On the Performance of an Array-Based ADT for OLAP Workloads

Yihong Zhao Kristin Tufte Jeffrey F. Naughton
Computer Sciences Department
University of Wisconsin-Madison

August 29, 1996

Abstract

There is currently a debate among OLAP vendors on the best way to provide OLAP functionality: Relational OLAP (ROLAP) vendors advocate using sophisticated front ends to provide a multidimensional view of a standard relational database, whereas Multidimensional OLAP (MOLAP) vendors provide custom servers that generally store their data as arrays (instead of tables.) An important question in this debate is the relative performance of arrays vs. tables for common OLAP operations. To shed some light on this question, we have implemented a MOLAP Abstract Data Type (ADT), which uses a multidimensional array as its principle storage mechanism, within the Paradise object-relational DBMS. Using this implementation, we have studied MOLAP and ROLAP performance on the same DBMS platform (by using either the MOLAP ADT or standard relational tables.) In particular, we have compared a new MOLAP consolidation algorithm with three ROLAP consolidation algorithms, and found that for a wide range of data densities, our MOLAP implementation significantly outperforms ROLAP both in terms of disk storage and query execution time. This suggests that the ADT mechanism of object-relational database systems is useful for constructing hybrid solutions that provide the benefits of both ROLAP and MOLAP systems.

1 Introduction

On-Line Analytical Processing (OLAP) has recently emerged as an important database technology. Currently, there are two dominant OLAP architectures: Relational OLAP (ROLAP) and Multidimensional OLAP (MOLAP). ROLAP vendors store data in tables in a traditional RDBMS and provide a sophisticated front end to support the complex functions needed by OLAP users. MOLAP vendors, on the other hand, provide custom OLAP servers which typically store the data as multidimensional arrays or custom storage structures specifically designed for efficient storage and retrieval of sparse multidimensional arrays.

There is an ongoing debate over which architecture more effectively supports OLAP functionality. ROLAP vendors find several faults with the MOLAP architecture. They argue that MOLAP systems do not effectively support ad-hoc querying, do not scale up to large data sizes, and do not conform to an existing open architecture (as RDBMSs do) [SAT95]. On the other side, MOLAP vendors criticize ROLAP systems for taking an inherently multidimensional data set and flattening it into two-dimensional relational tables. Two of the problems they claim with this representation are: 1) relational tables are not as natural or efficient as the MOLAP storage structures for storing multidimensional data; and 2) the complex functionality required by OLAP is mismatched with the SQL interface provided by RDBMSs, sometimes forcing complex OLAP functions to be executed in a specialized front end outside the database server [Fin]. While many of these complaints are found in marketing white papers rather than in the research literature, there are clearly fundamental technical differences between ROLAP and MOLAP, and each have their benefits and drawbacks.

Which architecture is better suited to meet the needs of OLAP consumers? One important piece of this puzzle is the performance of ROLAP tables versus MOLAP arrays on common OLAP operations, such as consolidation. To address this issue, we have designed a MOLAP Abstract Data Type (ADT) and implemented it within the Paradise object-relational database system [DKLPY94]. Our goal in implementing this ADT was two-fold: first, to compare the performance of array-based vs. table-based solutions on a common platform; second, to see how well object-relational technology supports the construction of a special ADT that tries to combine the benefits of MOLAP and ROLAP in a single system. The MOLAP ADT consists of a multidimensional array and a set of B-tree indices, one for each array dimension. The MOLAP ADT serves as both a storage and indexing structure. In addition, we have defined functions on the MOLAP ADT to execute complex OLAP analytical functions such as consolidation and aggregation.

Consolidation is one of the most important and frequently used OLAP functions; therefore, we selected it to test the performance of the MOLAP ADT. We implemented three ROLAP consolidation algorithms in Paradise, and compared the performance of these three algorithms with a new consolidation algorithm, designed specifically for the MOLAP ADT. A key performance determinant turns out to be the density of the data. In array terminology, the density of a data set is the percentage of array entries that contain data. Our tests show that if we use storage compression on the MOLAP array, the MOLAP consolidation algorithm outperforms the fastest ROLAP algorithms for data densities down to about one percent.

Our experiments demonstrate that this performance enhancement comes from three main sources:

1. Improved storage efficiency in the MOLAP implementation.

2. Improved data access efficiency in the MOLAP implementation due to replacing value-based lookups by simple array indexing.

3. Lowered overhead for individual data attribute referencing because the MOLAP array is stored in the database natively in main-memory format, whereas the tuples in the ROLAP implementation are stored in a standard (and very flexible) slotted-page format on disk.

The superior performance of the MOLAP ADT on an interesting range of data densities suggest that the ability to define an array-based ADT gives object-relational systems the potential to provide an OLAP solution that combines the advantages of ROLAP and MOLAP.

The rest of the paper is structured as follows: Section 2 provides some background on OLAP in order to make the paper self-contained. Section 3 describes the design and implementation of the MOLAP ADT, Section 4 describes the new MOLAP consolidation algorithm and the three ROLAP consolidation algorithms with which it is compared, and Section 5 describes the performance studies. Finally, Section 6 concludes and discusses future work.

2 An Example OLAP Schema

Perhaps the quickest way to understand the basic characteristics of OLAP workloads is through an example. In this section we present a greatly simplified example; readers already familiar with multidimensional data models, ROLAP, and MOLAP may wish to skip to Section 3.

The example OLAP database consists of sales data for a chain of retail stores. The database contains sales volumes for products sold at multiple stores over a ten year period. This data is best visualized as a three dimensional array with the dimensions store, product, and time. Each cell of the array contains the sales volume (in dollars) for a particular product at a particular store on a particular day. In general, some cells in the array may not contain valid data. For example, some stores may not sell color laser printers, therefore, all the array elements representing sales of color laser printers at those stores are not valid. The array is considered *dense* if most of the array data cells have valid data, and *sparse* otherwise.

Typically there are hierarchies defined within the dimensions of OLAP data models. For example, the time dimension consists of years, which contain quarters, which contain months, which contain days. This is better illustrated in the ROLAP schema below.

2.1 ROLAP Schema

The standard way of representing such a data set in the relational model is to use a Star Schema (or its slightly more complex variant, the Snowflake.) A star schema consists of a set of tables, including one table for the data itself (e.g., sales quantities in our example) and one table each for the dimensions. The ROLAP star schema for the example database is:

- **Sales** (*pid*, *sid*, *tid*, volume);
- **Product** (*pid*, pname, type, category);
- **Store** (*sid*, sname, city, state, region);
- **Time** (*tid*, day, month, quarter, year);

The Sales table contains the actual sales volume data and is often called the "fact" table. It has one column for the sales volume data, `sale`, and one column for each of the dimensions, `pid`, `sid`, and `tid`. In the interests of space, the actual dimension values (e.g pname, type, and category) are not stored in the Sales table, but are stored in a separate "dimension tables" and are related to values in the Sales table with foreign keys. Notice that each dimension does form a hierarchy. The dimension tables maintain information about the hierarchy (e.g. maintaining the fact that January is in the first quarter of the year) in addition to maintaining the mapping between a sales value and a particular store, product, and date. As Figure 1 shows, the term “star” in “star schema” arises here because Sales can be viewed as the center of a star, with each of the dimension tables representing a point of the star.

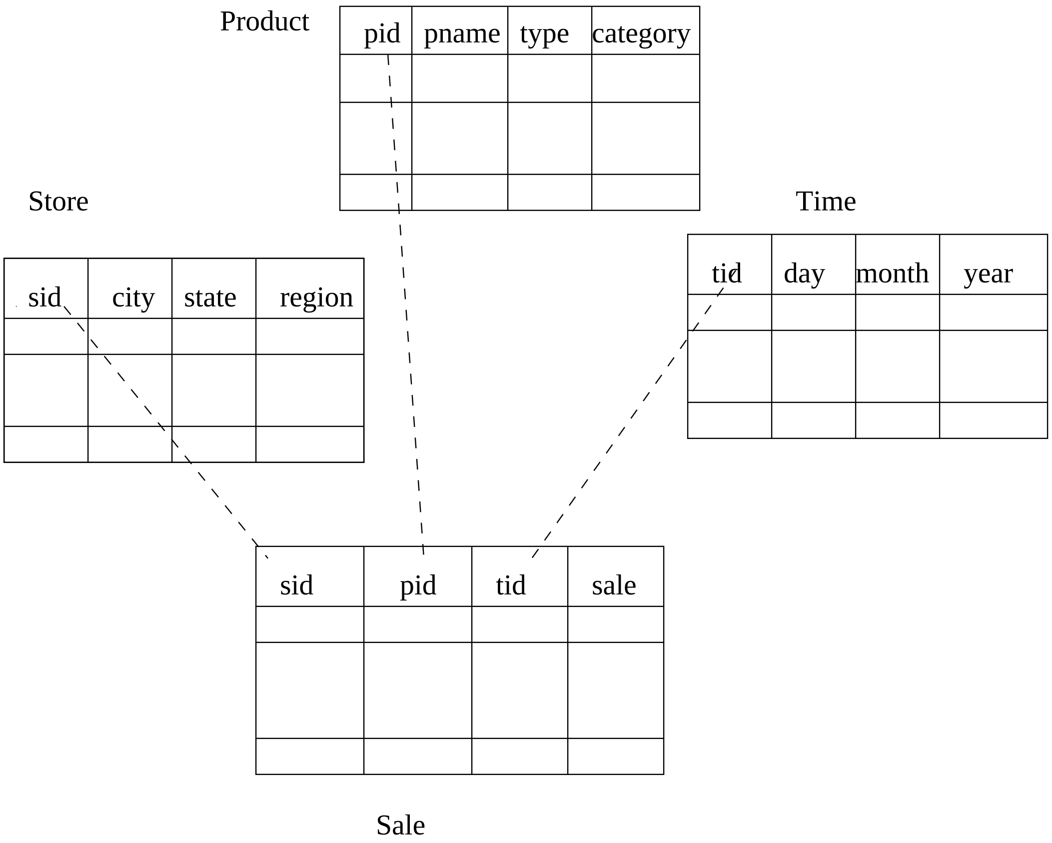

Figure 1: ROLAP Star Schema

2.2 MOLAP Schema

From the MOLAP perspective, the example data is a 3-dimensional array instead of a table, as shown in Figure 2. MOLAP systems store the data either directly in a multidimensional array or in a special data structure designed specifically for storage of sparse multidimensional arrays.

2.3 Operations

There is a reasonably standard set of OLAP operations. This section describes consolidation, one of the most important of these operations. Consolidation is a basic operation that can be used to describe the various “roll-ups” possible over the data set. The reader can find the descriptions of other OLAP operations such as drill down, selection, and slicing and dicing in a number of vendor white papers, see for example [Kenan].

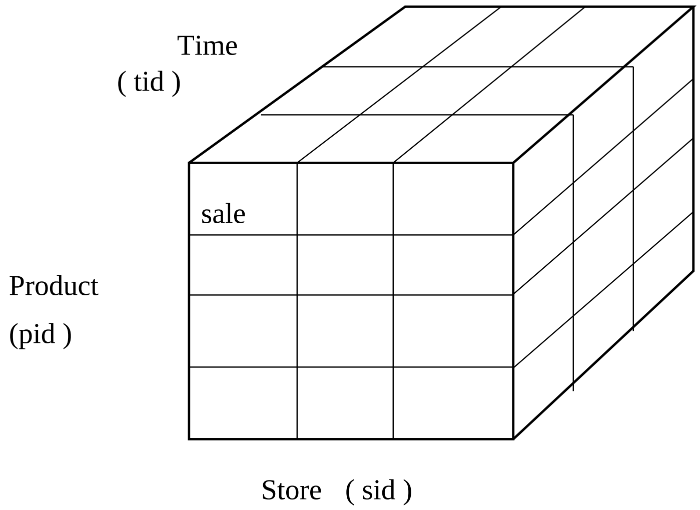

Figure 2: MOLAP N-D Array

Consolidation involves the aggregation of data over one or more dimension hierarchies or formulaic relationships within a dimension. While they are often as simple as summations, consolidations can involve complicated mathematical and statistical functions such as expected value and correlation. A typical consolidation query expressed in SQL is shown below.

```
select city, type, sum(volume)
from   Sales, Product, Store
where  Sales.pid = Product.pid and
       Sales.sid = Store.sid
group by Store.city, Product.type
```

3 A MOLAP ADT

In this section, we describe the design and implementation of a MOLAP ADT which integrates a multidimensional OLAP storage structure into the Paradise database system. As we have stated previously, our goal in implementing this ADT was two-fold: first, to compare the performance of array-based vs. table-based solutions on a common platform; second, to see how well object-relational technology allows the construction of a special ADT that tries to combine the benefits of MOLAP and ROLAP in a single system.

Paradise has an object-relational data model and query language which supports relational tables, Abstract Data Types, and functions on Abstract Data Types. This gave us the framework we needed to implement the MOLAP ADT and compare MOLAP and ROLAP performance on the same platform. The MOLAP ADT uses a multidimensional array as its primary storage structure and has functions defined on it to perform common OLAP operations such as consolidation and aggregation.

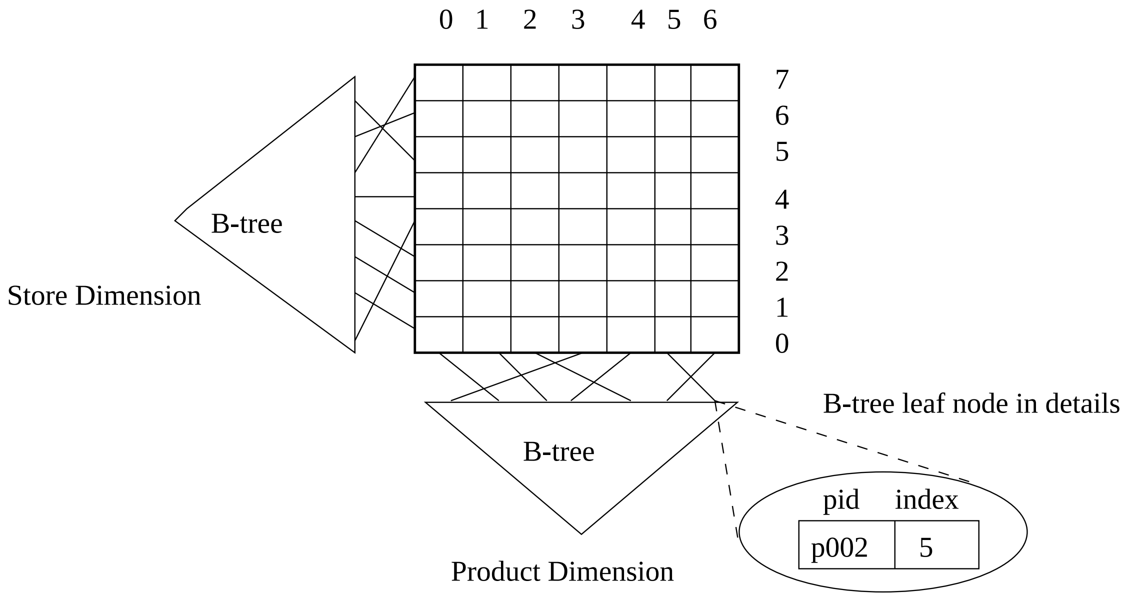

Figure 3: 2-D MOLAP ADT

3.1 Implementation

The MOLAP ADT contains a n-dimensional array and a set of B-tree indices, one for each dimension. The n-dimensional array stores the data from the ROLAP "fact" table. For example, if the MOLAP ADT were used to store the data from the retail sales example from Section 2, each cell of the array would contain one sales volume value. In addition to maintaining the sales volume values, we must also maintain the mapping from dimension value to array index. To do this, we store dimension tables, which are the same as those of the ROLAP Star schema, in the database. The schema for the dimension tables is shown below.

- **Product** (*pid*, pname, type, category);
- **Store** (*sid*, sname, city, state, region);
- **Time** (*tid*, day, month, quarter, year);

The *pid*, *sid*, and *tid* attributes are keys for the Product, Store, and Time relations. In order to maintain the mapping between a tuple in a dimension table (a dimension value) and its corresponding array index value, we store one B-Tree index for each dimension table within the MOLAP ADT. This B-Tree index maps dimension identifiers, e.g. *pid*, to array index values. Dimensions do not have duplicate values; therefore the B-Tree entries are unique. Notice that a vector ($dim1$, $dim2$, .., dim_n) of dimension values uniquely identifies a cell in the array in the MOLAP ADT. Figure 3 shows a small 2-D MOLAP object in detail.

The MOLAP ADT is built on top of the Paradise multidimensional array type. The Paradise multidimensional array uses tiling (also called chunking) to make array access more efficient [DKLPY94]. Storing large arrays on disk in row-major or column-major order may not be efficient because cells that are logically adjacent in the array can be far apart on disk. Tiling, on the other hand, breaks an n-dimensional array into n-dimensional tiles and stores each tile as a SHORE [CDFHM94] large object. (The SHORE storage manager is the storage repository used by Paradise.) In [SS94], the authors found that tiling improves access times for an n-dimensional array. The Paradise multidimensional array also implements compression on a tile by tile basis using the LZW algorithm [Wel84] to further improve performance.

The MOLAP ADT, like the ROLAP fact table, relies on SHORE for database support. As we described above, each tile of a Paradise multidimensional array is stored as a SHORE large object. SHORE provides database support for large objects, including recovery and concurrency control.

3.2 Storage Efficiency

Multidimensional arrays are clearly efficient for storing dense data. For example, consider a 100% dense n-dimensional array. To store this in a relational fact table, each tuple would contain an attribute for each dimension, and one attribute for the "data" value. On the other hand, in an array only the "data" value is stored, since the position of the data value in the array determines the values for the dimension attributes. This means that the ROLAP Star schema requires about $n + 1$ times the amount of storage space that a multidimensional array would need.

However, this storage efficiency does not come for free — on sparse data, the array can require far more storage. In the retail sales example, storing the "fact" table in a three-dimensional array instead of a four attribute ROLAP table saves storage space when the density of the fact table is 25% or greater. However with a density of less than 25%, the MOLAP ADT's array potentially requires much more disk space than the ROLAP "fact" table because the MOLAP ADT allocates storage for every array cell, regardless of whether the cell contains valid data or not. On the other hand, a relational table only stores a tuple for a cell if the cell has valid data.

Fortunately, for sparse data sets, another opportunity arises: sparse arrays are extremely compressible. Using the Paradise array compression code, we found that a compressed array takes less disk space than a ROLAP table all the way down to densities of about 1%.

3.3 Functions

As stated above, the Paradise data model supports functions on ADTs. These ADT functions can be invoked in the Paradise-SQL query language using the standard dot notation. We took advantage of this functionality to define several functions on the MOLAP ADT to provide OLAP functionality.

The functions on the MOLAP ADT include a Read_Write function to retrieve and update the array data in the MOLAP ADT, a function to compute the sum of a subset of the array, a consolidation function and a slice function. The result of the Read_Write and summation aggregation functions are instances of the MOLAP ADT. The Paradise ADT model will eventually allow us to implement complex OLAP analytical functions such as correlation and variance inside the DBMS server.

4 OLAP Consolidation Algorithms

Consolidation is one of the most costly and most widely used OLAP operations, much as the join is one of the most costly and widely used operations in standard relational query processing. We have developed a new consolidation algorithm specifically for the MOLAP ADT and have compared the performance of this algorithm against three ROLAP consolidation algorithms. This section describes our new algorithm in detail and gives brief descriptions of the ROLAP algorithms which we used for comparison.

4.1 MOLAP Consolidation Algorithm

Recall that a basic consolidation looks like this in SQL:

```
select city, type, sum(volume)
from   Sales, Product, Store
where  Sales.pid = Product.pid and
       Sales.sid = Store.sid
group by Store.city, Product.type
```

That is, the MOLAP consolidation operation consists of a Starjoin followed by a "group by" and an aggregation. In the context of the MOLAP ADT, a consolidation operation is a "join" of the MOLAP object's array with its associated dimension tables followed by a "group by" and an aggregation. An important feature of our new consolidation algorithm is that it merges the "Starjoin" and the aggregation into a single operation. This is faster than a Starjoin followed by an aggregation in a non-pipelined system. Furthermore, as we will discuss, the algorithm takes advantage of the fact that arrays are in some sense "index" structures, in that a set of array indices can be used to directly identify the position of a value in the array.

The result of a consolidation operation on an instance of the MOLAP ADT is another instance of the MOLAP ADT. We call these instances the input MOLAP object and result MOLAP object, respectively. In addition, the array associated with the input MOLAP object is called the input array. The result array is defined similarly.

This algorithm has two phases: the first phase scans the input MOLAP object's dimension tables and B-Trees and creates a structure called the **IndexToIndex** array for each dimension. The B-Trees for the result MOLAP object are also created during this phase. The second phase of the algorithm scans the input array and uses the values from the input array and the **IndexToIndex** arrays to create the result array.

The **IndexToIndex** structure is an in-memory structure to keep track of the mapping of the input array's indices to the result array's indices. Logically, the cells in the input array are divided into groups by the group by values. Each group is associated with exactly one cell in the result array. Therefore, each cell in the input array can be associated with exactly one cell in the result array. The **IndexToIndex** array maintains a function from the input array indices in a given dimension to the result array indices in that dimension as defined by the group by values. Given the indices of an element of the input array, we can use the **IndexToIndex** arrays for each dimension to uniquely determine the result element the input element is associated with. In the context of an aggregation

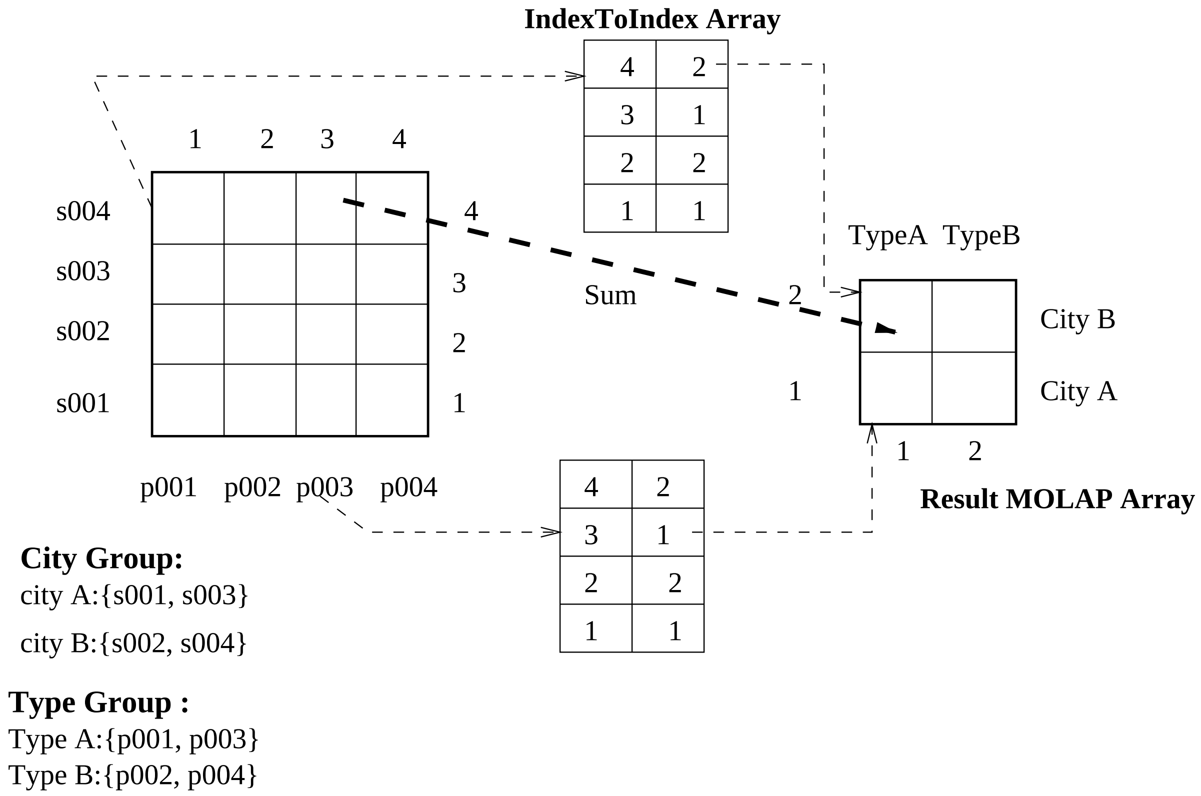

Figure 4: MOLAP Consolidation Algorithm Data Structures

function, we can think of each element of the result array as a "group" and all the elements of the input array that map to that "group" need to be aggregated together.

As stated above, the first phase of the algorithm creates the B-Tree for the result MOLAP object and the **IndexToIndex** arrays. This phase begins by scanning the dimension tables. For each dimension tuple retrieved, the dimension identifier from that tuple is used to probe the appropriate dimension B-Tree in the MOLAP ADT to obtain the tuple's associated index value. The group by value for the tuple is determined and is inserted into the appropriate B-Tree in the result MOLAP object. The insert returns the index of the group by value of this tuple in the result array. The result index value is inserted into the **IndexToIndex** array in the position of the input tuple's index value. Thus the **IndexToIndex** array translates an input index into an result index. Notice that some tuples will have the same group by values, in this case, the duplicate value is inserted into the B-Tree only once.

After building the **IndexToIndex** arrays and result B-Trees for each dimension, the algorithm scans the input array. For each index value of each array cell retrieved, the indices of that array cell and the corresponding **IndexToIndex** arrays are used to determine the indices of the result element associated with this input element. Once all result indices have been determined, the input element is placed in the appropriate array cell and aggregated as appropriate. For example, in a summation aggregation the input element is summed with the value in the result array cell. Figure 4 shows the data structures for the MOLAP consolidation algorithm.

Pseudo-code for the MOLAP consolidation algorithm is below.

MOLAP Consolidation Algorithm

```
For each join dimension table
{   // build the IndexToIndex array
    create an IndexToIndex array;
    scan the dimension table;
    populate the result MOLAP B-tree;
    insert the index pair into the array;
}
scan the input MOLAP array
For each array cell
{
       look up result indices using the IndexToIndex arrays;
       //Star Join
       find the corresponding result MOLAP array cell;
       //Aggregation
       add the input cell to the result array cell;
}
```

This algorithm is currently implemented for summation aggregation, but could easily be extended to aggregates such as count and average. This algorithm assumes that the result MOLAP object and all the **IndexToIndex** arrays fit into memory. The assumption that the **IndexToIndex** arrays fit in memory is reasonable, since dimension tables are typically small for OLAP applications. The assumption that the resulting MOLAP object fits in memory is more questionable. Since it is the result of an aggregation, this object will always be no bigger than the input MOLAP object, and in general it will be much smaller. The result MOLAP object approaches the input MOLAP object in size only when the number of groups in the "group by" implied by the consolidation approaches the number of cells in the input array. For such operations our algorithm would need to be extended to compute the result MOLAP object "chunk by chunk," where each chunk fits in memory.

4.2 A Cross Product ROLAP Consolidation Algorithm

In a standard multi-step relational join on a system that does not support pipelining, a star join produces very large intermediate tables. The I/O cost of reading and writing these large intermediate tables may be so high that executing a cross product of the dimension tables followed by a single join of this cross product with the fact table may actually be cheaper. This technique is often used in practice (along with some interesting hacks to circumvent query optimizers that will not consider cross product plans!) Accordingly we implemented a cross product consolidation algorithm, which generates a cross product of all the dimension tables and joins this cross product with the fact table. After the join, it uses a hash-based group by and an aggregation to generates the final result.

4.3 A Non-Pipelined ROLAP Consolidation Algorithm

This algorithm consists of a standard non-pipelined, multi-step relational join. We rely on the Paradise query optimizer to generate the join order based on its cost estimates. Paradise uses a hash join algorithm to compute each of the joins in the query. After the join is completed, a hash-based group by and an aggregation are used to produce the final result.

4.4 A ROLAP Pipelined Consolidation Algorithm

If the RDBMS supports pipelining, then a pipelined hash join is an attractive way to implement a consolidation. Such a join first builds a hash table for each dimension table. Then it scans the fact table and, for each tuple generated, probes the first hash table to generate a join tuple. This result tuple is then pipelined on to probe the next hash table, and so on. This algorithm is ideal for a typical OLAP setting, where we have relatively small dimension tables and a large fact table.

The version of Paradise used in this study does not support pipelining and therefore can not execute this plan. To simulate the performance of this algorithm, we implemented within Paradise a Starjoin algorithm, which is similar to the pipelined hash join. Our Starjoin Consolidation algorithm uses one hash table for each dimension table, and a hash table for the aggregate function. The algorithm first builds an in-memory hash table for each dimension table. Each hash entry contains the value of the tuple's group by attribute and the value of the tuple's key attribute. After building the hash tables, the algorithm scans the fact table. For each tuple from the fact table, the algorithm probes the hash tables to obtain the group by values for this tuple. It then constructs a new "joined" tuple containing all the group by values and the data value. If there exists a corresponding group tuple for the "joined" tuple in the aggregation hash table, the data value is aggregated to the group tuple. Otherwise, the "joined"tuple is inserted into the aggregation hash table. The results are output from the aggregation hash table at the end of the algorithm.

5 Performance

We have used two data configurations and three queries to compare the performance of the MOLAP ADT multidimensional array versus the ROLAP tables. Two different database sizes are used to determine how the MOLAP and ROLAP queries scale up for large OLAP data sets. This section describes our test database configurations and our performance results.

5.1 Test OLAP Database Schema

The test OLAP database schema is similar to that of the retail store database. The ROLAP schema for the test database is below:

- **fact** ($d0$ int, $d1$ int, $d2$ int, $d3$ int, volume int);
- **dim0** ($d0$ int, h01 string, h02 string);

- **dim1** ($d1$ int, h11 string, h12 string);
- **dim2** ($d2$ int, h21 string, h22 string);
- **dim3** ($d3$ int, h31 string, h32 string);

The h**X**1 and h**X**2 attributes of all the dimension tables are uniformly distributed. The h**X**1 values are drawn from 100 distinct values and the h**X**2 values are drawn from 10 distinct values. Thus the h**X**1 and h**X**2 attributes are hierarchically structured, similar to the dimensions in the retail sales example.

5.2 OLAP Queries

Query 1 is an OLAP consolidation query that joins the fact table or the MOLAP ADT array with all dimension tables, groups by each dimension table's h**X**2 attribute, and sums on the sales volume.

```
select sum(volume), dim0.h02, dim1.h12, dim2.h22
from   fact, dim0, dim1, dim2
where  fact.d0 = dim0.d0 and
       fact.d1 = dim1.d1 and
       fact.d2 = dim2.d2
group by h02, h12, h22
```

Query 2 scans the ROLAP fact table or the MOLAP ADT array, groups by two dimensions, and sums on the sales volume. Note that Query 2 does not need to refer to the dimension tables.

```
select sum(volume) d1, d2
from fact
group by d1, d2
```

Query 3 is an OLAP slice operation on dimension dim2.

```
select volume, d0, d1, d3
from fact where fact.d2 = 59
```

5.3 System Configuration

These tests were run on an 90MHZ Intel Pentium processor with 32MB main memory and a 2.0 GB Quantum Fireball disk. The operating system used was Solaris 2.4. We used version 0.5 of Paradise, configured with an 8MB buffer pool. The test databases were created using the Unix file system on a local disk. We flushed both the Unix file system buffer and Paradise buffer pool before running each query. The configurations of the two test databases are shown below.

To understand how we generated the data it is easiest to think in terms of the array representation. We generated two types of data: uniform, and Zipfian. To generate a uniform data set $x\%$ dense, we iterated through all cells of

Parameters	Value
Num of dimension	3
Dimension 0 Size	200
Dimension 1 Size	100
Dimension 2 Size	100
Fact Table Density	20% 10% 5% 1% 0.5%

Table 1: Small Database Configuration

Parameters	Value
Num of dimension	4
Dimension 0 Size	40
Dimension 1 Size	40
Dimension 2 Size	100
Dimension 3 Size	100
Fact Table Density	20% 10% 5% 1% 0.5% 0.1%

Table 2: Large Database Configuration

the array, as each cell was visited, we generated a random number between 1 and 100; if the random number r was in the range $1 \leq r \leq x$, we set the cell to a one, otherwise we set it to an invalid constant to represent "no data." (The performance of the OLAP queries we tested depend only on the presence or absence of a data value in the data cells, not the specific value in the cell.)

For the Zipfian data we again iterated through all the cells of the array, but this time each cell had a different probability of receiving a valid data value; the probabilities varied following Zipfian distribution. While we ran a number of experiments on Zipfian data, they did not provide any insight beyond that given by uniform data, so we do not present those results here.

The table representation of the data set was generated as determined by the array representation. One tuple was generated for each cell of the array that had valid data. For example, if the array had a value 1 at cell $A[i, j, k]$, then we generated a tuple $(i, j, k, 1)$.

5.4 Performance Results

5.4.1 Consolidation Query

The results of **Query 1** for the MOLAP and the three ROLAP consolidation algorithms on the small size OLAP data set are shown in Figure 5. In this figure, as in all other figures, we show performance results for data densities ranging from 0% to 20%. The non-pipelined ROLAP and cross product ROLAP consolidation algorithms are much slower than the other two algorithms since they both have a large overhead for storing the temporary results some of which are as large as the fact table itself. The pipelined ROLAP and MOLAP algorithms do not store any intermediate results and scan the fact table or MOLAP array only once and are therefore much faster.

Since the non-pipelined ROLAP and cross product ROLAP consolidation algorithms perform much worse than the other two algorithms, we only focus on the pipelined ROLAP and MOLAP consolidation algorithm for the rest of the section. With the good performance of these algorithms (relative to the other ROLAP algorithms) it was feasible to run tests on the larger database size. Figure 7 shows the performance of these two algorithms on the large database configuration. This figure also shows two variants of the MOLAP algorithm, with and without compression.

While we expected the MOLAP ADT to perform well on dense data, we were surprised to see how well it

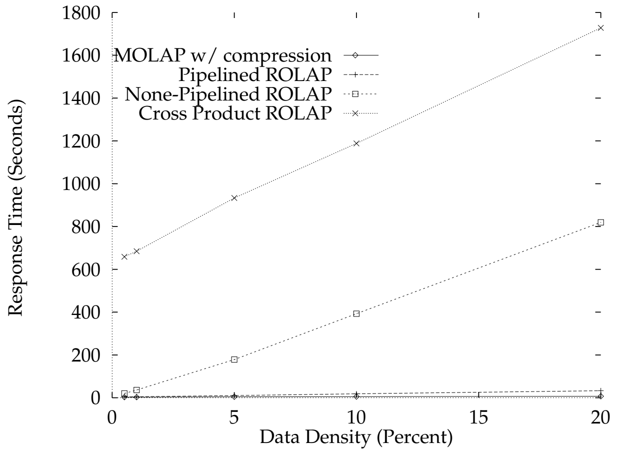

Figure 5: Query 1 - Small Database

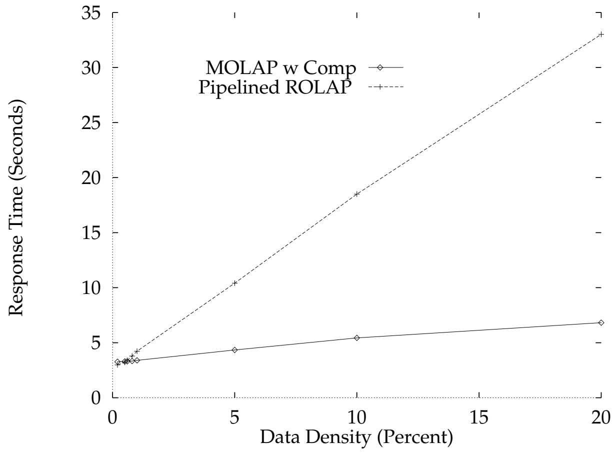

Figure 6: Query 1 - Small Database, Detail

performed as compared to the ROLAP algorithm on relatively sparse data sets. Here is an intuitive explanation for part of the performance difference; we will quantify these differences below.

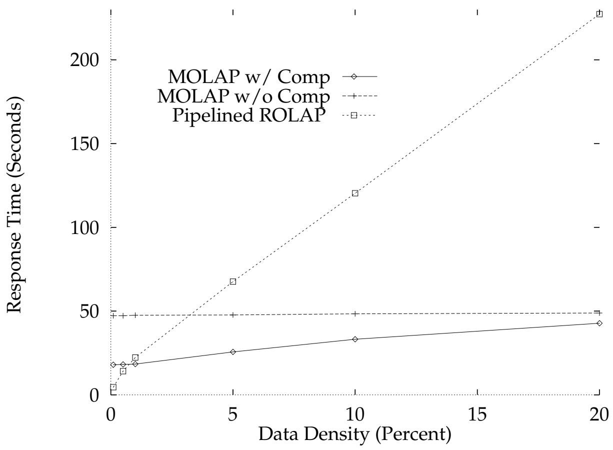

Figure 7: Query 1 - Large Database

Consider again the following consolidation query, Query 1:

```
select sum(volume), dim0.h02, dim1.h12, dim2.h22
from   fact, dim0, dim1, dim2
where  fact.d0 = dim0.d0 and
       fact.d1 = dim1.d1 and
       fact.d2 = dim2.d2
group by h02, h12, h22
```

Any algorithm to compute this consolidation must map all dimension attributes (d0, d1, and d2) in the fact table to their parents in their respective hierarchies (h02, h12, and h22.) In the MOLAP implementation, this mapping is done using simple index lookups in the **IndexToIndex** arrays. For each value in the MOLAP array, its array indices are used to determine the array indices in the output array to which that input value corresponds. The input value is then summed with the output value. Thus for each array element, aggregation involves a series of array lookups (one for each dimension) and a sum. This speed/simplicity of aggregation is made possible by virtue of the fact that all possible values for the fact table are stored by the MOLAP ADT (input and output).

The ROLAP algorithm, on the other hand, because it only stores valid entries, but not all possible entries, must use hashing or some other mechanism, instead of array lookups, to do the aggregation. For each tuple in the ROLAP fact table relation, we must map the tuple to its aggregation group, by looking at the dimension hash table, using the hash value of its group-by attributes to find the group entry in the result hash table, and adding the tuple volume to the result tuple volume. This is significantly more expensive than the array lookups and sum done by the MOLAP algorithm and is a direct result of storing only part of the possible values. In addition, the ROLAP algorithm must read, process, and write tuples, as opposed to the MOLAP algorithm's processing of array values. Storing the fact table values as tuples is very general and is a major advantage of ROLAP; however, as we discuss below, this generality contributes significant overhead to the algorithm.

In order to quantify where the performance differences between the MOLAP and ROLAP algorithms were coming from, we timed several segments of both algorithms. We found that doing the join and aggregation and fetching the fact table data are the two primary causes of the performance differences. In Figures 8 and 9, we present measurements of the execution time for the join and aggregation and data fetching sections of each algorithm for the small database.

In Figure 8, the "Get Tuple + Join + Aggregate" line shows the time taken to both fetch the fact table tuples and do the join and aggregation for the pipelined ROLAP algorithm. The "Get Tuple" line shows the time needed to fetch the fact table tuples, which includes the time taken to extract attributes values from the tuples. The difference between the two times gives the CPU time taken by the join and aggregation operations. The difference between the "Pipelined ROLAP Consolidation" and the "Get Tuple + Join + Aggregate" lines is the time to execute the rest of the query which includes scanning the dimension tables, writing the result tuples to the database, and Paradise system overhead for things such as query processing and transaction commit.

Similarly, in Figure 9, the "Get Chunk + Join + Aggregate" line shows the time needed to read array chunks, to uncompress those chunks, and to do the join and aggregation for the MOLAP algorithm. The "Get Chunk" line shows the time taken to read and to uncompress the array chunks. The gap of the top two lines is the time to execute scanning the dimension tables, writing the result array, and doing Paradise system overhead.

Based on the results in Figures 8 and 9, we can see that scanning the ROLAP fact table is slower than reading and uncompressing the MOLAP array chunks for data densities above about 1%. There are two explanations.

First, for the data densities above 0.5%, the ROLAP fact table is larger than the compressed MOLAP array. This can be seen from Figure 10. For example, at 20% data density, the size of the ROLAP fact table is 16.4 MBytes vs. 3.42 MBytes for the compressed MOLAP array. This is partially due to the high compressibility of the sparse MOLAP arrays and partially due to overhead in the general tuple storage structure used by the ROLAP table.

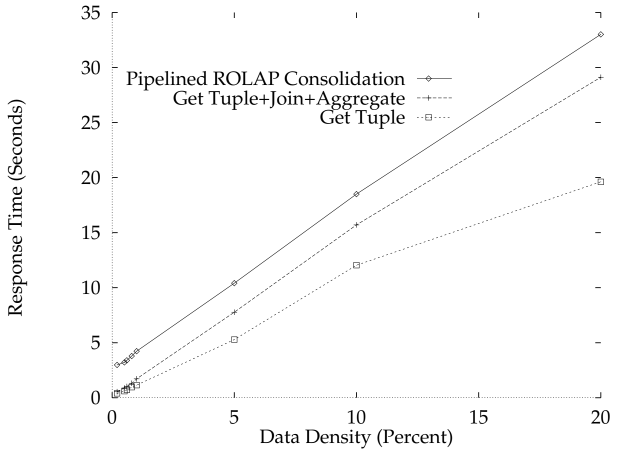

Figure 8: Performance Break Up for Pipelined ROLAP Consolidation for the Small Database

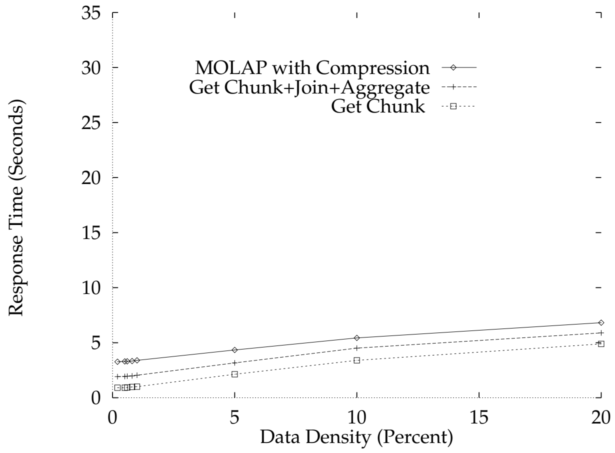

Figure 9: Performance Break Up for MOLAP Consolidation for the Small Database

About two thirds of the ROLAP fact table size is SHORE overhead. Each fact table tuple contains 16 bytes of data; however, SHORE adds 27 bytes of overhead making each tuple 43 bytes. This extra overhead is due to the general tuple structure and slotted page structure used by SHORE and would likely be present in any RDBMS.

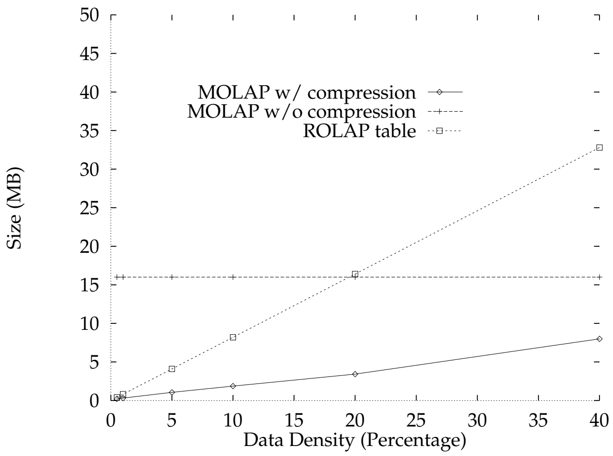

Figure 10: MOLAP ADT vs. ROLAP Fact Table - Storage Efficiency

Second, in the ROLAP table there is overhead to extract the attribute values from the tuples and to convert attribute values from database format to memory format. The time to fetch the tuples for the 20% dense ROLAP fact table contains 6 seconds to extract the attribute values in the small database configuration. So, ROLAP pays a price for using a very general storage structure. Scanning the array data, on the other hand, does not have any overhead to convert the array data from database format to memory format since the actual data within each chunk has the same format for memory and disk. Uncompressing each chunk does have some CPU overhead. For a dense fact table, the overhead for fetching ROLAP fact table tuples is greater than that of reading and uncompressing MOLAP array chunks.

In addition to taking more time to fetch the fact table data, the ROLAP algorithm also consumes more CPU time for the join and aggregation than the MOLAP algorithm does. We can see this by comparing the size of the gap between the bottom two lines in Figures 8 and 9. As we discussed in the previous section, the nature of the storage of sparse multi-dimensional data for MOLAP and ROLAP determines the nature of the join and aggregation for each algorithm. To do the join and aggregation, the pipelined ROLAP algorithm has to perform several hash table look-ups and string copies and at least one string comparison for each fact table tuple. Those operations are more expensive than the few array index look-ups required for each valid array value in the MOLAP algorithm.

For low data densities (< 1%), the results in Figure 11, tell a different story. The pipelined ROLAP consolidation algorithm outperforms the MOLAP algorithm. The ROLAP algorithm spends less time both scanning the fact table and doing the join and aggregation than the MOLAP does. This is because while the compressed MOLAP and ROLAP data sizes are roughly comparable for these densities, during processing the MOLAP algorithm uncompresses its data and processes a huge number of invalid (no data) cells. This represents a substantial overhead as compared to the ROLAP algorithm, which only handles valid tuples.

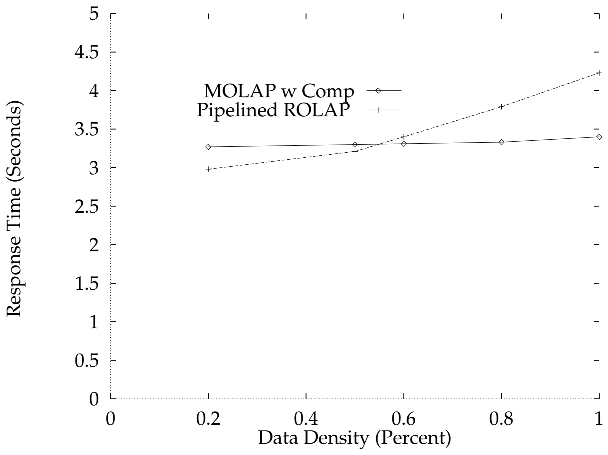

Figure 11: Query 1 - Low Data Densities

5.4.2 Other Queries

Figure 12 shows the results of Query 2. The execution time for the ROLAP algorithm for Query 2 is slower than that for the MOLAP algorithm, for data density no less than 1%. There are two factors contributing to this. One is that, as we discussed above, scanning the ROLAP fact table is slower than scanning the MOLAP array. The other is that writing the results to the database for ROLAP is slower than for MOLAP. It is more expensive to put tuples into a slotted page and write slotted pages than to write the large array chunks to the database. Our performance results show that writing the ROLAP results takes 4.24 seconds while writing the MOLAP result array takes 1.87 seconds.

Figure 13 shows the results of Query 3. For high data densities (>= 1%), the MOLAP slice operation is faster than the ROLAP one. Notice that the MOLAP slice operation only reads those chunks containing the sliced portion of the array, instead of all chunks. The ROLAP slice query, on the other hand, scans the entire fact table.

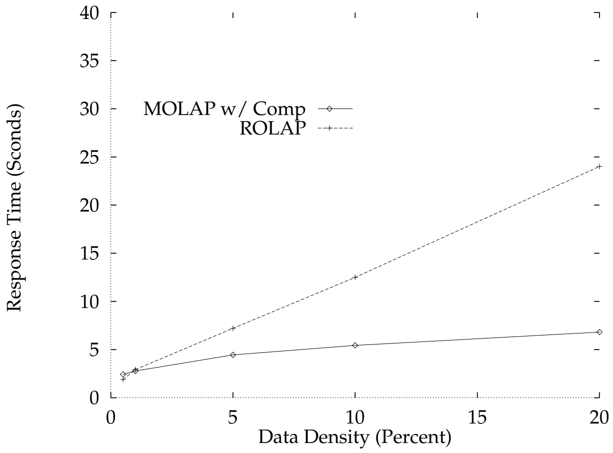

Figure 12: Query2 - Small Database

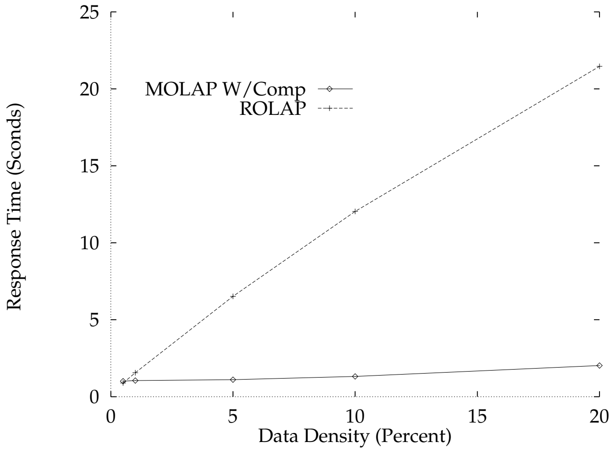

Figure 13: Query3 - Small Database

Finally, it is important to stress that the ROLAP algorithms and the MOLAP algorithms both use SHORE for database system support, including buffer pool management, recovery, and concurrency control. Although SHORE uses different recovery and concurrency control mechanisms for tuples and large objects, our performance results include realistic system overheads for both MOLAP and ROLAP implementations.

6 Conclusion and Future Work

In this paper, we have used an implementation of a MOLAP Abstract Data Type to show that multidimensional arrays can be more efficient both in terms of storage space and performance than ROLAP tables for multidimensional OLAP data sets. This performance comparison was done in the context of the Paradise object-relational database system. We conclude that multidimensional arrays are surprisingly efficient, when compared with ROLAP tables, for storing relatively sparse OLAP data sets. In particular, the MOLAP ADT outperforms ROLAP tables on the important operation of consolidation for data densities of 1% and above.

These results suggest that the ADT mechanism of object-relational database technology may be useful for providing a hybrid OLAP solution which would allow the implementation of a multidimensional OLAP structure within a standard database architecture. With the implementation of the MOLAP ADT, we have taken a first step in this direction; however, there is still much work to be done. We intend to extend this research to attempt to provide a complete framework for the implementation of OLAP within an object-relational system. To start with, we intend to extend the MOLAP ADT to support additional OLAP functions. In addition, currently all access to the multidimensional data stored within the MOLAP ADT must be done through MOLAP ADT member functions. We would like to investigate ways to integrate access to the MOLAP ADT's multidimensional data with SQL so that we can support ad hoc querying of the MOLAP ADT.

Currently many ROLAP vendors use special index structures such as bitmap indexes to speed up OLAP queries in the presence of very restrictive selections in the queries. We plan to compare the performance of these specialized ROLAP index structures and the MOLAP ADT for such queries. Finally, we believe that the large OLAP data set sizes require parallel computing and we would like to investigate parallelization of OLAP data structures and key OLAP operations in the context of a parallel object-relational database system.

7 Acknowledgements

We would like to thank Jignesh Patel, Navin Kabra, and Jiebing Yu for their help with the implementation of the ROLAP consolidation algorithms.

References

[Fin] Richard Finkelstein. *Understanding the Need for On-Line Analytical Servers*, Richard Finkelstein, President, Performance Computing, Inc. "**http://www.arborsoft.com/papers/finkTOC.html**"

[CDFHM94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White and M. Zwilling. "Shoring up Persistent Objects," Proc. of the 1994 SIGMOD Conference, May, 1994.

[Codd93] E.F. Codd, S.B. Codd, and C.T. Salley. *Providing OLAP (On-line Analytical Processing) to User-Analysts: An IT Mandate*, White Paper, E.F. Codd and Associates. "**http://www.arborsoft.com/papers/coddTOC.html**"

[DKLPY94] D. J. DeWitt, N. Kabra, J. Luo, J.M. Patel, and J. Yu. "Client-Server Paradise". In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, 1994

[Kenan] Kenan Technologies. *Guide to OLAP Terminology*, Kenan System, Cambridge, MA. "**http://www.kenan.com/acumate/olaptrms.html**"

[SAT95] M.J.Salyor, M.G.Achaya, and R.G.Trenkamp. True Relational OLAP: The Future of Decision Support, *Database Journal*, Dec 1995, p.38. "**http://www.strategy.com/tro_dbj.html**"

[SS94] Sunita Sarawagi, Michael Stonebraker, "Efficient Organization of Large Multidimensional Arrays". In *Proceedings of the Eleventh International Conference on Data Engineering*, Houston, TX, February 1994

[Wel84] T.A. Welch "A Technique for High-Performance Data Compression". *IEEE Computer*, 17(6), 1984.