Dynamic Tables: An Architecture for Managing Evolving, Heterogeneous Data in Relational Database Management Systems

John Corwin[*], Perry Miller[†], Avi Silberschatz[*], Luis Marenco[†]

March 29, 2006

Abstract

Data sparsity and schema evolution issues affecting bioinformatics and medical informatics communities have forced the adoption of vertical or object-attribute-value based database schemas to overcome limitations posed when using conventional relational database technology. Through our collaboration with the Yale Center for Medial Informatics (YCMI), we explore the reasons for this and show why their data is difficult to model using conventional relational techniques. We propose a solution to these obstacles based on a relational database engine using a *sparse, column-store* architecture. We provide benchmarks comparing the performance of queries and schema-modification operations using three different strategies: (1) the standard conventional relational design, (2) past approaches used by clinical and neuroinformatics researchers, and (3) our sparse, column-store architecture. Our performance results show that our architecture is a promising technique for storing and processing many types of data that were not handled well by conventional nor other derived semantic data models.

1 Introduction

The Yale Center for Medical Informatics (YCMI) is focused on the use of computing to benefit the fields of clinical medicine, neuroscience, and molecular biology. As part of these efforts, they have created databases that store and integrate diverse types of heterogeneous data, including models and properties of neurons and subcellular components, clinical patient data, genomic sequence, gene expression, and protein expression data, and the results of many different neuroscience experiments. These databases have properties that make them difficult to implement and maintain using conventional relational databases that rely on horizontal data storage, including the use of sparse and heterogeneous data, the high frequency of schema changes, and the extensive use of metadata.

The databases in question are TrialDB[10], which stores clinical patient data, and SenseLab[15], which stores an extensive set of information related to neurons and neuronal properties. TrialDB and SenseLab are implemented using the semantic data models known as EAV[11] and EAV/CR[13], respectively, which make use of vertical storage to store data values, plus an extensive set of metadata to allow generalized tools to be developed to query and maintain the data. Unfortunately, these tools must also re-implement features commonly taken for granted in a database engine, including typical methods of querying the logical schema of the data. Furthermore, the use of vertical storage can cause performance problems, particularly with attribute-centered queries[12].

[1]Department of Computer Science, Yale University, New Haven, CT
[2]Yale Center for Medical Informatics, Yale University, New Haven, CT

To alleviate some of the limitations imposed by vertical schemas, we propose a database system based on the use of sparse, column-based storage, which we call *dynamic tables.* The use of a column store, also known as decomposed storage, was first proposed by Copeland and Khoshafian[3][4], and is also used by the Sybase IQ[8] database engine. Our use of decomposed storage is to replace vertical schema at the level of the database engine itself, enabling the key advantages of vertical storage while maintaining the clarity and maintainability of querying a horizontal schema. Our storage implementation presents a standard horizontal view of the data to the database user, and was designed as an addition to – not a replacement of – the database engine's storage architecture, so dynamic tables can be flexibly and transparently mixed with standard horizontally-stored tables in queries.

The remainder of the paper is structured as follows. Section 2 describes the databases we're working with, the data model they're represented with, the tools developed to query them, and motivation for our architecture. In section 3, we present our architecture, describe it's implementation, and discuss the optimizations we employ to improve query performance. Section 4 contains performance results comparing the performance of queries and schema modifications using horizontal schema, vertical schema, and our decomposed schema. Section 5 talks about other implementation strategies we tried, section 6 discusses related work, and we conclude in section 7.

2 Background

The SenseLab project contains neuroinformatics databases created to support both theoretical and experimental research on neuronal models, membrane properties, and nerve cells using the olfactory pathway as a model. SenseLab is part of the national Human Brain Project which seeks to improve our understanding of brain function.

TrialDB is a clinical study data management system (CSDMS) created at Yale and that contains information on clinical trials from several organizations. TrialDB relies on the Entity-Attribute-Value (EAV) semantic data model, a type of vertical schema in which attributes are divided into tables based on type – there is a vertical table for string-valued attributes, another for floating-point-valued attributes, and so on. Attributes are stored as integers for space-efficiency, and some EAV tables have a fourth column with a time-stamp value to support versioning. Metadata tables are used to catalog which entities possess which attributes, as well as to provide a mapping between attribute numbers and names. Trial/DB is stored using the EAV model.

In clinical trials, like in electronic medical records systems, a given patient-event requires only a few from a possible myriad of attributes. These attributes also trend to change as new medical procedures are introduced and others retired. Storing patient-event data using straight relational tables and columns for attributes produce sparse tables and increased schema change overhead.

Neuroscience is a discipline highly characterized by constant evolution: Hypothesis can change in the course of an investigation and knowledge bases are difficult to maintain due to unstable conceptualization of the domain. Due to these issues, experimental databases are expensive to maintain or become obsolete. But neuroscience data is far more complex that patient data, requiring more than one type of entity, and relationships between them. For this reason, EAV/CR was created by extending the EAV model with *classes* and *relationships.* In this context, classes mean that the values stored in the EAV table are no longer required to be simple values but instead can be complex objects. The addition of relationships allow data values to reference other entities in the database, representing relationships amongst data items. SenseLab is stored using the EAV/CR model.

Horizontal query:

```
SELECT o_id, o_name, chromosome, nucleotides
FROM ordb_chemosensory_receptors
WHERE length=1363 and cr_type=1443
```

Vertical query:

```
SELECT OS.object_id, OS.object_name, V32.value, V40.value
FROM (
       SELECT senselab_objects.object_id, senselab_objects.object_name
       FROM (senselab_objects INNER JOIN senselab_eav_objects ON
                    senselab_objects.object_id = senselab_eav_objects.object_id)
       INNER JOIN senselab_eav_objects AS senselab_eav_objects_1 ON
                    senselab_objects.object_id = senselab_eav_objects_1.object_id
       WHERE senselab_objects.object_class =22
                    AND senselab_eav_objects.attribute_id = 39
                    AND senselab_eav_objects.value = 1363
                    AND senselab_eav_objects_1.attribute_id=80
                    AND senselab_eav_objects_1.value=1443)
       AS OS

LEFT JOIN (SELECT object_id, value
           FROM senselab_eav_int
           WHERE attribute_id=32) AS V32 ON
           OS.object_id = V32.object_id
LEFT JOIN (SELECT object_id, value
           FROM senselab_eav_memo
           WHERE attribute_id=40) AS V40 ON
           OS.object_id = V40.object_id
```

Figure 1: A typical query in the SenseLab database expressed using horizontal and vertical data representations.

In addition to the data models described above, YCMI has created a set of tools to query and manage data stored in the EAV and EAV/CR models. But their use of these semantic data models has come with a cost: a) The limited number of tables increases the number of tuples translating in longer query execution time, particularly for multi-attribute queries; b) Vertical systems have the potential of reducing the capacity of data stored on each of the attributes tables to the maximum number of rows in a single table; and c) Ad hoc queries' creation becomes more complex, as they have to be devised using a virtual schema but translated into the physical vertical schema.

The inability to execute standard ad-hoc relational queries against an EAV or EAV/CR database's logical schema is particularly troublesome – queries that are relatively simple to express in a horizontal schema become significantly more complex in a vertical schema. Consider the example in figure 1. Part of the complexity comes from the fact that when querying multiple attributes in a vertical schema, we need to join the vertical table with itself once for each desired attribute. Another loss of clarity comes from EAV's use of integers to identify attributes (if strings were used instead, the string value of each attribute would have to be repeated for each value in the vertical table, adding an unacceptable amount of storage overhead).

Even if the database engine contains extensions such as `PIVOT` and `UNPIVOT` operators to simplify the translation between vertical and horizontal schema, if the operators are implemented at a high enough level such that the translated queries are processed by the query planner, the resulting queries will involve a large number of joins. Most modern database engines perform poorly with queries that contain large numbers of project-join operations[5].

In order to overcome the problems imposed by the vertical storage approach, we have devised our dynamic table implementation discussed next.

3 Dynamic table implementation and optimizations

The design goals of our system are to support the simplicity of querying a horizontal schema and the flexible and efficient schema manipulations of a vertical schema, all while maintaining reasonable query performance. To achieve these goals, we have implemented a sparse, column-based storage architecture within the PostgreSQL [6] database engine.

We first present the system from the database user's point of view – how to create and manipulate dynamic tables. We then show how dynamic tables are implemented within the database engine, followed by several optimizations we have employed to improve query performance.

3.1 Interface

A table's storage format is specified at creation time by the presence of the `DYNAMIC` flag in the `CREATE TABLE` statement in SQL:

```
CREATE DYNAMIC TABLE table_name(
       column_1 type_1,
       column_2 type_2,
       ...
       column_n type_n
)
```

The `DYNAMIC` keyword indicates that the table is to be created using the decomposition storage model. From the database user's point of view, after the table has been created, operations on the the table are semantically indistinguishable from those on a standard, horizontally represented table – the user queries and updates the table as if it were stored horizontally. Dynamic tables also fully support indexes, primary and foreign-key constraints, and column constraints.

The addition of dynamic tables to the database engine does not affect the availability of regular, horizontally-stored tables; the user is free to choose the storage format of each individual table based on the sparsity of the data to be stored and the expected frequency of modifications to the table's schema. Dynamic tables can also be freely mixed with standard tables within queries.

3.2 Implementation

Let r be an $n+1$-ary relation consisting of n attributes plus an object identifier. To implement the decomposed storage of r, we create n two-column "attribute" tables, and a single one-column "object" table. Each two-column table stores pairs of object identifiers and attribute values. Null

Standard table (horizontal storage)

oid	A1	A2	A3
1	a	b	c
2	d	e	
3		f	
4	g		

Vertical storage

Object	Attribute	Value
1	A1	a
1	A2	b
1	A3	c
2	A1	d
2	A2	e
3	A2	f
4	A1	g

Dynamic table (decomposed storage)

oid
1
2
3
4

oid	A1
1	a
2	d
4	g

oid	A2
1	b
2	e
3	f

oid	A3
1	c

Figure 2: Horizontal, vertical, and decomposed storage models

Update $a \rightarrow a'$

	a is NULL	a is non-NULL
a' is NULL	No action	Delete a
a' is non-NULL	Insert a'	Update a to a'

Figure 3: Action performed when updating attribute a to a' on a dynamic table.

values of an attribute are not explicitly stored; their presence is inferred by the absence of an object-attribute pair in the attribute table. The one-column table stores a list of all object identifiers present in the relation. Finally, the correspondence between r, r's object table, and r's attribute tables are stored in a system catalog. Figure 2 shows the layout of a standard table compared to a dynamic table.

The object table was not present in Copeland and Khoshafian's original decomposed storage model. The addition of the object table allows us to quickly determine if a row is present in a relation, avoiding a potential scan of each of the attribute tables. It also allows us to maintain a single virtual location of a tuple set to the physical location of it's object identifier in the object table, and enables us to use a left outer join instead of a full outer join when computing query results. Additionally, the object table was also devised to allow future versions of the dynamic table infrastructure to provide row level security and versioning, which are present in the EAV/CR data model.

We implemented dynamic tables at the heap-access layer of PostgreSQL, which implements operations on individual tuples in a relation, namely inserting, updating, deleting, and retrieving a tuple. When the database user creates a new dynamic table, we create the attribute and object tables in a hidden system namespace, exposing only the virtual, horizontal schema given as the table's definition in the `CREATE TABLE` statement.

To retrieve a tuple for a query on r, we first scan the object table to find a qualifying object identifier, o. If found, we use o to query each of r's attribute tables. If the pair (o, a) is present in an attribute table for some value a, we set the corresponding attribute in the returned tuple equal to a. Otherwise, the tuple's attribute value is set to NULL. To facilitate the efficient lookup of object IDs in the attribute tables, we maintain a B-tree index on object IDs for each table.

Likewise, inserting a new tuple into r is implemented by inserting a new identifier into r's object table, then for each of the new tuple's attributes, if the new attribute value is non-NULL, we insert the new object ID and attribute value pair into r's corresponding attribute table. For deletes, given an object ID, we simply delete each tuple from r's object and attribute tables. Updates are slightly more complex in that the operation to perform is dependent on both the old and new values of each attribute. The semantics for updating an attribute a to a' in r are listed in figure 3.

Given the evolving nature of the data we intend to store, it is important that schema modifications on dynamic tables are implemented efficiently and that we avoid the cost of copying an entire table whenever possible. We support the schema operations of adding and removing columns, and changing a column's type, in addition to trivial schema operations such as renaming a column or renaming the entire table. We implement the non-trivial operations as follows:

- `ADD COLUMN` *col type*: To add a new column *col* to r, we create a new, empty attribute table for *col* and update the system catalog to associate it with r.
- `DROP COLUMN` *col*: To drop a column from r, we find and drop the attribute table associated

Query on schema $r(a, b, c, d, e)$:

```
SELECT b
FROM r
WHERE d > 10
```

Figure 4: A good opportunity for projection pushing

Query on schema $r(a, b, c, d, e)$:

```
SELECT *
FROM r
WHERE e = 5 AND d > 15 AND d < 25
```

Figure 5: A candidate for conditional select optimization

with the column and remove r's association with the dropped attribute table from the system catalog.

- `ALTER COLUMN` *col* `TYPE` *type*: Changing a column's type requires us to copy and convert the data in the column's associated attribute table to the new type. However, this is still significantly cheaper than copying and converting an entire horizontal schema, particularly when the schema has a large number of attributes.

3.3 Optimizations

We employ two main optimizations to improve the performance of queries on dynamic tables: projection pushing and optimizing conditional selects. The use of decomposed storage gives us additional opportunities for projection pushing in comparison to a standard horizontal schema. Consider the query listed in figure 4. If r is stored using decomposed storage, there is no reason to retrieve values from r's attribute tables for a, c, or e. For the general case, we compute the set of attributes we're required to fetch by taking the union of the attributes present in the target list of the `SELECT` clause with the attributes present in the `WHERE` clause of a query.

Next, consider the query listed in figure 5. We can't use projection pushing here because all r's attributes are requested via "`SELECT *`". However, when scanning r to find tuples that match the query condition, it would make sense to retrieve and test the values of attributes d and e against the condition, discarding the candidate tuple if the test fails before retrieving r's remaining attributes. To implement this optimization in the general case, we divide the requested attributes from a query into two groups: those with conditions and those without conditions. When retrieving attributes, we first retrieve attributes with conditions, one at a time. If the attribute does not satisfy the condition, we immediately discard the tuple, skipping the remaining attributes. If all of the conditions are satisfied, we read the remaining attributes and return the tuple to the query executor.

In both of these cases, these attribute-level I/O operations are not beneficial when using a standard horizontal schema because tuples are generally stored contiguously and packed into blocks, and sub-block I/O does not improve tuple throughput. However, with decomposed storage, attributes of each column are grouped together into blocks, giving us the opportunity to skip reading blocks for attributes we have determined that we don't need. Of course, the other side of the story is that if we do end up needing all the attributes of a tuple, a horizontal schema that allows us to fetch the entire tuple at once will be more efficient, so the optimal choice of storage format is dependent on both the sparsity of the data being stored and the types of queries that will be run against the data.

4 Performance results

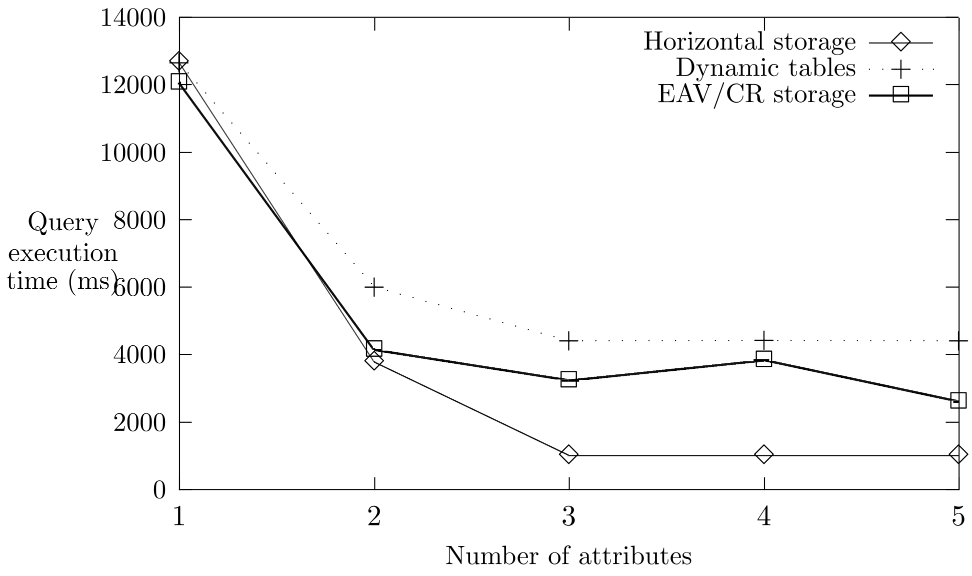

Figure 6: Comparison of query execution time for an attribute-centered query on SenseLab data without indexes.

To evaluate the performance of our implementation, we compared the performance of dynamic tables to both standard, horizontal storage and to storage using the EAV and EAV/CR vertical models. We test this using a typical series of queries executed by users of the SenseLab databases that retrieve a set of neurons plus several of their attributes, filtering on between one and five attributes to narrow the number of results. These queries have a similar structure to the queries shown in figure 1.

After each database was populated, the `ANALYZE` command was run to update the statistics used by the query planner. All tests were run on a 1.8Ghz Intel Pentium IV machine with 1GB of RAM running Fedora Core Linux version 4.01.

Figure 6 shows the results of running each query using each of the three storage models. No indexes were used on the data items, representing the case where we do not know beforehand which attributes will be of interest to the database users. The number of attributes in this case refers to the number of values that are filtered on in the `WHERE` clause of the query.

When we do know which attributes will commonly be queried on, we can create indexes on them to improve performance – this is the usual case for SenseLab data. Figure 7 shows the running time for each of the queries with indexes. Notice that the use of indexes virtually eliminates

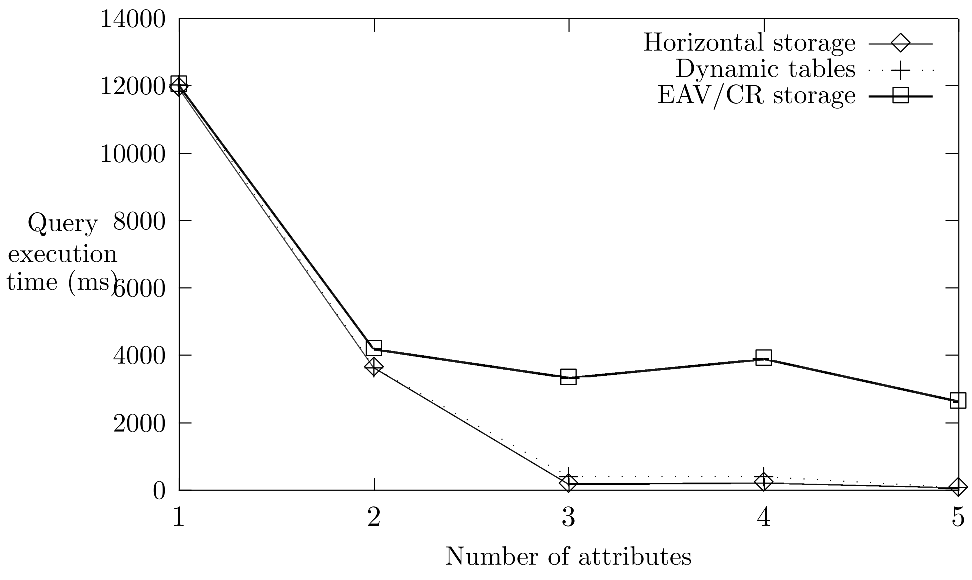

Figure 7: Comparison of query execution time for an attribute-centered query on SenseLab data with indexes on queried attributes.

the performance difference between horizontal storage and dynamic tables, whereas the vertical approach does not benefit as much from using indexes.

Unlike many other relational database engines, PostgreSQL already has support for the efficient implementation of some schema modification commands. In particular, most cases of adding or removing a column from a table are executed lazily – the storage format of the table is not actually modified until the table is later compacted using the `VACUUM` command. Thus, in our implementation, adding a column to a table is a trivial operation using any of the three storage formats, as they only involve updating the system catalog (the metadata catalog in the case of EAV/CR), and are independent of the data currently stored in the table.

However, some schema modification operations do require the data in a table to be modified, including altering a column's type and adding a new column with a default value. To implement altering a column's type using horizontal schema, the entire table is copied, converting the data values for the modified column to the new type as they are copied. For dynamic tables, we only have to convert the single attribute table corresponding to the modified column. For tables stored using EAV or EAV/CR, we need to move the data values from the object-attribute-value table for the old type to the object-attribute-value table for the new type. Figure 9 shows the SQL statements used to change the data type of a column representing the year of publication of a paper from a string value to an integer, and figure 10 shows the execution time for these queries. Dynamic tables are significantly faster than the other approaches for this operation.

We also ran a set of queries on the TrialDB data to compare the performance of each storage model on a large, sparse dataset. Since the TrialDB data contains millions of records, querying the dataset without the use of indexes is too slow to be useful using any of the storage methods. Figure 8 shows the results of running a set of typical attribute-centered queries with indexes on each of the attributes in question. The results are similar to the queries on SenseLab.

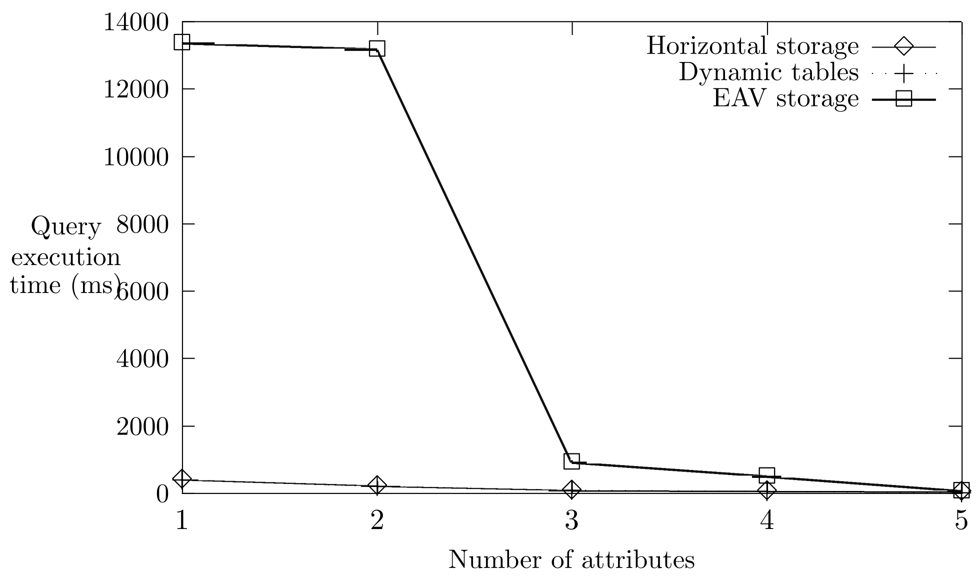

Figure 8: Comparison of query execution time for an attribute-centered query on TrialDB data with indexes on queried attributes.

Sample clinical data from TrialDB used in this project was properly deidentified following HIPAA[16] regulations by removing all patient demographics data, scrambling integer and string data values, and replacing all internal system id's (e.g.: patient event ids) with local sequential ones.

5 Other implementation strategies

The implementation strategy given above is actually our third implementation of an alternate relational storage architecture. Our first two approaches did not work out, but we believe the ideas may have merit if implemented in other database systems.

Our first approach was to use a dense column-store. In this architecture, a relation's attribute values are densely packed into one-column tables. Tuples from the conceptual schema are returned by joining values from the attribute tables based on their index in the table: to retrieve the ith tuple from relation r, we simply take the ith entry from the first attribute table, the ith entry from the second attribute table, and so on. This approach still allows for the efficient schema modifications described above, and provides greater storage efficiency for dense data. However, we realized that this approach is incompatible with PostgreSQL's use of multi-version concurrency control (MVCC). Using MVCC, PostgreSQL eliminates the need to hold locks on tables by storing multiple versions of tuples representing consistent snapshots of the data at different points in time, along with tuples that have been made obsolete by newer versions. When implementing the single-column store approach, even if we are careful to ensure that values from a tuple are inserted into the same logical position in each of the attribute tables, it is very difficult to ensure that values will remain in the correct logical order after a table compaction (VACUUM) operation.

Our second approach was to use sparse columns to store attributes, but to implement the

Horizontal and dynamic tables:

```
ALTER TABLE modeldb_model_paper_42
      ALTER COLUMN year TYPE int
      USING cast(year as int)
```

EAV/CR data model:

```
INSERT INTO senselab_eav_int
SELECT object_id, attribute_id,
       cast(value as int)
FROM senselab_eav_string
WHERE senselab_eav_string.attribute_id = 154

UPDATE senselab_attributes
       SET datatype = 'I'
       WHERE attribute_id = 154

DELETE FROM senselab_eav_string
WHERE senselab_eav_string.attribute_id = 154
```

Figure 9: SQL queries to alter the type of a column representing the year of publication of a paper from a string value to an integer value.

	Horizontal	Dynamic	Vertical
Execution time (ms)	501	297	4650

Figure 10: Execution time for the alter column type queries given in figure 9 for each of the three data storage formats.

translation from the decomposed storage model to the horizontal storage model at the level of rewrite rules in the database engine. Our hypothesis was that rewrite rules would give the query planner maximal opportunity to optimize query plans involving dynamic tables. We found, however, that PostgreSQL's query optimizer tried too hard to find the optimal join order when combining attribute tables, taking exponential time relative to the number of joins to plan the query. The generated query plans were efficient, but for tables with large numbers of columns, query planning took several orders of magnitude longer than actually executing the query. Constraining the join order decreased planning time, at the expense of potentially worse query plans. In the end, we found it was more efficient to implement decomposed storage at the heap-access level of the database, bypassing the query planner entirely.

6 Related work

Beckmann et al.[1] propose an alternate storage format for tables on disk in which the tuple layout of each row is described by a variable-length, "interpreted" record. The interpreted records are implemented at a lower level in the database engine compared to dynamic tables (storage level vs. heap-access level), and provide many of the same benefits for storing sparse or heterogeneous data with potentially greater space-efficiency. It is not clear, however, that interpreted records provide any benefit to the optimization of schema-modifying operations.

The C-Store[2] database engine is also based on a column-store architecture, but is designed for the application of read optimized databases, and thus may not be appropriate for typical neuroscience databases where writes are frequent.

The work of Agrawal, et al.[9] on e-commerce data shows that the domain of e-commerce data shares many similarities with neuroscience and clinical informatics data, particularly with respect to schema evolution and heterogeneous data. They, however, advocate the use of vertical storage and show a performance advantage for vertical schema compared to horizontal schema for this domain.

7 Conclusion and future work

This paper presents the architecture of *dynamic tables*, which are based on implementing relational database storage using the decomposed storage model while maintaining a logical horizontal view of the data. Our implementation has significant advantages over past approaches used by the bioinformatics and medical informatics communities that were based on the use of vertical storage, including improved query performance on attribute-centered queries, improved schema modification performance, and the greater manageability of working with a horizontal view of the data. Dynamic tables also maintain the capability of vertical schemas to efficiently store sparse data.

Our future work will continue our effort increasing the capabilities of database engines by adding features commonly needed by bioscience and clinical databases, including extensive support for metadata and a flexible system for row-level security.

8 References

References

[1] J. L. Beckmann, A. Halverson, R. Krishnamurthy, and J. F. Naughton. Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format. *In ICDE, 2006.*

[2] M. Stonebraker, D. Abadi, A. Batkin, X. Chen, M. Cherniack, M. Ferreira, E. Lau, A. Lin, S. Madden, E. O'Neil, P. O'Neil, A. Rasin, N. Tran and S. Zdonik. C-Store: A Column Oriented DBMS. In *Proceedings of VLDB, August, 2005, Trondheim, Norway.*

[3] 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.

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

[5] Benjamin J. McMahan and Guoqiang Pan and Patrick Porter and Moshe Y. Vardi. Projection Pushing Revisited. In *EDBT 2004*.

[6] PostgreSQL. PostgreSQL Global Development Group. `http://www.postgresql.org/`

[7] Jacob Anhoj. Generic Design of Web-Based Clinical Databases. *Journal of Medical Internet Research*, Volume 5, Issue 4, Article e27.

[8] Sybase IQ: Query Search Test Software And Operational Data Store Warehouse Application. Sybase Inc. `http://www.sybase.com/products/ informationmanagement/sybaseiq`

[9] R. Agrawal, A. Somani, Y. Xu. Storage and Querying of E-Commerce Data. In *VLDB*, pages 149 - 158, 2001.

[10] P. Nadkarni, C. Brandt, S. Frawley, F. Sayward, R. Einbinder, D. Zelterman, et al. Managing attribute-value clinical trials data using the ACT/DB client-server database system. *Journal of the American Medical Informatics Association*, 1998; 5(2):139-151.

[11] P. Nadkarni, C. Brandt. Data Extraction and Ad Hoc Query of an Entity-Attribute-Value Database. *Journal of the American Medical Informatics Association*, 1998;5:511-527.

[12] R. Chen, P. Nadkarni, L. Marenco, F. Levin, J. Erdos, P. Miller. Exploring Performance Issues for a Clinical Database Organized Using an Entity-Attribute-Value Representation. *Journal of the American Medical Informatics Association*, 2000;7:472-487.

[13] L. Marenco, P. Nadkarni, E. Skoufos, G. Shepherd, P. Miller. Neuronal database integration: the Senselab EAV data model. In *Proceedings of the AMIA Symposium; 1999.* 1999. p. 102-106.

[14] C. Friedman, G. Hripcsak, S. Johnson, J. Cimino, P. Clayton. A Generalized Relational Schema for an Integrated Clinical Patient Database. In *Proc. 14th Symposium on Computer Applications in Medical Care;* 1990; Washington, D.C.: IEEE Computer Press, Los Alamitos, CA; 1990. p. 335-339.

[15] P. Miller, P. Nadkarni, M. Singer, L. Marenco, M. Hines, G. Shepherd. Integration of Multidisciplinary Sensory Data: A Pilot Model of the Human Brain Project Approach. *Journal of the American Medical Informatics Association*, 2001;8:34-48.

[16] (HIPAA) US Office for Civil Rights. Medical Privacy: National standards to protect the privacy of personal health information. 2002 [cited 2006 Mar 15, 2006]; Available from: `http://www.hhs.gov/ocr/hipaa/`.