Storage and Querying of E-Commerce Data

Rakesh Agrawal **Amit Somani** **Yirong Xu**

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120
{ragrawal,asomani,yirongxu}@us.ibm.com

ABSTRACT

New generation of e-commerce applications require data schemas that are constantly evolving and sparsely populated. The conventional horizontal row representation fails to meet these requirements. We represent objects in a vertical format storing an object as a set of tuples. Each tuple consists of an object identifier and attribute name-value pair. Schema evolution is now easy. However, writing queries against this format becomes cumbersome. We create a logical horizontal view of the vertical representation and transform queries on this view to the vertical table. We present alternative implementations and performance results that show the effectiveness of the vertical representation for sparse data. We also identify additional facilities needed in database systems to support these applications well.

1. INTRODUCTION

Imagine you run a marketplace for the electronics industry. This marketplace consolidates information about parts from more than 1000 manufacturers and distributors. Your current catalog contains nearly 2 million parts classified into 2000 categories. There are more than 5000 part attributes across various categories. New suppliers are expected to join your marketplace every week. They bring with them new parts, causing new attributes to be added to the current categories and new categories to be added to the catalog. You have the enviable task of designing the back-end database system to support this marketplace. What do you do?

We found ourselves in this quandary while building such an experimental marketplace, called Pangea. In this paper, we summarize our experience from implementing this application with the hope that our observations will be useful to system developers interested in providing effective database support to e-commerce applications. The issues we faced are pervasive in the new generation of e-commerce applications, such as on-line shops, exchanges, marketplaces, and portals, which aggregate data from a large number of providers. The specific e-commerce software used in our implementation was the IBM Websphere Commerce Server running on top of the DB2 Universal Data Base System. However, we believe our observations are generally applicable.

Proceedings of the 27th VLDB Conference, Roma, Italy, 2001
.

1.1 Issues

In relational database systems, data objects are conventionally stored using a horizontal scheme. A data object is represented as a row of a table. There are as many columns in the table as the number of attributes the objects have. In trying to store all our electronic parts in one table using this scheme, we ran into the following problems:

- *Large Number of Columns* The current database systems do not permit a large numbers of columns in a table. This limit is 1012 columns in DB2 (also in Oracle), whereas we had nearly 5000 attributes across different categories.
- *Sparsity* Even if DB2 were to allow the desired number of columns, we would have had nulls in most of the fields. In addition to creating storage overhead[1], nulls increase the size of the index and they sort high in the DB2 B+ tree index.
- *Schema Evolution* We would need frequent altering of the table to accommodate new parts and categories. The schema evolution is expensive in the current database systems.
- *Performance* A query incurs a large performance penalty if the data records are very wide but only a few columns are used in the query.

Similar challenges are faced by those building large repositories of meta data about documents in a digital library. For instance, an experimental news portal [2] being built at IBM Almaden processes 5-10 thousand news stories every day. For every story, it extracts a couple of hundred features such as stemmed words, people, countries, etc. The features are not fixed a priori and new features emerge as new stories are processed. A conventional horizontal table would need more than 100,000 columns to store the data for the features that have been identified to date and would need frequent altering to add new columns to accommodate newly identified features. Other potential applications of the work reported in this paper include stores for XML [7], RDF [1], KBMS [15], LDAP [19] and data mining [8] [21] [18].

1.2 Vertical Representation

To address the above problems, many commercial e-commerce software systems (e.g. IBM Websphere Commerce Server, I2 Technology, Escalate) define the following 3-ary vertical scheme for storing objects in a table:

Oid (object identifier)	*Key* (attribute name)	*Val* (attribute value)

Figure 1 shows a horizontal table and its corresponding representation in the vertical format. The symbol $\perp$ represents a null value.

[1]For fixed-width fields (e.g. INTEGER), the size of a null value is same as a non-null value. A VARCHAR null value on the other hand incurs the overhead of only one byte.

Horizontal (H)

Oid	A1	A2	A3
1	a	b	$\perp$
2	$\perp$	c	d
3	$\perp$	$\perp$	a
4	b	$\perp$	d

Vertical (V)

Oid	*Key*	*Val*
1	A1	a
1	A2	b
2	A2	c
2	A3	d
3	A3	a
4	A1	b
4	A3	d

Figure 1: Horizontal and Vertical Table Representations

The vertical table contains tuples for only those attributes that are present in an object. Different attributes of an object are tied together using the same *Oid*. Schema evolution is now easy; simply add new tuples corresponding to new attributes.

However, once the data is stored in the vertical format, new problems arise. Writing SQL queries against this scheme becomes very cumbersome and error-prone. More importantly, the current application development tools designed for horizontal format for storing objects no longer work.

What is needed is a logical horizontal view on top of the vertical representation of the data and query rewrite algorithms to convert relational algebra operators from the horizontal view to the vertical representation. This approach is conceptually identical to the view mechanism used in the database systems. Note, however, that the values in the *Key* field in the vertical format become column names in the horizontal view. Such higher-order views are not supported in the current database systems. We also need well-tuned processing strategies to get good query performance.

This paper describes the enablement layer we built on top of DB2 to realize the above functionality. The algebra and query transformations we developed and the lessons we learned from the extensive performance experiments should be of interest to database practitioners interested in providing support to e-commerce and similar applications. In building this enablement layer, we consciously did not change the database engine code to make our solution portable and time-expedient. However, we did identify some capabilities we wish the database system had provided. These new capabilities should be of interest to the database engine architects and implementors.

There is rich heterogeneous database research literature on transformations between schematically disparate schemas and types of schematic differences. In [10], Krishnamurthy et al. eloquently elucidated how data values in one data source may be modeled as schema (attribute or relation) labels in another. Several languages have been proposed for querying over schema labels, including [13] [17] . There is also work on defining higher-order views for integrating heterogeneous data sources [10] [12] [14].

Closest to this paper is the interesting work presented in [11] on the implementation of SchemaSQL. We share with them the goal of a "non-intrusive" implementation (i.e. without requiring changes in the database engine code). The extended-algebra we use in our query transformations includes `v2h` and `h2v` operations that can be viewed as the specializations of *unfold* and *fold* respectively in [11]. As we will see later in the paper, we have been able to realize substantial performance gains from this specialization.

1.3 Alternative Representations

The work on decomposition storage model [5] [9] split a horizontal table into as many 2-ary tables as the number of columns. A common surrogate tied different fields of a tuple across tables. There are now as many tables as the number of attributes. This storage model has been implemented in the Monet System which also developed an algebra to hide the decomposition [4]. An early work [16], done in the context of data base machines, also explored the option of storing table data on a per attribute basis. This representation has also been used in the IBM's Enterprise Directory LDAP product [19].

Another alternative would be to create one separate table for each category. Yet another alternative would be to create one table for common attributes and per category separate tables for non-common attributes. See [7] for some other alternatives and a performance study done in the context of storing XML data. Messaging systems such as Lotus Notes and Microsoft Exchange have also developed specialized structures to support sparse rows containing optional columns. However, there is no support for SQL querying in these messaging systems.

We will focus on the 3-ary representation of data as outlined in Section 1.2 (henceforth referred to as vertical representation). This representation offers an interesting design point between the conventional n-ary horizontal representation (henceforth referred to as horizontal representation) and the 2-ary binary representation (henceforth referred to as binary representation). Like the horizontal representation, the vertical representation requires only one table to store data, whereas the binary representation splits the table into as many tables as the number of attributes. While there are applications (e.g. SAP) that store data across a large number of tables, having thousands of tables instead of one makes the system harder to manage and operate. Schema evolution is trivial with the vertical representation, whereas an addition (deletion) of a new attribute requires "altering" the table in the horizontal representation and an addition (deletion) of a table in the binary representation. On the negative side, the vertical representation loses data typing since all values are stored as VARCHARs in the *Val* field, although it is easy to design extensions to support data typing if desired[2].

The horizontal representation is well studied in the database literature and there has been excellent work in understanding the tradeoffs of the binary representation [4] [5] [9]. Because of its manageability and flexibility, the the vertical representation is increasingly finding its way in many commercial systems. It behooves the database community to investigate and study how best to support the vertical representation to bring the new emerging applications to its fold. The work we present is a step in this direction that compliments earlier work.

1.4 Organization of the Paper

The rest of the paper is organized as follows. In Section 2, we discuss rewriting of the queries from the horizontal format to vertical format. We discuss the implementation strategies in Section 3 and give performance results in Section 4. We conclude with a summary and some pointers for the database system practitioners in Section 5. We also refer the reader to the extended version of this paper [3] that contains additional explanations and performance results.

[2]Create a separate vertical table for every data type. A catalog table maintains data type information for each attribute. Thus we might have a scheme as shown below:

ATTRIBUTES (KEY CHAR(K) PRIMARY KEY, DATATYPE CHAR(N));
V_INT(OID INTEGER, KEY CHAR(K), VAL INTEGER);
V_FLOAT(OID INTEGER, KEY CHAR(K), VAL FLOAT);
V_VARCHAR(OID INTEGER, KEY CHAR(K), VAL VARCHAR(X));

2. TRANSFORMATIONS

Our overall approach is to define a horizontal view H over a vertical table V. The user poses regular SQL queries over this view, which are translated into queries that run against the underlying vertical table. We will describe these transformations in terms of an extended algebra in this section. In Section 3, we discuss their implementation in a SQL system.

2.1 Algebra

We start with the well-understood algebraic operations [6]: select(σ), project(π), join($\bowtie$), outer join($\supset\!\bowtie\!\subset$), left outer join($\supset\!\bowtie$), right outer join($\bowtie\!\subset$), cross product($\times$), difference($-$), intersection($\cap$), union($\cup$), and aggregation($\mathcal{F}$). We add two operations to this algebra: v2h(Ω) and h2v($\mho$). We define the semantics of these operations after briefly introducing a few notations.

Notation

Assume that the vertical table V has the scheme (Oid, Key, Val) with a non-nullable column Oid and that $A1, \ldots, An$ are the key values in V. The equivalent horizontal table H has the scheme $(Oid, A1, \ldots, An)$ with the column Oid being non-nullable. We use Ak+m to represent the $(k$+$m)^{th}$ attribute. The symbol $\perp$ represents a null value.

We will use $\Theta_{i=1}^{k}\Psi_i$ as a short hand for $\Psi_1\Theta\Psi_2\cdots\Theta\Psi_k$. For a join operation (including its outer species), unless otherwise stated, assume the join predicate to be the equality of Oid. An explicit join predicate Ψ will be specified as $\underset{\Psi}{\bowtie}$.

Sometimes we will use \$0, \$1, etc. to refer to the columns of a result table.

For visual clarity, we will sometimes add square brackets in an expression as shown below.

$$[\Psi_0]\ \Theta\ [\Theta_{i=1}^{k}\Psi_i]$$

These square brackets do not affect the order of evaluation in any way; they are there only to enhance readability.

v2h*Operation*

Intuitively, the v2h(Ω) operation takes as input a vertical table and a list of attribute names and returns a horizontal table with those attribute names as the column labels.

$\Omega^k(V)$ creates a horizontal table of arity k+1 whose first column is Oid and the first k key values form the rest of the k columns[3]. The content of the table is defined by:

$$\begin{aligned}\Omega^k(V) \quad = \quad & [\pi_{Oid}(V)] \supset\!\bowtie \\ & [\supset\!\bowtie_{i=1}^{k} \pi_{Oid,\,Val}(\sigma_{Key=`Ai'}(V))] \qquad (1)\end{aligned}$$

Because of the first term on the right hand side and the use of left outer join, Eq. 1 can yield tuples with nulls in all of the non-*Oid* columns. For example, Ω^2 applied to the vertical table V from Figure 1 results in the table shown in Figure 2. For Oid = 3, V does not contain tuples corresponding to key values $A1$ and $A2$. However, the result table contains a tuple with this Oid and null values for attributes $A1$ and $A2$.

This semantics is consistent with the null handling in SQL. For instance, a projection of the horizontal table in Figure 1 on attributes $A1$ and $A2$ will indeed preserve the tuple corresponding to Oid = 3 in SQL.

[3]Since the columns in a relation are supposed to be orderless, strictly speaking Ω should take column names as parameters. We have chosen this notational simplification for ease of exposition.

$\Omega^2(V)$

\$0	\$1	\$2
1	a	b
2	$\perp$	c
3	$\perp$	$\perp$
4	b	$\perp$

$\mho^2(H)$

\$0	\$1	\$2
1	A1	a
1	A2	b
2	A2	c
3	A1	$\perp$
3	A2	$\perp$
4	A1	b

Figure 2: Results of v2h and h2v Operations

But for null handling, which was not discussed, the v2h operation is equivalent to Unfold by *Key* on *Val*(V) in SchemaSQL [11]. This operation is also similar to the *Gather* operation in [18].

h2v*Operation*

The h2v($\mho$) operation is the inverse of the v2h operation. Intuitively, it takes as input a horizontal table and converts it into a vertical table where each column label in the horizontal table is converted to a key value in the vertical table.

Assume a horizontal table H having the scheme $(Oid, A1, \ldots, An)$ with the column Oid being non-nullable. $\mho^k(H)$ creates a vertical table with the scheme (Oid, Key, Val). The content of V is defined by:

$$\begin{aligned}\mho^k(H) \quad = \quad & [\cup_{i=1}^{k}\ \pi_{Oid,`Ai',Ai}(\sigma_{Ai \neq `\perp'}(H))]\ \cup \\ & [\cup_{i=1}^{k}\ \pi_{Oid,`Ai',Ai}(\sigma_{\wedge_{i=1}^{k} Ai = `\perp'}(H))] \qquad (2)\end{aligned}$$

For each tuple h in H, the first term in Eq. 2 creates the tuples $\{< Oid, `Ai', h.Ai > | i = 1, ..., k\ \wedge\ h.Ai \neq \perp\}$. The second term handles the special case of a horizontal tuple that has null values in all of the non-*Oid* columns. Figure 2 shows the result of applying $\mho^2$ to the horizontal table H from Figure 1.

Again, but for null handling, this operation is equivalent to Fold by *Val* on *Key*(V) in SchemaSQL [11]. This operation is also similar to the *Scatter* operation in [18].

2.2 Rewritings

We describe now the rewritings of the standard algebraic operations on the horizontal view over a vertical table. We give two forms of rewritings: one with and the other without using the v2h operation. The former can be used on a SQL-92 system whereas the latter can exploit the implementation of v2h operation using the object-relational extensions. See [3] for illustrative examples of these rewritings.

Projection

Let the projection be on attributes $A1, \ldots, Ak$. We have, from the definition of the v2h operation:

$$\begin{aligned}&\pi_{A1,\ldots,Ak}(H) \\ &\quad = \quad \Omega^k(V) \qquad (3) \\ &\quad = \quad [\pi_{Oid}(V)] \supset\!\bowtie [\supset\!\bowtie_{i=1}^{k} \pi_{Oid,\,Val}(\sigma_{Key=`Ai'}(V))] \qquad (4)\end{aligned}$$

Selection

We discuss the usual case of a selection followed by projection. Let the selection predicate be $\wedge_{i=1}^{k}(Ai\ \theta\ `ai')$ and the projection be on

the first $k+m$ attributes, $m \geq 0$.

$$\pi_{A1,\ldots,Ak+m}(\sigma_{\wedge_{i=1}^{k} Ai\ \theta\ `ai'}(H))$$

$$= \pi_{A1,\ldots,Ak+m}(\sigma_{\wedge_{i=1}^{k} Ai\ \theta\ `ai'}(\Omega^{k+m}(V))) \quad (5)$$

$$= \pi_{\$1,\ldots,\$k+m}([\cap_{i=1}^{k} \pi_{Oid}(\sigma_{Key=`Ai' \wedge Val\ \theta\ `ai'}(V))] \supset\!\bowtie [\supset\!\bowtie_{i=1}^{k+m} \pi_{Oid,Val}(\sigma_{Key=`Ai'}(V))]) \quad (6)$$

A disjunctive selection

$$\pi_{A1,\ldots,Ak+m}(\sigma_{\vee_{i=1}^{k} Ai\ \theta\ `ai'}(H))$$

can be transformed by replacing the intersection $\cap_{i=1}^{k}$ in Eq. 6 with the union $\cup_{i=1}^{k}$.

Join

Take a horizontal table H having the scheme $(Oid, A1, \ldots, An)$ which is really a logical view over a vertical table V. Its join with a true horizontal table RH having the scheme $(R1, \ldots, Rr)$ is given by:

$$\pi_{A1,\ldots,Ak,Rk+1,\ldots,Rk+m}(H \underset{\wedge_{i=1}^{k} Ai\ \theta\ Ri}{\bowtie} RH)$$

$$= \pi_{A1,\ldots,Ak,Rk+1,\ldots,Rk+m}(\Omega^{k}(V) \underset{\wedge_{i=1}^{k} Ai\ \theta\ Ri}{\bowtie} RH) \quad (7)$$

$$= \pi_{\$1,\ldots,\$k,Rk+1,\ldots,Rk+m}([\bowtie_{i=1}^{k} \pi_{Oid,Val}(\sigma_{Key=`Ai'}(V))] \underset{\wedge_{i=1}^{k} \$i=Ri}{\bowtie} [RH]) \quad (8)$$

Aggregation

We use the following notation from [6] to specify aggregation:

$$\text{Grouping attributes}\ \mathcal{F}\ \text{Function list}\ (\text{Table name})$$

Function list consists of (*function, attribute*) pairs, where *function* can be one of the allowed aggregate functions such as SUM, COUNT, AVG, MAX, and MIN. The transformations are:

$$_{A1,\ldots,Ak}\mathcal{F}_{F\ Ak+1}(H)$$

$$= {}_{A1,\ldots,Ak}\mathcal{F}_{F\ Ak+1}(\Omega^{k}(V)) \quad (9)$$

$$= {}_{\$1,\ldots,\$k}\mathcal{F}_{F\ \$k+1} ([\pi_{Oid}(V)] \supset\!\bowtie [\supset\!\bowtie_{i=1}^{k} \pi_{Oid,Val}(\sigma_{Key=`Ai'}(V))]) \quad (10)$$

For aggregate functions, SUM, MIN, and MAX, which are unaffected by null values in the column being aggregated, Eq. 10 can be simplified to:

$$_{A1,\ldots,Ak}\mathcal{F}_{F\ Ak+1}(H) = {}_{\$1,\ldots,\$k}\mathcal{F}_{F\ \$k+1}(\supset\!\bowtie\!\sqsubset_{i=1}^{k} \pi_{Oid,Val}(\sigma_{Key=`Ai'}(V))) \quad (11)$$

SetOperations

The set operations cross product($\times$), union($\cup$), intersection($\cap$), and difference($-$) can be transformed by first applying the v2h operation on the vertical table(s) and then carrying out the desired operation.

Updates

Updates are easy. Insertion requires decomposing a data object into a set of attribute name and value pairs and inserting them into V with a common *Oid*. A predicate-based deletion requires determining the *Oid* set of objects satisfying the predicate and deleting the corresponding tuples from V. An update results in a change of the value field in some tuples in V. It can also cause some insertions and deletions.

Output

There may be need for transforming the result of an operation involving a vertical table back into the vertical format (e.g. for storing the result). This can be accomplished by applying the h2v operation on the result table.

3. IMPLEMENTATION

With the algebra described above in hand, we are in a position to develop a non-intrusive enablement layer on top of the database engine that hides from the user (application) the vertical table. A horizontal view H is defined for the vertical table V using an extended DDL:

CREATE HORIZONTAL VIEW H ON
VERTICAL TABLE V USING COLUMNS $(A_1, A_2, \ldots, A_n)$

where $A_{i=1,n}$ represent attribute names (keys) in the vertical table. The DDL is generated by the enablement layer. The user poses regular SQL queries over the view. The enablement layer parses the SQL query, validates it, and transforms it to another SQL query that runs against the underlying vertical table. It uses a query graph structure to facilitate this translation.

We consider three transformation strategies.

3.1 VerticalSQL

This implementation assumes only the SQL-92 level capabilities from the underlying database engine. The enablement layer uses the second set of equations for translating each of the algebraic operations given in Section 2 for this implementation. See [3] for an example.

3.2 VerticalUDF

This implementation attempts to exploit object-relational extensions to SQL, particularly the user-defined table functions. The underlying engine is extended with the table functions for v2h and h2v operations. The v2h table function reads tuples of vertical table sorted on *Oid* and outputs a horizontal tuple for each *Oid*. The h2v table function takes as input column names and a horizontal tuple and splits it into vertical tuples.

The enablement layer uses the first set of equations for translating each of the algebraic operations given in Section 2 for this implementation. For example, the projection query:

```
SELECT A1, A2 FROM H
```

is translated into[4]:

```
SELECT t.Attr1, t.Attr2
FROM V v , TABLE(v2h(v.Oid, v.Key, v.Val)) AS t(Oid,
Attr1, Attr2)
WHERE v.Key='A1' or v.Key='A2'
```

The query appears to be a Cartesian product between the vertical relation V and the table function v2h. What happens in effect is that the relevant fields from any qualifying tuple v after applying the select predicates on the V table are passed as parameters into the v2h function, which in turn produces horizontal tuples t from which the fields in the select list are extracted.

The v2h table function requires v tuples to be streamed in the *Oid* order so that it can buffer the key-value pairs until the *Oid*

[4]The actual translation is more complex and includes an additional clause in the join list for selecting distinct *Oid*'s from V.

changes. At that point, it can output the tuple corresponding to the horizontal view. Unfortunately, the current SQL syntax does not allow the specification of the order in which the tuples should be streamed into a table function and that causes problems. Note that a good plan for the above query will push down the *Key* predicates on $A1$ and $A2$ so as to select only the relevant tuples from the V relation. However, the output of this selection would generate tuples in *Key* or physical row-id order which is different from the *Oid* order required for `v2h`.

A workaround this problem is to introduce a join of the tuple stream produced by the selection with a table of *Oid*'s and cajole the optimizer to pick a merge sort join plan, thereby forcing a sort on *Oid*. By introducing this join and adjusting the optimization level for the the DB2 query optimizer, we could generate the correct plans. We were able to play similar tricks for other algebraic operations.

3.3 SchemaSQL

This implementation employs the non-intrusive strategies proposed for the SchemaSQL implementation [11]. Specifically, we implemented *unfold I* and *unfold II* strategies[5]. These strategies result in SQL translations that are different from the ones given in Section 2. See [3] for details.

4. PERFORMANCE EXPERIMENTS

We now present the results of our extensive experiments to study the performance of the alternative implementations of the enablement layer just described. We include in this study the performance comparison with the horizontal representation as well as binary representation discussed in Section 1.3. They will be referred to as HorizontalSQL and Binary respectively.

4.1 Experimental Setup

All experiments were run on a 600 MHz dual processor Intel Pentium machine with 512 MB of physical memory. The operating system was Windows NT 4.0 and the database system used was DB2 UDB 7.1. The machine had two 30GB IDE drives. Data was placed on one disk and the temporary table spaces and the logs were created on the other. The buffer pool size was set to 50MB and the prefetch size to 512KB.

To study performance characteristics over a wide range of operating regions, we used synthetic data that allowed us to vary the following parameters:

- Number of columns in the horizontal table
- Number of rows in the horizontal table
- Non-null density (i.e. percentage of field values that are not null)
- Selectivity of a predicate for each column
- Number of distinct values in each column
- Size of each column

Given a set of these parameter values, we first generated data in the horizontal format and then transposed it into its equivalent vertical and binary formats. We kept the size of a table (number of rows $\times$ number of columns in a row) constant by adjusting the number of rows as we varied the number of columns. See [3] for details of the data generation algorithm.

We generate the following schemes for horizontal, vertical, and binary tables respectively:

[5] [11] proposed another strategy, called *unfold III*, which avoids the cost of creating temporary tables incurred by *unfold I* and *unfold II*. However, as pointed out in [11], a non-intrusive implementation of *unfold III* turns out to be less efficient than *unfold I* due to the tuple-at-a-time nature of the implementation.

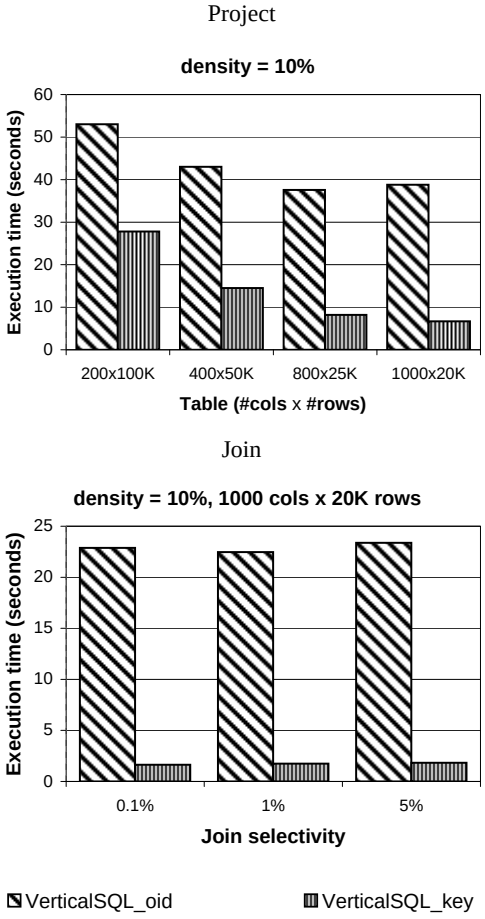

Figure 3: Clustering by *Key* versus *Oid*

H (OID INTEGER, A0 VARCHAR(X), A1 VARCHAR(X),. . ., An VARCHAR(X))
V (OID INTEGER, KEY CHAR(K), VAL VARCHAR(X))
TAB_Aj (OID INTEGER, VAL VARCHAR(X))

where X is the size in number of bytes (set to 16) and K is the size of the key field (set to 5). There were n binary tables TAB_Aj corresponding to n columns in H. The queries for various operations were generated using the additional parameters such as the columns involved in the operation and selectivities.

4.2 Layout

The data for horizontal as well as binary tables was clustered by *Oid*. For the vertical table, we have two choices for the physical layout: i) cluster by *Oid*, or ii) cluster by *Key*. We performed extensive experiments and found that clustering by *Key* consistently resulted in much higher performance than clustering by *Oid*. Figure 3 shows the performance of clustering by *Key* versus *Oid* for projection and join operations. The settings for these experiments are exactly the same as used for experiments reported later in the paper in Figures 4 and 7 respectively. We have not included those graphs, but we found large gaps in performance for selection as well as aggregation operations.

To understand this performance difference, let us consider the projection operation. The SQL translation of the projection operation corresponding to the algebraic transformation presented in Eq. 4 in Section 2 contains selection predicates on *Key* values. After applying the selection predicate using an index on the *Key* col-

umn, the qualifying tuples are fetched. If the data is physically laid out in the *Key* order, we get the benefit of clustered I/O. If, on the other hand, the data is clustered by *Oid*, fetching the tuples results in unclustered I/O which is very inefficient.

4.3 Indices

Every column involved in a query on the horizontal table was indexed. We also indexed both the columns of every binary table. For the vertical table, we indexed each of the three columns.

Note that we end up indexing the entire data in the vertical table. Contrast this to the case of the horizontal table where just the columns involved in the typical query workload are indexed. We found that the total size of the indices for vertical table was typically two orders of magnitude larger compared to the horizontal indices and large portions of the vertical indices were not useful for any of the queries. Having large indices adversely impacts the performance of the vertical representation because of the time spent in loading the indices and the increase in path length due to deeper index trees. It would have helped if the database supported partial indices [20] that allow only the rows of interest to be indexed.

4.4 Performance Results

We now summarize the important results from a very large number of experiments we performed. We will present the results for project, select, join and aggregation operations. We flushed buffer pool, main memory and file system cache before the start of each run to get cold start numbers.

We will report the performance of a single operation at a time in order to isolate the trade-offs for each operation. Of course, a typical database query contains a combination of operations. We ran several such composite queries but did not find any trend we could not predict having understood the trade-offs for individual operations.

We found the implementations using the SchemaSQL strategies performed 2-3 times slower compared to VerticalSQL[6]. The culprit was the creation of a large number of intermediate temporary tables, a problem recognized by the implementors of SchemaSQL [11]. We therefore do not include the SchemaSQL numbers in the results.

In the initial set of results, we include numbers for HorizontalSQL, VerticalSQL and Binary. Later in Section 4.5, we present results that are indicative of the performance achievable from a VerticalUDF implementation.

Projection

Figure 4 shows the performance of various strategies for the projection operation. The number of projected columns is 10, thereby requiring 10-way joins with the Binary and VerticalSQL strategies. In each graph, the execution times are shown for four horizontal tables and their equivalent vertical and binary tables. The horizontal tables differ in the number of rows and columns but their total size (#rows × #columns) is kept constant. The number of rows decreases as we move from left to right in the graphs (which explains the reduction in the execution time for all the strategies). We show graphs for non-null density ρ = 5% and 10%.

The surprising result from these experiments is that VerticalSQL uniformly outperforms HorizontalSQL in spite of requiring multiple joins. The superior performance of Binary over HorizontalSQL reconfirms the results in [4] [7] [9].

The reason for the relative poor performance of the horizontal format is that the whole tuple needs to be fetched before the relevant fields can be extracted. There is additional cost of finding where the relevant fields for a tuple lie on a page, which can be substantial for a wide tuple with a large number of fields. In the case of a vertical table, the index on the *Key* field allows only the tuples corresponding to attributes participating in the projection to be retrieved. The net is a decrease in total I/O.

[6]This observation should not be construed as a negative statement against SchemaSQL, which addresses a more general problem.

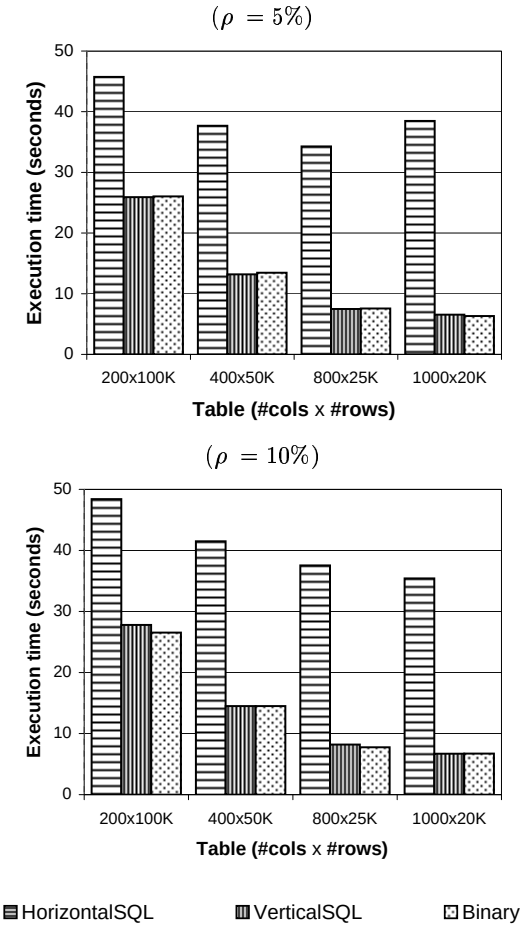

Figure 4: Projection performance (10 cols)

Between Binary and VerticalSQL, Binary performs slightly better. A tuple in the binary representation contains only the attribute value, whereas a tuple in the vertical representation contains both attribute name (key) and value. Thus, the total I/O will be less in the binary scheme. Moreover, the vertical scheme requires an additional selection on the key field for locating tuples having the desired attribute name, which is not needed in the binary scheme since it has a separate table for each attribute. Later in Section 4.5, we see how the performance of the vertical scheme can be made better than binary by using the VerticalUDF implementation.

Figure 5 shows the effect on performance as the number of projected columns is varied. The experiments were run for the dataset 1000 × 20K and for non-null densities ρ = 5% and 10%. The relative performance of the various strategies remains the same as in the previous experiments when number of projected columns was fixed at 10 (Figure 4).

We see that the performance of HorizontalSQL is insensitive to the number of projected columns. This trend can be understood by recalling that when the tuples are wide, the projection performance in the horizontal scheme is dominated by the cost of fetching tu-

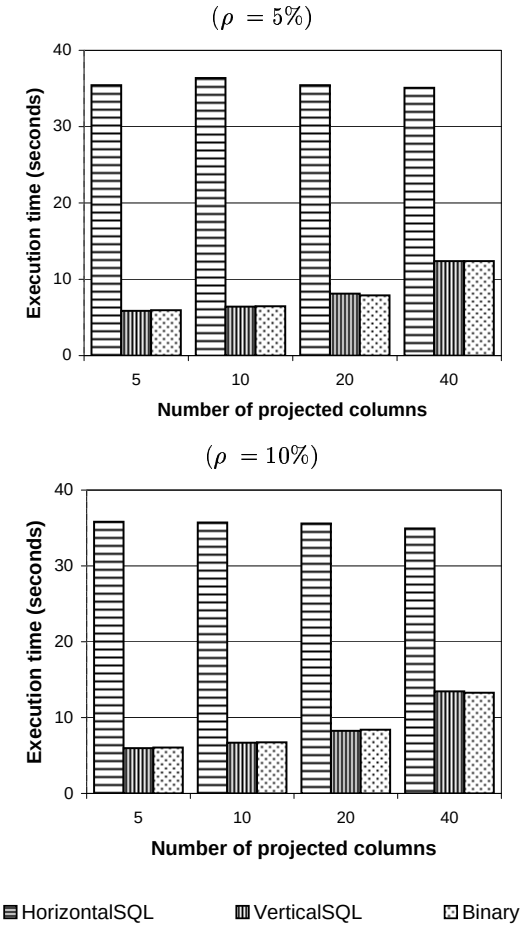

Figure 5: Impact of varying the number of projected columns (1000 cols × 20K rows)

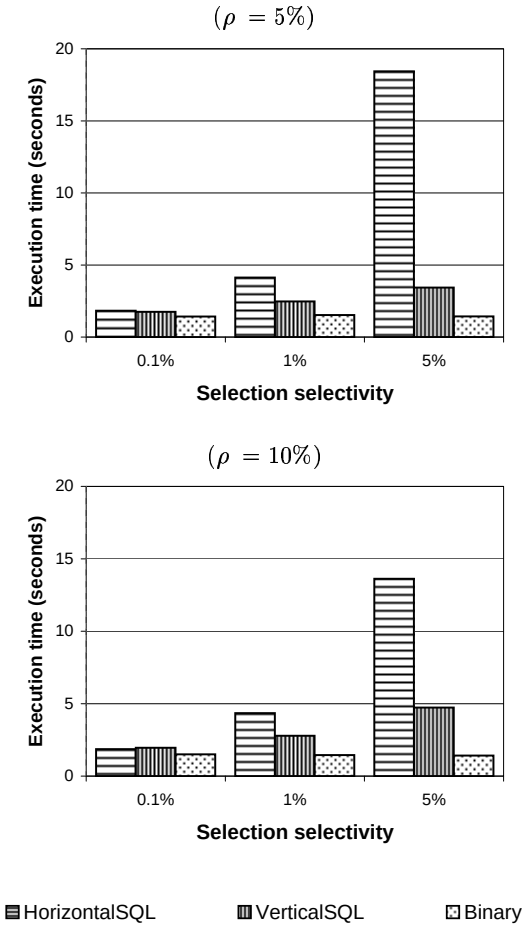

Figure 6: Selection performance (1000 cols × 20K rows)

ples. The performance of Binary as well as VerticalSQL improves as the number of projected columns decreases. This trend is quite understandable since a decrease in number of projected columns results in a decrease in number of joins these schemes need to perform. However, the execution time increases rather slowly as the number of projected columns increases and VerticalSQL and Binary continue to outperform the horizontal scheme.

Selection

Selection experiments were run using the following query:

```
SELECT A40, A200 FROM H WHERE A200 = 'A200V0'
```

The experiments were run for the dataset 1000 × 20K with non-null densities ρ = 5% and 10%. The selectivity of the selection predicate was set to 0.1%, 1% and 5%. Figure 6 shows the results.

The plan for HorizontalSQL applies the selection predicate on $A200$ and then fetches the qualifying tuples to extract attributes in the select list. However, since the data is clustered by *Oid*, this causes unclustered I/O of wide tuples, hurting the performance of HorizontalSQL. As the predicate becomes less selective, the amount of I/O increases and the query performance degrades.

VerticalSQL has to apply a predicate based on the value of $A200$ as well as the *Key* predicate for the projected column $A40$. The fetch following the application of the *Key* predicate on $A40$ causes clustered I/O while the one following the application of the value predicate on $A200$ causes unclustered I/O. It then sorts each of these streams on *Oid* to do the join. However, since the tuples are narrow, VerticalSQL ends up performing better than HorizontalSQL.

For Binary, the optimizer chooses a table scan for the $A200$ table, followed by a sort, and a merge sort to join it with the $A40$ table. Since a table scan was chosen instead of an index scan, we do not see much effect of selectivity on the query execution time. Because of smaller indices and narrower tuples, Binary performs a little better than VerticalSQL.

Join

For this set of experiments, we joined H with a horizontal table *HR* whose scheme was:

```
HR(A1 VARCHAR(16), A2 VARCHAR(16))
```

HR has $\rho \times H$ rows, each of which has no null value. The join query was:

```
SELECT h1.A40, h2.A2 FROM H h1, HR h2 WHERE h1.A1=h2.A1
```

All the columns involved in the query were indexed.

The experiments were run for the dataset 1000 × 20K with non-null densities ρ = 5% and 10%. The join selectivity was set to 0.1%, 1% and 5%. Figure 7 shows the results.

Both Binary and VerticalSQL considerably outperform HorizontalSQL, with Binary performing a little better. The join selectivity

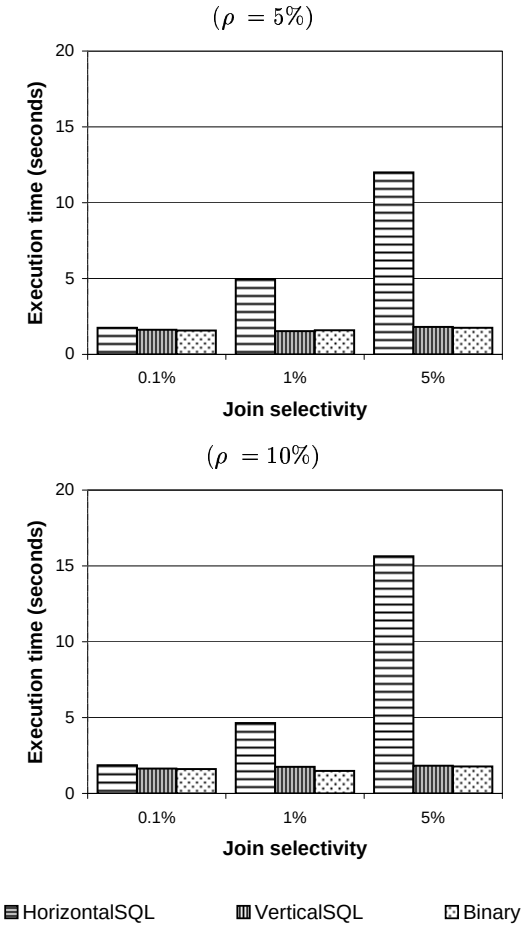

Figure 7: Join performance (1000 cols × 20K rows)

did not exhibit much affect on the performance of Binary and VerticalSQL. The query plans for both first compute a horizontal relation consisting of attributes *Oid*, $A1$ and $A40$. For VerticalSQL, this involves selection on the key predicates, fetching the tuples, sorting them on *Oid*, and doing a merge sort join. For Binary, the selection on the key predicate is not required since each attribute has its own table (which explains the slight performance advantage). It therefore only requires fetching the tuples and joining them on *Oid*. For both the schemes, this interim result is then joined with the *HR* table using a merge sort join. It is only this last step that is affected by the join selectivity. Since the cost of entire plan is dominated by the I/O required to fetch the input tuples as opposed to the final join, hence the execution time is not significantly affected by join selectivity.

The plan for the Horizontal scheme involves a nested loop join with the *HR* table as the outer and an index scan on the join column in the inner H table. However, since we have an additional column in the select list, a fetch on the inner table does unclustered I/O to get the tuples (H is clustered on *Oid*). This unclustered I/O of wide tuples result in the poor performance of HorizontalSQL. The amount of I/O depends on the number of tuples in H that join. Hence the query performance degrades with increasing join selectivity.

Aggregation

We measured aggregation performance by using the following query:

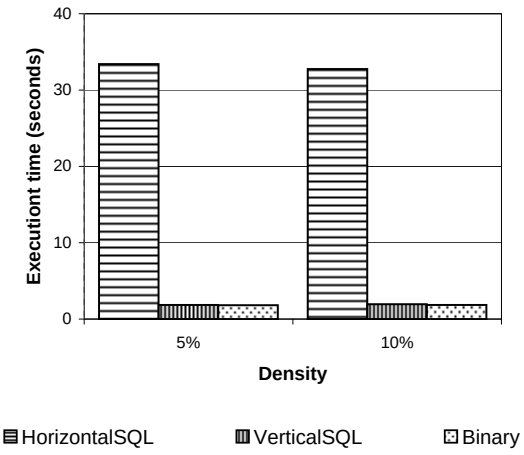

Figure 8: Aggregation performance (1000 cols × 20K rows)

```
SELECT A500, AVG(LENGTH(A0)) FROM H
GROUP BY A500
```

The experiments were run for the dataset 1000 × 20K with non-null densities ρ = 5% and 10%. The number of groups in the result was 100 for both densities. The average number of tuples processed per group was 10 and 20 for densities ρ = 5% and 10% respectively. Figure 8 shows the results.

HorizontalSQL needs to fetch the entire tuple to extract the fields required in the aggregation computation. The horizontal tuples being wide, HorizontalSQL takes the performance hit of fetching a large number of unnecessary fields. The query cost is dominated by the I/O, not by the computation of the aggregation function. Hence we do not see much difference in performance as the density of the data increases (which mostly increases the aggregation function computation cost but affects the I/O marginally).

VerticalSQL and Binary have comparable performance, with Binary performing marginally better. The query plans for both the strategies are similar to doing a projection of 2 columns (a `v2h` operation), followed by the computation of the aggregation function. However, the execution time of the query is dominated by the I/O time to implement `v2h` and therefore we only see a marginal increase in execution time as the density increases.

4.5 Exploiting the Object-Relational Features

We saw from the performance results just presented that VerticalSQL uniformly outperforms HorizontalSQL but slightly under performs Binary. We show in this section that with a little better support from table functions, the VerticalUDF strategy can outperform Binary. VerticalUDF can avoid multi-way joins VerticalSQL performs to assemble the attribute values of a tuple. What needs to be ensured is that the relevant tuples from the vertical table are streamed into the `v2h` table functions in the *Oid* order. The table function then can do the assembly and output the horizontal tuple.

Let us first consider the projection operation. Figure 9 compares the performance of VerticalUDF to Binary and VerticalSQLfor the same datasets as in Figure 4. We show the graph for ρ = 10%; the performance advantage of VerticalUDF was relatively larger for ρ = 5%. The performance numbers for VerticalUDF were obtained after adding all the contortions described in Section 3.2 for forcing the tuples to stream into the `v2h` table function in the *Oid* order. Thus, the performance numbers for VerticalUDF should be viewed

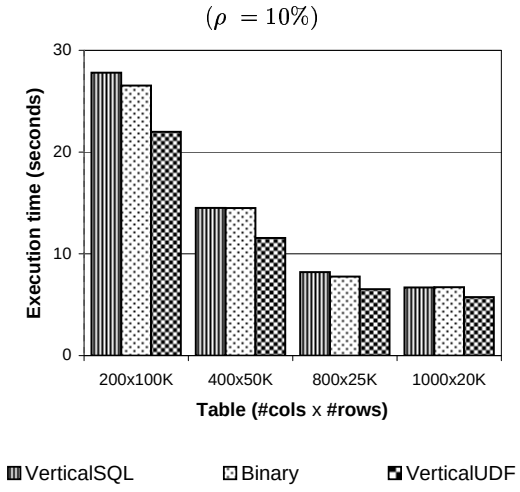

Figure 9: Projection performance (Projection of 10 cols)

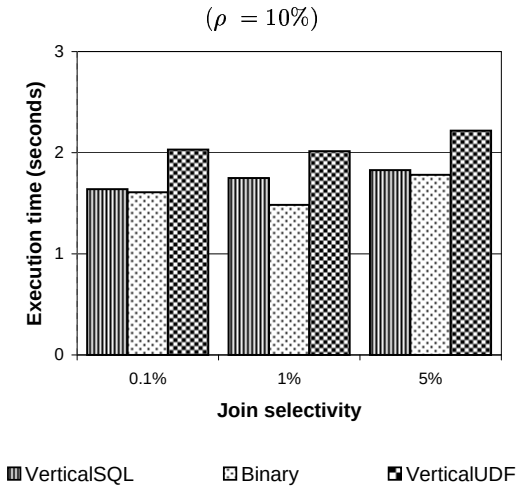

Figure 10: Join performance (1000 cols $\times$ 20K rows)

as the worst-case numbers. In spite of the unnecessary performance penalty, VerticalUDF, performs uniformly better than both Binary and VerticalSQL. The main reason for this performance win is the avoidance of multi-way joins present in Binary and VerticalSQL.

Now consider the join query used in the performance evaluation in Section 4.4. Figure 10 shows the performance of VerticalUDF, VerticalSQL, and Binary for the same datasets and join selectivities as used in Figure 7. For executing this query using the VerticalUDF strategy, we would like to first apply the select predicates on the key columns prior to streaming the tuples in the *Oid* order into the table function. However, as discussed earlier, there is no way to specify this order on the output of the select operation. The workaround again is to create an additional artificial join on *Oid*'s and trick the optimizer into choosing a merge-sort plan for this join, thereby creating a tuple stream sorted on *Oid*. The performance numbers for VerticalUDF, therefore, should again be viewed as the worst-case numbers. It is remarkable that the performance of VerticalUDF comes so close to Binary in spite of all the unnecessary overhead. Also, in a typical query, joins are often followed several projections. The performance gain in the projection operation allows VerticalUDF to outperform Binary for such composite queries. We found similar issues for selection and aggregation.

We would like the table function to be able to control the order in which input arguments are supplied to it. The need for such facility has been identified for other applications also [18]. If the table function syntax is extended with this feature and the optimizer takes advantage of it, we would avoid the performance penalty of introducing additional operations just to force the desired order on input arguments. In that case, we expect VerticalUDF to outperform the other strategies.

5. CONCLUSIONS

Emerging applications such as e-commerce and portals are creating new threats and opportunities for database technology. The prevalent conventional horizontal representation is optimized for applications in which the data is dense and evolves slowly. The new generation of applications require data schemas that are rapidly evolving and sparsely populated.

To meet the requirements of these applications, many commercial software systems have converged on a 3-ary vertical representation for storing objects in a table. This paper recounts our experience from building an e-marketplace using this vertical scheme for representing data. The application was built using IBM Websphere Commerce Server running on top of DB2. Our two main contributions are:

- Design of an enablement layer that hides the complexity of the queries over the vertical table and gives a horizontal view of the vertical representation to the user (application). We provide transformation algebra and techniques for its non-intrusive implementation on top of a SQL database system.
- A thorough investigation of the performance trade-offs of the vertical representation and a comparison of its performance with the horizonal and binary (2-ary) representations. The key results are:
  - The performance of the vertical representation is sensitive to the choice made for clustering the data. Clustering on *Key* has much higher performance than clustering on *Oid*.
  - The vertical representation uniformly outperforms horizontal representation for sparse data (in spite of the extremely efficient representation of null values in DB2).
  - The performance of the vertical representation using only the SQL-92 capabilities is comparable to the binary representation, the latter performing a little better. By using table functions, the vertical representation starts performing better than binary for the projection operation. If the table function could provide some extra functionality (see below), the vertical representation can outperform binary representation for other operations also.

  The major arguments in favor of the vertical representation have been its flexibility in supporting schema evolution and manageability (single table versus as many tables as the number of attributes in the binary scheme). Based on the results of this study, we can provide the following matrix for comparing the three representations:

We finally give a wish list of the capabilities we would like from the database system to be able to further enhance the performance of the vertical representation:

	Horizontal	Vertical	Binary
Manageability	+	+	-
Flexibility	-	+	-
Performance	-	+	+

- *Partial indices* We create an index on each of the three columns of the vertical table. In the process, we end up indexing the entire data in the vertical table. Having database support for partial indices [20] that allow only the rows of interest to be indexed will help improve the performance of the vertical representation.

- *Enhanced table functions* The table function syntax needs to be extended with additional clauses to specify the required order of input arguments. This facility is critical for benefitting from the `v2h` table function for assembling attribute values from the vertical table into a horizontal tuple without performing multiple joins.

- *First class treatment of table functions* Table functions are currently not treated as first class objects during the query optimization phase. Current systems do not allow table functions to register the ordering of tuples they receive, the output cardinality, or the order property for the tuples they produce. Because of these limitations, optimizers often produce less than optimal plans for executing queries that include table functions.

- *Native support for* `v2h` *and* `h2v` *operations* Since `v2h` and `h2v` operations are fundamental primitives for the efficient execution of queries over vertical table, they should be supported natively by the database system for best results.

Acknowledgments

We wish to thank Jay Shanmugasundaram for providing us the XQGM code, which we used in the enablement layer to represent parsed queries.

6. REFERENCES

[1] Storing RDF in a relational database. http://www-db.stanford.edu/ melnik/rdf/db.html.

[2] R. Agrawal, R. Bayardo, D. Gruhl, and S. Papdimitriou. Vinci: A service-oriented architecture for rapid development of web applications. In *WWW10*, Hongkong, May 2001.

[3] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce data. Research report, IBM Almaden Research Center, San Jose, CA 95120, June 2001. Available from `http://www.almaden.ibm.com/cs/quest`.

[4] P. Boncz and M. Kersten. MIL primitives for querying a primitive world. *VLDB Journal*, 8(2):101–119, October 1999.

[5] G. P. Copeland and S. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD International Conference on Management of Data, Austin, Texas, May 28-31, 1985*, pages 268–279.

[6] R. Elmasri and S. B. Navathe. *Fundamental of Database Systems*. Benjamin/Cummings, Redwood City, California, 1989.

[7] D. Florescu and D. Kossman. A performance evaluation of alternative mapping schemes for storing XML data in a relational database. Technical report, INRIA, France, May 1999.

[8] Internationl Business Machines. *IBM Intelligent Miner User's Guide*, Version 1 Release 1, SH12-6213-00 edition, July 1996.

[9] S. Khoshafian, G. P. Copeland, T. Jagodis, H. Boral, and P. Valduriez. A query processing strategy for the decomposed storage model. In *Proceedings of the Third International Conference on Data Engineering, February 3-5, 1987, Los Angeles, California, USA*, pages 636–643.

[10] R. Krishnamurthy, W. Litwin, and W. Kent. Language features for interoperability of databases with schematic discrepancies. In *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data, Denver, Colorado, May 29-31, 1991*, pages 40–49.

[11] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. On efficiently implementing SchemaSQL on an SQL database system. In *Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland*, pages 471–482.

[12] L. V. S. Lakshmanan, F. Sadri, and S. N. Subramanian. SchemaSQL – a language for querying and restructuring multidatabase systems. In *Proceedings of 22nd International Conference on Very Large Data Bases, September 1996, Bombay, India*.

[13] W. Litwin and A. Abdellatif. Multidatabase interoperability. *IEEE Computer*, 19(12):10–18, 1986.

[14] R. J. Miller. Using schematically heterogeneous structures. In *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 189–200.

[15] M. Minsky. A framework for representing knowledge. Technical Report MIT-AI Laboratory Memo 306, Massachusetts Institute of Technology Artificial Intelligence Laboratory, June 1974.

[16] M. Missikoff. A domain based internal schema for relational database machines. In *Proceedings of the 1982 ACM SIGMOD International Conference on Management of Data, Orlando, Florida, June 2-4, 1982*, pages 215–224.

[17] K. A. Ross. Relations with relation names as arguments: Algebra and calculus. In *Proceedings of the Eleventh ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, June 2-4, 1992, San Diego, California*, pages 346–353.

[18] 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), July 2000.

[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):360–383, 2000.

[20] M. Stonebraker. The case for partial indexes. *SIGMOD Record*, 18(4):4–11, 1989.

[21] M. Wang, B. R. Iyer, and J. S. Vitter. Scalable mining for classification rules in relational databases. In *IDEAS 1998*, pages 58–67.