Column-Stores For Wide and Sparse Data

Daniel J. Abadi
MIT
dna@csail.mit.edu

ABSTRACT

While it is generally accepted that data warehouses and OLAP workloads are excellent applications for column-stores, this paper speculates that column-stores may well be suited for additional applications. In particular we observe that column-stores do not see a performance degradation when storing extremely wide tables, and column-stores handle sparse data very well. These two properties lead us to conjecture that column-stores may be good storage layers for Semantic Web data, XML data, and data with GEM-style schemas.

1. INTRODUCTION

Although the idea of a column-oriented database or "column-store" has been around for a while [12], there has been a recent revival in column-oriented research and commercial products [1, 8, 9, 13, 14, 16]. A column-store stores each attribute in a database table separately, such that successive values of that attribute are stored consecutively. This is in contrast to most common database systems (e.g., Oracle, DB2, SQLServer, Postgres, etc.), "row-stores", where values of different attributes from the same tuple are stored consecutively (i.e., column-stores store data column-by-column, while row-stores store data row-by-row).

The trade-offs between column-stores and row-stores are still being explored. The following are some cited advantages of column-stores:

- **Improved bandwidth utilization** [12]. In a column-store, only those attributes that are accessed by a query need to be read off disk (or from memory into cache). In a row-store, surrounding attributes also need to be read since an attribute is generally smaller than the smallest granularity in which data can be accessed.
- **Improved data compression** [3]. Storing data from the same attribute domain together increases locality and thus data compression ratio (especially if the attribute is sorted). Bandwidth requirements are further reduced when transferring compressed data.
- **Improved code pipelining** [8, 9]. Attribute data can be iterated through directly without indirection through a tuple interface. This results in high IPC (instructions per cycle) efficiency, and code that can take advantage of the super-scalar properties of modern CPUs.
- **Improved cache locality** [6]. A cache line also tends to be larger than a tuple attribute, so cache lines may contain irrelevant surrounding attributes in a row-store. This wastes space in the cache and reduces hit rates.

On the the other hand, the following are some disadvantages of column-stores:

- **Increased disk seek time**. Disk seeks between each block read might be needed as multiple columns are read in parallel. However, if large disk pre-fetches are used, this cost can be kept small.
- **Increased cost of inserts**. Column-stores perform poorly for insert queries since multiple distinct locations on disk have to be updated for each inserted tuple (one for each attribute). This cost can be alleviated if inserts are done in bulk.
- **Increased tuple reconstruction costs**. In order for column-stores to offer a standards-compliant relational database interface (e.g., ODBC, JDBC, etc.), they must at some point in a query plan stitch values from multiple columns together into a row-store style tuple to be output from the database. Although this can be done in memory, the CPU cost of this operation can be significant. In many cases, reconstruction costs can be kept to a minimum by delaying construction to the end of the query plan [4].

These advantages and disadvantages need to be considered when deciding whether a row- or column-store should be used for a particular application. Certainly it seems that data warehouses and OLAP workloads, with their batch writes, high bandwidth requirements, and query plans rife with table scans are well suited for column-stores [8, 16]. In this paper, we take the position that column-stores should be considered for other applications as well.

This article is published under a Creative Commons License Agreement (http://creativecommons.org/licenses/by/2.5/).
You may copy, distribute, display, and perform the work, make derivative works and make commercial use of the work, but you must attribute the work to the author and CIDR 2007.

3rd Biennial Conference on Innovative Data Systems Research (CIDR) January 7-10, 2007, Asilomar, California, USA.

In particular, we make two observations about column-stores, derived from the first two advantages listed above, that open up the potential for other column-store applications. These observations are:

- **Wide tables are not a problem for column-stores**. In a column-store, if a query accesses a fixed number of attributes from a table, it does not matter if the table is 5 columns wide or 5 million. Only those columns that are needed by the query need to be read in and processed. In a row-store, these extra columns cannot so easily be ignored.
- **Sparse columns are not a problem for column-stores**. Since column-stores can choose a domain specific compression algorithm for each attribute, an appropriate NULL suppression algorithm can be chosen for each column, selected based on the column sparsity.

Consequently, we believe that column-stores should be considered to store data with wide, sparse schemas. In this paper, we focus on the Semantic Web, XML, and databases with GEM-style schemas (an extension of the relational model to include richer semantics with respect to the notion of an entity [19]) as applications where these advantages might be beneficial.

In the next section, we describe in more detail why column-stores handle wide tables well and why column-stores handle sparse data well. Then in Section 3, we describe why this allows columns-stores to be applied to the Semantic Web, XML, and databases with GEM-style schemas.

2. TWO OBSERVATIONS

In this section, we describe in detail why column-stores can store wide, sparse data well.

2.1 Wide Schemas

Agrawal et. al. [5] point out situations where it would be desirable to have very wide schemas, but where performance can be problematic. For example, imagine an e-commerce marketplace for the electronics industry that consolidates information about parts from thousands of manufacturers. The catalog may contain two million parts classified into 500 categories with 4000 attributes per category. Storing each category as a 4000-column wide table is a potential performance disaster in a row-store, since a table scan for a query that accesses only a small percentage of these attributes will result in a significant waste of disk bandwidth.

The problem is that data is read from disk in blocks (and from memory in cache lines) and these data blocks are generally much larger than an individual attribute, even usually larger than a tuple. Consequently, in order to read in a desired attribute, one must read in the surrounding unnecessary attributes as well, wasting bandwidth. In a column-store, since values from the same attribute are stored together consecutively, this is not a problem.

Thus, column-stores open up the possibility for schemas that are orders of magnitude wider than current limitations. Note also that adding new columns to a column-store is a trivial task.

2.2 Sparse Attributes

In a row-oriented DBMS there are a variety of options for handling a NULL attribute. One option is to prefix the tuple with a bit-string that indicates which attributes are NULL. If an attribute is NULL, the corresponding bit in the bit-string is set, and either a "don't care" symbol is inserted into the tuple or the attribute is omitted all together. The former option is a waste of space, while the latter option slows down tuple random access; the nth attribute is no longer located at the nth position in the tuple (which it would be if n is in the fixed-width attribute section of the tuple); instead the NULL bit-string has to be processed to derive where the nth attribute is located.

There are a few other ways NULLs can be handled. Instead of storing a bit-string in front of the tuple, a value in the attribute domain can be reserved to represent NULL. However, this option always wastes space for NULLs. Alternatively (as is done in Oracle 10g), every attribute in a tuple can be preceded by a length attribute. NULLs still waste a little space (1 byte to indicate a length of 0), and tuple random access is still slow (every attribute becomes a variable length attribute). Another option, for very sparse data, is an interpreted attribute layout [7] where a tuple is a set of attribute-value pairs, and omitted attributes are assumed to be NULL. However, this scheme reduces performance of tuple random access - the nth attribute can only be found by scanning the attribute-value list rather than jumping to it directly. Thus, for each of the above described options for handling NULLs, NULLs either waste space or hinder performance. (Note that wasting space also has performance consequences since table scans become slower).

In a column-oriented DBMS, NULLs are much easier to handle, and impose a significantly smaller performance overhead. From a high level, NULLs in a column-store can be thought of as another potential column value that can be compressed using column-oriented compression. If a column is sparse, these NULLs can be run-length encoded; otherwise other encoding techniques can be used (described below). Different NULL compression techniques can be used for different columns of different sparsities.

To demonstrate the ability of column-stores to handle NULL data, we extended C-Store, an open source column-oriented database that we have been building for the past several years [2], with three techniques for handling NULLs. In each case, a page is divided into three parts: a header indicating the position (the ordinal offset of a value within the column, also sometimes called a virtual tupleID) range that the page covers and the number of non-NULL values inside that range; a list of the non-NULL values in that page; and a representation of the positions (tupleIDs) for that column where the value is not NULL. Depending on the sparsity of the column, these positions can be represented in one of three ways: for sparse columns these positions are just a list (see Figure 1a); for columns with intermediate sparsity these positions are represented using a bit-string with '1's located at the non-NULL positions (see Figure 1b); and for low sparsity columns non-NULL positions are represented using position ranges (see Figure 1c).

Thus, from a space overhead, NULLs take up minimal space.

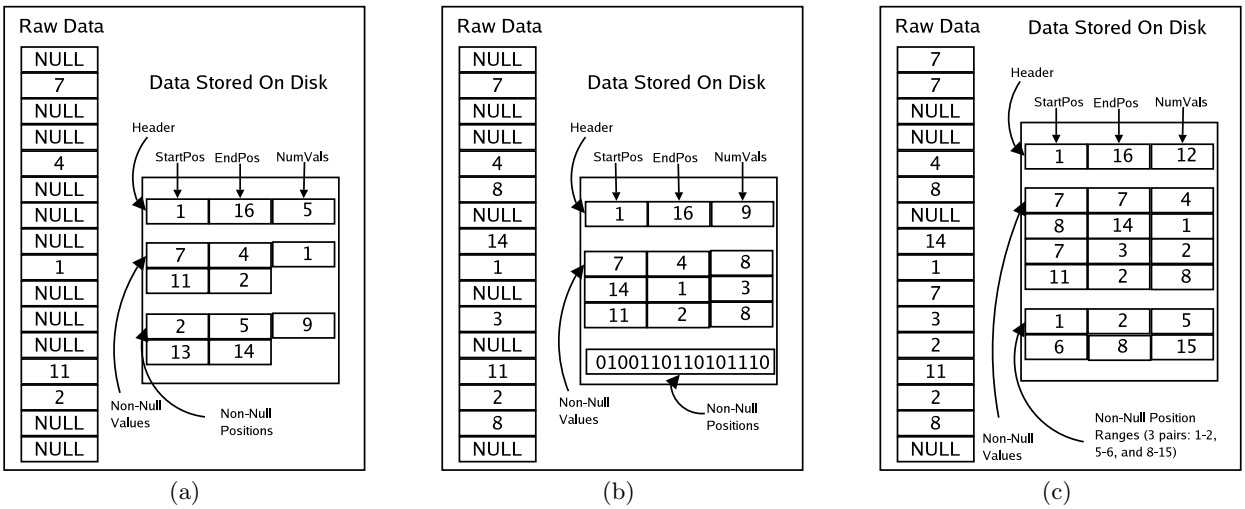

Figure 1: Positions represented using a list (a), a bit-string (b), and as ranges (c) for sparse columns

NULL values are omitted from the column. The cost is the additional overhead of representing the positions of the non-NULL values. For the position list scheme, the cost is an extra 32 bits per non-NULL value. However, since this is only used for sparse columns, for a column with 98% sparsity, an average of 0.65 bits are wasted per NULL value. For the bit-string scheme, the total space needed to represent the positions is equal to 1 bit per value in the column (NULL or non-NULL). Thus, for a column with 50% sparsity, this cost is 2 bits per NULL value. For the position-range scheme, the total space needed to represent the positions is equal to 64 bits per non-contiguous NULL value. But since this scheme is only used for dense columns, the total space wasted on NULLs is small. Note that if the column is sorted (or secondarily sorted), long runs of NULLs will appear, and this scheme will perform particularly well.

The key difference from row-stores is that while these schemes avoid wasting space in storing NULL data, they do not suffer the tuple attribute extraction cost that similar schemes in row-stores do. Consider the following simple example: a predicate needs to be applied to a column that may contain NULLs (say for the Xth column in the table). In a row-store, for each tuple, the Xth attribute must be found. For schemes that save space by not representing NULLs, the Xth attribute will not be in a fixed location in the tuple, so iteration through the tuple header or the X-1 attributes must occur. Further, if there is no index on that attribute, this Xth attribute must be identified in every tuple - even for those tuples whose value is NULL in the Xth attribute. In contrast, for column-stores, finding the Xth attribute is trivial (simply jump to the Xth column). Further, only the not-NULL values need to be iterated through since the NULL values are not stored. The list of positions for which the predicate succeeded is determined by iterating through the position representation in parallel with the non-NULL values of that column. (See [4] for a more detailed description of how selections are performed in C-Store).

As a case in point, we show the results from a simple experiment performed on a version of C-Store we extended to handle sparse attributes. We created a table with ten columns and filled them with 10,000,000 values drawn randomly from an integer domain. A percentage of the values in each column were given a NULL value according to the table sparsity (multiple tables were created with varying sparsities ranging from 0 sparsity – no NULLs, to 99% sparsity – 99 out of every 100 values were NULL). The sparsity of each column in the table was the same.

We applied the following query to each table of varying sparsity[1]:

```
SELECT count(*)
FROM T
WHERE T.1 > CONST
   OR T.2 > CONST
   OR T.3 > CONST
   OR T.4 > CONST
   OR T.5 > CONST
   OR T.6 > CONST
   OR T.7 > CONST
   OR T.8 > CONST
   OR T.9 > CONST
   OR T.10 > CONST
```

Thus, a selective predicate is applied to each column in the table, and the result aggregated. Figure 2(a) shows the time needed to run this query as the column sparsity is varied. This number is compared with a naive version of C-Store that wastes space on NULLs by physically storing them, and the best possible query time (i.e., if the NULLs were not present – instead of the table being 50% sparse, it is 50% smaller). The difference between the sparse C-Store line and the target performance line is thus the overhead imposed by C-Store for representing and processing the NULLs in the column. As can be seen, this overhead is small.

(Our benchmarking system is a 3.0 GHz Pentium IV, running RedHat Linux, with 2 Gbytes of memory, 1MB L2

[1]We chose 2147481974 to be the constant value since the was the most selective predicate that could be applied while making sure that each column returned at least one selected value which was useful to cross-check query results between C-Store and the commercial row-store we experiment with.

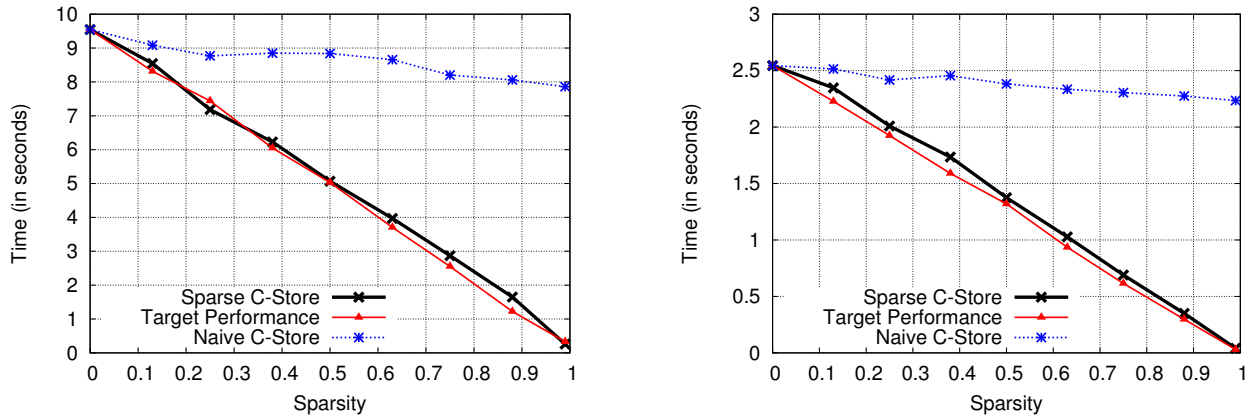

Figure 2: Total query time (a) and its CPU component (b) of running an aggregation query on a table with varying sparsity on C-Store

cache, and 750 Gbytes of disk. The disk can read cold data at 50-60MB/sec.)

Since the query is simple, it is disk limited, so in essence Figure 2(a) is just showing the time needed to read the table off disk. Figure 2(b) shows the CPU component of query time, thus showing that the processing overhead of NULLs is also small.

Figure 3(a) and (b) shows the graphs when the same query is run on the same data on a popular commercial row-store system. The time needed to run this query as the column sparsity is varied is again compared to the best possible query time (i.e., if the NULLs were not present – instead of the table being 50% sparse, it is 50% smaller). Clearly, the overhead imposed by the row-store for representing and processing the NULLs is larger than C-Store.

The differences in the total query times between C-Store and the commercial row-store (even for dense data) are not the same since the column-store uses vectorized code for predicate application and has a smaller per-tuple storage overhead. For this reason, it is more interesting to compare each system against the target time rather than against each other.

3. THREE WIDE, SPARSE APPLICATIONS

Now that we have established that column-stores are a good way to store wide, sparse tables, we look at three examples where these advantages of columns-stores are very important, such that column-store technology may be applied to areas other than data warehouses and OLAP workloads.

3.1 RDF

RDF (Resource Description Framework) is a foundation for processing information stored on the Web. It is the data model behind the Semantic Web vision whose goal is to enable integration and sharing of data across different applications and organizations. RDF describes a particular resource (e.g., a Website, a Web page, or a real world entity) using a set of RDF statements of the form <subject, property, object>. The subject is the resource, the property is the characteristic being described, and the object is the value for that characteristic: either a literal or another resource. For example, an RDF triple might look like: <http://www.example.org/index.html, http://www.example.org/ontology/dateCreated, "12/11/06">.

The naive way to store a set of RDF statements is in a relational database with a single table including columns for subject, property, and object. While simple, this schema quickly hits scalability limitations, as common queries such as those that want multiple property-object value pairs for a given subject require a self-join on subject. One common way to reduce the self-join problem [10, 17, 18] is to create separate tables for subjects that tend to have common properties defined. The rows in the table are subjects, columns are properties, and values are objects (i.e., a row in this table is a set of object values for some predefined properties of a particular subject). NULLs are used if a subject does not have a property defined. These tables are called property tables or subject-property matrix materialized join views. For example, for the raw RDF data (URI prefixes are not shown for simplicity):

Subject	Property	Object
Isaac	rdf:type	student
Isaac	Age	26
Rachel	rdf:type	post-doc
Isaac	Year	3rd
Rachel	Office	925
Rachel	Age	29

The following property table can be created:

Subject	rdf:type	Age	Year	Office
Isaac	student	26	3rd	NULL
Rachel	post-doc	29	NULL	925

Since Semantic Web data is often semi-structured, storing this data in a row-store can result in very sparse tables as more subjects or properties are added. Hence, this normalization technique is typically limited to resources that contain a similar set of properties. Thus, many small tables are

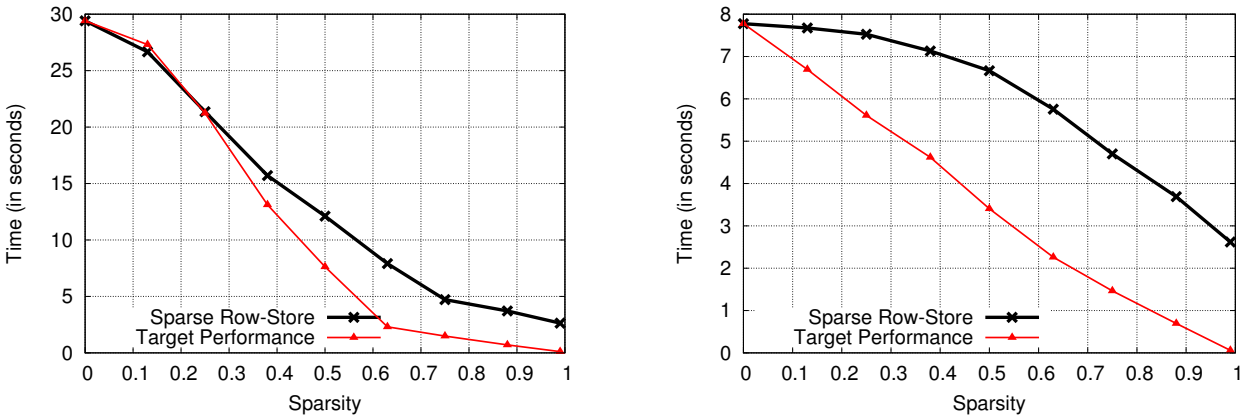

Figure 3: Total query time (a) and its CPU component (b) of running an aggregation query on a table with varying sparsity on a commercial row-store

usually created. The problem is that this may result in union and join clauses in queries since information about a particular subject may be located in many different property tables. This complicates the plan generator and query optimizer and can degrade performance. When stored in a column-store, these additional property tables are not needed, since, as we have discussed, there is no penalty to storing wide, sparse tables of this sort.

Another limitation of the property table approach is that RDF data often have multiple object values for the same subject-property pair (e.g., in the example above, if Rachel had two offices rather than just one). This means that property tables can have multi-valued attributes (e.g., office). The common implementation (such as in the Jena Semantic Web toolkit [17]) is to store multi-valued attributes separately from the other property tables in their own separate table with two columns: subject, and object value (e.g., if Rachel's two offices were 925 and 261, then <Rachel,925> and <Rachel,261> would be two rows in the office property table). For data with many multi-valued properties (which is common for Semantic Web data), this results in the creation of many of these two column tables which may have to be joined together at query time.

The ability of column-stores to store wide, sparse tables enables multi-valued attributes to be stored in the same row as single-valued attributes. They can either be stored in an array, or even can be stored in multiple columns. This further improves performance by reducing the number of joins that must be performed on the data.

Thus, if built on top of a column-store, property tables could be made arbitrarily wide with NULLs at undefined properties and multi-valued attributes for properties that can have multiple object values for the same subject. We have recently begun a research project where we are building a RDF store on top of a column-store. Our initial results, on a 50 million triple library data benchmark, are very promising, with queries that used to take thousands of seconds on a naive triple store, and hundreds of seconds on a schema that includes multiple property tables, being executed in less than a minute on a wide, sparse single property table.

3.2 XML

There has been a body of work [11, 15] that attempts to store XML data in standard relational DBMSs. Most techniques that store XML data in relational tables store XML elements in relations and XML attributes as attributes of these relations. Parent/child and sibling order information are also stored as attributes. Path expressions, which are prevalent in XML queries, require joins between element tables where the number of joins is proportional to the size of the path expression.

Although this can get fairly expensive, [11] shows that such an approach (they call it the *binary* approach), despite the many joins, beats an alternative scheme (they call it the *universal table* approach) which fully denormalizes all of the element tables into a single universal table (the universal table is so large from repeated data and NULLs, that the I/O cost of accessing it is prohibitive). [15] describes an inlining technique that reduces the number of joins required to evaluate path expression queries by including as many descendant elements of a particular element as possible in the same relation. The feasibility of having wide, sparse schemas in column-oriented design allows inlining to be used for elements with many descendants, and could even allow for a limited amount of recursive element expansion. Indeed, the universal table approach can be revisited (though repeated data is still an issue).

If no DTD is defined, the data is less structured, and the attributes and child elements for a particular element are less predictable. However, adding columns to a relation can be trivially done in column-stores even if the new column is almost entirely NULL. This allows new attributes and children to be added without impacting storage and queries on the other columns.

3.3 Databases With GEM-style schemas

Zaniolo argues in his work on GEM [19] that "the main limitation of the relational model is its semantic scantiness, that often prevents relational schemas from modeling completely and expressively the natural relationships and mutual constraints between entities." He thus proposes an extension of the relational model that includes generalization,

set-valued attributes, and sparse attributes in the same conceptual schema entity as the rest of the entity attributes.

The canonical example in database textbooks of generalization is the banking example. A bank account might have some attributes (i.e., customer name, branch name). Further, it might be a checking account or a savings account. Each account type has different attributes – a savings account might have an interest rate, while a checking account might have an annual fee. Thus three tables are formed: a bank account table, a checking account table, and a savings account table (the latter two tables have foreign keys into the bank account table).

The conventional wisdom is to add one table to a schema for each generalized type (e.g., one table for checking account and one table for savings account). However, using multiple tables for generalization can result in poor query performance (e.g., an additional join is now required to find interest rates on savings accounts for different branches since the savings account table must be joined with the bank account table). Further, GEM argues that it results in unnatural schemas (it might easier to reason about bank accounts as single entities as tuples in a single table). On the other hand, representing generalization in a single table increases table width and introduces NULL data (checking accounts will have NULL values in all savings account attributes and vice versa). If a column-store is used, the performance consequences of the additional width and sparsity are minimal.

The argument is similar for set-valued attributes and sparse attributes. The former makes the table wider, while the latter makes the table more sparse.

The GEM model is not widely used today. We speculate that the prevalence of row-store DBMSs, coupled with the lack of physical data independence in database systems (performance considerations of the row-store physical layer appear to influence schema design decisions) have held back this model. There is potential for a column-oriented implementation to rejuvenate the GEM debate.

4. CONCLUSION

While it is generally accepted that data warehouses and OLAP workloads are excellent applications for column-stores, this paper speculates that column-stores may well be suited for other applications in addition to these recognized uses. We observed that column-stores handle wide, sparse tables very well, and conjecture that because of these advantages, they may be good storage layers for Semantic Web, XML, and databases with GEM-style schemas.

5. ACKNOWLEDGMENTS

Thanks to Samuel Madden and Michael Stonebraker for their constructive advice, ideas, and feedback. Thanks also to Joey Hellerstein and David DeWitt for their feedback on earlier versions of this paper. This work was supported by the National Science Foundation under NSF Grant numbers IIS-048124, CNS-0520032, IIS-0325703 and by an NSF Graduate Research Fellowship.

6. REFERENCES

[1] Vertica. `http://www.vertica.com/`.

[2] C-Store code release under bsd license. `http://db.csail.mit.edu/projects/cstore/`, 2005.

[3] D. J. Abadi, S. R. Madden, and M. C. Ferreira. Integrating Compression and Execution in Column-Oriented Database Systems. In *SIGMOD*, pages 671–682, 2006.

[4] D. J. Abadi, D. S. Myers, D. J. DeWitt, and S. R. Madden. Materialization strategies in a column-oriented dbms. In *ICDE*, 2007.

[5] R. Agrawal, A. Somani, and Y. Xu. Storage and Querying of E-Commerce Data. In *VLDB*, 2001.

[6] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving relations for cache performance. In *VLDB*, pages 169–180, 2001.

[7] J. Beckmann, A. Halverson, R. Krishnamurthy, and J. Naughton. Extending RDBMSs to support sparse datasets using an interpreted attribute storage format. In *ICDE*, 2006.

[8] P. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. In *CIDR*, 2005.

[9] P. A. Boncz and M. L. Kersten. MIL primitives for querying a fragmented world. *VLDB Journal: Very Large Data Bases*, 8(2):101–119, 1999.

[10] E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *Proc. of VLDB*, pages 1216–1227, 2005.

[11] D. Florescu and D. Kossmann. Storing and querying XML data using an RDMBS. *IEEE Data Eng. Bull.*, 22(3):27–34, 1999.

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

[13] R. MacNicol and B. French. Sybase IQ multiplex - designed for analytics. In *VLDB, pp. 1227-1230*, 2004.

[14] R. Ramamurthy, D. Dewitt, and Q. Su. A case for fractured mirrors. In *VLDB*, pages 89 – 101, 2002.

[15] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: Limitations and opportunities. In *VLDB*, pages 302–314, 1999.

[16] M. Stonebraker, D. J. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. J. O'Neil, P. E. O'Neil, A. Rasin, N. Tran, and S. B. Zdonik. C-Store: A column-oriented DBMS. In *VLDB*, pages 553–564, 2005.

[17] K. Wilkinson. Jena property table implementation. In *SSWS*, 2006.

[18] K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *SWDB*, pages 131–150, 2003.

[19] C. Zaniolo. The database language GEM. In *Proc. of SIGMOD*, pages 207–218, 1983.