The Tradeoff between Horizontal and Vertical Representations of Sparse Data Sets

Paper ID: 469

ABSTRACT

Sparse data is an important and challenging type of information for RDBMS to store and query. The traditional “horizontal” representation for sparse data results in tuples with a lot of null values. Observing this, in 2001, Agrawal et al. explored the alternative “vertical” representation and showed that this vertical representation “uniformly outperforms” the horizontal representation for sparse datasets. In this paper, we show that, in contrast to what was observed in 2001, in many cases the horizontal representation is substantially better than the vertical. We show that the choice of data representation depends on a number of factors including the efficiency with which nulls are handled, the sparseness of the data, and the number of attributes requested for the query output. We explore the effect of these and other parameters on the relative performance of the two formats in experiments on two commercial relational databases. We also report some problems we experienced with the databases on these workloads and offer suggestions for improvements in RDBMS to make them better able to handle sparse data sets.

1. INTRODUCTION

In the last few years, a variety of new applications, perhaps most importantly e-commerce, have brought *sparse* datasets into prominence. The main characteristic of a sparse dataset is that the space of all possible attributes for an entity is huge (several hundreds or even thousands of attributes), but each data instance may have only a small subset of these attributes (hence the name *sparse*). In relational database systems, entities are usually mapped to the rows of a relation (often referred to as a *horizontal* relation) in which there is a column for each attribute. If an entity is missing a particular attribute, that attribute in the row for the entity will be null. While this approach is straightforward, unfortunately the horizontal approach is problematic as we scale to large numbers of attributes. The problems arise for a variety of reasons, including system limitations on the number of columns in a single relation, performance issues created by overheads incurred by storing and processing null values, and difficulty handling evolving schemas.

An alternative method for storing a sparse dataset in an RDBMS is known as the *vertical* approach, in which each entity, in general, gets mapped to a number of rows in a table. Each row in this approach 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 sets, and it looks like

```
Vertical(oid integer, key varchar(x), value varchar(y))
```

Unlike the horizontal approach, the vertical approach scales to thousands of attributes, avoids storage overheads for missing values, and supports evolving schemas. Nevertheless, as we will see, writing queries over the vertical schema is cumbersome and error prone (although as proposed by Agrawal et al. in [1], this can be ameliorated by an enablement layer that presents a horizontal view of the data to the user.) However, what was an inspection of a single tuple in the horizontal approach becomes a multi-way self-join of the vertical table in the vertical approach. These self-joins “bring together” the attributes that have been spread over multiple vertical tuples. Thus a critical question is the performance implications of the differences between the two formats.

The focus of this paper is to study this performance tradeoff between the horizontal and vertical formats. We show that there is no single *best* way to handle sparse datasets in an RDBMS, because different approaches dominate under different scenarios. As shown in our experimental study, a variety of factors influence the tradeoff between the two formats, including the efficiency of null handling, how many attributes are mentioned in the queries, and properties of the dataset such as the number of columns and the degree of sparsity.

Our tradeoff analysis starts with two simple queries, projection and select-project. With these queries, we break down the plan options that a relational optimizer can consider and identify the system parameters that affect the performance of these plans. For example, as one varies the number of projected columns in the projection query, we show that the execution time for the horizontal approach is almost constant, while for the vertical approach the execution time can be expected to increase linearly with the number of columns projected. This suggests (and our experiments indeed confirm) that there should in general be a tradeoff between the formats, where the exact "break even" point occurs depends on system parameters such as the overhead for null values, the scan speed, and the incremental join cost.

To investigate whether and where this tradeoff occurs in actual systems, we use two commercial database systems, A and B[1]. We explore this issue by varying the non-null density and number of columns in the data, and see how the crossover point between the two formats changes for the above queries. An interesting side effect of these experiments is that they illustrate very clearly the dangers in drawing conclusions from single experiments, since we find that the tradeoff point is critically dependent upon the tuning of the DBMS.

One of the surprising results of our experiments is that they contrast with the results reported by Agrawal et al. [1]. In that work, the authors found that the vertical format "uniformly outperforms" the horizontal format for a wide range of datasets. In our experiments we found that the situation is not that simple, and that always choosing the vertical format can result in substantially suboptimal performance in many cases. We explain in detail why our results differ in Section 2.4. Briefly, the primary reasons are (1) the improved support for storing null values in commercial databases within the last few years has made the horizontal format more competitive, and (2) the query plans chosen by the relational database for selection queries over the horizontal relation, as reported in [1], were suboptimal.

The next part of our study illustrates how the characteristics of the query workload play an important part in the choice between the horizontal and vertical formats. We consider two additional classes of queries for this purpose. First, we consider selection queries that return all non-null attributes associated with selected objects (*"select *"* queries), and show that for this important class of query (which corresponds to "tell me everything you know about these objects") the horizontal approach looks much better, compared to the vertical approach, than it does on select-project queries that return only a small specified subset of attributes. Second, we look at join queries with the Oid field as the joining column and show how this has a profound effect on the choice of proper clustering strategy for the vertical relation.

In the process of our study, we uncovered some problems current RDBMS implementations have with sparse dataset workloads. These problems, which highlight areas in which current RDBMS should be improved if the goal is better handling of sparse data sets, include poor optimizer statistics for sparse data sets, problems with handling wide tuples, and difficulty in expressing even simple English queries over sparse data in SQL.

2. BASIC TRADEOFFS BETWEEN HORIZONTAL AND VERTICAL

In this section, using simple projection and selection queries, we compare the performance of the horizontal and vertical formats. We start by looking at candidate plans and analyze the expected trends for the two formats. We then present our experimental results, which show the effect of properties of the dataset and characteristics of the system on the crossover point between the two formats. Finally, we discuss the reasons behind why we obtain contrasting results compared to previous work [1].

[1]The names are withheld due to license restrictions.

In our experiments, the schema for the horizontal relation is

```
H(Oid INTEGER, A0 VARCHAR(16), A1 VARCHAR(16),
               ..., An VARCHAR(16))
```

and the corresponding schema for the vertical relation is

```
V(Oid INTEGER, Key CHAR(5), Value VARCHAR(16))
```

We refer to queries over the horizontal format as HorizontalSQL and queries over the vertical format as VerticalSQL. In this section, we assume that the vertical relation is clustered by the *key* column.

2.1 Predicting trends

In this section, we present the candidate plans for the two formats for selection and projection queries and qualitatively analyze the trends we would expect to see in their performance.

2.1.1 Projection

We begin with a query that asks for k attributes of the objects stored in the system. In HorizontalSQL this is straightforward:

```
select A1, A2, ..., Ak from H
```

The equivalent VerticalSQL query, by contrast, is a series of k self-joins on the V relation, one per projected column, as shown below.

```
select A1, A2, ..., Ak from
  (select distinct oid from V) as t0
  LEFT OUTER JOIN
  (select oid, value as A1 from V where key = 'A1') as t1
  ON t0.oid = t1.oid
  LEFT OUTER JOIN
  (select oid, value as A2 from V where key = 'A2') as t2
  ON t0.oid = t2.oid
  LEFT OUTER JOIN
  ...
```

An obvious execution plan for the horizontal query is a table scan followed by a projection of the required columns. We refer to this plan as **H_ProjScan**. The primary cost of this plan is the I/O cost for scanning the horizontal relation, which is directly related to the size of the relation. For sparse datasets, the relation size depends critically on how efficiently the relational database stores null values, the sparsity of the data, and the average size of a non-null value. Another important factor that determines the cost is the scan speed, that is, the rate at which the database is able to read data from the disk.

For the vertical query the natural plan is a little more complex. It begins by using the index on the *Key* field to retrieve the tuples corresponding to the attributes occurring in the projection list (recall that in the vertical schema each attribute of each object is in its own row) and joins them together on Oid to "reconstruct" the objects. Each of the k join operations can either be nested loops, sortmerge or hash join. We refer to this plan as **V_ProjJoin**.

This plan has two main overheads: (1) the I/O cost of the index lookups that retrieve the vertical tuples, and (2) the CPU cost for performing the join operations. When the data is sufficiently sparse, the CPU costs are likely to be comparable to the I/O costs, because each attribute list will only be a small fraction of the whole relation.

For a given dataset, as we vary the number of columns projected, the execution time for H_ProjScan is likely to be fairly constant as the dominating cost is the scan cost. On the other hand, the execution time for V_ProjJoin is likely to increase linearly with the number of columns projected as both the I/O and CPU costs increase linearly with the number of columns projected. Given this observation, the question that arises is when the actual crossover occurs for real systems. We look at this in our experimental evaluation in Section 2.3.1.

2.1.2 Selection

We now turn to consider the following simple selection query over the horizontal relation. We distinguish between "projection" queries, which return a subset of attributes for all objects, and "selection" queries, which return a subset of attributes for a subset of objects. Most precisely, we should refer to the first type of query as "project" queries and the second as "select-project" queries, but for conciseness of expression we use the terms "projects" and "select". Assuming we only want attributes A1 and A2 in the answer, then the HorizontalSQL query is:

```
select A1, A2 from H where A1 = 'V0'
```

The corresponding VerticalSQL query is

```
select A1, A2 from
  (select oid, value as A1 from V
   where key = 'A1' and value ='V0') as t1
  LEFT OUTER JOIN
  (select oid, value as A2 from V
   where key = 'A2') as t2
  ON t1.oid = t2.oid
```

Assuming the presence of indexes on all relevant columns, three reasonable plans for the horizontal approach are

- **H_SelScan:** Scan the entire relation and project the two columns while filtering based on the where clause.
- **H_SelIndexFetch:** Use an index on column A1 to select the tuples with value 'V0' and then perform an unclustered index fetch to get the value of column A2 for these qualifying tuples.
- **H_SelIndexMerge:** Use an index on column A1 to select the tuples with value 'V0'. Similarly, use an index on column A2 to select all the values in A2. Merge the two lists based on RIDs.

The cost of the H_SelScan is most likely dominated by the I/O cost for scanning the entire relation, while the primary cost for H_SelIndexFetch is likely to be the cost of the unclustered lookup for column A2. For the H_SelIndexMerge plan, the main cost is most likely the index scan on column A2.

For sparse datasets, the H_SelIndexMerge plan is likely to dominate the other two plans in most cases because the index on column A2 will be very small. For a given dataset, as one varies the selectivity of the predicate `A1 = 'V0'`, the costs for the H_SelScan and H_SelIndexMerge plans are likely to remain constant, while the cost for the H_SelIndexFetch plan is likely to increase linearly with the selectivity.

There are also several alternative plans for the VerticalSQL query, as the query involves three selection predicates and a join. We present three that are likely to dominate when the vertical relation is clustered by the *key* column.

- **V_SelIndexFetch:** Use the index on the value column to find all the RIDs for tuples that have values equal to V0. Next, do an unclustered fetch to get the tuples with these RIDS to check if the corresponding key is A1. For all such tuples, perform an unclustered I/O to get the corresponding value for column A2.
- **V_SelIndexJoin:** Use the index on the value column to get the RIDs for tuples with value V0. Next, again, do an unclustered fetch by RID of these tuples to check if the corresponding key is A1. If so, add them to a list of (Oid, value) pairs — all the tuples in this list will have the Oids and A1 values for all selected tuples. Next, get (Oid, value) pairs for all tuples with key A2, by using the clustered index on the key column. Join the two lists on Oid using one of nested loops, sortmerge or hash join.
- **V_SelIndexMerge:** Use the clustered index on the key column to get (Oid, value) pairs for all the values in column A1, applying the selection predicate on the way. Next, get (Oid, value) pairs for all tuples with key A2, by using the clustered index on the key column. Join the two lists on Oid using one of nested loops, sortmerge or hash join.

The dominating cost for the V_SelIndexFetch and V_SelIndexJoin plans is most likely the cost of the unclustered lookups. On the other hand, the main cost for the V_SelIndexMerge plan is the I/O for scanning the two indexes.

For a given sparse dataset, as the selectivity of the predicate `A1 = 'V0'` increases, the costs for V_SelIndexFetch and V_SelIndexJoin are likely to increase linearly with the selectivity because of the unclustered fetch in each plan, while the cost for V_SelIndexMerge is likely to stay relatively constant. The V_SelIndexMerge plan will be the dominating plan for the vertical relation beyond a threshold selectivity (the exact value of that threshold selectivity is dependent on several input parameters).

Across the two formats, the best horizontal plan (H_SelIndexMerge) and the best vertical plan (V_SelIndexMerge) are likely to perform comparably across a wide range of selectivities because both merge two lists, one for each of the projected attributes. The actual relative performance of the plans will depend upon the size of these lists.

2.2 Experimental Setup

In this section we describe the setup for a quantitative empirical investigation of the tradeoffs identified above by using two commercial database systems A and B. We ran all the experiments on a Windows 2003 Server with a 2.4 GHz Intel Pentium machine and 1 GB of physical memory. Data was placed on one disk and logs were created on another. The buffer pool size was set to 50MB.

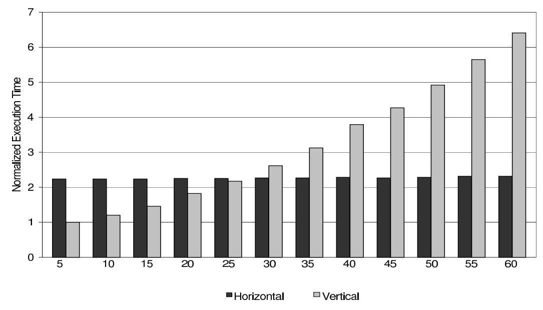

Figure 1: Projection performance for database system A on $400x50k$ $\rho = 5\%$.

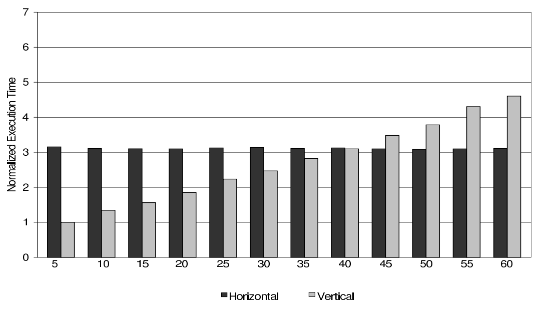

Figure 2: Projection performance for database system B on $400x50k$ $\rho = 5\%$.

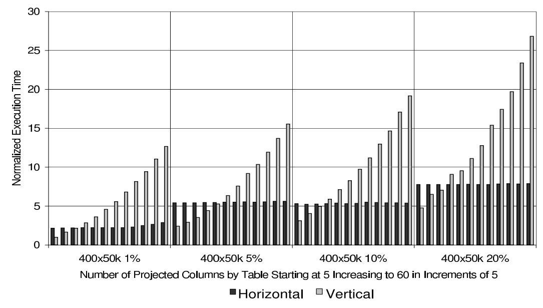

Figure 3: Projection performance for database system A on $400x50k$ tables.

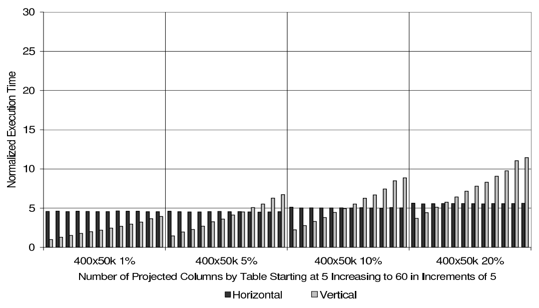

Figure 4: Projection performance for database system B on $400x50k$ tables.

We used the datasets described in [1] for the experiments. The synthetic data generator has the following parameters: the number of columns in the horizontal table, the number of rows in the horizontal table, the non-null density (that is, percentage of field values that are not null), the predicate selectivity for each column, and the number of distinct values in each column.

We used four different configurations for the table dimensions, $200x100k$, $400x50k$, $800x25k$ and $1000x20k$, where CxR indicate the number of columns C and the number of rows R in the table. We used four values for the non-null density ρ: $1\%, 5\%, 10\%$ and 20%.

The data for the horizontal and vertical tables was clustered by the Oid and Key fields respectively. Every column involved in a query on the horizontal table was indexed. We also built individual column indexes on each of the three columns of the vertical table and a two-attribute index on (Key, Oid) as well.

We obtain cold numbers by flushing the buffer pool and file system cache before the start of each run. Throughout the paper, whenever we compare across different databases, we present normalized execution times. For each system, we normalize the corresponding execution times by the smallest number for that system, unless otherwise noted.

2.3 Performance results

In this section, we report the results of projection and selection queries for the two formats. We show how the different parameters affect the tradeoff using the two databases.

2.3.1 Projection queries

Figures 1 and 2 show the results for the $400x50k, \rho = 5\%$ dataset using database systems A and B, respectively. In each graph, we present the horizontal and vertical numbers for projecting $5-60$ columns, in increments of five columns. The main observations from these graphs are as follows:

- As predicted by our analysis in Section 2.1.1, the execution time for horizontal is almost constant, while the execution times for vertical increase linearly with the number of columns projected.
- For both databases, there is a crossover point beyond which horizontal performs better than vertical.
- This crossover point occurs in different places for the two databases: at 30 columns for database A and at 40 columns for database B.
- The rate at which the vertical execution times increase is steeper for database A than it is for database B, resulting in an earlier crossover point for database A.

We next look at how the trends change as we vary the non-null density. The results for the $400x50k$ dataset over database A are given in Figure 3 for four different non-null densities $\rho = 1\%, 5\%, 10\%$ and 20%. Again, we report normalized running times for projecting $5-60$ columns, in increments of five columns.

From the results, we notice how the execution times for vertical increase as the non-null density increases. Recall that the plan for the VerticalSQL query is a join of k lists,

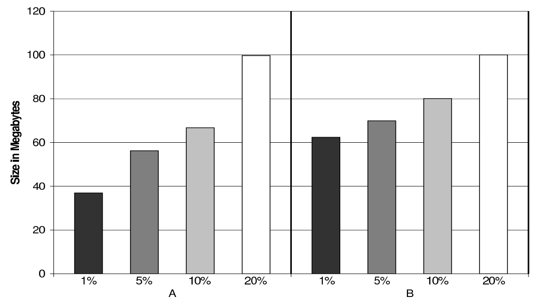

Figure 5: Table sizes for the $400x50k$ dataset.

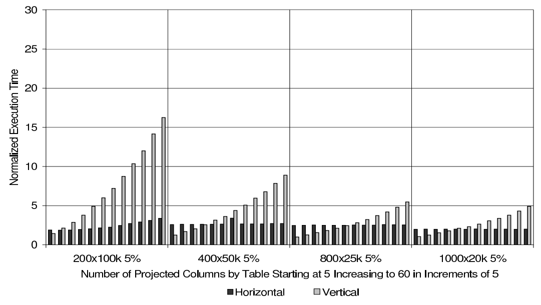

Figure 6: Projection performance for database system A on all tables with $\rho = 5\%$.

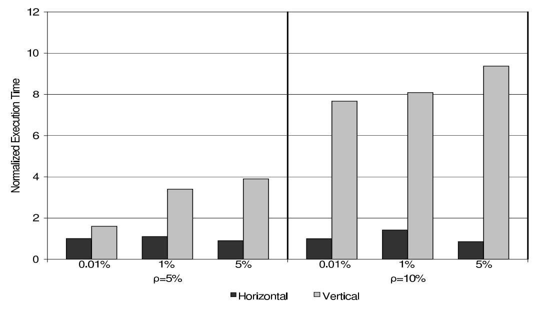

Figure 7: Selection performance for database system A on $1000x20k$ tables.

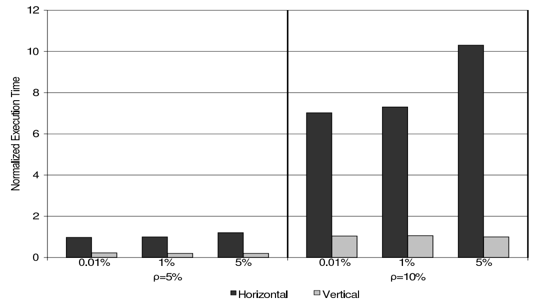

Figure 8: Selection performance for database system B on $1000x20k$ tables.

each of which are the tuples corresponding to a single attribute in the select clause. As non-null density increases, the length of each of these lists increases, resulting in longer execution times. The horizontal execution times also increase as the non-null density increases, except across the 5% and 10% datasets. This has a direct correspondence to the actual size of the relations on disk, the details of which are presented in Figure 5. Since the execution times for both formats changes in analogous ways the crossover points remain fairly stable (between 20 and 30) across the four values of ρ .

On the other hand, database B exhibits a slightly different behavior as we vary the non-null densities. The results for the $400x50k$ dataset over database B are given in Figure 4. The execution times for horizontal vary in different ways for the two databases, resulting in different trends for the crossover points as ρ is varied. For B, the execution time for horizontal increases only by a small amount as the non-null density goes up. This is related to the table sizes on disk, which are shown in Figure 5. Notice that the horizontal relation size increases only by 60% from $\rho = 1\%$ to 20%. The corresponding sizes for database A increase by 170%. This lesser increase results in only incremental increases in horizontal running time.

We now take a final look at the projection queries in order to investigate how the performance changes across table dimensions. Figure 6 shows the experimental results for database A, across different table dimensions for a fixed value of $\rho = 5\%$. Notice how, as the tables get wider, the execution times for vertical reduce, and as a result the crossover points occur later. This trend holds for the $200x100k$, $400x50k$ and $800x20k$ table dimensions, where the crossover occurs at $10, 20$ and 30 columns. The execution times for horizontal go down for the $1000x20k$ dimension, resulting in an earlier crossover point at 25 columns.

2.3.2 Selection queries

We present the selection results for the $1000x20k$ dataset with non-null densities $\rho = 5\%$ and $\rho = 10\%$ over database A in Figure 7. We notice that for a fixed value of ρ, as the selectivity of the predicate increases, the horizontal execution times are almost constant, while the vertical execution times are increasing linearly. The horizontal format is better than the vertical format in all the cases.

The corresponding numbers for database B are given in Figure 8. Here, we see the opposite trend and the vertical format is better than the horizontal format in all the cases. For a fixed value of ρ, as the predicate selectivity increases, the vertical execution times stay fairly constant while the horizontal execution times are much higher and increase slowly.

The behavior we expect from our analysis in Section 2.1.2 is that the execution time for both formats should remain constant as selectivity increases and the two formats should be comparable to each other. Neither of the databases agree with this conclusion.

We then looked at the plans selected in the various cases. For database A, we saw that the horizontal plan was correctly chosen as the H_SelIndexMerge plan (refer to Section 2.1.2). On the other hand, the vertical plan was either V_SelIndexFetch or V_SelIndexJoin, both of which have un-

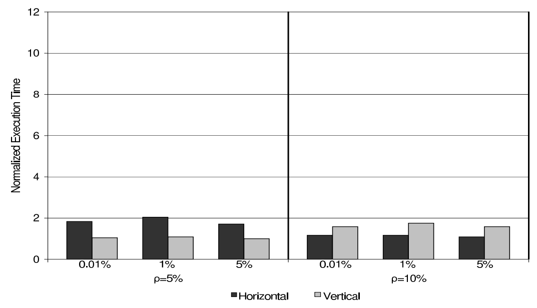

Figure 9: Selection performance for database system A on $1000x20k$ tables with forced vertical plans.

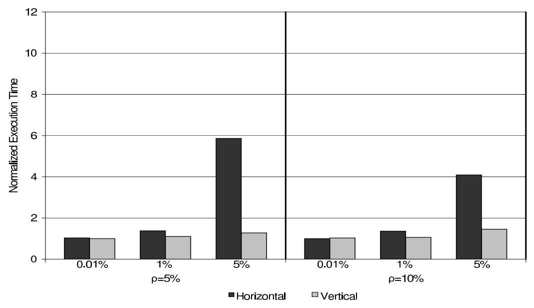

Figure 11: Selection performance from [1] on $1000x20k$ tables.

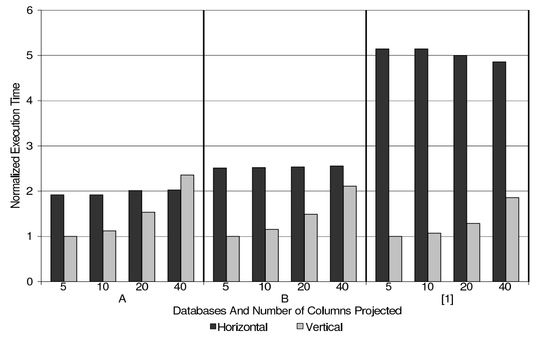

Figure 10: Projection performance for databases A, B, and [1] on $1000x20k$ $\rho = 10\%$.

clustered fetches resulting in linear increase in costs. For database B the problem was with the horizontal plan. It was choosing the H_SelIndexFetch plan, which has unclustered fetches as well resulting in high execution times. So, plan selection is the main reason for this conflicting behavior across the two databases.

We then forced database A to use the optimal V_SelIndexMerge plan for the vertical queries and the resulting comparison is shown in Figure 9. Notice how, for a fixed value of ρ, the execution times for the two formats are almost flat across the three selectivities. This agrees with our prediction in Section 2.1.2 that the two numbers are likely to be constant and close to each other. In addition, for $\rho = 5\%$ vertical is better than horizontal by about 50%, while for $\rho = 10\%$, horizontal is better than vertical by about 30%. This again illustrates the tradeoff between the two formats and its dependence on non-null density.

2.4 Comparison to Prior Work

The results above show that there is a tradeoff between the two formats and several parameters play an important role in deciding the crossover point. These results contradict the claims from previous work [1], in which Agrawal et al. claim that vertical *uniformly outperforms* horizontal over a wide range of datasets. In this section, we provide an explanation for why they observed a different trend.

In Figure 10, the normalized execution times we obtained for projection queries and the corresponding normalized execution times from Agrawal et al. are shown. We are able to compare the different numbers as the underlying dataset used in both cases is the same. The vertical numbers show a similar trend for the three databases. The horizontal numbers are fairly constant within a database, but are much higher for the database in the previous paper. Comparing the published overheads for storing a null value in a Varchar field, we saw that databases A and B both have a two byte overhead per null value, while the database they used had a five byte overhead (four bytes for any variable length field and a one byte null overhead). This means that commercial databases have improved the null handling capabilities in the last few years, resulting in smaller horizontal relations and better scan times.

In Figure 11, we present the normalized selection numbers for two different values of ρ. In Agrawal et al., it is mentioned that the plan for horizontal is H_SelIndexFetch, where the cost increases as query selectivity increases. This is a suboptimal plan (Section 2.1.2). Similarly, the paper mentions that the plan chosen for vertical is V_SelIndexJoin, which is also a suboptimal plan. Both plans have unclustered fetches, the horizontal plan over the entire horizontal table and the vertical plan within a single attribute. So, vertical performed better than horizontal in this case. On the other hand, if the optimizer correctly chose the best plans for both cases, we expect the two numbers to be comparable like in Figure 9.

2.5 DBMS-Specific Issues

In this section we highlight some interesting facts related to the performance and capabilities of the two systems we tested.

In the introduction we mentioned that one interesting side effect of our work is that it illustrates how DB performance can depend on configuration settings. When we first ran the above experiments, we observed contrasting behaviors of the two databases. While the results for database A were the numbers shown above, the horizontal execution times for database B were much higher than the numbers reported above. As a result, there were no crossover points for that DBMS, and we saw, as did Agrawal et al., that vertical was uniformly better than horizontal.

Upon further investigation, we found that while both databases choose to use a scan plan for the projection query over the horizontal relation, the performance of

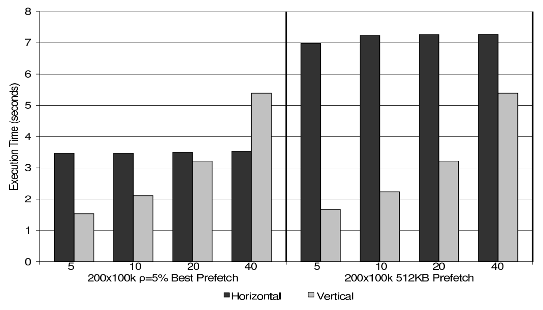

Figure 12: The effect of prefetch on database system B.

database B was worse compared to that of database A. This was perplexing because they both ran on the same hardware and OS, so scan times should be nearly equivalent. We tried changing different system parameters and found that changing the prefetch size greatly affected the scan time for HorizontalSQL in database B. Our initial prefetch size was 512 KB, and by changing the prefetch size, we were able to reduce the running times for HorizontalSQL, over database B by over a factor of two. This resulted in the two DBMS showing comparable running times. Changing the prefetch size marginally improved the running times for VerticalSQL in database B, but overall HorizontalSQL became better than VerticalSQL in many cases.

We present a sample result from projecting $5, 10, 20$ and 40 columns over the $200x100k$ ($\rho = 5\%$) relation in Figure 12. The left presents the comparison between HorizontalSQL and VerticalSQL with the best prefetch size and the right shows the numbers for a prefetch size of $512KB$. Notice how the running times for horizontal have gone down by a factor of two, resulting in a reversal of conclusion that vertical is uniformly better. So, with the correct prefetch size, we observe that HorizontalSQL can compete with VerticalSQL, even in database B, confirming the results that we observe with database A and the predictions from our trend analysis. The numbers reported earlier in Section 2.3 for database B are those obtained after tuning the prefetch size.

Another interesting issue to explore is other options for implementing queries over the vertical schema. In [1], Agrawal et al. explored two different techniques for writing queries over the vertical format: VerticalSQL and VerticalUDF. While VerticalSQL uses SQL-92 level capabilities, VerticalUDF exploits object-relational extensions to SQL; namely, user-defined table functions. In their experiments, the authors observed that VerticalUDF is better than VerticalSQL for projection operations (by about 10%), while it is slower than VerticalSQL for other operations. Since neither of the options uniformly dominated the other, and they were always within 10% of each other, our broad conclusions would be unlikely to change based upon the choice between VerticalSQL and VerticalUDF, so we only present results with VerticalSQL.

3. WORKLOAD DEPENDENCY

In this section, we show how the nature of the query workload can play an important part in the tradeoff between the horizontal and vertical formats. We first consider a modified version of the selection queries and show how returning all attributes for every selected item has a significant impact on the choices made. We then use a different sparse dataset inspired by a hierarchical categorization of e-commerce items to show how the ideal clustering strategy for the vertical relation changes with the workload.

3.1 "Select * " selection queries

The selection queries in Section 2 project only two attributes for every selected item. While that is a reasonable query, we also think a logical variation of this query is to return all attributes of the selected items. Intuitively, this new query corresponds to "give me all the non-null attributes" of the selected item, and in the horizontal framework it can be expressed as follows:

```
select * from H where A200 = 'V0'
```

While this query is easy to express in HorizontalSQL, expressing it in VerticalSQL is difficult. Performing one self-join per attribute does not scale for this query, because there are potentially several hundred attributes. Even the VerticalUDF approach, suggested as an alternative in [1], requires that the external table function have many parameters—twice the number of projected columns, one for the variable name and one for the null indicator. Clearly writing or even generating a table function for every projection scenario is a tedious task.

In view of this problem, a third approach is to no longer require the system to return a horizontal view of the vertical table and present the vertical result to the application layer. That is, instead of presenting a tuple with the values of the attributes as fields in the tuple, for each object we return a list of tuples, one for each non-null attribute of the object. The query to do this is as follows:

```
select   v2.*
from     V v1, V v2
where    v1.key = 'A200' and v1.val = 'V0'
         and v1.oid = v2.oid
order by v2.oid
```

The query is similar to the corresponding VerticalUDF query, with the difference being that the VerticalUDF query additionally converts each item's attributes from the vertical to the horizontal format. So, the VerticalUDF query incurs a higher cost than the above query. We use the above query in our experiments and thus underestimate the total cost for the vertical approach. This is "fair" because the general trend in the experiments in this section is that horizontal is cheaper than vertical in many cases.

We present experimental results over database A for the $400x50k$ dataset, with $\rho = 5\%$ and 10%. The execution times are shown in Figures 13 and 14 respectively. The selectivity varies from 0.05% to 50%. This selectivity is computed only over the non-null values. So, a non-null factor of 10% reduces the stated selectivity by a factor of 0.1 over the whole table. For example, if the table had 10% non-nulls, a selectivity of 10% would return 1% of the tuples. For each query, we experimented both with optimizer chosen plans and also several forced plans. We report the minimum execution time over all plans in this section.

In both the graphs, we see that horizontal is much better than vertical for low selectivities but that the gap narrows as

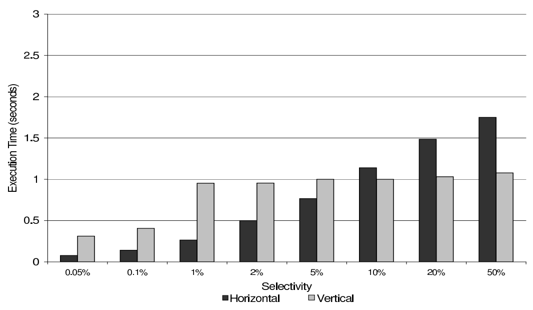

Figure 13: Selection performance with projecting all columns for database system A on $400x50k$ $\rho = 5\%$.

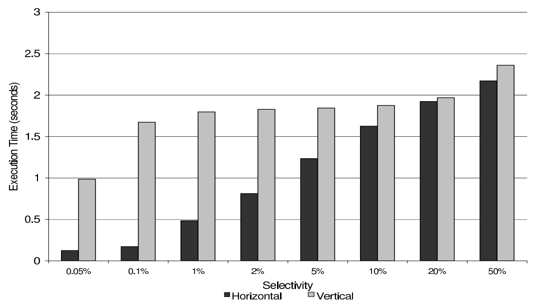

Figure 14: Selection performance with projecting all columns for database system A on $400x50k$ $\rho = 10\%$.

the selectivity increases. For the lowest selectivities, vertical uses an index nested loops join, but very soon hash join becomes the best plan. This explains the almost constant execution time for vertical as the scan of the entire table to build the hash table dominates the total cost. As a result, the selectivity of the predicate has little effect on the total time. On the other hand, for horizontal, an index-based plan is better than a table scan for the entire range and the linear increase in running time with increased selectivity reflects this fact.

For the 5% non-null case, vertical performs much better than horizontal for the last three selectivities, while for the 10% case, horizontal does marginally better than vertical in these cases. The relationship in the performance of the two approaches is related to the relative sizes of the relations in the two approaches. The vertical relation is about half the size of the horizontal relation for the 5% case, while they are the same size for the 10% case. Since the main cost for the vertical plan is a scan to build the hash tables, and the maximum cost for HorizontalSQL is a table scan, we see that HorizontalSQL is never really bad compared to VerticalSQL for $\rho = 10\%$ (when the tables are of comparable size), while it is worse off for $\rho = 5\%$ (when the horizontal relation is double the size of the vertical.)

Finally, we investigate the impact of the number of columns projected on the tradeoff between the two formats. Let us compare the "select *" selection queries with the selection queries from Section 2.3.2. Recall that for those queries, which project only two columns, the running time is virtually constant for both formats and that the performance of the two formats is nearly the same. By contrast, when projecting all columns, things are quite different. When projecting more columns, the horizontal times experience increased running times with increased selectivity. Also, when projecting all columns, although vertical execution times stay constant beyond a threshold selectivity, the vertical cost is much higher than the two column case. This is because returning more attributes corresponds in the vertical case to scanning more tuples, so the vertical approach has to scan the entire vertical relation instead of just the subset corresponding to the two projected attributes. In summary, for the two column case, there is very little difference between the two formats, while in the project all column case, there is a tradeoff that depends on the selectivity of the predicate and the relative size of the

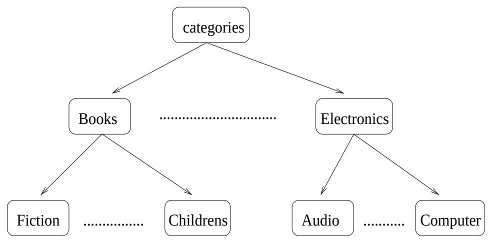

Figure 15: Sample hierarchical arrangement of categories

two relations.

3.2 Sparse regular dataset

In the previous sections, we looked at a dataset in which there is no correlation across different columns. In this section, we consider a more "regular" dataset in which subsets of attributes are highly correlated in that they are either all likely to be null or all likely to be non-null. This workload exhibits two interesting facts (i) the tradeoff between horizontal and vertical exists even in this more structured case and (ii) the best way to cluster the vertical relation depends upon whether or not these correlations exist among the attributes.

3.2.1 Experimental Setup

We use the same setup as described in Section 2.2, but focus on a new data set that is inspired by a closer inspection of some e-commerce datasets. A typical e-commerce catalog has a wide variety of items organized into categories, with the categories themselves hierarchically grouped. An example appears in Figure 15. Items that belong to a common category are likely to have common attributes, while those within the same subcategory are likely to have more common attributes. In order to capture this property in our dataset, we assign fixed and optional attributes to all of the nodes in the hierarchy. For each item, we randomly assign a leaf level category and, based on its root to leaf path, establish its possible attribute set. The fixed attributes for the item always get non-null values, while the optional attributes may or may not have non-null values (whether or not they have non-null values is controlled by a random probability.)

The parameters of the synthetic data generator are:

- the number of levels in the hierarchy, L;
- the fanout at each level in the hierarchy, F;
- the number of fixed fa and optional attributes oa for each hierarchy node;
- the probability p that an optional attribute gets a non-null value; and
- the number of items R.

We report experimental results for the following configuration: $L = 2$, $F = 12$, $fa = 2$, $oa = 2$, $p = 0.25$ and $R = 300K$. The resulting table has a non-null density of $\rho = 1.1\%$ and the number of columns is 628. There is a category relation

```
C(OID INTEGER, Cat1 INTEGER, Cat2 INTEGER)
```

that maps each item to a particular category. We use this relation to map the vertical tuples to categories. Although the schema for vertical is the same as described earlier, we inline the category relation into the horizontal relation to get the following schema:

```
H(OID INTEGER, Cat1 INTEGER, Cat2 INTEGER,
A0 VARCHAR(16), A1 VARCHAR(16), ..., An VARCHAR(16))
```

This allows the clustering of the horizontal data by categories Cat1 and Cat2. The alternative of clustering the horizontal relation by Oid and doing a join with the category relation results in an unclustered fetch and poor performance for the workload considered below. The above schema for vertical is best because storing the category information in the same relation does not save any join operations (unlike the horizontal case), but instead increases the size of the relation.

3.2.2 Performance Results

We present results for experiments on projection and selection queries on the hierarchical dataset. In these queries, we place conditions on the categories, which translates into a selection on the horizontal table and a join between the category and the vertical table. We consider three formats: horizontal, vertical clustered by key (vertical(key)) and vertical clustered by Oid (vertical(oid)).

Projecting entire subcategories

The following query projects all values in a subset of the categories:

```
select * from H where
and H.cat1 = ‘c1’ and H.cat2 = ‘c2’
```

Here cat1 and cat2 correspond to the first and second level categories for each item. For example, if we require cat1 = 7 and cat2 = 3, the resulting query projects $\frac{1}{144}$ of the entire relation, while cat1 = 7 and cat2 in $\{1, 2\}$ projects twice as much data ($\frac{1}{72}$). Figure 16 shows results from this experiment for various category projections up to projecting the entire relation. For vertical, we use a query similar to the one in Section 3.1 and return results in the vertical format without constructing the horizontal tuples. So, as discussed previously, the graphs largely underestimate running times for vertical.

```
select  v1.*
from    V v1, C c
where   v1.oid = c.oid and c.cat1 = ’c1’
        and c.cat2 = ’c2’
orderby v1.oid
```

There are two main points to note from the results: (i) Horizontal is better than vertical for low selectivities, while the reverse is true for higher selectivities, and (ii) vertical (clustered by Oid) is better than vertical (clustered by key) in all the scenarios.

In [1], the authors note that clustering by key is better than clustering by Oid for the vertical relation in all scenarios (including projection and join). Our conclusions differ for two reasons:

- The join with the category table is on Oid, unlike in [1], where the join was on a sparse column.
- The query returns all columns for every selected item, unlike the case in [1] where only a *small known* subset of columns is returned.

The graph in Figure 16 shows HorizontalSQL at a large disadvantage, because both formats return the same data, but in different formats. When projecting most of the table, HorizontalSQL spends a majority of the time fetching the result tuples and returning them to the application layer. When we measure just the time spent executing the query, by dropping the resulting tuples at the server rather than dropping them at the application layer, then the times for HorizontalSQL are comparable to VerticalSQL (oid).

A comparable operation for VerticalSQL (oid) requires reconstructing the horizontal format at the database layer through a stored procedure or a UDF, but this is also expensive. In our tests, constructing a horizontal tuple in a stored procedure took over 7 minutes for projecting items from one first-level category (the third column from the left in Figure 16). This means that for vertical, if possible, it might be best to return the results in the vertical format to the application layer.

Selection queries

We next look at selection queries of the following form:

```
select * from H
where H.c11 = ’v1’
```

The running times for this query are presented in Figure 17 for varying selectivities for the predicate. Here, again, a selectivity of 25% means that the query returns 25% of the items with non-null values in that column. Notice how vertical(oid) and horizontal are both better than vertical(key) for all the selectivities.

3.3 Summary

From the above results, we see how the query workload has a significant impact on the relative performance of the two formats. Our main conclusions are:

- The horizontal format outperforms the vertical format in many cases. This is especially true in selection queries, where indexes can be used to speed the query evaluation. In our experiments, the vertical format outperforms the horizontal format primarily in selection queries that have a high selectivity over data sets in which the non-null density is low.

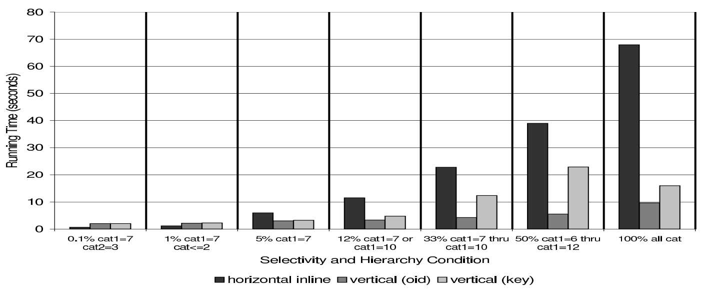

Figure 16: Performance for projecting all values in a subset of categories.

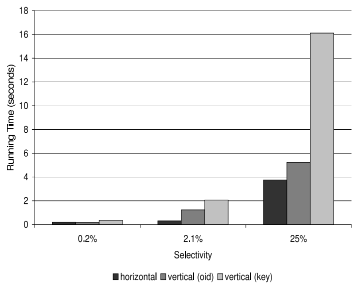

Figure 17: Performance for selecting items with a certain value.

Table 1: Comparing clustering alternatives for vertical relation

clustered	Projection	Projection	Selection	Selection	Join	Join
	known columns	all columns	return known columns	return all columns	on sparse column	on Oid
key	x		x		x	
oid		x		x		x

- The tradeoff between the approaches for selection queries depends heavily on whether a small number of known columns are projected, or the entire set of columns is projected.

- Maintaining a horizontal view of the vertical table becomes expensive when a query returns all columns of the selected tuples. This means, as an extreme but perhaps not rare case, that returning the entire dataset is very expensive. On the other hand, vertical outperforms horizontal if we allow the system to return its answer in vertical format (that is, as a list of OID, attribute name, value) triples.

- In contrast to the conclusion in [1], we found that the Vertical(oid) strategy is better than the Vertical(key) strategy in certain scenarios. The relationship between the clustering strategy and the query workload is summarized in Table 1.

4. SHORTCOMINGS OF CURRENT RELATIONAL SYSTEMS

In the previous sections, we explored the strengths and weaknesses of the horizontal and vertical formats. As part of this investigation, we found that storing and querying sparse data in current RDBMSs in either format can be somewhat problematic. In this section we describe problems we encountered; these problems suggest areas in which systems could be improved to better handle sparse data.

4.1 Representing wide tuples

Wide tuples are unavoidable when using the horizontal format to represent sparse data, and we found that the RDBMSs had problems handling them during storage and query processing. Three specific problems we encountered were:

- Storing wide tuples in the horizontal format is not scalable. Since the RDBMSs we tested do not allow a single tuple to span multiple pages, the system page size limits the maximum number of columns allowed in a single relation. While database B enforces a limit on tuple width during schema creation, database A is more flexible and enforces this limit at data load time. Because we found that horizontal yields better performance than vertical in many cases, finding techniques for scaling the horizontal schema to thousands of columns would be desirable.

- While both databases have null compression, the storage per null is still about two bytes, which hurts storage efficiency as well as placing a bound on the maximum number of columns (due to the page size limit on tuple width). One alternative would be to support a "sparse row" representation where the name and value of non-null attributes are stored. (Such a format really amounts to a "tagged" representation within the RDBMS.) This could result in up to two orders of magnitude improvement in the number of columns supported in a sparse horizontal table. The downside of implementing such an approach would be that relational operators throughout the system would have to be modified to handle this new tuple format.

- Wide tuples are a problem during query evaluation as well as storage. Irrespective of the format used to store the data, if query evaluation requires intermediate results consisting of wide tuples, then current commercial systems do not scale well. For example, consider a self join query on the horizontal relation. Even if tuples of the original relational schema fit on a page, problems arise if the final result tuples may be wider than a single page. Again, we found that database B does not execute the query if at compilation time it suspects that intermediate tuples may be wide during query evaluation, while database A executes the query and raises a runtime error if it finds it needs to write wide tuples to disk.

	all categories	cat1 = 1	cat1 = 1 and cat2 = 1
$col4 =' v1'$	732	55	2
$col8 =' v2'$	58	58	3
$col12 =' v3'$	3	3	3
$col4 =' v2'$	0	0	0

Table 2: Actual cardinality estimates. Predicted cardinality estimate was 1 in all cases.

- Previous work [1] advocates that the vertical format should be a physical format, while users should still interact with the horizontal format. This implies that the query results have to conform to the horizontal format. Unfortunately there is no obvious scalable solution for constructing horizontal tuples from the vertical format. Recall that there are two existing approaches for this: VerticalSQL and VerticalUDF. Suppose the output horizontal tuples have n columns. Then the VerticalSQL approach requires a query with n joins. In the VerticalUDF approach, a user-defined table function is used to construct sparse wide tuples. In order to write this table function, one needs to know the actual column names in the resulting horizontal schema. Also, the number of parameters that the table function requires is linear in n. So, the VerticalUDF approach is also not a scalable approach for reconstructing flat tuples from the vertical format.

4.2 Suboptimal statistics

In many cases we noticed that the statistics used by the optimizer were bad. We illustrate this point with the following example query on the vertical relation:

```
Select all items in category c for
which attribute ‘a’ has value ‘v’
```

By varying the values of the different parameters, we get a family of queries. Table 2 gives the actual result cardinality for some queries obtained over the data set in Section 3.2.

For all the queries, the optimizer estimates the cardinality as 1. On the other hand, notice that the actual cardinalities vary considerably. The main problem here is the absence of multi-column statistics. The above queries have a selection predicate of the form *p: V.key = 'a' and V.value = 'val'*, which is the place where the optimizer makes errors. Notice how estimating the cardinality for the above query requires keeping track of the correlations between values in the two columns. Due to the absence of multi-dimensional histograms [3, 8, 9], the optimizer makes an *independence assumption* resulting in a huge error. A similar problem results with the horizontal format for certain other queries.

Even with multicolumn distribution statistics, there is another problem for the vertical approach. To see it, note that the horizontal relation allows for statistics on a per attribute basis. Performing a similar task over the vertical relation is not straightforward, since all the values for all attributes appear in the same relational column. Workload aware techniques for building multi-dimensional histograms [3] help address this problem to some extent.

4.3 Ease of expressing queries

An interesting aspect of the vertical format compared to the horizontal format is that the schema for the vertical format has no meaningful metadata. (The authors of this paper, and unfortunately presumably the readers as well, suffered from this, as it is difficult to keep track of what “key” and “value” mean. Other choices for the vertical column names seem perhaps even more confusing, for example, naming an attribute “attribute.”) The names of attributes are stored as data in the vertical relation. In one sense, this makes writing SQL queries against the vertical format cumbersome and error-prone. This argues that it might be better to give a horizontal view of the vertical representation to the user.

On the other hand, the horizontal format makes expressing certain metadata queries difficult, while these same queries can be expressed in a straightforward way over the vertical format. For example, consider the following query:

Q : Find all items with at least k non-null values.

In order to express the query in HorizontalSQL, either the *where* clause will have $\binom{n}{k}$ conditions, or it will require a user-defined row function that includes similar logic. In contrast, this query can be expressed in VerticalSQL using a nested sub query with a group-by operation as shown below:

```
select v.*
from   V v
where  v.oid in ( select   v1.oid
                  from     V v1
                  group by v1.oid
                  having   count(*) >= k )
```

These types of queries are very simple to write in vertical because the format naturally allows queries over the metadata. It is an interesting area for future work to come up with approaches that balance the opacity of the vertical schema with the ease with which it supports queries about attribute names.

4.4 Handling different datatypes

Following the example of [1], in this paper we consider values of type varchar(16) for all the sparse columns. An issue arises when the sparse columns have different datatypes and the *value* column for vertical needs to support several datatypes. One approach that is mentioned in [1] is to create a separate vertical table for every data type along with a catalog table maintaining data type information for each attribute. This approach works well when the query involves a few *known* columns, but it is problematic for queries that involve a *select ** operation. For example, the simple selection query *Return all items where col1 = 'value'* translates into an outer union query similar to this one:

```
with SelectedTuples(integer oid) as (
     select oid
     from V_VARCHAR v
     where v.key = ’col1’ and v.value = ’value’
    )
select v1.oid, v1.key, v1.value, null, ...
from   SelectedTuples S, V_VARCHAR v1
where S.oid = v1.oid
union all
select v2.oid, v2.key, null, v2.value, ...
from   SelectedTuples S, V_INT v2
where S.oid = v2.oid
union all ...
```

Even here, we assume that the query writer knows the exact datatype of col1 beforehand. Otherwise, even the selection part of the query is the union of multiple selection queries. These observations indicate that it would be easier to use RDBMS for sparse data if there were support for columns of multiple types.

5. RELATED WORK

Many commercial e-commerce software systems such as IBMWebsphere Commerce Server, I2 Technologies, and Escalate use the vertical scheme for storing and querying sparse data. In [1], Agrawal et al. present a system that uses the vertical format to store sparse data and, with an enablement layer, presents a horizontal view to the user. The authors propose two new operators, v2h and h2v, to convert data from the vertical to horizontal format and vice versa. The UNPIVOT and PIVOT operators in SQL Server "Yukon" [14] perform a similar functionality. While the above approaches focus on how to use the vertical format to efficiently handle sparse datasets, the focus in this paper is to study the tradeoff between the horizontal and vertical formats.

In [7], the "unfold" and "fold" operators are presented as restructuring operators in the context of implementing SchemaSQL on a SQL database system. These operators can be viewed as generalizations of the v2h and h2v operators from [1].

The Decomposed Storage Model (DSM) [4] splits a horizontal relation into many 2-ary relations, one for each column in the relation. A common surrogate is used to identify different fields of a tuple across the 2-ary relations. Comparisons of the performance of the DSM model and the horizontal model have appeared in [4, 6]. The DSM storage model is used by the Monet system [2], where a detailed performance evaluation of DSM using the TPC-D benchmark is presented. In [10], a simple indexing strategy is suggested for DSM and different reconstruction algorithms are proposed. The DSM storage model is also referred as the Binary format in literature. All the above work is in the context of traditional dense relations in which there are a small number of columns and few null values, while the focus of our work is the tradeoff between the horizontal and the vertical formats for sparse data. The binary representation has also been used in IBM's Enterprise Directory LDAP product [13].

A study of different ways for storing XML data in an RDBMS and the performance implications of these alternatives is presented in [5]. The *Edge* and *Binary* approaches proposed here are similar to the vertical relation and the binary relation respectively. Once again, that work considered traditional dense data sets.

The vertical format has been used for dense data in a data mining context to implement association-rule mining algorithms in a relational database [11, 12] and shown to have better performance than their classical horizontal counterparts. This is different from our focus of comparing the two formats for simple relational queries over sparse data sets.

6. CONCLUSIONS

In this paper, we examined the tradeoffs involved in choosing between the horizontal and vertical formats for storing sparse data sets in a relational database. We showed that contrary to claims from previous work, there is no single best way to handle this data. We presented several factors that play an important role in this tradeoff, including properties of the dataset, the query workload and the null handling capabilities of the database. During this process, we identified some problems current RDBMS implementations have with handling sparse datasets, which suggest improvements for helping them handle these scenarios better.

Several avenues for future research remain. Improving the way RDBMS handle the problems mentioned in Section 4 constitutes one intriguing and relevant direction to pursue. In another direction, it is suggestive that many of the special demands of handling sparse data sets (including supporting a large and varying number of attributes, supporting frequent schema evolution, and mixing schema and data in queries) seem to fit very naturally in the XML data model. Investigating whether native XML DBMS have anything to offer in this domain is an interesting and promising area for future work.

7. REFERENCES

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

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

[3] N. Bruno, S. Chaudhuri, and L. Gravano. Stholes: A multidimensional workload-aware histogram. In *SIGMOD*, 2001.

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

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

[6] 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.

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

[8] M. Muralikrishna and D. J. DeWitt. Equi-depth histograms for estimating selectivity factors for multidimensional queries. In *SIGMOD*, 1988.

[9] V. Poosala and Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *VLDB*, 1997.

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

[11] S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. *Data Mining and Knowledge Discovery*, 4(2/3), 2000.

[12] P. Shenoy, J. R. Haritsa, S. Sudarshan, G. Bhalotia, M. Bawa, and D. Shah. Turbo-charging vertical mining of large databases. In *SIGMOD*, 2000.

[13] 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.

[14] SQL Server "Yukon". `http://www.microsoft.com/sql/yukon`.