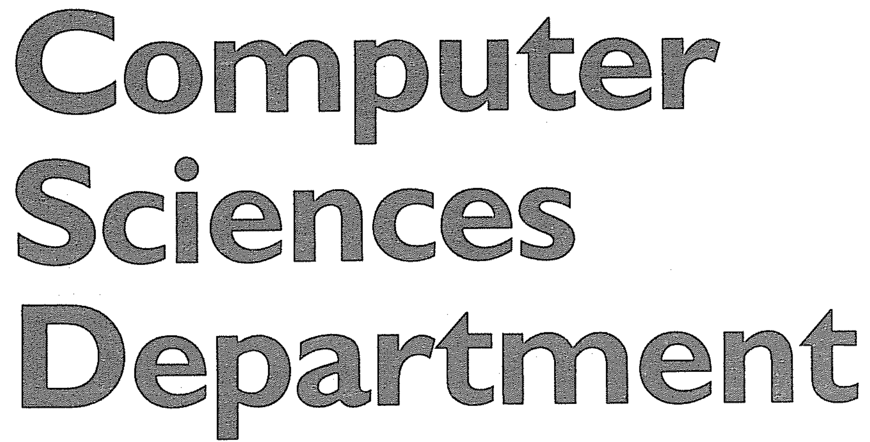

On the Performance of an Array-Based ADT for OLAP Workloads

Yihong Zhao
Kristin Tufte
Jeffrey Naughton

Technical Report #1313

May 1996

UNIVERSITY OF
WISCONSIN
MADISON

1313

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

May 6, 1996

Abstract

There is currently a raging 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 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 a new and important database technology. OLAP provides business managers with the tools they need to effectively analyze large volumes of data. The analytical tools provided by OLAP include a multidimensional view of business data, the ability to aggregate the data in arbitrary ways, and the ability to apply simple and complex analytical functions to

the data. 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 see 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].

Which architecture is better suited to meet the analytical needs of OLAP consumers? An 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. This suggests 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 an introduction to OLAP, Section 3 describes the debate between ROLAP and MOLAP in more detail, Section 4 describes the design and implementation of the MOLAP ADT, Section 5 describes the new MOLAP consolidation algorithm and the three ROLAP consolidation algorithms with which it is compared, and Section 6 describes the performance studies. Finally, Section 7 concludes and discusses future work.

2 What is OLAP?

As an introduction to OLAP, imagine an electronics company executive attempting to make decisions for this year's marketing strategy for computer equipment based on an analysis of last year's sales figures and using an OLAP product to do the analysis. The executive begins by asking for total sales volume for computer products for each month in the past year. Since December is the month with the highest sales, the executive "drills down" to the month of December and requests total sales volume of computer products for each day in December. Wondering which products produced the most revenue, the executive "rotates" the sales data cube by requesting total sales volume by computer product for December. After noticing that both color laser printers and Pentium PCs sold well in December, the executive requests the a calculation of the correlation between sales of color laser printers and Pentium PCs. A strong correlation is discovered giving the executive a powerful piece of information which will help in the design of the marketing strategy.

This simple scenario illustrates three distinguishing features of OLAP. First, **OL**AP is an **O**n-**L**ine process; the decisions about which queries to ask are based on answers to the previous queries. Since it is an interactive process, each query requires a sub-second response time. Given that OLAP data sets are typically large, ranging in size from gigabytes to even terabytes, providing sub-second response times poses a technical challenge. Second, supporting OL**A**P means supporting the execution of complex **A**nalytical functions. OLAP queries often involve mathematical or statistical functions such as correlation, expected value, variance, and moving averages. Third, OLA**P** products must support the **P**rocessing of a wide variety of queries. OLAP systems are used by many levels of business management, and therefore, should support queries that vary significantly in detail and complexity.

In the rest of this section, we continue to provide background on OLAP by introducing an example of a retail sales database and describing a set of typical OLAP operations.

2.1 Example OLAP Database

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 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. Most OLAP datasets are sparse.

The dimensions of an OLAP array are typically hierarchical. 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.

2.2 ROLAP Schema

ROLAP people view this multidimensional array as a set of tables consisting of one table for the array data itself and one table for each of the dimensions. The ROLAP 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

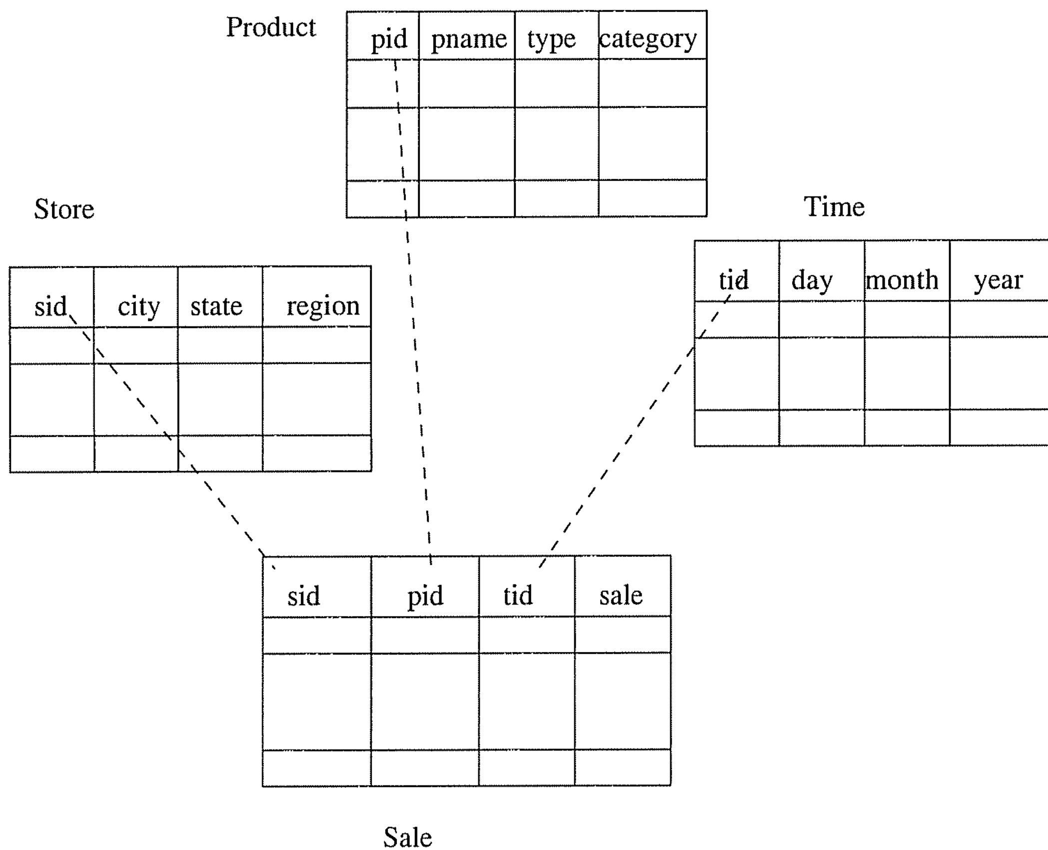

Figure 1: ROLAP Star Schema

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.

This schema is typical of OLAP databases and is called a "Star Schema" in ROLAP terminology. As illustrated in Figure 1, Sales is in the center of the star and is connected to each dimension in the table by a foreign key.

2.3 MOLAP Schema

MOLAP people view the example data as a 3-dimensional array, as shown in Figure 2. MOLAP systems store the data either in a multidimensional array or in a special data structure designed specifically for storage of sparse multidimensional arrays.

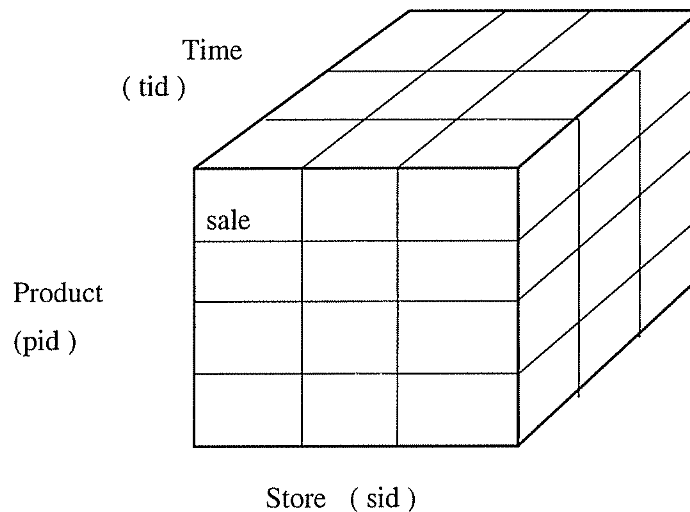

Figure 2: MOLAP N-D Array

2.4 Operations

There is a reasonably standard set of OLAP operations. This section describes the most important of these operations, including consolidation, drill down, selection, and slicing and dicing [Kenan].

Consolidation 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
```

Standard SQL can express simple consolidation queries as shown above, but can not express consolidation queries involving analytical functions such as variance and correlation.

In terms of query processing, an OLAP consolidation query involves a join of the "fact" table with the dimension tables followed by a group by and an aggregation. The join portion of the consolidation is called a "Starjoin" in OLAP terminology. In our example query, the Sales table would be first joined

with the Store and Product tables to produce `Store.city, Product.type, Sales.volume` tuples. Finally, a group by and aggregation would be performed to calculate total sales for each city, type pair.

Drill down A drill down operation involves navigating through a dimension hierarchy from highly aggregated data to the most detailed data. Drill down typically involves a series of queries. For example, the following series of queries/questions demonstrates a drill down through the time dimension hierarchy: 1) request total sales for each year for the past ten years, 2) request total sales for each quarter in a particular year; 3) request sales data for each month in a quarter; 4) finally request sales volumes for each day within a particular month.

Slicing and Dicing A slice of a n-dimensional array is a n-1 dimension (or lower) subset of the array. For example, 2-dimensional plane is considered a slice of a 3-dimensional array. A slice is specified by fixing one (or more) dimension(s) of the array. Below is a SQL query selecting a slice of the three dimensional Sales data set. In this case, the product dimension is fixed to be equal to Pentium PC. The slice contains all data related to the Pentium PC and excludes all data for products other than the Pentium PC.

```
select Time.day, Store.sname, Sales.volume
from Sales, Product, Store, Time
where  Product.type = "Pentium PC" and
       Sales.pid = Product.pid and
       Sales.sid = Store.sid and
       Sales.tid = Time.tid
```

Slicing and Dicing is the interactive process of navigating through an array requesting the display of slices of the data. Typically two-dimensional slices are displayed. The slices requested may be consolidations or drill downs and may include analytical functions such as variance or correlation.

Selection A selection operation returns a subset of the fact table. A selection specification consists of specifying range predicates on some of the array dimensions. Selection is a generalization of slicing and does not necessarily return a lower dimension table as slicing does. The following is a typical selection query expressed in SQL:

```
select Store.sname, Product.pname, Sales.volume
```

```
from   Sale, Product, Store
where  Sale.pid = Product.pid and Sale.sid = Store.sid and
       Product.type in {"LaserPrinter", "486PC"} and
       Store.state in {"CA", "WI"}
```

3 ROLAP vs. MOLAP

As stated before, ROLAP stores data in relational tables while MOLAP stores it in custom multidimensional structures. These different approaches have significant advantages and disadvantages. An advantage of using relational tables to store OLAP data is that OLAP data can be easily integrated with other business data that is also stored in relational databases. MOLAP systems on the other hand are closed systems and it is difficult to integrate data from a MOLAP system with data stored in a RDBMS. A disadvantage of using relational tables for OLAP data is that relational tables can be a less intuitive and less efficient way to store multidimensional data.

For example, consider a 100% dense n-dimensional array. To store this in a 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. Another disadvantage of using relational tables is that the Star schema introduced in Section 2.2 requires multi-way joins to create the results. These multi-way joins on large tables, such as OLAP "fact"tables, consume enormous amounts of system resources which can cause degraded performance especially with multiple users in the system.

A serious concern about ROLAP is the limited functionality of the SQL interface. Standard SQL provides only simple aggregation functions such as sum, average, and count while OLAP analysis requires more complicated analytical functions. It is difficult to perform these operations within an RDBMS, so ROLAP products often resort to pulling tuples out of the DBMS and computing these analytic functions in an application program. MOLAP systems in general support their own customized languages rather than SQL, and have chosen to implement more complex analytical functions within the MOLAP server.

Finally, ROLAP systems can scale up to the large data set sizes required for enterprise solutions by using parallel RDBMS technology. MOLAP systems currently can not scale beyond say 20-30 gigabyte data

sets [SAT95], since parallel MOLAP technology has not yet been developed.

4 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. 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.

4.1 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 similar to 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

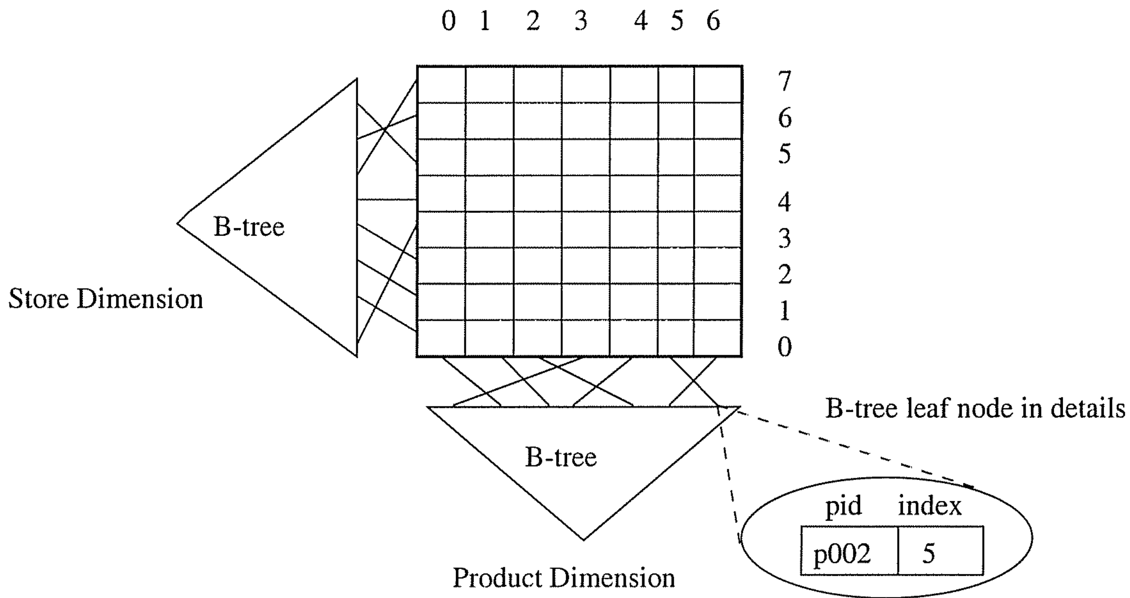

Figure 3: 2-D MOLAP ADT

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. Storing large arrays on disk in row-major or column-major order is inefficient 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 an object on disk. This helps keep elements that are close together in the array close together on disk, which improves access times [DKLPY94] [SS94]. The Paradise multidimensional array also implements compression on a tile by tile basis using the LZW algorithm [Wel84] to further improve performance.

4.2 Storage Efficiency

Multidimensional arrays are clearly efficient for storing dense data. 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 more disk space than the ROLAP "fact" table because only 25% of the array cells contain valid data. Fortunately, below this density, 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% (see Figure 11).

4.3 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 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 Starjoin function. The Starjoin function joins an instance of the MOLAP ADT with one or more dimension tables. 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.

5 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.

5.1 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 OLAP 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** 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 uniquely determine the result element it is associated with. In the context of an aggregation 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

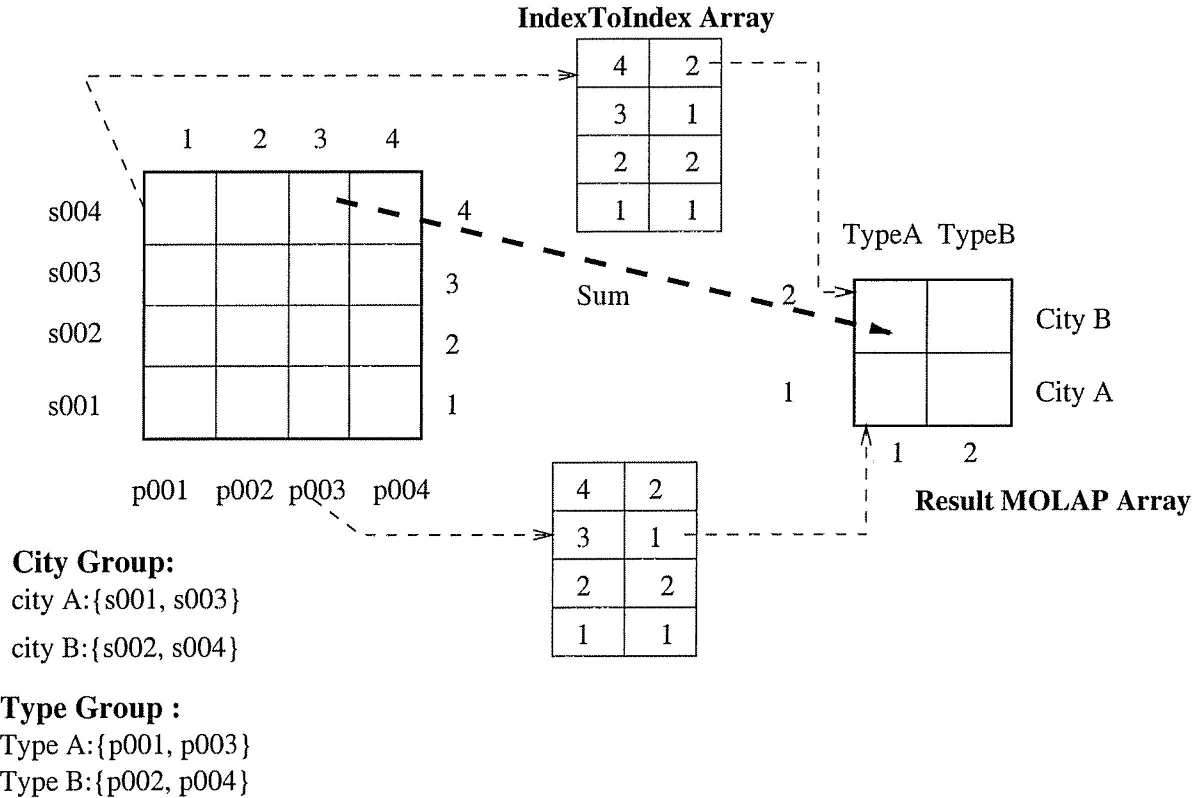

Figure 4: MOLAP Consolidation Algorithm Data Structures

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
{   // building 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 original MOLAP array
For each array cell
{
       look up the IndexToIndex arrays;
       //Star Join
       find the corresponding result MOLAP array cell;
       //Aggregation
       add the 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.

5.2 Cross Product 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.

5.3 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.

5.4 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 dimension

values for this tuple. It then constructs a new "joined" tuple containing all the dimension values and the data value. This tuple is inserted into the aggregation hash table in the standard way. The results are output from the aggregation hash table at the end of the algorithm.

6 Performance Tests

We have designed two data configurations and a set of OLAP 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.

6.1 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.

6.2 Test OLAP Queries

We used the following set of queries to measure the performance of the MOLAP ADT and the ROLAP tables. For ease of understanding, each query is presented in SQL below.

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. The results of Query 2 and Query 3, which is a scan query, together may provide insight into the cost of hash based aggregation. Note that Query 2 does not need to refer to the dimension tables.

```
select sum(volume) d0, d1
from fact group by d0, d1
```

Query 3 scans the MOLAP ADT array or the ROLAP fact table. Scan is a very simple query and is part of the MOLAP and pipelined ROLAP consolidation operations. It will help us dissect the cost of both algorithms.

```
select volume
from fact where volume = 0;
```

Query 4 is an OLAP slice operation on dimension dim2.

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

6.3 System Configurations

These tests were run on an 90MHZ Intel Pentium processor with 32MB main memory. The software platform is a **Paradise** server version 0.5 configured with an 8MB buffer pool. The configurations of the two test databases are shown below.

Small Size Database Configuration

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%

Large Size 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%

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 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.

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 $(i,j,k,1)$.

6.4 Performance Results

The results of **Query 1** for the MOLAP and the three ROLAP consolidation algorithms on the small size OLAP data set is 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 will only compare pipelined ROLAP and MOLAP algorithms for the rest of the tests.

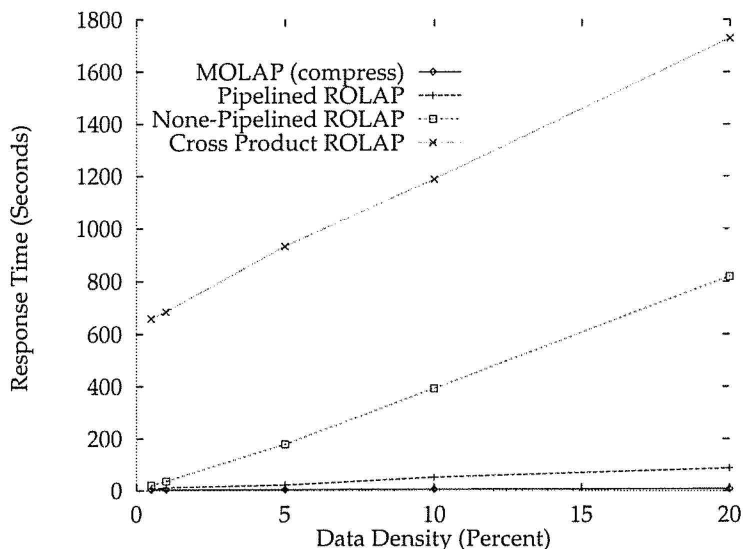

Figure 5: Performance of Query 1 for the Small Database Configuration

Figure 6 shows the performance of **Query 1** for the MOLAP and pipelined ROLAP consolidation algorithms for the large database. Figure 7 shows a closer view of the 1% to 5% density region of the graph. We have two different results for the MOLAP consolidation algorithm: one in which the MOLAP array is stored without compression; and one which uses compression. From 1% to 20% density, the MOLAP consolidation with compression beats the pipelined ROLAP consolidation algorithm by a large margin. There are two reasons for this. First, the ROLAP table uses slightly less disk storage than the compressed array. This means that both algorithms have almost identical I/O costs. Second, hash based aggregation on the fact table is much more expensive than the MOLAP consolidation algorithm's index look ups. For each fact table tuple, the ROLAP algorithm has to extract a string from each dimension table, concatenate the four strings into one string, and calculate the hash value for the result string. In addition, decom-

pressing the MOLAP array in memory is cheap for sparse data. Therefore, the consolidation algorithm on the compressed MOLAP array is two times faster than the pipelined ROLAP for 1% dense data. For a very sparse array with less than 0.5% density, the ROLAP consolidation algorithm outperforms the MOLAP algorithm, even when compression is used. With such sparse data, the compressed MOLAP array is much larger than the ROLAP table. I/O cost dominates the cost of the query and therefore the ROLAP algorithm is faster. We also observe that the consolidation algorithm performs better with compression performs better than without compression when the data is less than 10% dense.

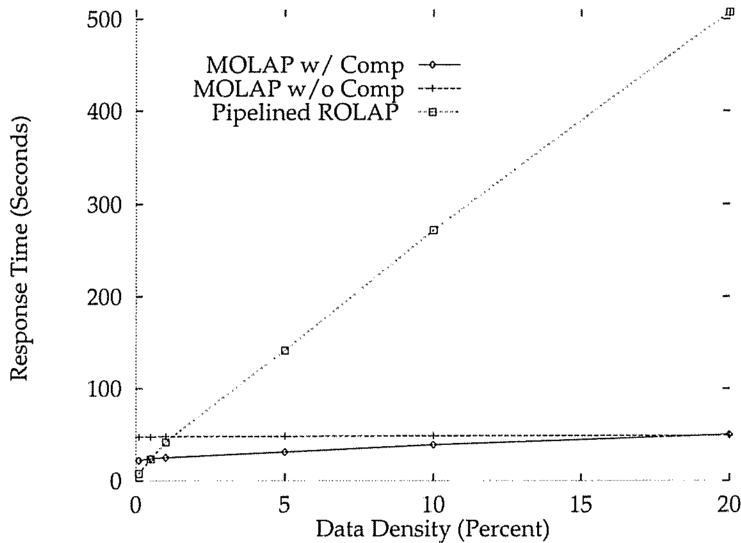

Figure 6: Performance of Query 1 for the Large Database Configuration

Comparing the performance curves in Figure 5, we can see that the MOLAP and the pipelined ROLAP consolidation algorithms scale up nicely for the large databases.

Figure 8 and 9 show the results of **Query 2** and **Query 3** for the compressed MOLAP object and the ROLAP table. There are two interesting results in these figures. First, the MOLAP aggregation (sum) is much cheaper than the hash based aggregation (sum). Second, for data that is more than 5% dense, the MOLAP algorithm with a compression performs better than the ROLAP algorithm on both queries. The ROLAP aggregation (sum) is 20 seconds slower than the ROLAP scan for data that is 10% dense. This tells us that the cost of the aggregation is around 20 seconds. On the other hand, the MOLAP aggregation

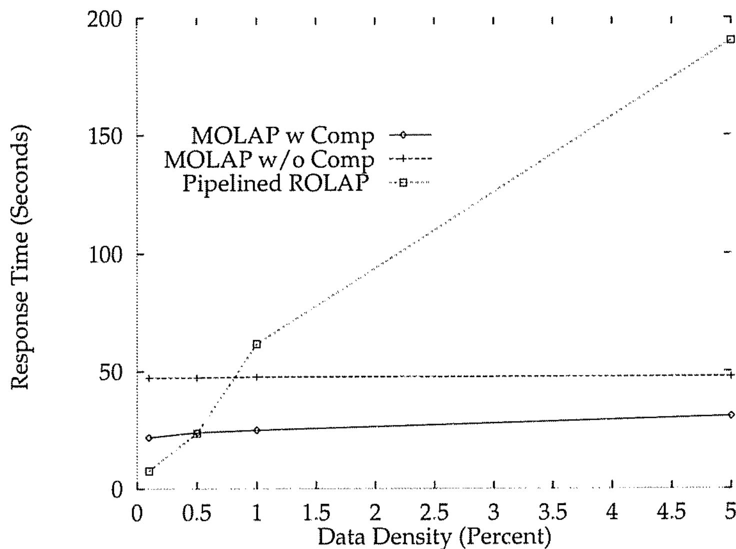

Figure 7: Performance of Query 1 for the Large Database Configuration - 0% to 5% Density

(sum) only cost a few seconds.

Figure 10 shows the results of **Query4** for the MOLAP algorithm with compression and the pipelined ROLAP algorithm. It appears that the slice operation on the MOLAP algorithm outperforms the one on the ROLAP table for data densities greater than 1%.

6.4.1 Storage Efficiency

Figure 11 shows the disk storage for a MOLAP data array with a Zipfian distribution, a MOLAP data array with a uniform distribution, an uncompressed MOLAP array, and a ROLAP table for different data densities. The test array is a 200x100x100 array of integers. For data densities less than 20%, the algorithm compresses a Zipfian distribution data array slightly more efficiently than it compresses a uniformly distributed array. For a very sparse array, with less than 1% data density, the compressed array size is very close to the ROLAP table size. For a dense array, with data density larger than 20%, the MOLAP array requires less disk space than a ROLAP table.

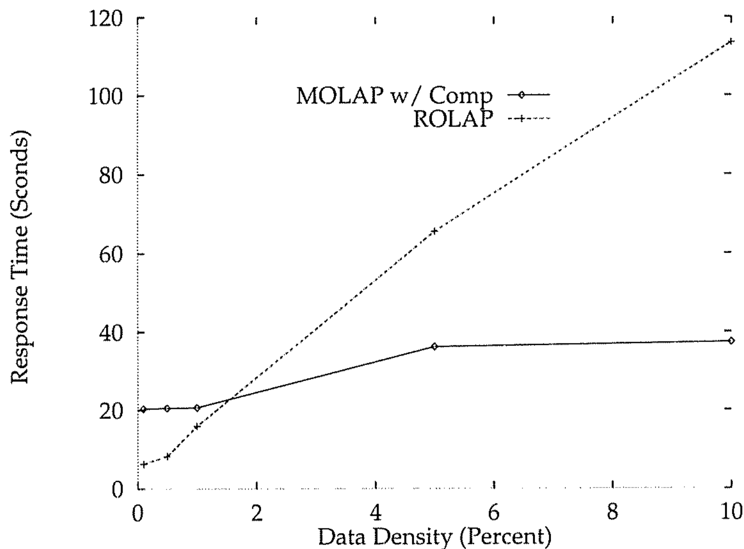

Figure 8: Performance of Query 2 for Large Database Configuration

7 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

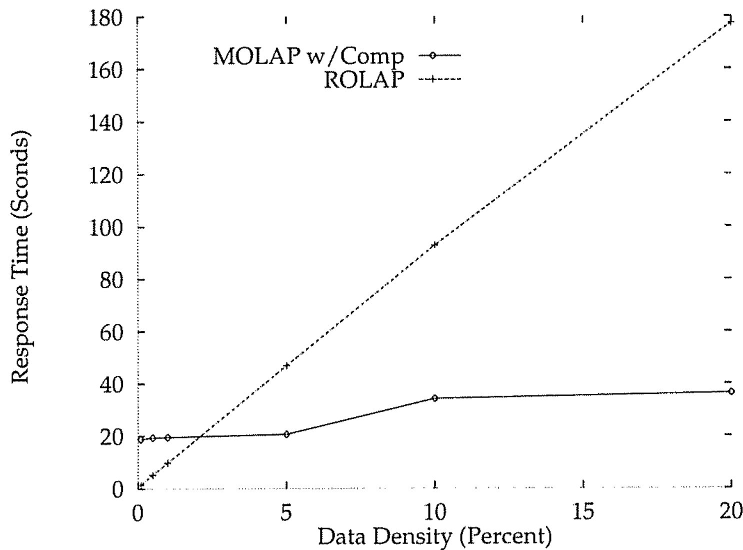

Figure 9: Performance of Query 3 for Large Database Configuration

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. As we have shown, the MOLAP ADT is efficient for data densities down to 1%; however, below that level the ROLAP tables appear to be more efficient. We would like to develop a better storage structure for very sparse OLAP data. 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.

Acknowledgements

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

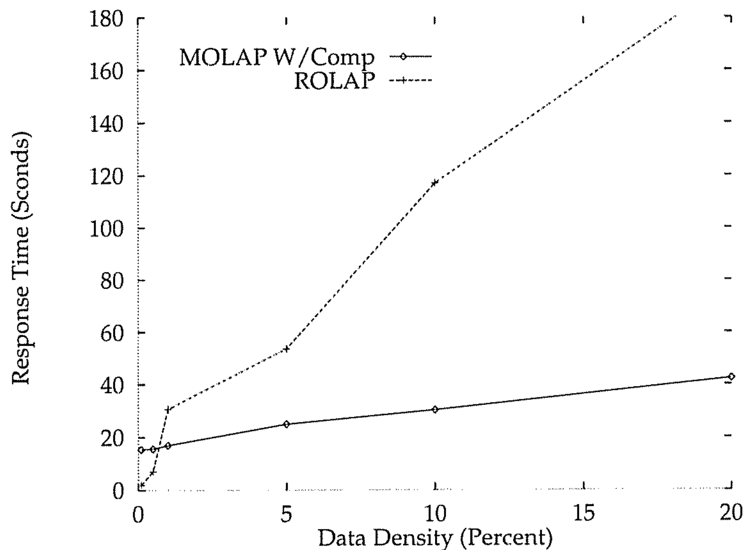

Figure 10: Performance of Query 4 for Large Database Configuration

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**"

[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**"

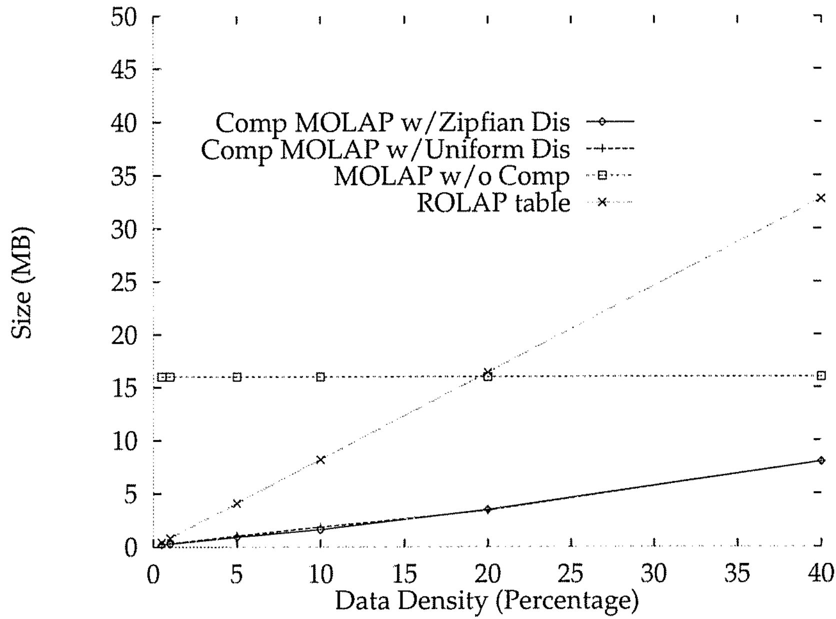

Figure 11: MOLAP ADT vs. ROLAP Fact Table - Storage Efficiency

[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.