Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format

Jennifer L. Beckmann Alan Halverson Rajasekar Krishnamurthy*
Jeffrey F. Naughton

University of Wisconsin
Madison, WI
{jbeckmann,alanh,naughton}@cs.wisc.edu

IBM Research—Almaden*
San Jose, CA
rajase@us.ibm.com

Abstract

"Sparse" data, in which relations have many attributes that are null *for most tuples, presents a challenge for relational database management systems. If one uses the normal "horizontal" schema to store such data sets in any of the three leading commercial RDBMS, the result is tables that occupy vast amounts of storage, most of which is devoted to* nulls*. If one attempts to avoid this storage blowup by using a "vertical" schema, the storage utilization is indeed better, but query performance is orders of magnitude slower for certain classes of queries. In this paper, we argue that the proper way to handle sparse data is not to use a vertical schema, but rather to extend the RDBMS tuple storage format to allow the representation of sparse attributes as* interpreted *fields. The addition of interpreted storage allows for efficient and transparent querying of sparse data, uniform access to all attributes, and schema scalability. We show, through an implementation in PostgreSQL, that the interpreted storage approach dominates in query efficiency and ease-of-use over the current horizontal storage and vertical schema approaches over a wide range of queries and sparse data sets.*

1 Introduction

"Sparse" data arises in a variety of applications. As one example, in Condor [18], a distributed workload management system, any user can define new attributes for any job they submit. The result is a data set in which more than half of the attributes are *null* in all but a handful of tuples. While Condor may seem like a special case, sparse data also arises in e-commerce datasets, where each participant may declare their own idiosyncratic attributes for products and work orders, which result in data that has thousands of attributes (Agrawal et al. [1] deal with nearly 5,000 attributes), most of which are *null* for particular entities; in medical information systems [4, 9], where only a small subset of the universe of attributes apply to any given patient; in customer demographic datasets, where a wide variety of attributes are stored on each customer (Pyle [15] mentions a brokerage firm with over 700 attributes, half of which are *null* in 98% of the entities); and in any other similar application where entity sets can have many attributes only a few of which apply to a given entity.

Horizontal

Oid	A1	A2	A3	A4
1			v4	v2
2		v1		
3	v3		v1	
4	v1	v4		
5			v5	

Vertical

Oid	Attr	Value
3	A1	v3
4	A1	v1
2	A2	v1
4	A2	v4
1	A3	v4
3	A3	v1
5	A3	v5
1	A4	v2

Figure 1. A sparse dataset represented in the horizontal and vertical schema alternatives.

The sheer number of possible attributes and number of undefined values that sparse applications demand presents a question of how such data should be dealt with in a relational database management system. The most natural and straightforward approach, commonly referred to as the "horizontal" schema approach, is to map the entity set to a table, with the attributes of the table comprising all the attributes that apply to any of the entities stored in the table. It is natural because it handles the data as a traditional relational table and provides a familiar SQL interface along with all of the advantages of storing relational data in an RDBMS (such as query optimization, statistics collection, column and multi-column indexes, and table constraints).

The problem with the horizontal schema for sparse data is that in any given row, most of the attributes will be *null*, and while commercial systems employ a number of techniques to reduce the space occupied by *nulls*, they do not reduce this space to nothing. The result is large tables that are predominantly filled with *nulls*, and hence are slow to scan and that "pollute" the buffer pool with empty data.

In response to deficiencies in current horizontal storage, Agrawal et al. [1] investigated an alternative known

as the "vertical" schema. In the vertical schema, a single row in a horizontal table is split into multiple rows. For example, if the horizontal schema is $R(A_1, \ldots, A_n)$, the vertical format will have the schema $R_v(oid, attr, val)$; a tuple $(v_1, \ldots, v_n)$ would be stored in n tuples $(oid, A_1, v_1), (oid, A_2, v_2), \ldots, (oid, A_n, v_n)$. In the vertical table, only non-*null* values need to be stored, so sparse data storage in the vertical schema requires less space than the horizontal table.

The vertical schema takes an approach similar to the decomposed storage model (DSM) [7], which decomposes horizontal tables into many 2-ary relations, one for each column in the relation. However, DSM also traditionally stores *nulls* and helps to accelerate horizontal queries over dense data that reference few attributes, which is orthogonal to the topic of this paper. Unlike horizontal storage, DSM and vertical decouple the logical and physical storage of entities. Unfortunately, the space savings of vertical for sparse data does not come for free, as any operation that requires the reconstruction of some fragment of the original horizontal tuple by requesting several attributes (either as part of the query evaluation or in order to provide a result to a client) is expensive. In our experiments, we saw that in some cases this reconstruction of fragments of horizontal tuples generates such complex queries that at least two commercial systems refuse to execute them, where one responds with the message "query too complicated."

We show how the contributions of this paper fit in with previous work with a quad chart in Figure 2. Note that the quad chart mixes two different aspects of relational storage: what is the schema (horizontal or vertical or DSM) and what is the underlying representation (positional or interpreted). We use this quad chart because, as the rest of this paper will demonstrate, the choice of underlying storage representation has such a profound effect on performance that the choice of schema cannot be properly evaluated without it. In fact, one of our main conclusions is that with interpreted storage, previously proposed conclusions about which schema is good for sparse data are no longer valid.

Our contribution falls in the upper right corner of the chart that targets minimizing reconstruction of logical tuples while at the same time not allocating any space to *null* values by considering a (minor) change to the traditional RDBMS tuple storage format. The "interpreted" storage format differs from standard horizontal *positional* formats in that the association between a data value and its attribute is represented by tagging the data value, rather than by its position in the tuple schema. That is, when storing a value v_i for some attribute A_i, we store the tag A_i along with the value v_i in the tuple. The interpreted format is a lot like vertical storage because it stores only non-*null* data values (*null* values take zero space), but like horizontal positional storage, it ties together logical and physical storage of the tuple

Null values occupy space	Expensive Reconstruction of Tuples: Yes	Expensive Reconstruction of Tuples: No
No	Vertical [1]	Interpreted [this paper]
Yes	DSM [7]	Positional [current systems]

Figure 2. The alternatives for storing sparse data.

and eliminates the need to do expensive reconstruction of entities.

The main contribution of this paper is the evaluation of the interpreted storage format as an extension to modern RDBMSs for handling sparse data. We show through a prototype implementation in PostgreSQL that at the storage level, existing systems can easily be extended to use the format. We demonstrate that the format is the best option for

- Intermediate results from queries over vertical tables
- Table storage for queries that require vertical results
- Table storage for queries that require horizontal results

Our evaluation looks the approaches of horizontal and vertical schema solutions and the current physical storage options in RDBMSs (Section 2). As an alternative to the traditional storage techniques, the interpreted storage representation provides better scalability for sparse data (Section 3). Finally, our experiments show a detailed performance analysis of the storage options on synthetic datasets (Section 4) that are based on statistics collected from the online catalog CNET [6].

To the best of our knowledge, the interpreted format has never been evaluated in the published literature and implemented as an optimization for handling sparse data in a RDBMS. Gray and Reuter [11] give the interpreted representation a cursory mention, but do not evaluate it as a storage solution. Finally, the interpreted format is similar to the sparse matrix representations studied outside the context of RDBMSs [5, 20].

2 Current RDBMSs and Sparsity: The State of the Art

When discussing how to store sparse data in relational database systems, two orthogonal issues arise. The first is whether to use a horizontal or vertical schema when mapping entities to tables. The second, which to date has not been considered in the context of sparse data handling, is the underlying physical storage of attributes in tuples. In this section we consider each issue in turn.

2.1 Horizontal vs. Vertical Schemas

A generic horizontal table schema for a sparse data set with **C** attributes with varying types is

Select	Project

Horizontal
 	
`SELECT *`
`FROM H`
`WHERE A1='v1'`
`AND A2='v2'`
`AND ...`
`AND Aj='vj'`
	Horizontal
	
	`SELECT A1, ..., Ak`
	`FROM H`

Vertical
 	
`SELECT *`
`FROM V`
`WHERE oid IN`
`(SELECT V1.oid`
`FROM V V1, ..., V Vj`
`WHERE (V1.attr = 'A1'`
`AND V1.value = 'v1')`
`AND ...`
`AND (Vj.attr = 'Aj'`
`AND Vj.value = 'vj')`
`AND (V1.oid = V2.oid`
`AND V1.oid = V3.oid`
`AND ...`
`AND V1.oid = Vj.oid))`
	Vertical
	
	`SELECT *`
	`FROM V`
	`WHERE attr = 'A1'`
	`OR attr = 'A2'`
	`OR ...`
	`OR attr = 'Ak'`

Figure 3. Select and project queries for horizontal and vertical.

```
H(oid     INTEGER NOT NULL,
  A1      TYPE-OF A1,
  A2      TYPE-OF A2,
  ...
  AC      TYPE-OF AC)
```

In the corresponding vertical schema, each entity in general gets mapped to a number of rows in a table. Each row in a vertical table contains an object identifier (intuitively, this identifier says to which entity this row belongs) and an (attribute name, value) pair. The relational schema for the vertical approach is the same for all data, and it looks like this:

```
V(oid     INTEGER NOT NULL,
  attr    CHAR(sz_attr) NOT NULL,
  value   TYPE NOT NULL)
```

As evident from the schema, datatypes are a source of problems for a vertical schema because only one column is provided for the values of attributes with differing types. Applications that deal with vertical tables employ ad-hoc solutions for types by defining multiple type-specific vertical tables or by interpreting the type of values at runtime. For simplicity of exposition, we assume that all values have the same SQL type and that there is only one vertical table.

One consequence of the vertical schema is that queries that are simple over the horizontal schema become complicated in the vertical approach. Figure 3 demonstrates the differences between horizontal and vertical queries where all attributes are of the same SQL type. Notice that simple projection queries over a horizontal table are transformed into selection queries over a vertical table. The projection query for vertical retrieves, for each entity selected in the query, the rows corresponding to the non-*null* attributes specified in the query. A select query over a vertical table is complex because it has to retrieve all of the attributes of an entity that match the selection predicate. From here on, again for clarity of exposition, when we use "select-project" over a vertical table, we mean the query that is the equivalent over a horizontal schema table, even though when applied to a vertical schema the equivalent query selects tuples with a complex predicate and projects all three columns.

2.1.1 Vertical-to-Horizontal (V2H) Translation

The queries in Figure 3 return the result of a query over a vertical table as a set of "vertical" tuples. Leaving data in this format may be acceptable to applications that expect their data to arrive in this form. However, if an application expects the more standard horizontal form for the answers to its queries, the RDBMS must perform extra processing to convert the set of vertical tuples to the equivalent horizontal tuples. Similarly, if the query writer "thinks" of the data as a horizontal table, they will have to expend substantial effort translating their queries over this horizontal table to their vertical equivalents.

We call this process of viewing vertical tables as if they were horizontal the "V2H" approach. Agrawal et al. proposed this approach and advocated hiding the complexity by implementing special functions to do the translation. There are two main approaches to V2H translation in the published literature, the first we call the left-outer-join (LOJ) approach [1] and the other is called PIVOT [8]. We present the SQL to make the conversion explicit, since it has performance implications; in a "real" implementation of these special functions it would be desirable to hide the intricacies of the vertical queries from users, although these functions would do little to improve performance (because they do not reduce the complexity of the vertical query itself).

LOJ Translation. The LOJ approach to V2H translation takes a vertical view of a data and constructs the equivalent horizontal table by projecting each attribute separately from a vertical table and then joining all of the columns to construct a horizontal table. By using the `oid` in each vertical row, joins "bring together" the attributes that have been spread over multiple vertical tuples. We illustrate this through a simple projection of a table that has two columns, A1 and A2. The projection of this table using LOJ is

```
SELECT A1, A2
FROM
     (SELECT DISTINCT oid FROM V) AS t0
     LEFT OUTER JOIN
     (SELECT oid, value AS A1
      FROM V WHERE attr = 'A1') AS t1
     ON t0.oid = t1.oid
     LEFT OUTER JOIN
     (SELECT oid, value AS A2
      FROM V WHERE attr = 'A2') AS t2
     ON t0.oid = t2.oid
```

Left outer joins are the key to constructing a horizontal row. Like natural joins, they return tuples that match the

predicate; but they also return any row from the first table that has no matching rows from the second table and return the non-matching rows from the second table as *null* values. The process starts by obtaining all of the possible `oids` in the resulting horizontal table, which insures that all tuples are represented in the result. A query that projects more columns simply adds left outer joins in a similar way.

An approach similar to vertical with LOJ is the Decomposed Storage Model (DSM) [7] because both LOJ and DSM reconstruct horizontal tables by merging vertical partitions of the data on a surrogate. DSM is also referred to as the *binary format* in literature and it has also been used in IBM's Enterprise Directory LDAP product [19]. Comparisons of the DSM model with horizontal storage on dense data [3, 10, 12, 13, 17] have shown DSM to be more efficient for queries that use a small number of attributes. For sparse data, Agrawal et al. compare vertical storage and LOJ reconstruction to a binary storage solution where *null* values are not represented in the 2-ary relations and found that LOJ performs similarly to the binary storage approach.

PIVOT Translation. As an alternative to LOJ construction of horizontal tuples, Cunningham et al. [8] define a PIVOT scheme. One way to implement PIVOT is to use group-by and aggregation to produce a horizontal version of a tuple. A PIVOT query that produces a two column horizontal projection of data from a vertical table is

```
SELECT oid,
  MAX(CASE WHERE attr='A1' THEN value ELSE null) as A1,
  MAX(CASE WHERE attr='A2' THEN value ELSE null) as A2
FROM V
GROUP BY oid
```

In this query, the group by collects all rows corresponding to one entity. Next, the query matches each row in the entity against the MAX aggregates (which assume that *null* is smallest of all values). Each aggregate produces the value for each matching attribute and *null* otherwise. Cunningham et al. show that group-by optimizations can be extended to PIVOT when it is implemented inside an RDBMS.

LOJ and PIVOT Performance. The performance difference between PIVOT and LOJ reconstruction of attributes is on two levels: the processing order of tuples and the types of access methods used to perform the query. PIVOT requires an `oid` grouping and localizes its reconstruction to within groups of `oid`. Comparatively, the best strategy for LOJ requires `attr` ordering to select portions of the vertical table that correspond to the attributes being projected and then in the merge phase, it uses the `oid` ordering. Thus, an (`attr`, `oid`) clustering is best for LOJ.

Although it seems that the PIVOT operator would perform best when clustered on `oid`, it is not the case for classes of queries that project attributes from a vertical table or select a few entities. For example, most horizontal queries over a vertical schema queries project attributes by using predicates such as `attr = 'A1'`. When the number of attributes projected in the query is small, the best clustering for the vertical table is on `attr` for both LOJ and PIVOT. We return to the clustering issues in section 4.3.

2.1.2 Horizontal-to-Vertical Translation

Even though a horizontal representation has many benefits over a vertical schema for managing and querying sparse data in an RDBMS, queries that return results in a vertical format may be what some applications prefer to handle rather than wide horizontal results with many *null* values. H2V translation converts a horizontal table into a vertical table and is conceptually defined as the union of the projection of each attribute in a horizontal table. For example, a two column H2V translation is

```
SELECT oid, 'A1', A1 FROM H where A1 is not null
UNION ALL
SELECT oid, 'A2', A2 FROM H where A2 is not null
```

Agrawal et al. and Cunningham et al. (who defines a more general operator called UNPIVOT) both define H2V as the union of projections of a horizontal table. The semantics for the `value` column is that the types must be union compatible across the projected columns. The H2V operator is useful in situations where legacy applications expect vertical results from queries. We also consider H2V as an option and Section 4.3.2 shows that H2V has the best performance in some situations.

2.2 Current Physical Storage Options

In this section we consider the physical storage in current relational systems, specifically those found in the leading commercial vendors and in the open-source database PostgreSQL. In general, these formats have inherent inefficiencies when scaling to large numbers of sparse attributes.

2.2.1 Commercial Systems Positional Format

Most commercial RDBMS [11, 16] use a positional layout for their relational records. While the exact details vary from system to system, the layout of a positional record begins with a tuple header that might include fields such as the relation-id, tuple-id, and tuple length. Next is the *null*-bitmap, which indicates the fields that are *null*. The fixed-width data follows the *null*-bitmap where the tuple has a fixed amount of storage pre-allocated for each fixed-width attribute, regardless of whether the values in the tuple are *null* are not. Finally, an array of variable width offsets point to and precede the variable width data.

The positional format is tuned to dense data and allows for quick access to the values of attributes, since they are either located by a fixed offset from the start of the record (for fixed-width data), or by adding to the start of the record an offset that is located at a fixed position in the record (for variable-width data). The system catalog maintains the mapping from attribute name to value within a record by recording the order of attributes in the record.

Sparse data sets present a challenge for this kind of storage, because the *null* values take up space. A *null* value for a fixed-width attribute takes up a bit in the *null*-bitmap and the full size of the attribute (e.g., a *null* four-byte integer field takes four-bytes); a *null* value for a variable-width attribute takes a bit in the *null*-bitmap and a pointer in the record header. Commercial RDBMSs employ some techniques to reduce this overhead somewhat. Indeed, commercial improvements in *null* handling have shifted the tradeoffs between the vertical and horizontal schema approaches; and the conclusion in Agrawal et al. [1] that vertical "uniformly outperforms horizontal" is no longer valid (we will return to this issue in Section 4). Still, the cost of storing a *null* is not zero, and for that reason we claim that the interpreted format is a valuable addition to relational storage techniques.

2.2.2 PostgreSQL Bitmap-Only Format

PostgreSQL has a format not currently used by any of the three leading commercial systems [14]. We call the PostgreSQL storage strategy the bitmap-only scheme. In a bitmap-only scheme, the tuple also has a header with typical tuple information and a *null*-bitmap, which indicates the fields that are *null*. But instead of pre-allocating the space for all of the attributes, the data portion of the record only contains the non-*null* values.

The retrieval of a value for a bitmap-only representation is more complex than the pre-allocated positional record format because the location of the value of an attribute is different in each tuple and, thus, the location of attribute values are not known prior to query execution. A request of an non-*null* attribute A_n from a tuple requires knowledge of data-lengths of non-*null* fields in the prior $n-1$ attributes of the record. The algorithm uses the datatype information in the catalog to determine the length of the prior non-*null* attributes and uses the aggregate of their sizes to find the position of the requested attribute.

Although the bitmap-only representation of tuples helps with the sparse data because it does not pre-allocate space for *null* attributes, we will show that the solution does not scale to the number of attributes many sparse applications demand. Allocating any amount of space to *null* values has an effect on scalability—even a bit per attribute is costly in tables with hundreds, or especially thousands, of possible attributes.

3 Toward RDBMS Support for Sparsity: The Interpreted Format

The positional and bitmap-only approaches to physical tuple storage are intimately tied to the number of attributes defined in a table schema, which fundamentally limits both of these formats from being able to scale to large numbers of sparse attributes. In order to efficiently scale to applications that require hundreds or even thousands of sparse attributes, RDBMSs need to provide an alternate storage format that is independent of schema width. In this section we present the interpreted storage format, which stores only non-*null* values and performs well on sparse data, as experiments in Section 4 show.

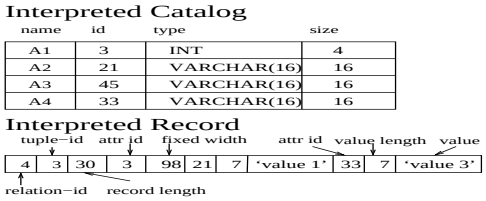


Figure 4. Interpreted record layout and corresponding catalog information.

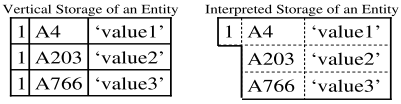


Figure 5. The relationship between the vertical and interpreted storage options.

3.1 Interpreted Format

In contrast to a positional representations, the interpreted record format stores a list of attribute-value pairs. Figure 4 shows a representative interpreted format that starts with a header, which, as in the positional notation, contains fields such as relation-id, tuple-id, and a tuple length. When a tuple has a value for an attribute, the attribute identifier, a length field (if the type is of variable length), and the value appear in the tuple. The attribute identifier is the id of the attribute in the system catalog (in general the id is much smaller than the full attribute name). Attributes that appear in the system catalog, but not in the tuple, are *null* for that tuple.

Since the interpreted format stores nothing for *null* attributes, sparse data sets in a horizontal schema can in general be stored much more compactly in the format. In practice, if some attributes are dense and others sparse, a system would use a record format in which the dense attributes are stored positionally toward the beginning of the record and are followed by an interpreted section that contains any sparse attributes that are defined for the tuple.

The interpreted format can either be viewed as a storage option for the horizontal approach, or as an optimization of the vertical approach. Figure 5 shows an entity stored vertically and as an interpreted tuple. Note that both store the same repeating "attr, value" pairs. The primary distinction is that (a) in interpreted, all the pairs are viewed as a single object so there is no need to tie them together with a common tuple id or reconstruct the tuple during query evaluation; (b) in interpreted, the attributes are collected together as one object, and in contrast, the vertical entity is a set of independent tuples that can be organized (or clustered) in any order; and (c) in interpreted, the system catalog records the

attribute names, whereas in the vertical format these names must be managed externally by the application.

While the interpreted format clearly has storage benefits for sparse data, retrieving the values from attributes in tuples is more complex. In fact, the format is called *interpreted* because the storage system must discover the attributes and values of a tuple at tuple-access time, rather than using precompiled position information from a catalog, as the positional format allows. For this purpose we introduce a new operator, called EXTRACT. In a query plan the EXTRACT operator precedes any reference to attributes stored in the interpreted format. It returns the offsets to the referenced interpreted attribute values, and these resulting offsets are then used to retrieve the values.

Value extraction from an interpreted record is a potentially expensive operation that is dependent on the number attributes stored in a row, or the length of the tuple. Also, if a query evaluation plan fetches each attribute individually, with one EXTRACT call per attribute, the record will be scanned for each attribute, which will be very slow unless only a few attributes are required. It is far more efficient, therefore, to batch extract attributes and find offsets for multiple attributes in one record scan. We explore this issue in Section 4.2.1. In practice, systems already employ techniques for retrieving batches of attributes from tuples [11] and the EXTRACT operator is an extension of such an optimization.

4 Experimental Evaluation

In the next sections we define the experimental setup, synthetic data sets, and our PostgreSQL implementation. Section 4.3 shows that vertical tables can benefit from interpreted storage by storing reconstructed fragments of the original horizontal tuple in the interpreted format. In Section 4.3.2, we show that even if applications want query results in a vertical form, they can get the best performance when they store data in a interpreted horizontal format and convert it to vertical "on the way out." Finally, of the horizontal storage options, we show that interpreted is the best format for storing and querying horizontal tables in Section 4.4.

4.1 Experimental Setup

We report on experiments with different tuple formats and schemas for storing sparse data. We use three PostgreSQL v.8.0.1 systems, the first is the unmodified PostgreSQL, which supports the bitmap-only scheme natively; the second is PostgreSQL with our modifications for it to use a positional storage format; and the final one is PostgreSQL modified with the interpreted storage format.

Our H2V implementation transforms horizontal tuple by scanning a record and producing a vertical tuple for each present attribute with a value. The transformation happens after execution of the horizontal query. Similarly, for the V2H translation, PIVOT, we use an implementation that uses the optimizations presented by Cunningham et al. and perform the PIVOT after the execution of the vertical query.

Our experimental platform was a Tao Linux 1.0 system running on a 2.4 GHz Intel Pentium machine with 512 MB of physical memory. We clustered horizontal tables by the `oid` field. For comparison, we also stored two separate vertical tables, one clustered on (`attr`, `oid`) and the other clustered on `oid`. We indexed every horizontal column involved in a predicate and created single-column indexes on each of column of the vertical tables. The buffer pool size was set at 64 MB.

We performed experiments with a cold buffer pool where the filesystem was unmounted and remounted before each run. We ran queries 5 times and when we report exact times, they include a 95% confidence interval. We measured running times for the execution of the queries within the engine and do not include the time to output the results.

Although we experimented with a warm buffer pool, the results did not provide any additional insight. We saw a substantial difference in the positional format compared to the other formats because in the positional format, the data sets were all bigger than memory itself. The relative performance between the bitmap and interpreted formats were similar to the cold performance. Although we expected that buffer pool size would affect performance of the bitmap format, PostgreSQL runs only on the native filesystem and cannot store and manage files on raw disk. Consequently, our tests also ran with a warm filesystem cache and, thus, the performances for missing in the buffer pool were low.

The fundamental difference between the horizontal storage formats is retrieval of attributes from the storage and nothing changes in the query processor. Our experiments focus on selection and projection, which in turn use table scan and index selection of tuples. We report the performance of a selection and projection in order to isolate the trade-offs of table access that happens at the leaf-levels of more complex queries. For complex queries that we ran, we found that we could predict the results by understanding the trade-offs for table scans and index selection.

4.2 Datasets

First, we explain a representative sparse e-commerce data set on which we based our synthetic data. Next, we define the synthetic parameters and data sets that we used. Finally, we look at the respective table sizes of each format over our 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

Width	Sparsity	Pos.	Bitmap	Interp.	Vertical
5	0%	73.7	67.4	75.1	153.8
320	98.4	419.8	87.1	74.8	153.8
640	99.2	781.3	106.7	74.8	153.8
1280	99.6	2000.0	146.2	74.9	153.9

Table 1. Size of tables in MB with 500k rows and 5 present values per row as stored in our PostgreSQL prototype.

System	Horizontal	Vertical
System A	345.59	83.97
System B	660.22	84.55
System C	976.72	88.02

Table 2. Size of tables in the three leading commercial systems for the dataset with 640 possible attributes, 5 present values per row, and 500k rows in MB.

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 11 and the mode is 5. More details on the data collection and statistics of the dataset can be found in [2].

We use synthetic datasets that show the performance of the approaches on a spectrum of sparsity and wide schemas. Each dataset has the same number of average present values per entity at 5, each has a constant number of entities at 500k, and there are four tables with 5, 320, 640 and 1280 attributes. The data generator randomly distributed present values across the attributes in each row. The sparsity of the synthetic data used in our experiments never exceeds that of the real world CNET data, therefore the synthetic data is not artificially sparse.

Table 1 lists the size of the respective storage schemes for the tables as stored in PostgreSQL. The table illustrates how the storage schemes scale with the number of possible attributes. The interpreted and vertical schemes scale well with more attributes and remain constant in size. However, the positional horizontal and bitmap-only horizontal schemes scale poorly because they both allocate some space to missing values.

Table 2 presents the table sizes for the three leading commercial RDBMS vendors for the dataset with 640 possible attributes. System A has a smaller footprint for the data because the data pointers only take 1 byte per attribute. Also, System A and System B use *null* compression that only preallocate space for a tuple through to the last non-*null* attribute and, thus, save space on tuples that have many *nulls* at the end of a tuple. System C compresses *nulls* by reducing size of data pointers from the 4 bytes typically used in its tables to 2 bytes when there is a *null*. Even though each of these systems try to cope with *nulls* in some way, all of the tables are more than 4.6 times larger than interpreted storage of the same table.

4.2.1 Batch vs. Once-per-attribute Extract

During query execution, the main differences between the positional, bitmap-only, and interpreted storage formats is how the attribute-values are retrieved from the tuples. PostgreSQL has two methods to extract values from tuples: one that retrieves all values of a tuple (effectively, `select *`), and one-at-a-time retrieval of attributes. When we report time to batch EXTRACT-ion of values, we use the time it takes to extract all attributes of a tuple.

Extract	Interpreted	Bitmap	Positional
Batch	2.49 ± 0.13	3.89 ± 0.3	15.51 ± 0.10
Once-per	46.73 ± 2.65	34.02 ± 4.46	39.05 ± 1.58

Table 3. Cold running time in seconds for batch and once-per-attribute extraction.

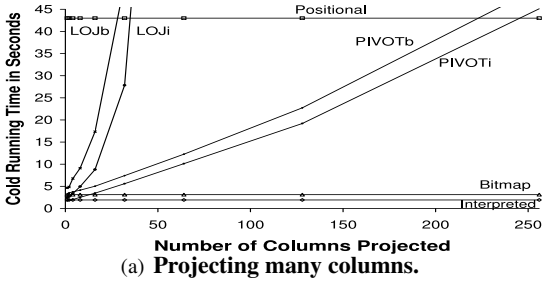
Table 3 shows the performance for projecting all columns from a table with 640 columns using batch extraction method and a once-per-attribute extraction (640 extract calls per tuple). The data show that batch extraction of attributes is far more efficient than fetching each attribute separately. Even the positional format benefits from making one function call (instead of 640) with a 60% decrease in execution time over once-per-attribute extraction. However, interpreted and bitmap-only gain the most from batch extraction leading to a 95% and 89% decrease in execution time, respectively. Bitmap is faster than interpreted in the once-per-attribute case because the bitmap representation uses the bitmap to check if an attribute is *null* and most of the attributes are *null* in a tuple. For interpreted, extracting each column individually scans the list of present values each time, thus using batch EXTRACT allows one scan of the present values and saves time. Because of the benefits of batch extraction, the rest of the experiments only consider batch-extraction of attributes.

4.3 Horizontal vs. Vertical Schema

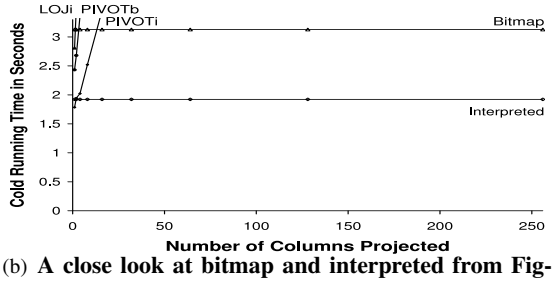
Our experimental evaluation of sparse storage begins with the differences between horizontal and vertical schemas. Recall that some applications query vertical tables as if they are horizontal tables using V2H. We present results from projection and selection queries when using V2H and compare them to the horizontal storage alternatives.

4.3.1 Returning Horizontal Results

Projection Queries and V2H. Figures 6(a) compares positioinal, bitmap, interpreted, and vertical storage for projecting columns from the table with 1280 columns. Figure 6(b) shows results from the same experiments, but takes a closer look at the performance of bitmap and interpreted.



(a) **Projecting many columns.**



(b) **A close look at bitmap and interpreted from Figure 6(a).**

Figure 6. Projecting a set of attributes from a dataset with 1280 possible attributes.

In both graphs, the number of columns projected is varied from 1 column to 256 columns. All of the horizontal queries have near constant performance across the number of columns projected because the dominant cost is the I/O to read the tables off of disk which is independent of the number of columns projected. The interpreted approach performs the best amongst all strategies and is on average 1.63 times faster than its closest rival, the bitmap-only format, and 22.36 times faster than the positional storage.

The figure shows LOJ and PIVOT both clustered on (`attr`, `oid`) and with two intermediate storage representations. The intermediate results of V2H translation must be stored in some tuple storage format, whether positional, bitmap or interpreted. The figure shows the interpreted and bitmap approaches to intermediate storage for results from LOJ and PIVOT and indicate them by LOJi (PIVOTi) and LOJb (PIVOTb), respectively. For LOJi, the results are nearly 2 times faster than the LOJb approach that stores the intermediate results in the bitmap storage format—in fact, the LOJi approach is faster than the bitmap horizontal approach up to 6 columns. In the PIVOT approach, the interpreted storage averages a 1.55 times improvement over PIVOTb. The performance of LOJi and PIVOTi indicates that even in situations when applications have stored sparse data vertically (perhaps for legacy reasons), the interpreted approach is an optimization that helps processing vertical queries over sparse data. Neither LOJ or PIVOT are ever better than the interpreted format and perform worse than positional horizontal at around 256 columns (to project all columns using LOJ takes hours and using PIVOT 160 seconds).

The figure shows the best clustering (`attr`, `oid`) for PIVOT. In a separate experiment, we verified that the (`attr`, `oid`) clustering for PIVOT is the best up to 256 columns, the running times for `oid` clustering grow from 7.5 seconds to project one column to 50 seconds for projecting 256 columns.

Figure 7 shows the projection performance for the table with 640 attributes. The smaller table width is reflected in the performance of the horizontal tables where the running time has decreased by more than half for the positional table. The bitmap performance has slightly decreased, but the

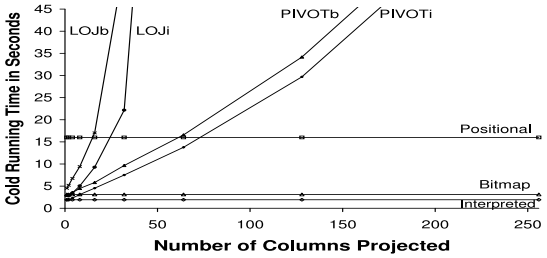


Figure 7. Projecting a set of attributes from a dataset with 640 possible attributes.

interpreted table is the least affected by the thinner table. The V2H approaches, LOJ and PIVOT, are both clustered on (`attr`, `oid`). Besides the width of the tables, the difference between the table with 640 columns and the one with 1280 columns is that the columns of the table are more dense in the 640 column table (i.e. one column in the 640 table has twice as many values as that of the 1280 table). The increase in column density is apparent in the running times of the V2H functions. With denser columns, the PIVOT approach has to group more tuples to reconstruct the horizontal results. Similarly, the LOJ approach has to process more values per attribute during the merge phase of its reconstruction. Again, we show the cost of reconstructing a horizontal table using the intermediate formats of interpreted and bitmap and interpreted is on average 1.35 times faster than bitmap for PIVOT.

Selection Queries and V2H. Figure 8 shows the running times on the dataset with 640 possible attributes for a selection query with one simple-selection predicate that projects all attributes from the data. Since the experiment projects all columns, we focus on PIVOT with interpreted intermediate results; the V2H operation that the previous section shows as best when projecting many columns. The graph shows two lines for the PIVOT operation that correspond to the clustering of the vertical table. Notice, that in the previous section, projection, the (`attr`, `oid`) clustering was best for PIVOT and now in selection the `oid` is the best clustering when returning many entities. However, the query optimizer is not able to choose the better of the two since each requires a specific physical ordering of the vertical table. Unlike the horizontal tables, for PIVOT clus-

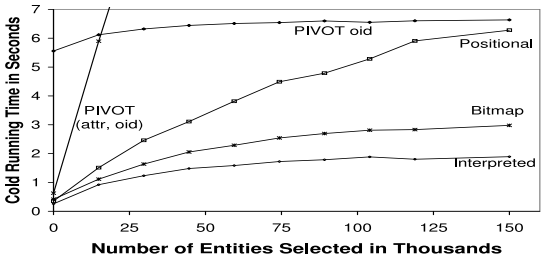


Figure 8. Selecting a set of tuples from a dataset with 640 possible attributes.

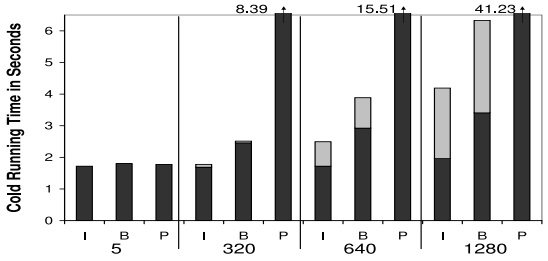


Figure 9. Projection performance where the black is time to project one column and the additional gray is the time to batch-project all columns.

tered on `oid`, the cost to select few entities is nearly the same as selecting many entities. The near constant cost is because the query plan scans the table to find the attribute occurrences and match the value predicate. The scan is the dominant cost of the query.

We now turn our focus to the performance of the horizontal tables. Again, we see that the memory footprint of the tables are also influencing the performance. In each storage format, the query plan uses an index on the requested column to fetch tuples from the table. Since the positional table is the largest table, it benefits the least from caching and each access is a random I/O. The bitmap and interpreted tables are smaller and thus, as more tuples match the predicate, matching pages are more likely to be in the buffer pool. Again, the interpreted format is 1.5 times faster than the bitmap format and up to 3.3 times faster than positional.

Query	Vertical	Interpreted	Bitmap
project all	5.63 ± 0.01	4.12 ± 0.02	6.54 ± 0.02
select 15k	6.56 ± 0.03	2.20 ± 0.02	3.14 ± 0.03

Table 4. Cold running time in seconds queries returning vertical results from the 1280 column table.

4.3.2 Returning Vertical Results

In the previous section, we saw that horizontal storage can be efficient at storing and querying sparse data. H2V translation converts a horizontal query to a vertical format "on the way out" to the application. Table 4 shows the performance for returning vertical results from the 1280 attribute data set. We show a *project* query that projects all tuples from the vertical table and a *select* query that has one simple selection predicate that selects 15k tuples and projects all attributes of an entity.

The project query simply returns the data as vertical result, which for the vertical storage is a scan over the vertical table. The table shows that the interpreted approach, although stored horizontally, is the fastest at returning vertical tuples. The performance difference is due to the relative size of the tables (the vertical table is bigger than the interpreted table). Also, the interpreted format does very little processing to convert an interpreted tuple into a vertical equivalent. On the other hand, the bitmap approach, although smaller than vertical on disk, is not faster than the vertical table scan because it has to traverse all 1280 columns to find the values that are not-*null* and transform them to vertical tuples.

In the select query, the interpreted format is faster than all others for returning results in the vertical format even though the vertical table is already in the three column format. The reason for the the poor performance of vertical is that selection over a vertical table requires a join to collect all of the attributes of the entities selected in a query (recall Figure 3). The join over the vertical table is the reason for the nearly 3 times performance improvement of V2H over vertical alone.

4.4 Horizontal Tuple Storage

The final issue that we consider is the performance of horizontal storage of tuples in positional, bitmap-only, and interpreted storage. In this section, we focus on the performance of physical storage alternatives for horizontal table storage.

Extract	Interpreted	Bitmap	Positional
First (1)	1.72 ± 0.01	2.91 ± 0.03	15.76 ± 0.11
Center (320)	1.72 ± 0.01	2.92 ± 0.003	15.42 ± 0.03
Last (640)	1.73 ± 0.02	2.95 ± 0.02	15.42 ± 0.02

Table 5. Cold running time in seconds for extracting attributes at different positions in the schema.

An issue related to value extraction is the difference between accessing just one attribute from a tuple and accessing all attributes from a tuple. In the all attribute case, each attribute of the table schema is extracted into memory and there may be extra overheads relating to that extraction. The one attribute case represents the least amount of overhead because it only extracts one value instead of all values. In the bitmap only scheme, extraction of one attribute scans the bitmap from the beginning up to the attribute requested to calculate the location of the attribute. Table 5 shows the cold execution time to extract the first attribute, the center attribute, and the last attribute in the table with 640 attributes. It shows that there is little difference among the costs to extract attributes at different positions in the bitmap.

Figure 9 presents the running times for projecting one or projecting all attributes of a table. The black part of each bar in the figure shows the execution time for projecting one column from the table and the gray portion of each bar shows the additional time for projecting all columns from the table. For the tables with less than 320 attributes, the cost to batch extract all attributes is similar to the cost to extract one attribute. However, for the table with 1280 attributes, the time to project all attributes is much higher with at least twice as much additional time over just projecting one column. The differences between projecting 320 and projecting 1280 attributes is related to the amount of memory that the extract routine has to manage, in the former it is managing 8KB and the later it is managing 22KB. Interestingly, the overheads of projecting wide tables can be reduced by projecting vertical tuples using H2V.

5 Conclusions

Relational database systems are increasingly facing the demands of applications with sparse datasets. Current relational systems are inefficient at handling sparse data sets because the underlying storage formats are inefficient at storing the data. For horizontal schema, the storage of *null* values burdens query processing. If one tries to avoid storing *nulls* by using vertical schema and storage, the performance is better for queries that access few attributes, but is poor when accessing many attributes.

In this paper we argue that the complexities of sparse data management should be handled inside an RDBMS with the addition of an interpreted storage format. When considering queries that return a horizontal result, the best option is to store the data horizontally in the interpreted format. In fact, the performance is better than if the data is stored in a vertical schema and transformed to a horizontal result. With interpreted storage, the conclusion of Agrawal et al., who concluded that a vertical schema transformed to a horizontal schema "uniformly outperforms horizontal [positional storage]," is no longer valid. Even if an application uses a vertical schema and returns horizontal results, perhaps for legacy reasons, the interpreted format is an optimization that helps to store intermediate results. Finally, when applications prefer a vertical schema view of the data, but do not constrain the underlying actual storage to be vertical, the best option is to store data using interpreted horizontal and convert to a vertical representation "on the way out" to the application.

With the addition of an interpreted data format, a database administrator would face two alternatives for the storage of the attributes: interpreted and positional. An interesting area for future work is to explore the use of a "storage wizard" to automatically decide between these formats for attributes of a data set based on density, frequency of access, and possibly other factors.

Acknowledgments. This work was supported by the National Science Foundation under grant ITR 0086002. Jennifer Beckmann was also supported by AT&T Labs Fellowship Program.

References

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

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

[3] P. Boncz, A. N. Wischut, and M. L. Kersten. Flattening an object algebra to provide performace. In *ICDE*, 1998.

[4] C. Brandt, A. Deshpande, and et al. TrialDB: A web-based Clinical Study Data Management System. In *AMIA Annu Symp Proceedings*, page 794, 2003.

[5] N. Chapin. A comparison of file organization techniques. In *Proceedings of the 1969 24th national conference*, pages 273–283. ACM Press, 1969.

[6] CNET Networks, Inc. CNET Product Directory. `http://shopper.cnet.com/4296-3000_9-0-0-0.html`, March 2005.

[7] G. P. Copeland and S. Koshafian. A decomposition storage model. In *SIGMOD*, pages 268–279, 1985.

[8] C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. Pivot and unpivot: Optimization and execution strategies in an rdbms. In *VLDB*, pages 998–1009, 2004.

[9] A. Deshpande, C. Brandt, and P. M. Nadkarni. Metadata-driven Ad Hoc Query of Patient Data: Meeting the Needs of Clinical Studies. In *AMIA Annu Symp Proceedings*, page 794, 2003.

[10] D. Florescu and D. Kossman. Storing and Querying XML Data using an RDBMS. *Data Engineering Bulletin*, 22(3), 1999.

[11] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.

[12] S. Koshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *ICDE*, pages 636–643, 1987.

[13] L. V. S. Lakshmanan, F. Sardi, and S. N. Subramanian. On efficiently implementing SchemaSQL on a SQL database system. In *VLDB*, 1999.

[14] PostgreSQL. `http://www.postgresql.org`.

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

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

[17] R. Ramamurthy, D. J. DeWitt, and Q. Su. A case for fractured mirrors. In *VLDB*, 2002.

[18] R. Raman, M. Livny, and M. H. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *HPDC*, pages 140–, 1998.

[19] S. Shi, E. Stokes, D. Byrne, C. Corn, D. Bachmann, and T. Jones. An enterprise directory solution with DB2. *IBM Systems Journal*, 39(2), 2000.

[20] R. E. Tarjan and A. C.-C. Yao. Storing a sparse table. *Commun. ACM*, 22(11):606–611, 1979.