in only one table. One way to implement this requirement is to allow attributes to appear in multiple tables. Although this approach avoids fragmentation, it can lead us back to sparse tables. Furthermore, projection queries and updates on attributes that appear in multiple tables will need to name and access all those tables.

Inflexibility for Evolving Schemas: Many applications that generate sparse data allow attributes to be defined freely. One example is crawling over a corpus of unstructured text documents to find potential attribute-value pairs. In this scenario, a tuple represents a document, and its non-null attributes are the attribute-value pairs that we discover in the document. Unless we pre-define a finite set of target attributes, we may encounter new attributes every time we process a document. Another example is Condor [18], a distributed workload management system, in which users can define new attributes in addition to using existing ones, for any job they submit.

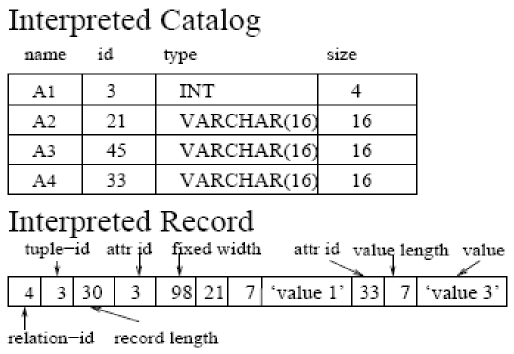
Unless we know the future non-null data distribution *a priori*, there is no good way to integrate a set of new attributes to a multi-table schema. We can either add the new attributes to some existing tables, or store them in a new table. However, both approaches can cause sparseness and complications for queries and updates.

2.2 A Review of Interpreted Storage

Storing a sparse data set in a wide table avoids the problems concerning tuple fragmentation and evolving schemas; however, this approach has been frowned upon because it causes a huge storage space blow-up by storing the null values. Fortunately, recent work has shown that by using the interpreted storage format [7], which we briefly review in the following, we can avoid the storage overheads of the null values.

Unlike the predominant *positional* storage, which allocates a pre-determined amount of space to all attributes in a horizontal table, the *interpreted* storage format, shown in Figure 1, avoids storing the null values. To "interpret" a tuple of this format, the system uses a catalog that records for each attribute its name, id, type, and size. A tuple in the interpreted format starts with a header, which contains fields such as relation-id, tuple-id, and record length; then, for each of its non-null attributes, the tuple stores the attribute's identifier, length field (if the type is of variable length), and value. Attributes that appear in the catalog, but not in the tuple, are implicitly null for that tuple. For instance, the interpreted tuple in Figure 1 has non-null values for A1, A2, and A4 after the header. This storage format is highly flexible for adding new attributes – we only need to update the system catalog, and the way we define tuples remains the same.

Of course, merely storing the sparse data set in a wide table using interpreted storage does not solve the problems of query construction and evaluation. In the rest of the paper, we turn our attention to those two problems.


Interpreted Catalog

name	id	type	size
A1	3	INT	4
A2	21	VARCHAR(16)	16
A3	45	VARCHAR(16)	16
A4	33	VARCHAR(16)	16

Interpreted Record

4	3	30	3	98	21	7	'value 1'	33	7	'value 3'

Figure 1. Interpreted catalog and an interpreted record using this catalog.

3. QUERYING SPARSE DATA

3.1 Keyword Search

Writing ad hoc SQL queries over a sparse data set is challenging because there are so many attributes to choose from. For example, suppose that some user wants to query the e-commerce data set described by Agrawal [3], which has over 5000 attributes. One could try to apply the standard query-building approaches of displaying the schema on the user's screen, or providing a drop-down list from which the user can select the desirable attributes. However, with over 5000 attributes, these and similar approaches will not be effective.

Recently, some researchers have investigated the use of keyword search as an alternative to query a structured database [12, 13, 14]. Keyword search is a natural solution for the "too many attributes" problem because users need not specify the attributes. However, if the keywords appear in many attributes, and the users prefer the keywords to appear in only some specific attributes, this approach will include extraneous objects. In this case, the answer set can have a low precision. More formally, let $Q_K = [x_1, ..., x_n]$ be a keyword query from a user. We assume that there exists a list of attributes $A = [a_1, ..., a_n]$, in which a_i can be equal to a_j for $i \neq j$, such that the SQL query

```
Qs:  SELECT *
     FROM table_t
     WHERE a1.contains(x1) AND ... AND
     an.contains(xn)
```

represents the ideal query for the user. The answer set Z_S of Q_S is then the set of relevant objects. Let Z_K be the answer set of Q_K. We define the precision of Z_K to be the percentage of objects in Z_K that is also in Z_S, or $|Z_K \cap Z_S|/|Z_K|*100$.

We analyze two real-world sparse data sets to try to answer the question: how often is low precision a problem for keyword search over real-world sparse data sets? The first data set we use in our evaluation, CNET, is an e-commerce product catalog that consolidates product information from different vendors [10]. As a relational table, the data set has 2,984 attributes across a total of 233,304 products. The average number of attributes in each row is 11 and the mode is 5. The second data set, EBuild, is a home-building product catalog. It has 1,101 attributes, a total of 302,631 objects, and a degree of sparseness similar to CNET. Because we obtained similar results for the two data sets, in the following we will only discuss CNET due to lack of space.

We ran our experiments as follows. We ignored "dense" attributes (there are only 4 dense attributes in CNET). After tokenizing all data values in the sparse attributes into terms, for each term, we recorded 1) the number of attributes that contain the term, and 2) the number of rows that contain the term. We refer to these numbers as Attr-Num and Row-Num respectively.

We plot the top 500 terms ranked by Attr-Num in Figure 2 and the top 500 terms ranked by Row-Num in Figure 3. Along the X-axis are the top 500 terms ranked by the Attr-Num or Row-Num values on the Y-axis. We include only the top 500 terms because both distributions have a very long tail, so including all terms would heavily skew the plot to obscurity.

Both distributions look Zipfian – a small number of terms appear in many attributes (or rows), but most terms appear in relatively few attributes (or rows). Zipf's Law describes many natural phenomena, the most famous one being the frequency of English words in a corpus. Indeed, we observe
that the term-frequency distribution in the data set is also Zipf-like.

Table 1 shows the percentage breakdowns of all terms in CNET with respect to Attr-Num, Row-Num, and both at the same time. We show the breakdowns with respect to both values across the first 5 rows that represent different ranges of Attr-Num, and across the 5 left-most columns that represent different ranges of Row-Num. The last row and column show the percentage breakdowns of the Row-Num and Attr-Num distributions respectively.

Table 1 reveals the heavy tails of the distributions. Under Attr-Num, we see that 85% of all terms appear in fewer than 6 attributes, and only 5% appear in more than 15 attributes. This result suggests that low precision is not an issue for most values in the table. As for Row-Num, 72% of all terms appear in fewer than 25 objects, and 13% appear in more than 150 rows, so that doing keyword search on even just one term would usually give us relatively few rows. Also, the two distributions are positively correlated. The terms that appear in more than 150 rows are the same ones that appear in more than 15 attributes. The tails of the two distributions share many terms in common; 71% of terms appear in fewer than 6 attributes and 26 rows.

To summarize, most terms appear in few attributes and in few rows. Unless users query the small set of keywords that appear in the head of the distributions, the answer set will be surprisingly focused. If the query contains multiple keywords, this pattern would become even more extreme because of the multiplicative selectivity of conjunction.

But what do these results say about keyword search over sparse data in general? We note that most keyword queries over the sparse data sets we studied are focused because (a) the universe of values appearing in the data sets roughly follow a Zipf-like distribution, and (b) the rows in a sparse data set define only a few non-null values (Because most rows in a sparse data set define few attributes, the values that appear in few objects most likely also appear in few attributes, which explains the overlap of the tails of the two distributions). Property (b) is a property of all sparse data sets (by definition), so we will see the same focused results from keyword queries for any data set that follows a Zipf-like distribution. We consider this observation encouraging because many data sets follow Zipf-like term distributions.

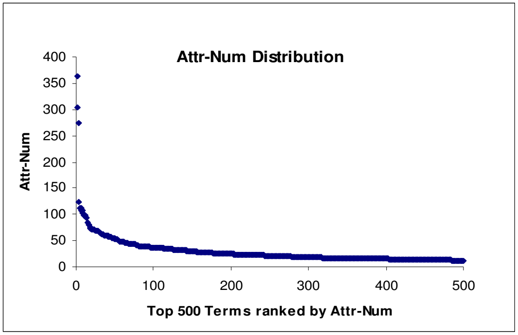


Figure 2. Term distribution with respect to Attr-Num.

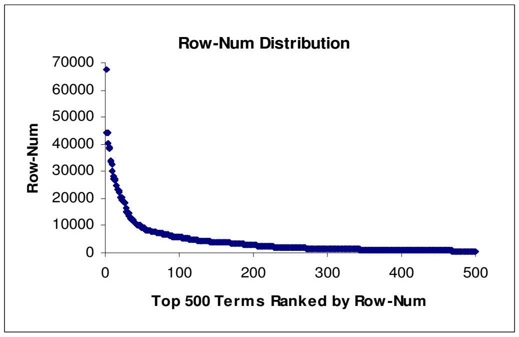


Figure 3. Term distribution with respect to Row-Num.

CNET	1 only	2-25	26-50	51-150	>150	Attr-Num
>15	0%	0%	0%	0%	5%	*5%*
11 - 15	0%	0%	0%	1%	2%	*3%*
6 - 10	0%	1%	1%	2%	3%	*7%*
2 - 5	1%	18%	4%	4%	2%	*29%*
1 only	21%	31%	2%	1%	1%	*56%*
Row-Num	*22%*	*50%*	*7%*	*8%*	*13%*	*100%*

Table 1. Percentage breakdowns of term distributions with respect to Attr-Num, Row-Num, and both at the same time.

3.2 Fuzzy Attributes for Keyword Search and SQL

Besides giving low-precision results in the hopefully rare cases when keywords appear in many attributes, keyword search is inapplicable when users need more-advanced queries, such as range queries and aggregates. In both situations, although users may not know the exact attribute names they should use, they may have some idea about them. Therefore, allowing users to specify "fuzzy" attributes in keyword and SQL queries can be very useful. That is, users make guesses about the names of the attributes they want, and the system tries to find attributes in the schema that match the guesses with high similarity scores. This task can be done by using name-based schema-matching techniques (e.g., consulting a domain-specific thesaurus, performing standard stemming and transforming operations, etc.), which have been shown to be effective in the data integration domain [18] (and is often regarded as the "easy part" of data integration).

For a SQL query, once we have identified attributes in the database that match the fuzzy attributes, we can replace the fuzzy attributes with these "real" attributes and execute the revised query. This approach, which we refer to as F_SQL, presents a problem when it returns multiple candidate attributes for a fuzzy attribute. Based on the similarity scores of each candidate with respect to the corresponding fuzzy attribute, we can revise the query in two ways. One possibility is to pick the candidate with the highest similarity score for the fuzzy attribute. This approach is simple, but if the candidate turns out to be wrong, we may miss some tuples that we should have returned. Alternatively, we can use more than one candidate and get several queries, whose results can then be merged to get the final result. For example, consider the query "$A_1 = V_1$ and $A_2 = V_2$," in which A_1 and A_2 are fuzzy attributes that match the two sets of "real" attributes $\{C_{11}\}$ and $\{C_{21}, C_{22}\}$ respectively. We can rewrite the query as:

"$A_1 = C_{11}$ AND $A_2 = V_{21}$ OR $A_1 = C_{11}$ AND $A_2 = V_{22}$"

This approach has a greater chance of getting the right tuples; however, the query rewriting quickly gets unwieldy when multiple fuzzy attributes have multiple candidates, and the final result may once again be a superset of the result that the users intended. Similar approaches that seek a compromise between these two extremes also face problems with either low precision or low recall.

For keyword search with fuzzy attributes (in the form of "f_attr = value" where f_attr is a fuzzy attribute), we can execute the query as follows:

1) Run keyword search on the data value and obtain a set of objects Z_K.
2) Identify a set of candidate attributes, A_C, by performing name matching between f_attr and the attributes in Z_K that contain the value.
3) Return an object in Z_K only if the object contains the value in an attribute in A_C.

Compared to F_SQL, this approach, which we call F_KS, has the advantage that instead of matching with the set of all attributes, we match the fuzzy attributes with only the set of attributes that contain the keywords (Step 2). Compared to pure keyword search, F_KS could possibly achieve higher precision because it applies the attribute constraint to the set of objects obtained by keyword search (Step 3).

None of the three approaches is always better than the others. F_SQL could result in extra work if the fuzzy attributes match many attributes, few of which contain the data values. F_KS avoids that problem, but does not apply for range queries. Also, one can imagine queries for which F_SQL is highly effective (e.g., only a single attribute matches the user's fuzzy attribute) and for which F_KS is inefficient because of an expensive and imprecise keyword query in Step 1. Exploring the tradeoffs between these approaches is a promising area for future research.

The utility of fuzzy attributes depends on the users' knowledge of the data set. In Section 5.4, we describe a complementary query-building approach to help users who cannot exploit fuzzy attributes. Specifically, we use a hidden schema that we automatically infer from the data set as a logical organization of the attributes, which users can browse through and select the attributes they need for writing structured queries.

4. SPARSE B-TREE INDEXES

4.1 Motivation

When the entire sparse data set is stored in a single table, it is crucial that we minimize the need to scan the whole table. A common approach to avoiding table scans is indexing. Although inverted indexes avoid table scans for keyword queries, they cannot be used for range queries, so B-tree indexes or their equivalent must be used. Unfortunately, building and maintaining hundreds of B-tree indexes on a table is generally considered infeasible because of the high storage and update costs that they incur. In this section, we show that when the data set is sparse, we can overcome this problem by using sparse B-tree indexes.

A *sparse index* on an attribute maps only the non-null values to the object identifiers (oids). Therefore, on a sparse data set, it incurs much lower storage overhead and maintenance cost than its full counterpart, which indexes both null and non-null values. Specifically, the size of a sparse index on an attribute is proportional not to the number of rows in the table, but to the number of rows that have a non-null value for that attribute. Moreover, for each tuple insertion or deletion, we only need to update the indexes on the attributes that are non-null in the tuple. In other words, a table may have a large number of sparse indexes, but if a tuple that is being inserted or deleted contains few non-null attributes (which is the norm in a sparse data set), only those few indexes will be updated.

Sparse indexes are a special case of *partial indexes* proposed by Stonebreaker [21]. A partial index contains only a subset of the tuples in a table. To define this subset, it uses a conditional expression called the predicate of the index. Only tuples that are evaluated true in this predicate are included in the index. For instance, we can define a sparse index on an attribute A in a table H in terms of a partial index as follows:

```
CREATE INDEX A_sparse_index ON H(A)
WHERE A is not NULL
```

Although feasible, using generic partial indexes to implement sparse indexes is not the best solution, especially if we want to build many sparse indexes. The reason is that partial indexes require predicate checks during index maintenance and query evaluation. When a tuple is inserted or deleted, the system must evaluate the predicate of each index on the table to determine if the index needs to be updated – a table with hundreds of partial indexes will have hundreds of predicate evaluations for each tuple insertion or deletion. As for query evaluation, because partial index supports arbitrary predicates, the query optimizer needs to check if the index is applicable, e.g., a partial index containing only tuples that satisfy the predicate "Color = red" cannot be used for the query "Color = blue." Sparse indexes avoid these overheads because they consider the specific predicate of whether an attribute is non-null. A lookup of the non-null attributes of a tuple would determine which indexes need maintenance. Also, sparse indexes are applicable for queries with any non-null predicates.

Finally, with the ability to build sparse indexes over a large number of attributes, we can consider building multi-column covering indexes, which could be useful for vertically partitioning the data set. The challenge is to determine which attributes should be grouped together in these indexes. We address this problem in Section 5.

4.2 Bulk-loading Sparse Indexes

Many database systems support index construction via bulk-loading, a single operation that scans the table to retrieve the index keys, sorts the index entries (with external merge-sort), and builds a B-tree over them. When multiple non-clustered indexes are to be built, these systems employ bulk-loading for each index in succession; in other words, the table is scanned once for each index construction. Although the operations are done offline, this *scan-per-index* approach is undesirable when we want to build hundreds of indexes on a table.

We propose to improve the efficiency of large-scale index construction by using a *scan-per-group* approach, which scans the table once per group of m indexes. Shown in Figure 4, our algorithm for bulk-loading divides a memory buffer pool of size B into m equal sections of $\lfloor B/m \rfloor$ buffer pages (β1 to βm). In the table scanning phase, the algorithm retrieves the index keys from each tuple and stores the key-oid pairs in the corresponding β_i. As β_i becomes full, its content is written to a file on disk. In the index building phase, for each index, the algorithm performs external merge-sort on the keys from the file of key-oid pairs, then builds the internal nodes of the B-tree.

Although the algorithm shown here is for creating sparse indexes, it will create full indexes if we remove the requirement that an attribute is non-null (Line 1). In the following, we analyze the cost difference in building m indexes between scan-per-index and scan-per-group, for both full and sparse indexes. To simplify our comparison, we show only the two aspects of the bulk-loading operation that make a difference: the I/O cost and the fetch cost, which is the cost of retrieving the index keys. We ignore the costs of doing external merge-sort and building the internal nodes of the B-trees because they are the same in both approaches.

For scan-per-index, we assume that a portion of the table T is cached in the buffer pool after building an index. Let BP be the number of pages of this portion, ||T|| be the number of pages in T, and IO be the cost of sequentially reading a page. The I/O cost of scan-per-index is:

$$||T||*IO + (||T|| - BP)*IO*(m-1) \qquad \text{(Eq. 1)}$$

In comparison, scan-per-group scans the table once, and writes and reads each index entry once (Lines 2, 3, and 4). Let ||I|| be the number of pages of an index I. We make the simplifying assumption that ||I|| is the same for each index and get the following I/O cost for scan-per-group:

$$||T||*IO + 2*||I||*IO*m \qquad \text{(Eq. 2)}$$

To estimate the fetch cost, let |T| be the number of index entries in T, and FETCH be the constant cost of retrieving an index key from T. For scan-per-index, the fetch cost is:

```
Scan-per-group Approach for Bulk-loading
INPUT: Table $T(c_1, ..., c_n)$
          Index definitions $IDef_1, ..., IDef_m$
          Buffer of size B divided evenly into $\beta_1, ..., \beta_m$
OUTPUT: Indexes $I_1, ..., I_m$
Algorithm:
  // Table scanning phase
  for each tuple $(t_1, ..., t_n)$ in T do
    for each $IDef_i$ do
      if $t_i$ is not null then                          // Line 1
        INSERT $(t_i$, oid) into $\beta_i$
        if $\beta_i$ is full then
          WRITE tuples in $\beta_i$ to file $F_i$          // Line 2
        end if
      end if
    end for
  end for
  for each $\beta_i$ do
    WRITE remaining tuples to $F_i$                      // Line 3
  end for
  // Index building phase
  for each $IDef_i$ do
    READ $F_i$ to B pages of buffer pool                 // Line 4
    EXTERNAL MERGE-SORT of $F_i$
    BUILD B-tree $I_i$ over $F_i$
  end for
```

Figure 4. Creating m sparse indexes per table scan.

$$|T|*FETCH*m \qquad \text{(Eq. 3)}$$

Scan-per-group retrieves only non-null index keys. Therefore, when a tuple has on average c non-null attributes, the fetch cost for scan-per-group is:

$$|T|*FETCH*c \qquad \text{(Eq. 4)}$$

For full indexes, c is the same as m, so the difference in fetch cost is zero. We get the following difference in I/O cost between scan-per-index and scan-per-group by subtracting Eq. 2 from Eq. 1:

$$(||T|| - BP)*IO*(m-1) – 2*||I||*IO*m \qquad \text{(Eq. 5)}$$

A negative difference means that scan-per-index is better, whereas a positive difference means that scan-per-group is better. Scan-per-group loses its edge as more of T fits in the buffer pool. When T fits completely in the buffer pool (i.e., ||T|| = BP), scan-per-group is never better than scan-per-index.

Although Eq. 5 also describes the I/O cost difference for both full and sparse indexes, the difference is much more significant for sparse indexes because ||I|| is much smaller without the null values. At the extreme, when β_i is large enough to hold all entries of I_i, the algorithm does not have to flush the index entries to disk at all. In addition, scan-per-group incurs a lower fetch cost for sparse indexes than scan-per-index:

$$|T|*FETCH*(m – c) \qquad \text{(Eq. 6)}$$

The difference increases when we are building more indexes (a larger m). Because of this and the I/O cost difference, scan-per-group is much more efficient than scan-per-index in creating many sparse indexes. We present experiments on these tradeoffs in Section 6.

5. HIDDEN SCHEMAS

5.1 Defining "Good" Partitions

Besides maximizing index coverage, another approach to avoid scanning the entire sparse table is to vertically partition the data set with materialized views or covering (sparse) indexes. Using a vertical partition is more efficient than scanning the base table because there are fewer columns to process. This advantage is huge for tables with many attributes, especially positional storage is used. Interestingly, vertically partitioning a sparse data set also achieves horizontal partitioning (as all-null rows are omitted), so the vertical partitions actually have both fewer rows and columns than the base table.

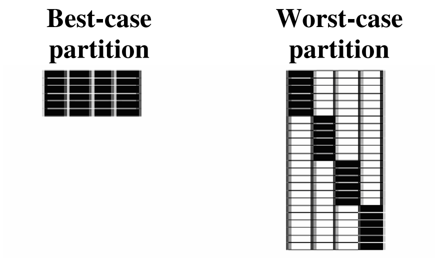


Figure 5. The best-case partition has no null values, whereas in the worst-case partition no two columns are non-null for the same row.

The challenge for vertical partitioning is to determine good partitions of attributes. This task is essentially the same as designing a good schema for the multi-table approach, except that in our approach, we materialize the partitions as views or indexes on top of the base table. Therefore, their objective is the same – maximizing the scan benefit of the partitions (or tables) while minimizing their maintenance costs. We list the following desiderata for vertical partitions:

- A reasonable number of partitions (e.g., getting one partition for each tuple is useless).
- Partitions contain minimal null values.
- Each base-table tuple is preferably stored in its entirety in one partition.

Given a sparse data set, it is unclear if we can obtain partitions with these qualities, because the tuples can have non-null values for any combination of attributes. However, our desiderata suggest that one way to approximate good partitions is to try to group together *co-occurring* attributes, or attributes that have non-null values in the same rows. Intuitively, a partition comprising co-occurring attributes will be dense.

Identifying groups of co-occurring attributes in a sparse data set, however, can be difficult for a schema designer because the attributes have varying degrees of co-occurrence, and co-occurring attributes may not appear adjacent in the wide schema. Therefore, our goal is to automatically discover these groups of co-occurring attributes, which from now on we collectively refer to as a *hidden schema*. In Section 5.2, we describe how we infer a hidden schema by clustering attributes based on co-occurrence. In Sections 5.3, we discuss our approach of materializing partitions on top of the base table. In Section 5.4, we discuss the benefits of a hidden schema for query construction.

5.2 Inferring Hidden Schema via Attribute Clustering

We consider vertical partitioning as a k-nearest-neighbor (k-NN) partitioning of the attributes based on co-occurrence. Given n attributes A_1, ..., A_n and a target of k partitions, we want to find k clusters of co-occurring attributes. Our approach is as follows.

We model the relationship between attributes in a sparse data set as a connected weighted graph. In this graph, each node represents an attribute, and every pair of nodes is connected by a weighted edge, whose weight represents the strength of co-occurrence between the two attributes. We use the Jaccard coefficient to define this weight. Given two attributes A_X and A_Y, let X be the set of rows for which A_X is non-null, and let Y be defined analogously. The Jaccard coefficient for A_X and A_Y is then defined as:

$$\text{Jaccard}(A_X, A_Y) = |X \cap Y| / |X \cup Y|$$

In other words, the numerator is the number of rows that have non-null values for both A_X and A_Y, whereas the denominator is the number of rows that have non-null values for A_X, A_Y, or both. The value of this coefficient ranges from zero to one. It is zero when no rows have non-null values for both A_X and A_Y, and one when A_X and A_Y are either both null or both non-null for all tuples.

After creating an adjacency matrix on the attributes with the weights as the values, we use a k-NN clustering algorithm implemented in CLUTO [9] to find a hidden schema. We consider disjoint partitioning schemes in which each attribute is assigned to only one partition.

As co-occurrence is the main criterion for clustering, the algorithm's objective is to minimize sparseness. To evaluate the quality of a partition with respect to sparseness, we define *NullRatio*, which compares the actual number of null values in the partition to the largest possible number of null values. For a partition P that has c attributes and a total of m non-null values in r rows, its NullRatio is defined as:

$$\text{NullRatio}(P) = (c^*r - m)/(c-1)^*m$$

The numerator is the number of null values in the partition, whereas the denominator is the highest possible number of null values in a partition with c attributes and m total non-null values. This worst-case scenario occurs when all the attributes are disjoint, meaning that no two attributes have non-null values in the same row. The NullRatio can range from zero, when the partition has no null values, to one, when the partition has the maximum number of null values. Figure 5 shows an example of best-case and worst-case partitions. In Section 6, we report that our approach finds clear clusters of attributes with low NullRatio from our sample data sets; moreover, the partitions make semantic sense.

Finally, we note that although our current clustering approach tries to minimize sparseness in the hidden schema, it has no constraints on the total number of partitions or on the percentage of fragmented tuples per partition. An interesting direction for future work is to extend the clustering algorithm to support these constraints.

5.3 Hidden Schema for Query Evaluation

Once we obtain a hidden schema, we can use it to build either materialized views or covering indexes on top of the base table.

The following is an example of defining a sparse view over a group of attributes A_n, ..., A_m in a relation H:

```
CREATE MATERIALIZED VIEW sparse_view ON
H(oid, A_n, ..., A_m)
WHERE NOT (A_n is null AND ... AND A_m is null)
```

As the partitions are relatively dense and narrow, we may consider storing them with positional storage, rather than with interpreted storage that we suggest for the sparse table. The reason is that although interpreted storage incurs no storage overheads over null values, it is less efficient than positional storage in retrieving values from the attributes. Specifically, with interpreted storage, the system must "interpret" the attribute identities and their values for each tuple at query access time, whereas with positional storage, the position information of the attributes is pre-compiled. For dense partitions, the benefit of not storing null values becomes less relevant, while the performance gain in retrieving values from attributes becomes more desirable.

One might ask whether our approach of storing a number of dense, narrow views over the data set is consistent with our comments earlier that partitioning a sparse data set into narrow, dense tables is not a good idea. The answer is yes, for several reasons. First, the partitions discussed here are discovered automatically, and used in queries automatically (standard view matching algorithms can be employed transparently to determine when a query can be evaluated from the view), without imposing any burden on the users. Second, because the partitions are materialized views, there is always the option of referring to the base table if the optimizer decides that using the view is less efficient than using the base table for a given query. Third, if after some number of future updates the views no longer match the hidden schema embedded in the current underlying data set, we can always drop the views, and create new ones as determined by the latest hidden schema.

Another reasonable question is how expensive it is to build and maintain these materialized views. For reasons similar to those that explain why sparse B-tree indexes are efficient over sparse data, storing and maintaining these views is far more efficient than it would be for dense data. Regarding storage, the additional overhead of these views is approximately equal to that of the base table that uses interpreted storage. To see this, note that except for the oids, each non-null value is stored in only one view because each attribute appears in only one view. When the views are stored using interpreted storage, only the non-null values take space; when the views are stored using positional storage because they are dense, the null overhead is again small. Regarding maintenance costs, consider a table with 100 attributes A_1 to A_{100} and we materialize ten projection views on it: A_1 to A_{10}, A_{11} to A_{20}, and so on. If the table is sparse and the tuples usually have values for only one partition, each tuple insertion, deletion, or update on the sparse table will trigger two updates on average. In comparison, if the table is dense in the same setting, each update will cause eleven updates – one to the sparse table, and one to each of the ten views.

5.4 Using the Hidden Schema for Query Construction

With a hidden schema, we can extend our effort in aiding users to query sparse data sets. When facing a sparse data set, users are often overwhelmed not only because of the vast number of attributes, but also because the data set is unorganized. The lack of logical organization leads to two problems: 1) It is difficult to make sense of a schema that has so many attributes, and 2) it is difficult to browse through the data. A hidden schema can improve these problems by imposing order on chaos. Specifically, we can build a directory of attributes from which users can choose the appropriate attributes for structured queries, or even build a browsing-based interface that lets users browse the data set itself as multiple dense "mini-tables."

In Section 3, we discussed pure keyword search, F_KS (keyword search with fuzzy attributes), and F_SQL (SQL with fuzzy attributes). All these approaches satisfy users with different needs and knowledge about the data set. Pure keyword search is for users who know about the data, but not the attributes. F_KS is for users who have some idea about the attributes they want and do not need the expressive power of SQL. F_SQL is for users who are somewhat familiar with their attributes and need to ask complex queries that cannot be expressed via keyword search. When these approaches are inapplicable, browsing-based approaches based on a hidden schema provide a helpful alternative, especially for users who know nothing about the schema, and for those who do not even have a specific query in mind.

6. EXPERIMENTS

6.1 Sparse Indexes

We conducted our experiments with sparse indexes in PostgreSQL, which implements both full and partial B-tree indexes, and uses the scan-per-index approach for bulk-loading. During bulk-loading, it packs the leaf pages to 90% capacity and the non-leaf pages to 70% full. We implemented the scan-per-group algorithm as described in Section 4.2, and set the buffer size B to 40 MB. For partial indexes, PostgreSQL uses a heavy-weight predicate evaluation that makes several function calls. Based on this implementation, we implemented sparse indexes by using a lighter-weight test that looks for non-null attributes in the inserted tuple.

We used synthetic data so that we could control the parameters in the experiments. We modeled the properties of the data after the CNET sparse data set. The table had 250k rows and on average 5 non-null values per row distributed over 640 varchar(16) attributes. We used the interpreted storage format to store the table in all but one experiment, in which we used positional storage. The size of the table was 37.6MB and 391MB for interpreted and positional storage, respectively. Each single-column full index was 4.34MB and each single-column sparse index was 88KB. In other words, a sparse index was approximately 50 times smaller than a full index, and the amount of space that sparse indexes took to cover all 640 attributes in the table was only enough for 13 full indexes.

We compared scan-per-group with scan-per-index for bulk-loading both full and sparse indexes. Figure 6 shows the times of building different numbers of full indexes with both approaches on a table that fits in memory. While scan-per-index increased linearly as the number of indexes increased, scan-per-group increased even more rapidly. Recall from Section 4.2 that although scan-per-group scans the table only once, it incurs an overhead on buffering tuples to disk and this cost is proportional to the number of indexes (Eq. 2). Therefore, when the base table fits in memory, scan-per-index's advantage over scan-per-group increases as more indexes share the buffer pool in the scan-per-group approach.

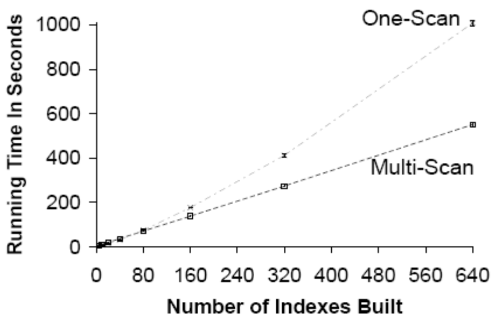


Figure 6. Bulk-loading full indexes when table fits in memory.

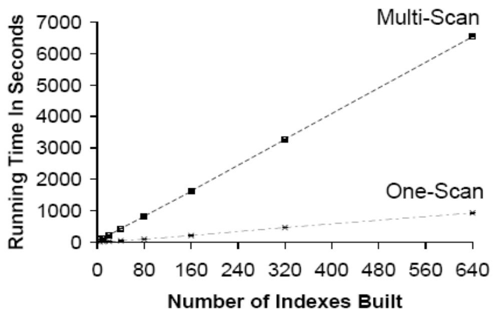


Figure 7. Bulk-loading full indexes when table does not fit in memory.

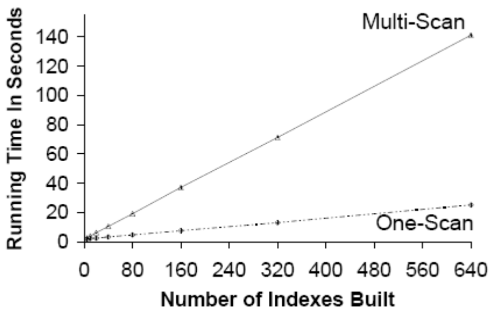


Figure 8. Bulk-loading sparse indexes when table fits in memory.

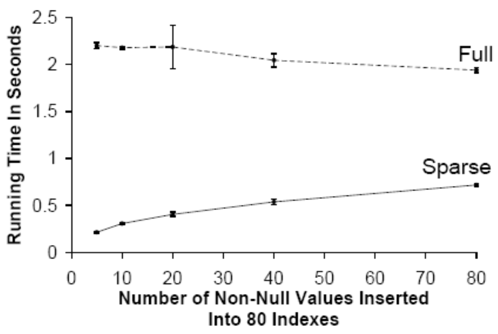


Figure 9. Inserting different number of non-nulls into 80 indexes.

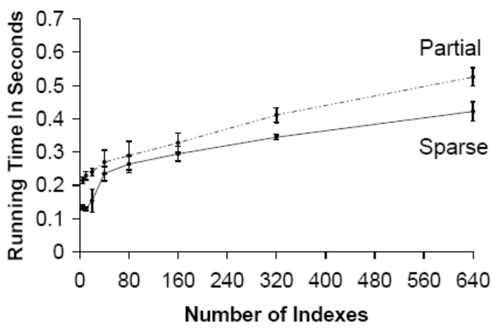


Figure 10. Inserting 5 non-nulls into table with varying numbers of full and sparse indexes.

To observe the difference when the base table does not fit in memory, we experimented with the same data set in positional storage. Figure 7 shows the comparison of building full indexes on this big table with the two approaches. Scan-per-group always outperformed scan-per-index because scan-per-index had to scan what was not in the buffer many times, and the cost advantage increased as the number of indexes increased (Eq. 1).

Figure 8 compares the performances of the two approaches in creating sparse indexes on the table using interpreted storage (so the table fits in memory). It shows the same pattern as in Figure 7 – scan-per-group was much better and the advantage increased with the number of indexes. This experiment shows the cost savings in doing fewer scans as we batch indexes together. Also, comparing Figures 6 and 8, we see that for the same set of attributes on the same table, the amount of time to build the sparse indexes was much less than that to build the full indexes.

Next, we compared the maintenance costs of the two types of indexes. The maintenance cost depends on the density of a row inserted into the table and the number of indexes affected by the insert. To illustrate these factors, we built 80 indexes over the base table, then inserted rows with non-null values for different numbers of the indexed attributes. Figure 9 shows that for full indexes, the insert costs were relatively constant because an update was required whether the value was null. However, for sparse indexes, the update costs started low when a new tuple had few non-null values for the indexed attributes, and increased as more of the 80 indexed attributes were non-null.

Finally, we compared the insert performance between a generic partial index and the more specialized sparse index. We inserted a row with 5 values into a table with different numbers of indexes. Figure 10 demonstrates that the partial index had higher overheads associated with predicate evaluation – even when there were few inserts into the indexes, the predicates of all partial indexes had to be checked to see if an update was needed. Toward the right end of the graph, we see that the partial indexes incurred more costs as the table had more indexes. This result supports our claim that sparse indexes perform better than generic partial indexes and warrant a separate implementation.

6.2 Hidden Schemas

We begin by exploring whether the patterns of attribute co-occurrence in our real-world data sets can really be exploited to find hidden schemas. We performed experiments on both the CNET and the EBuild data sets.

Figure 11 shows the adjacency matrix of the attributes of EBuild before applying the k-NN clustering algorithm, whereas

Figure 12 shows the adjacency matrix of the same set of attributes after clustering the attributes by co-occurrence (with k = 13). The lines dividing the matrices indicate the clusters of attributes. We can observe the values of the Jaccard coefficient in gradation of intensity. As the Jaccard coefficient goes to one, the color gets darker; as the coefficient goes to zero, the color approaches white. The clear pattern of blocks on the diagonal in Figure 12 means that we have found very strong clusters based on co-occurrence.

Tables 2 and 3 list five groupings of attributes with high average Jaccard coefficients, for CNET (with k = 406) and EBuild (with k = 203), respectively. These partitions have very low NullRatios. Also, about 83% of all partitions from CNET, and about 88% of all partitions from Ebuild, have a NullRatio less than 0.5. The CNET and EBuild data set have 233,304 and 302,631 rows respectively; in comparison, the sizes of the partitions in Tables 2 and 3 are much smaller.

We also observed if the partitions in these hidden schemas make semantic sense. Judging on semantic quality is highly subjective, but these partitions mostly make semantic sense to the authors. The clusters in CNET are related to printers, projectors, hardware storage, cameras, and speakers. The clusters in EBuild are related to toilets, vehicles, cabinets, whirlpools, and small refrigerators.

We now turn to experiments that explore the impact of the structure of hidden schemas on scan times and maintenance overhead. Recall that our goal is to form views by clustering co-occurring attributes, and that these views will speed query evaluation even more than views defined over attributes that have no special co-occurrence pattern. To test our claim, we created six synthetic data sets that had different values of NullRatio. Each data set was stored in a base table that had 110 attributes of type varchar(16). We created 11 materialized views, each with 10 consecutive attributes, on the base table. The data sets had 100k rows and each row had 10 non-null values. To determine which 10 attributes were non-null for a row, we first picked a view, then assigned non-null values to c' random attributes that were in the view and 10-c' attributes that were not. The value of c' was 0, 2, 4, 8, and 10 for the six data sets. The base table was stored with interpreted storage, whereas the views were stored with positional storage, which we considered to be the common approach because we expected the views to be relatively dense. Figure 13 shows the actual scanning times for the views. As expected, the view with a zero NullRatio had the best performance and the scanning time increased with increasing NullRatio.

To observe how data distribution affects maintenance costs, we created single-attribute views and compared them with the 10-attribute views. In this experiment, we inserted tuples with 10 non-null attributes that belonged to different number of views. Figure 14 shows the result. The single-attribute views represent the worst-case scenario because inserting 10 non-null values always triggered 10 view updates. For the 10-attribute views, the update costs were low when the non-null attributes belonged to few views, and increased when the attributes were in more views. The result shows that given a hidden schema, whether it incurs low maintenance costs depends on whether the data distribution of the new tuples conforms to the partitions of the hidden schema.

We studied the scan performance of the views for EBuild. We wanted to observe how the NullRatio of the actual views correlated with the percentage improvement of the actual views to the worst-case views, which had no co-occurring attributes. To obtain the worst-case views, we created a table with the same schema, projected the values of each attribute individually along with null

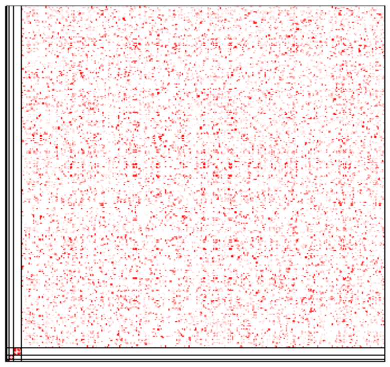


Figure 11. Adjacency matrix of attributes from EBuild before applying k-NN.

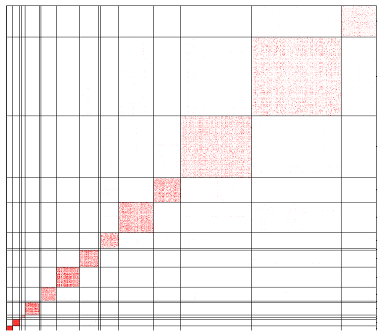


Figure 12. Adjacency matrix of attributes from EBuild after applying k-NN with k = 13.

Row Count	Average Jaccard	Null Ratio	Attributes in Cluster
1423	0.944	0.007	printer output type, printer type, media feeder(s), media type, printer output ...
346	0.932	0.015	audio output type, input device type, projector image brightness ...
3116	0.949	0.012	configuration device type, device type, hard drive size, storage controller type...
442	0.984	0.002	camera flash type, connections type, lens systems type, still image format ...
125	0.860	0.005	speaker form factor, speaker qty, speaker driver diameter, speaker type ...

Table 2. Five attribute groupings from k-NN for CNET with k = 406.

Row Count	Average Jaccard	Null Ratio	Attributes in Cluster
5894	0.874	0.060	bowl style, flushing system, gallons per flush, mounting type, rough-in ...
2707	0.816	0.042	convenience features, gross weight, ground clearance, maximum payload, maximum torque, sound system ...
10729	0.035	0.035	cabinet construction, door mount detail, door style, interior cabinet finish ...
44112	0.105	0.105	outlet position, no. of jets, no. of pumps, package type, pump horsepower...
320	0.075	0.075	bottle storage capacity, temperature range, storage capacity...

Table 3. Five attribute groupings form k-NN for Ebuild with k = 203.

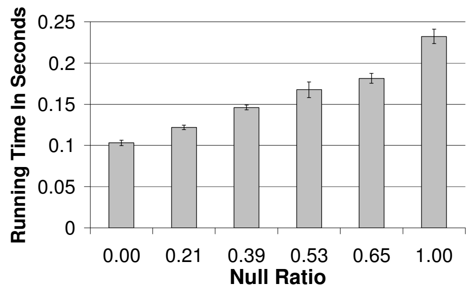


Figure 13. Scan times for 10-attribute views with different NullRatios.

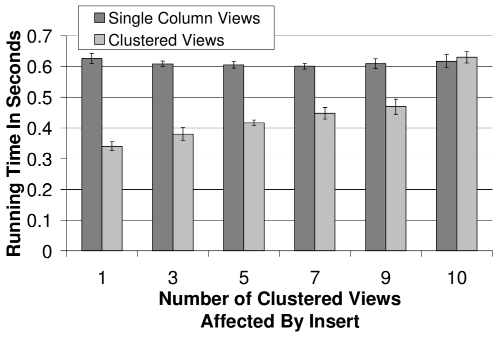


Figure 14. Insert cost depends on how many views are affected.

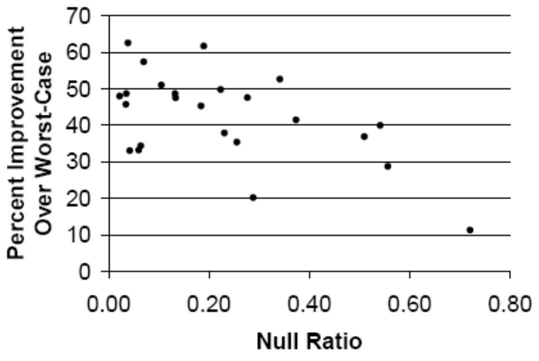


Figure 15. Percentage speedup of clustered views to the worst-case view for Ebuild data (k = 203).

values for the rest of the attributes, and combined the result of each projection into one table. For example, the following SQL statement creates a two-column worst-case view WorstCase over attributes A_1 and A_2 of table H:

```
INSERT INTO WorstCase
SELECT * FROM
((SELECT oid, A₁, NULL as A₂ FROM H)
UNION ALL
(SELECT oid, NULL as A₁, A₂ FROM H))
```

The result in Figure 15 verifies our claim that views with lower NullRatio results in better performance. Moreover, most points in the figure appeared in the upper left-hand corner, which means that the clusters found by the k-NN algorithm led to tightly-packed views. Next, we compared the scanning times of the views to that of the base table for EBuild. Scanning the base table that used interpreted storage took 15.247 second, whereas the largest view suggested by k-NN took 1.84 seconds, and a majority of the views took no more than 1 second.

7. RELATED WORK

In pioneering work on the sparse data problem, Agrawal et al. discussed using vertical tables as an alternative to horizontal tables (with positional storage) for handling sparse data [3]. More recently, comparing these approaches to using horizontal tables with interpreted storage, Beckmann et al. [6] concluded that interpreted storage outperformed both positionally stored horizontal and vertical tables. The paper did not address schema design issues and the impact of indexes and views with respect to supporting efficient query construction and evaluation over sparse data sets.

There is a large body of work on keyword search over relational [13] and XML data [12, 14]. For relational data, the focus is on efficiently joining tuples that have at least one keyword. Our approach does not have this problem because the data set is stored in a single base table. To improve the quality of keyword search, some previous work proposed to support metadata hints and exploit the hierarchical structure of XML documents [14], but the latter option is not as useful for the flat relational model.

Agrawal et al. [4] noted that when a numeric data set has low-reflectivity, correspondence between attribute names and values becomes less important for finding the right answer because a query with multiple numbers is likely to have few interpretations. For example, in a completely non-reflective, 2-dimensional data set, the query "1 and 3" can only refer to the point "x = 1 and y = 3" for attributes x and y because its reflection, the point "x = 3 and y = 1," does not exist. Low-reflectivity and Zipf-like Row-Num and Attr-Num distributions have similar implications in that keyword search over data sets with these qualities tend to get high-precision results. However, reflectivity applies to only numeric values, whereas the term distributions apply to all terms.

Oracle [5] implements indexes that do not store null values, but to our knowledge no published literature has evaluated the performance of these indexes on sparse data sets.

Vertical partitioning is a well-studied optimization technique [16]. Our work is the first that considers it in the context of sparse data. It is similar to the work by Edmonds et al. [11], which described a scalable algorithm to find empty rectangles in 2-dimentional data sets; however, the latter was meant as a complementary data mining approach, rather than for query optimization.

Column-based storage techniques, such as C-Store [20], take vertical partitioning to an extreme by projecting all individual columns from the tables. Though not specific to sparse data, Abadi [1] discussed some schema-design constraints similar to the problems described in Section 2, and argued that these constraints are no longer valid if a column-oriented layout is used. C-Store is optimized for read-mostly workloads, whereas our approach makes no assumption about workloads.

8. CONCLUSION

The management of sparse data sets is a challenge for relational database systems. To our knowledge, the research literature has not addressed this challenge beyond studying the problem of how to efficiently store such data sets. In this paper, we argue that the initially unappealing approach of "stuffing" the sparse data set into a very wide, very sparse table, is actually an attractive alternative. The approach is initially unappealing for a variety of reasons, including the discouraging prospects of data storage explosion, lack of indexability, slow scans of tuples that are full of attributes not requested by the query, and the unfortunate prospect of asking users to select from thousands of attributes when building their queries. Fortunately, the previously proposed interpreted storage format removes the storage explosion problem. In this paper, we show that sparse indexes and materialized views over an automatically discovered hidden schema can solve the indexability and inefficient scan issues.

The issue of helping users query this kind of data remains. We have argued that a combination of keyword search, "fuzzy" SQL, and a directory of attributes based on the hidden schema can assist users in building their queries. Of course, like all user interface work, the utility of our approach cannot be proven definitively without a user study involving real-world users, data, and workloads. Even in the absence of such a study, we think there is reason to be optimistic, because our proposed approach exploits techniques that have been proven to work in other aspects of data management, such as data integration and information retrieval.

ACKNOWLEDGEMENT

This work was funded by National Science Foundation Award SCI-0515491.

REFERENCES

[1] D. Abadi. Redefining Physical Data Independence. To appear in CIDR 2007.

[2] E. Agichtein, L. Gravano: Querying Text Databases for Efficient Information Extraction. ICDE 2003: 113-124.

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

[4] R. Agrawal, R. Srikant. Searching with Numbers. WWW2002.

[5] R. Baylis. Oracle Database Administrator's Guide, 10g, 2003.

[6] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In Proc. of ICDE, 2006.

[7] N. Chapin. A Comparison of File Organization Techniques. In Proc. of 24th national conference, pg. 273-283, USA, 1969. ACM Press.

[8] S. Chaudhuri, V. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In VLDB, 1997.

[9] CLUstering TOolkit (CLUTO). WWW, available at: http://www.cs.umn.edu/karypis/cluto.

[10] CNET Networks, Inc. Product Directory. http://shoppper.cnet.com.

[11] J. Edmonds, J Gryz, D. Liang, R. Miller. Mining for Empty Rectangles in Large Data Sets. ICDT 2001: 174-188.

[12] D. Florescu, D. Kossmann, I. Manolescu, "Integrating Keyword Search into XML Query Processing", WWW Conf., 2000.

[13] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *Proc. of VLDB*, 2002.

[14] Y. Li, C. Yu, H. Jagadish. Schema-Free XQuery. In *VLDB*, 2004.

[15] D. Maier, J. Ullman. Maximal Objects and the Semantics of Universal Relation Databases. ACM Trans. Database Syst., 1983.

[16] S. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. ACM Trans. Database Syst., 9(4):680-710, 1984.

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

[18] E. Rahm, P. A. Bernstein, A survey of approaches to automatic schema matching. VLDB Journal 10, 4 (Dec. 2001), pp. 334-350.

[19] R. Raman, M. Livny, and M. H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, 1998.

[20] M. Stonebraker et al. C-Store: a Column-Oriented DBMS. In VLDB 2005.

[21] M. Stonebraker. The Case for Partial Indexes. SIGMOD Rec., 18(4):4-11, 1989.