# Array-Based Evaluation of Multi-Dimensional Queries in Object-Relational Database Systems

**Yihong Zhao**
Computer Sciences Department
University of Wisconsin-Madison
zhao@cs.wisc.edu

**Karthikeyan Ramasamy**
Computer Sciences Department
University of Wisconsin-Madison
karthik@cs.wisc.edu

**Kristin Tufte**
Computer Sciences Department
University of Wisconsin-Madison
kristint@cs.wisc.edu

**Jeffrey F. Naughton**
Computer Sciences Department
University of Wisconsin-Madison
naughton@cs.wisc.edu

June 2, 1997

**Abstract**

Since multi-dimensional arrays are a natural data structure for supporting multi-dimensional queries, and object-relational database systems support multi-dimensional array ADTs, it is natural to ask if a multi-dimensional array-based ADT can be used to improve O/R DBMS performance on multi-dimensional queries. As an initial step toward answering this question, we have implemented a multi-dimensional array in the Paradise Object-Relational DBMS. In this paper we describe the implementation of this compressed-array ADT, and explore its performance for queries including star-join consolidations and selections. We show that in many cases the array ADT can provide significantly higher performance than can be obtained by applying techniques such as bitmap indices and star-join algorithms to relational tables.

## 1 Introduction

Multi-dimensional data analysis has been around for at least twenty years [OR95], but has recently taken the spotlight in the context of OLAP (On-Line Analytical Processing) systems. A fundamental demand of OLAP systems is extremely fast response times for multi-dimensional queries. The relational database community is investigating and implementing a variety of techniques intended to meet the stringent OLAP response time requirements, most notably special techniques for "star join" consolidation queries and bit-map indices for select-join queries over star or snowflake schemas. These techniques can indeed greatly improve relational performance for multi-dimensional queries when compared with more traditional alternatives such as pipelined left-deep hash joins for consolidation queries [Su96] or B-tree indexes for star select-joins [OQ97].

However, the recent emergence of object-relational database systems [Ki95, St96] has opened the door to another potential approach to solving the OLAP query performance problem: since object-relational database systems can be extended to support the storage and manipulation of multi-dimensional arrays, why not use these arrays as a storage structure for multi-dimensional data sets, and use specialized processing algorithms on these arrays to answer multi-dimensional queries? To begin to answer this question, we have implemented multi-dimensional arrays

in the Paradise [DKLPY94] object-relational database system, and tested their effectiveness for multi-dimensional query workloads. In previous work [ZDN97], we investigated the performance of multi-dimensional arrays for the specialized OLAP "compute the cube" function as well as bulk loading the array ADT from a relational table. In this paper we discuss the design of the array ADT and several algorithms for the previously mentioned star select-joins and consolidation queries. To enable a performance comparison to its relational database counterpart, we have also implemented bit-map indices and specialized star-join consolidation operators in Paradise.

Of course, the idea of using multi-dimensional arrays as a storage structure for multi-dimensional data analysis is far from new — many of the leading OLAP companies (including Arbor Software, Pilot Software, and others) in fact use arrays in their multi-dimensional (non-relational) database systems as their basic storage mechanism. However, since these systems are proprietary, and their vendors view many of their processing techniques as trade secrets, very little implementation or performance information is available. The Arbor Software's MOLAP storage structure has been patented and made public, but the processing algorithms used to implement operations such as consolidation on top of this structure have not. Some object relational database vendors are in the process of implementing an array ADT in their database engine, but to our knowledge none of them have published information about implementation details or performance implications when using such an array ADT as a storage structure for OLAP databases.

The results from our study indicate that multi-dimensional arrays in object-relational systems can indeed improve performance significantly over bit-map indexing and star-join consolidation algorithms. The main reasons for this are that (1) with proper compression, the same data set is much smaller when stored in a multi-dimensional array than in a relational table (even when the relational table is stored in an extremely dense format customized for fixed-length records), and (2) with arrays, lookups and aggregations are position-based rather than value-based.

The rest of the paper is structured as follows: Section 2 presents an OLAP data model. Section 3 describes the design and implementation of the OLAP Array ADT, while Section 4 describes the new query evaluation algorithms used by this ADT, and the corresponding relational algorithms to which they are compared. Section 5 gives the results from our performance study, and Section 6 presents our conclusions and directions for future work.

## 2 A Basic OLAP Data Model

In order to make this paper self-contained, we outline some basic characteristics of OLAP data sets relevant to the queries and algorithms we will be discussing here. Consider a database containing sales data describing the volumes of products sold in a given store on a given date. The attributes *product*, *store* and *time* functionally determine the *sales* volume. These attributes are referred to as *dimensions*, while the *sales* volume is referred to as a *measure*. Intuitively, this data is best visualized as a three dimensional array with *product*, *store* and *time* being the dimensions of the array. In addition, each dimension has its own attributes that describe its members. For example, the *product* dimension might contain for each product the *product name*, the product *type*, and the *category* to which it belongs. Returning to the measure, each individual cell in the array contains the sales volume for a particular product at a particular store on a particular date. Even though the array logically has cells for all possible combinations of values of dimensions, some of the cells might not be valid. For example, a store in Madison, Wisconsin is unlikely to sell beach clothing in January.

Typically, dimensions have associated with them hierarchies. These hierarchies represent the level of aggregation and hence successive refinement of data being viewed. For example, the store dimension might have a hierarchy *store name* $\rightarrow$ *city* $\rightarrow$ *region*. Similarly, *product type* $\rightarrow$ *category* could be a hierarchy on the product dimension.

Now we generalize the preceding example. Let us assume that the data contains $n$ dimensions. Let $C$ be the hypercube with $n$ dimensions. Let $M = \{m_1, ..., m_p\}$ be the set of $p$ measures stored in individual cell in the hypercube. Each of $m_i$ can draw values from the domain $dom(m_1)$, ... , $dom(m_p)$. Each of the dimensions is referred to as $D_1$, ... ,$D_n$. Each dimension $D_i$, contains the $k_i$ attributes $D_i(A_{i1})$, ... , $D_i(A_{ik_i})$ that describe the

dimension. Each $D_i$ contains a key attribute to index to into the cube. For simplicity, we will assume $D_i(A_{i1})$ is the key attribute for each dimension $D_i$ $(1 \leq i \leq n)$. Each dimension attribute $D_i(A_{ij})$ takes values from $dom(D_i(A_{ij}))$ $(1 \leq i \leq n, 1 \leq j \leq k_i)$. Now each $n$-tuple $(a_1, a_2, ..., a_n)$ uniquely identifies the cell at $C(a_1, ..., a_n)$, where each $a_i$ is a element of the domain $dom(D_i(A_{i1}))$.

With this minor formalism in place we can describe the classes of queries we are considering.

## 2.1 Consolidation Queries

Consolidation is one of the most important of OLAP operations. 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 generalized consolidation query can be formally written as

**SELECT** $P, \quad F_1(m_1), \cdots, \quad F_p(m_p)$
**FROM** $C(D_1(A_{11}), \cdots, D_n(A_{n1}))$
**WHERE** $\phi(D_1)$ **AND** $\cdots$ **AND** $\phi(D_n)$
**GROUP BY** $G$

Perhaps the strangest-looking part of this query to SQL hackers is the FROM clause term $C(D_1(A_{11}), \cdots, C(D_n(A_{n1}))$, which we use to indicate that the "source" of the data for the query is the cube indexed by the dimension tables. As with "real" SQL this does not imply any particular implementation. Also, we have that $\phi(D_i)$ is the predicate $D_i(A_{ij}) = v_{ij}$, where $v_{ij} \in dom(D_i(A_{ij}))$ $(1 \leq i \leq n, 1 \leq j \leq k_i)$. Also, we have that $G \subseteq \{D_i(A_{ij}) \mid 1 \leq i \leq n, 1 \leq j \leq k_i\}$, and $P \subseteq G$. $F_i$ is the aggregation function used for measure $m_i$.

## 2.2 Mapping the Data Model to a Relational Schema

In this section, we show how to map the OLAP data model to a ROLAP schema. The standard way of representing such a data set in the relational model is to use a star schema [Ki95] (or its slightly more complex variant, the snowflake schema). Each of the dimension tables $D_i$ $(1 \leq i \leq n)$ with $k_i$ attributes maps onto a relational table $T_{D_i}$ with $k_i$ attributes

$$D_i(A_{i1}, A_{i2}, ..., A_{ik_i}) \rightarrow T_{D_i}(A_{i1}, A_{i2}, ..., A_{ik_i})$$

such that

$$dom(D_i(A_{ij})) = dom(T_{D_i}(A_{ij})), \ (1 \leq j \leq k_i)$$

. The hypercube $C$ is mapped into a table $F_C$ with $n + p$ attributes

$$C \rightarrow F_C(A_{11}, A_{21}, ..., A_{n1}, m_1, m_2, ..., m_p)$$

such that

$$dom(F_C(A_{i1})) = dom(D_i(A_{i1})) \quad 1 \leq i \leq n$$

and

$$dom(F_C(m_i)) = dom(m_i) \quad 1 \leq i \leq p$$

with $n$ attributes from the foreign keys of $n$ dimensions and $p$ attributes from the measures. This table is often called the "fact" table. The relational star schema for our running 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. Also notice that each dimension does form a hierarchy. The dimension tables maintain information about the hierarchy. 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.

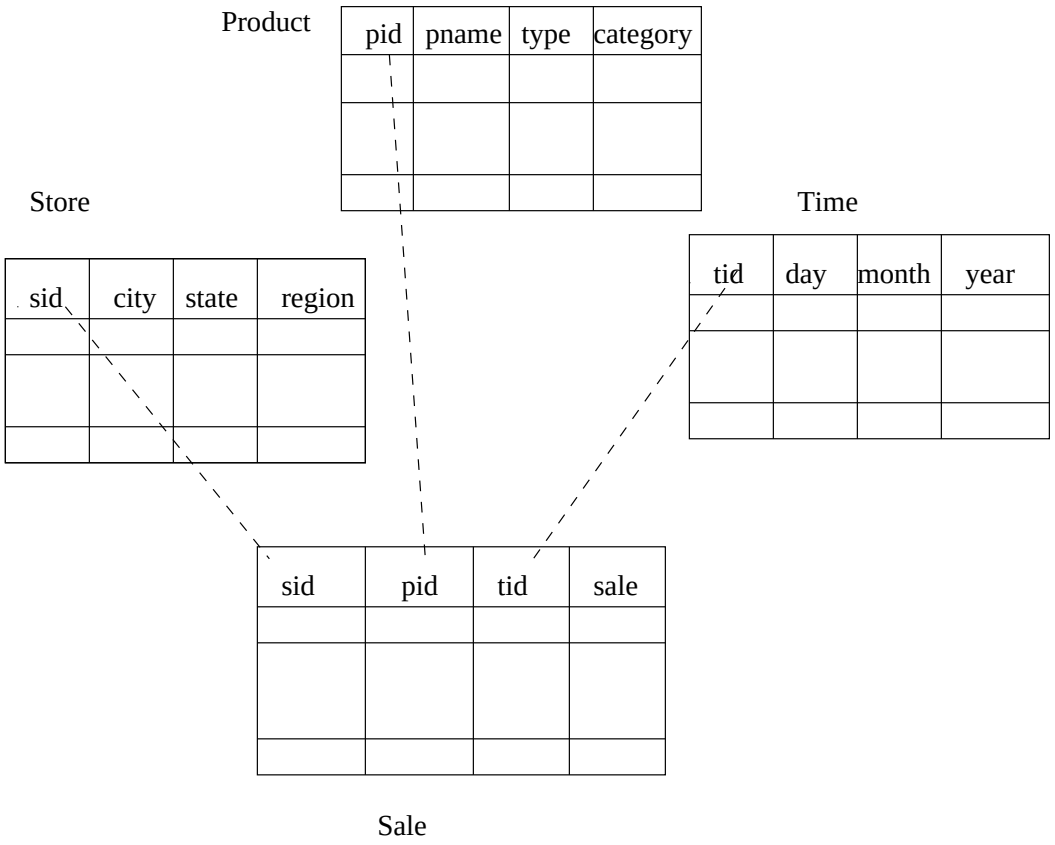


Figure 1: Relational Star Schema

In terms of query processing, an OLAP consolidation query over a star schema 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 often called a star join in OLAP terminology.

## 2.3 Mapping the Data Model into an O/R DBMS Array

In an O/R DBMS, we have an alternative for implementing our abstract data model. We still map each dimension $D$ into a dimension table, but we store the hypercube $C$ in a multi-dimensional array instead of a fact table. This array is implemented as an ADT, and the mapping between the key value of a dimension table and the index value of the array is maintained by the instance of the ADT. The ADT also keeps track of the mapping between the dimension table and the array dimension. A consolidation query is evaluated by executing a function on the ADT. The implementation details of the ADT are discussed in the next section.

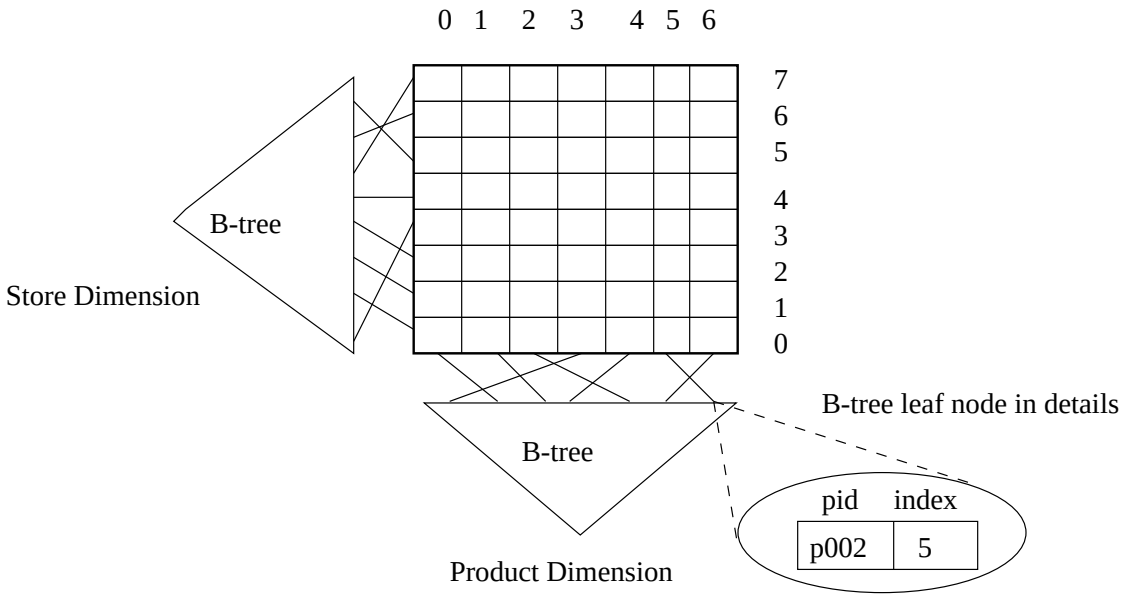


Figure 2: 2-D OLAP ADT

# 3 An ADT for Multi-Dimensional Data Sets

In this section, we describe the design and implementation of an array-based OLAP ADT that integrates a multi-dimensional OLAP storage structure into the Paradise database system. Paradise has an object-relational data model and query language that supports relational tables, ADTs, and functions on ADTs. The OLAP Array ADT uses a multi-dimensional array as its primary storage structure and has functions defined on it to perform common OLAP operations such as consolidation and aggregation.

## 3.1 Implementation

The OLAP Array ADT contains an $n$-dimensional array and a set of B-tree Indices, one for each dimension. The $n$-dimensional array stores the same data that was stored in the "fact" table in the relational implementation of the data model. For example, if this 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. (Unfortunately here and elsewhere in this paper we run into overloading of the word "index" — in this case "index" refers to a value used as a subscript for an array access.)

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 OLAP Array ADT. This B-tree index maps dimension identifiers, e.g. *pid*, to array index values. Notice that a vector $(a_1, a_2, .., a_n)$ of dimension values uniquely identifies a cell in the array in the ADT. Figure 2 shows a small 2-D OLAP Array ADT object in detail.

The OLAP Array ADT is built on top of the Paradise multi-dimensional array type. The Paradise multi-dimensional 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 multi-dimensional array also implements compression on a tile by tile basis using the LZW algorithm [Wel84] to further improve performance. As we discuss in the next subsection, the OLAP Array ADT does not use LZW compression, and uses instead a compression method that is specific to arrays.

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

## 3.2 Storage Efficiency

Multi-Dimensional arrays are clearly efficient for storing dense data. For example, consider a 100% dense $n$-dimensional array. To store this in a relational table, each tuple would contain a foreign key for each dimension $D_i$ and $p$ attributes for the set of measures $M$. On the other hand, in an array only the corresponding $p$ measures are stored, since the cell in the array is identified by the key attributes $D_i(A_{i1})$ for $1 \leq i \leq n$. If $T_s$ and $A_s$ are the storage requirements for a table and array, then $T_s = \frac{n+p}{p} \times \rho A_s$ where $\rho$ stands for data density. This means that the relational star schema requires about $\frac{n+p}{p}$ times the amount of storage that a multi-dimensional array would need, when data density is 100%.

However, this storage efficiency does not come for free — on sparse data, the array might require far more storage. When $\rho < \frac{p}{n+p}$, the table storage requires less space than the array. 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 $\rho$ is 25% or greater. However, with a density of less than 25%, the OLAP Array ADT's array potentially requires much more disk space than the relational "fact" table because the OLAP Array 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 only if the cell has valid data.

Fortunately, for sparse data sets, another opportunity arises: sparse arrays are extremely compressible. As we show in the next subsection, using our "offset compression" algorithm, we found that a compressed array takes less disk space than a relational fact table for surprisingly low densities. In the next subsection, we discuss the chunk-offset compression in details.

## 3.3 Chunk-offset compression

In our OLAP Array ADT we use a form of compression that to our knowledge is new, and which we call "chunk-offset compression." In chunk-offset compression, we omit those invalid data cells in each chunk and store a pair, (offsetInChunk,data), for each valid array cell. The offsetInChunk integer can be computed as follows: consider the chunk as a normal (uncompressed) array. Each cell $c$ in the chunk is defined by a set of indices; for example, if we are working with a three-dimensional square chunk of $c$ units in each dimension, a given cell will have an "address" $(i, j, k)$ in the chunk. To access this cell in memory, we would convert the triple $(i, j, k)$ into an offset $s = ((i \times c) + j) \times c) + k$ from the start of the chunk. This offset is the "offsetInChunk" integer we store.

We also sort the valid array cells of each chunk in a increasing order of array cells' chunk offsets. For a given set of array index values $(a_1, \cdots, a_n)$ , we can calculate the chunk number and the chunk offset and use a binary search to find whether there is valid array cell in the position $(a_1, \cdots, a_n)$. This feature can be used when indexing the multi-dimensional array, which is explored in the Array-based algorithm for consolidation with selection.

Since in this representation chunks will be of variable length, we use some meta data to hold the OID and the length of each chunk and store the meta data at the beginning of the data file. Given a chunk number, the meta data is used as index structure to fetch the corresponding array chunk.

### 3.4 IndexToIndex Arrays

The OLAP Array ADT, in addition to the array corresponding to the relational fact file, includes a set of arrays, one per dimension table, called the IndexToIndex arrays. These arrays map from the array index corresponding to one level of a dimensional hierarchy to the array index corresponding to another. For example, if we have a dimension table geography(city, state), we would have an IndexToIndex array mapping from the city array index in the OLAP Array ADT's array to the state array index in a (virtual) consolidated version of the OLAP Array ADT. If Madison is the 10,344th distinct element of the City column of the geography table, and Wisconsin is the 47th distinct element of the State column, then IndexToIndex array for the geography dimension would contain the pair (10344,47) in the 10,344'th slot. This is really the array equivalent of the hierarchy information in the dimension table.

Returning to our formalism, this can be expressed as follows. For each dimension $D_i$, where $1 \leq i \leq n$, there is an indexToIndex array $I_i$. If $D_i$ has $k_i$ attributes, the array $I_i$ can be declared of size $I_i[countDistinct(A_{i1}), k_i]$. Consider some row $m$ of this array. This row will be an $q$-tuple of the form $(m, c_2, \ldots, c_q)$ $(q = k_i)$, with the interpretation that the $m$th distinct element of attribute $A_{i1}$ maps to the $c_2$th distinct element of attribute $A_{i2}$, and to the $c_3$th element of attribute $A_{i3}$, and so forth.

### 3.5 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 take advantage of this functionality to define several functions on the OLAP Array ADT to provide OLAP functionality, which, though very limited, is enough for our experiments.

The functions on the OLAP Array ADT include a Read/Write function to retrieve and update the array data in the OLAP Array ADT, a function to compute the sum of a subset of the array, a consolidation function, and a slicing function. The result of the Read/Write and summation aggregation functions are instances of the OLAP Array 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 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 OLAP Array ADT and have compared the performance of this algorithm against a relational consolidation algorithm. This section describes our new algorithm in detail and gives brief descriptions of the relational algorithm which we used for comparison.

### 4.1 An OLAP Array Consolidation Algorithm

Recall that a basic consolidation looks something 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 consolidation operation consists of a star join followed by a "group by" and an aggregation. In the context of the OLAP Array ADT, a consolidation operation is a "join" of the OLAP Array 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 star join and the aggregation into a single operation. This is faster than a star join 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 OLAP Array ADT is another instance of the OLAP Array ADT. We call these instances the input OLAP Array object and the result OLAP Array object, respectively. In addition, the array associated with the input OLAP Array object is called the input array. The result array is defined similarly.

This algorithm has two phases: the first phase scans the input OLAP Array object's dimension tables, B-trees, and IndexToIndex arrays for each dimension. The B-trees for the result OLAP Array 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.

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 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 OLAP Array object. 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 OLAP 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 OLAP Array ADT object.

After building the 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 3 shows the data structures for the OLAP Array ADT consolidation algorithm.

Pseudo-code for the OLAP Array consolidation algorithm is below.

**OLAP Array Consolidation Algorithm**

```
For each joined dimension
{
    create result B-tree;
    load the IndexToIndex array;
}
scan the input array
For each array cell
{
       look up result indices using the IndexToIndex arrays;
       //Star Join
```

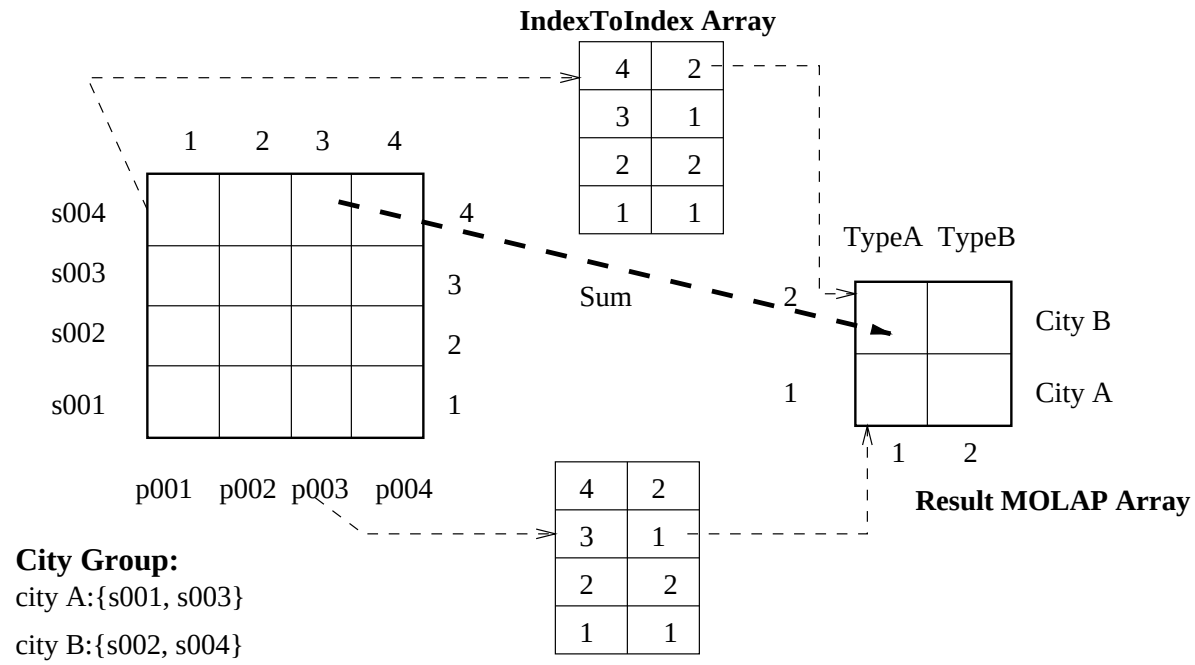


Figure 3: OLAP Array Consolidation Algorithm Data Structures

```
        find the corresponding result 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 OLAP 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 OLAP Array 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 OLAP Array object, and in general it will be much smaller. The result OLAP Array object approaches the input OLAP Array 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 OLAP object "chunk by chunk", where each chunk fits in memory. While such an extension would be straightforward we did not implement it, as it was not necessary in our experiments.

## 4.2 OLAP Consolidation algorithm with Selection

To handle the consolidation queries with selection we designed another algorithm. The algorithm consists of two main steps. First, it searches the B-tree built on the selected attribute on each dimension table for the "selected" value. It returns a list of index values for the dimension. As we mentioned before, each tuple in a dimension table has a key value and its key value corresponds to a unique index value of the OLAP Array object's $n$-dimensional array.

The list of index values associated with a given "selected" value serves as a join index for the "selected" value. After retrieving the index lists for all "selected" values on each dimension table, the algorithm merges those index lists into a final list for each dimension table. Second, it generates a cross-product of the final lists from each dimension on the fly. No memory is allocated to the cross-product elements. For each generated cross-product element, the algorithm calculates the element's chunk number and the offset in the chunk from its index values. It uses the chunk number to retrieve the chunk and uses the offset to probe the chunk. If the cross-product element is a valid element, the algorithm aggregates the corresponding array cell value to the result.

Pseudo-code for the OLAP Array algorithm is below.

**OLAP Array Consolidation Algorithm with Selection**

```
For each join dimension table
{
    Use the B-tree to retrieve the index list
    for the selected values;
    Merge those index lists to generate the final list;
}
Generate the cross-product of the final lists;
For each cross-product element
{
     calculate the chunk number and chunk offset;
     probe the chunk;
     if (cross-product element is valid)
        aggregate the array cell to the results;
}
```

In our implementation of the algorithm, we adopted a few optimizations to enhance its performance. First, we generate the cross-product elements according to the chunk number. In other words, we generate the cross-product elements contained in Chunk 0 first, then those in the Chunk 1, and so on. If we calculate that a chunk does not overlap with any cross-product elements, we do not read the chunk into memory. If the array chunks are laid out on the disk in the same order as their chunk number order, we read chunks in the same order as the chunks' physical order. Second, we sort each array chunk's elements according to their offset values. This means that given a cross-product element's chunk offset, we can use a standard binary search to see whether the element is valid in the chunk. Third, we generate the cross-product elements in an increasing order of their chunk offsets. We use this order to optimize the search.

## 4.3 A Relational Consolidation Algorithm

To compare the array-based consolidation algorithm with the table-based relational consolidation algorithm, we implemented within Paradise a Starjoin algorithm. To see why this was necessary, we first repeat a common assumption of star schemas: the dimension tables are dwarved by the fact table, so much so that the assumption is that the dimension tables fit in memory while the fact table does not. Furthermore, the dimension tables share no join attributes; that is, they join only through the fact table, not with each other.

If we try to do a star join on such a set of tables using only left-deep hash-based query plans, we are in trouble. At issue is where in the plan to put the (large) fact table. It must be one of the lowest leaves of the tree, or else the plan will contain a cross product. If it is the left leaf, then we are faced with building a hash table on the (huge)

fact table. If it is the right leaf, we only delay this by one stage; after a join with the first dimension table, we must build a hash table on the large result.

One alternative is to do a cross product of the dimension tables and join this with the fact table. We explored this possibility, but even with small dimension tables this cross product quickly grew in size to the point where this alternative was unworkable. Another alternative is to modify the system to accept right-deep trees. Our star join algorithm approximates such an approach, albeit at lower overhead (since it is a single operator rather than a pipeline.)

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.

### 4.4 Selections on Star Joins: Bitmap Index and Fact File

The previous section considered only consolidations without selections. While such consolidations are important (e.g., when computing an aggregate table), many times queries involve selection conditions on the dimension tables.

We implemented and tested several algorithms for these selections, including standard B-tree indexing, a specialized "skipping multi- attribute B-tree" algorithm, and bitmap indexing. Here we present only bitmap indexing, since our tests showed that it dominated the other techniques over the full range of queries tested in this paper. For details of these tests please see [RQZN].

Bitmap Indexes are becoming widely used in relational systems [SybaseIQ, Redbrick, Informix] to speed up the evaluation of consolidation queries with selection [OQ97]. We have implemented bitmap indices in Paradise. We have also implemented a specialized file structure optimized for tables with small, fixed-size records. We call this structure a "fact file" for obvious reasons. Our main reason for implementing this file is to do everything possible to ensure that the relational table is as fast as possible (since our conclusions indicate that the OLAP Array is faster.)

This fact file improves performance in two ways: (1) it provides a fast path for retrieving tuples corresponding to "1"s in a bitmap, and (2) it eliminates the space overhead associated the with slotted page structure used in most relational database systems. Note that we also used this "fact file" structure for the consolidation queries to take advantage of this second benefit.

In more detail, in OLAP applications, the fact table tuples are fixed length. We can use this feature to eliminate the overhead incurred by using slotted pages. In addition, given a tuple position relative to the first tuple i.e. the tuple number, we can find the corresponding page and offset within the page. One possible method is to allocate all fact table pages contiguously. Given a tuple number, we can calculate the page number and the offset within the page containing the tuple. However, the disadvantage with such direct mapping is that it is not often possible to allocate large set of pages contiguously for large fact tables.

To address this problem, the fact file allocates $n$ pages in groups called extents. Within each extent, all the pages are contiguous. It uses an internal tree structure to keep the pointers to the first page of each extent. Given a tuple number, we can compute the extent, the page within the extent, and the offset within the page for the tuple. The fact file provides a interface that takes a bitmap and retrieves the tuples corresponding to non-zero bit positions. For more details on the design, implementation and performance of bitmap indices and fact file refer to [RQZN].

Note that it was possible to implement a fact file complete with concurrency control, recovery, and SQL query

processing in a reasonable amount of time because the SHORE storage manager used by Paradise is C++ from the bottom up. This means that by exposing the proper public interface all these database functions were available to the fact file with essentially no additional programming effort.

### 4.5 Relational Algorithm for Consolidation Queries with Selection

The basic idea of the algorithm is to retrieve the bitmaps satisfying the predicate on the dimension tables and AND them to get the result bitmap. Using the result bitmap, we fetch the tuples and aggregate. Initially, we create a join bitmap index on each selected attribute for each dimension table. (This bitmap creation is done ahead of time, not as part of the query evaluation.) The relational algorithm for consolidation with selection on $n$ dimensions is as follows

```
Set all bits of ResultBitmap to ones;
foreach selected dimension
{
    retrieve the bitmaps for the selected values;
    AND ResultBitmap with the bitmaps;
}
retrieve the tuples for ResultBitmap;
aggregate the tuples' measure to the results;
```

The ResultBitmap is used to open up a fact file scan on the fact file. For each "1" in the final bitmap, it gets the tuple from the fact file and aggregates the tuple to the results.

## 5 Performance

We have used six data sets to compare the performance of the OLAP Array ADT versus the relational tables. 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 relational star 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 and h**X**2 attributes are hierarchically structured, similar to the dimensions in the retail sales example.

## 5.2 OLAP Queries

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

```
select sum(volume), dim0.h01, dim1.h11, dim2.h21, dim3.h31
from   fact, dim0, dim1, dim2, dim3
where  fact.d0 = dim0.d0 and fact.d1 = dim1.d1 and
       fact.d2 = dim2.d2 and fact.d3 = dim3.d3
group by h01, h11, h21, h31
```

**Query 2** is Query 1 with selection on four dimensions.

```
select sum(volume), dim0.h01, dim1.h11, dim2.h21, dim3.h31
from   fact, dim0, dim1, dim2, dim3
where  fact.d0 = dim0.d0 and fact.d1 = dim1.d1 and
       fact.d2 = dim2.d2 and fact.d3 = dim3.d3 and
       dim1.d01 = "AA1" and dim1.d11 = "AA1" and
       dim2.d21 = "AA1" and dim3.d31 = "AA1"
group by h01, h11, h21, h31
```

**Query 3** is the same consolidation query with selection on three dimensions.

```
select sum(volume), dim0.h01, dim1.h11, dim2.h21
from   fact, dim0, dim1, dim2
where  fact.d0 = dim0.d0 and fact.d1 = dim1.d1 and fact.d2 = dim2.d2 and
       dim1.d01 = "AA1" and dim1.d11 = "AA1" and dim2.d21 = "AA1"
group by h01, h11, h21
```

**Query 4** is a "slicing" OLAP query.

```
select volume, fact.d0, fact.d1, fact.d2, dim3.h31
from fact, dim3
where fact.d3 = dim3.d3 and dim3.h31 = "AA1"
```

## 5.3 System Configuration

These tests were run on an 200MHZ Intel Pentium Pro processor with 64MB 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 16MB 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.

## 5.4 Data Sets

We used synthetic data sets to study the algorithms' performance. There are a number of factors that affect the performance of an OLAP algorithm. These include:

- Number of valid data entries.

  That is, what fraction of the cells in a multi-dimensional space actually contain valid data? Note that the number of valid data entries is just the number of tuples in a relational table implementing the multi-dimensional data set.

- Dimension size.

  That is, how many elements are there in each dimension? Note that for an OLAP Array ADT implementation, the dimension size determines the size of the array. For a relational table implementation, the table size remains constant as we vary dimension size, but the range from which the values in the dimension attributes are drawn changes.

We used the following two types of data sets and scaling in our tests.

- Data Set 1: Keep the number of valid data elements constant, vary the dimension sizes. The data set consists of three 4-dimensional arrays. For these arrays, three of the four dimensions sizes are fixed at 40, while the fourth dimension is either 50 (for the first array), or 100 (for the second), or 1000 (for the third). Every array has 640000 valid elements. This results in the data density of the arrays (fraction of valid cells) ranging from 20%, to 10%, to 1%.

- Data Set 2: Keep the dimension sizes fixed, vary the number of valid data elements.

  All members of this data set are logically 4-dimensional arrays, with size 40x40x40x100. We varied the number of valid data elements so that the array data density ranges from 0.5% to 20%.

We generated uniform data for both data sets. We used 20x20x20x10 array chunks for the arrays in our performance tests.

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

## 5.5 Performance Results

### 5.5.1 Consolidation Query

The results of Query 1 for the OLAP Array and the relational consolidation algorithms on Data Set 1 and Data Set 2 are shown in Figure 4 and Figure 5.

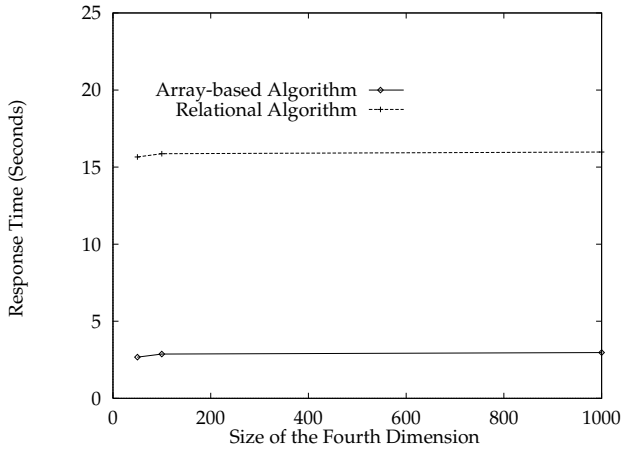


Figure 4: Query 1 - Data Set 1

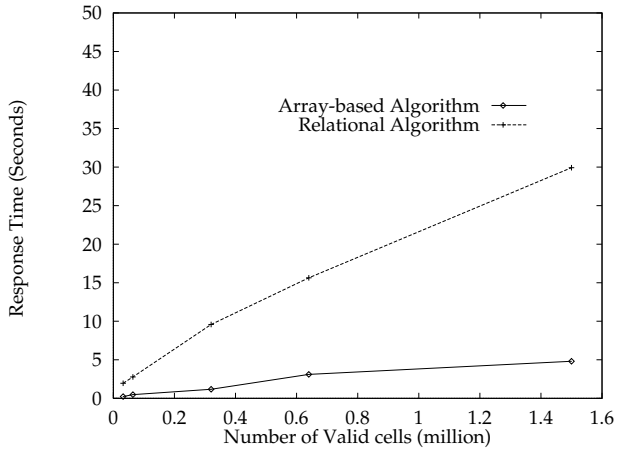


Figure 5: Query 1 - Data Set 2

For both data sets, the OLAP Array consolidation outperforms the relational algorithm by a wide margin. There are two factors contributing the performance difference. First, the array, compressed by the offset compression

algorithm, is smaller than the Fact File. For this reason scanning the fact file is much expensive than scanning the compressed array. (It takes about 10 seconds to scan the Fact File for the Data Set 1 while the entire array-based algorithm takes less than 3.5 seconds.) For example, at 1% data density with Data Set 1, the size of the relational fact file is about 18.5 MBytes vs. 6.5 MBytes of the compressed OLAP array.

Second, the relational algorithm uses a value-based aggregation algorithm, which costs more CPU time than the OLAP Array's position-based aggregation. For Data Set 1, the relational algorithm spends 3 seconds on aggregation while the array algorithm takes less than 1 second to aggregate the results.

To see why, consider again the following consolidation query, Query 1:

```
select sum(volume), dim0.h01, dim1.h11, dim2.h21, dim3.h31
from   fact, dim0, dim1, dim2, dim3
where  fact.d0 = dim0.d0 and fact.d1 = dim1.d1 and
       fact.d2 = dim2.d2 and fact.d3 = dim3.d3
group by h01, h11, h21, h31
```

Any algorithm to compute this consolidation must map all dimension attributes (d0, d1, d2, and d3) in the fact table to their parents in their respective hierarchies (h01, h11, h21, and h31.) In the array implementation, this mapping is done using simple index lookups in the IndexToIndex arrays. For each value in the OLAP 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.

The relational 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 relational 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 array algorithm.

## 5.6 Performance Result for Query 2, Query 3 and Query 4

The results of Query 2 for the Array-based and the Relational consolidation algorithms on the 40x40x40x100 and 40x40x40x1000 arrays are shown in Figure 6 and Figure 7.

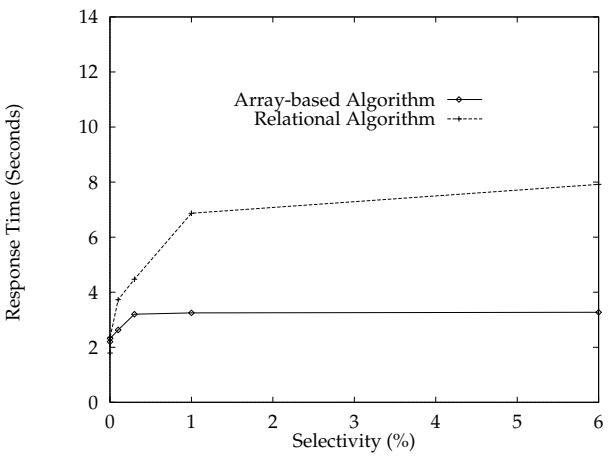


Figure 6: Query 2 on the 40x40x40x1000 array

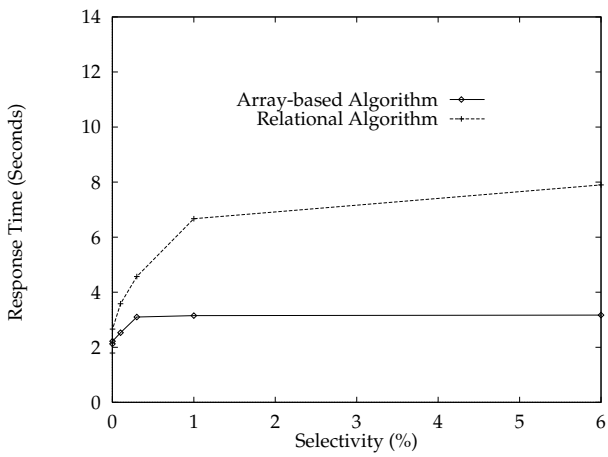


Figure 7: Query 2 on the 40x40x40x100 array

For Query 2, we vary the number of distinct values for the second attribute of each dimension table from 2, 3, 4, 5, 8, to 10. The selectivity $S$ for the star join is the $s^r$, where $s$ is the selectivity on each dimension table and $r$ is the number of joined dimension tables. When $s$ value ranged from 1/2, 1/3, 1/4, 1/5, 1/8, to 1/10, the $S$ changes from 0.0625, 0.0123, 0.00391, 0.0016, 0.00024, 0.0001 for a four dimension star-join query.

From Figure 6 and 8, we see that the crosspoint of the array alogrithm and the relational algorithm is at 0.00024 selectivity for the 40x40x40x1000 array. The relational algorithm is faster than the array algorithm when the selectivity is less than 0.00024. There are two reasons to explain this. First, the relational algorithm has to spend an extra 0.5312 seconds to retrieve the selected bitmaps and to 'AND" them. Second, for the 0.00024 selectivity, the relational algorithm retrieves only 160 tuples while the array algorithm has to read 800 chunks. However, the difference between the two execution times is less than the ration of 800 to 160 would suggest. This is because the reading of the 800 array chunks consists entirely of sequential reads, since the array consists of 800 chunks and the array reading algorithm reads those chunks in physical order. On the other hand, reading 160 relational tuples from a file of 640000 uniformly distributed tuples is equivalent to randomly reading the same number of data pages from the fact file. For this data set the sizes of the chunks and pages are about the same — 6.4K for the compressed array chunk, 8 KBytes data page used in the fact file.

For the array 40x40x40x100, Figures 7 and 9 show that the crosspoint of the two algorithms is once again at the 0.00024 selectivity. Notice the 40x40x40x100 has only 80 chunks, but each its compressed chunk is 10 times larger than 40x40x40x1000 array's compressed chunk after using the offset-compression algorithm since it 10 times denser. So, for Query 2 and Query 3, two arrays have almost identical performance for the array algorithm.

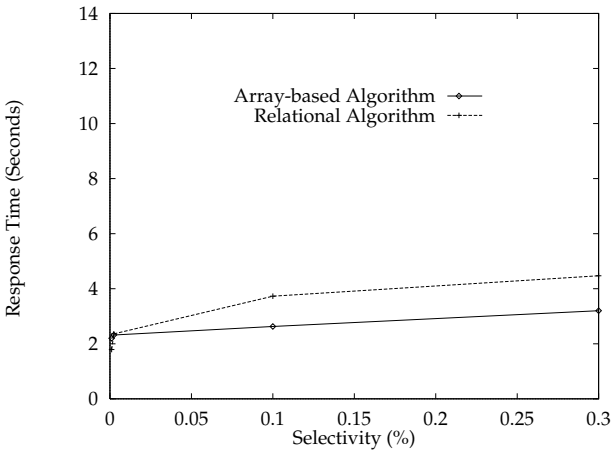


Figure 8: Query 2 on the 40x40x40x1000 array

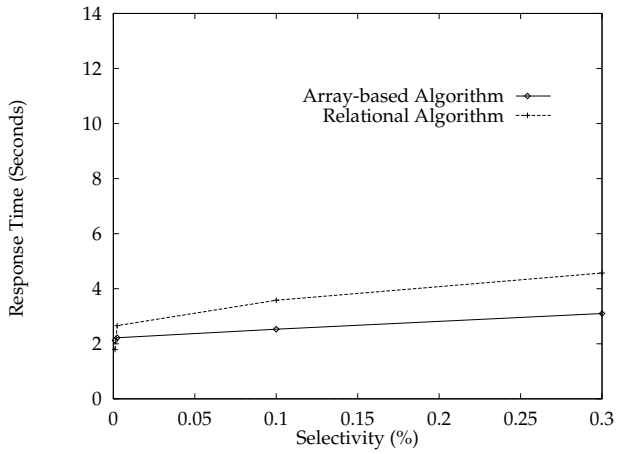


Figure 9: Query 2 on the 40x40x40x100 array

For Query 3, we measured the time to retrieve the bitmaps and to "AND" them. If we compare the results for this query to those of Query 2, the cost of reading one extra bitmap and one extra bitmap "ANDing" is less than 0.1 second. But, from Figure 7 and 10, we see that Query 3 is more than 0.1 seconds slower than Query 2 since Query 3's result bitmap has more bits set to "1"s. This means that Query 3 has to pull out more result tuples than Query 2 does.

Before we look into the performance results of Query 4, lets us see how the array and relational algorithms work for this "slice" query. The tests were run on the 40x40x40x1000 array. Recall that there are 100 distinct values in the second attribute $h31$ of the fourth dimension table. The result of the slice query consists of roughly 6400 "cells" (in the array format) or tuples (in the relational format.)

For the relational algorithm, we built a bitmap index on attribute $h31$. At query time, the query evaluation algorithm first retrieves the bitmap for the value "AA1" and then uses this bitmap to pull out the result tuples. For the array algorithm, we built a B-tree index on the attribute $h31$ of the fourth dimension table. At query time, the

array algorithm probes this B-tree to get the array index list for the value "AA1" and uses the list to retrieve the corresponding chunks for the slice.

The array algorithm took 1.6 seconds to finish the query while the relational algorithm took 4.6 seconds. The reason for this is that the array algorithm reads only 80 out of 800 chunks to retrieve the require 6400 cells, while the relational algorithm reads about 6400 tuples scattered randomly about the table. Another interesting observation is that, for the array algorithm, the "slicing" query takes less time than the Query 2 at 0.0001 selectivity. There are two reasons to explain this. First, the array "slicing" algorithms probes only one b-tree while the array consolidation algorithm with selection needs to probe four b-trees. Second, the "slicing query" reads only 80 chunks, but Query 2 has to read 800 chunks.

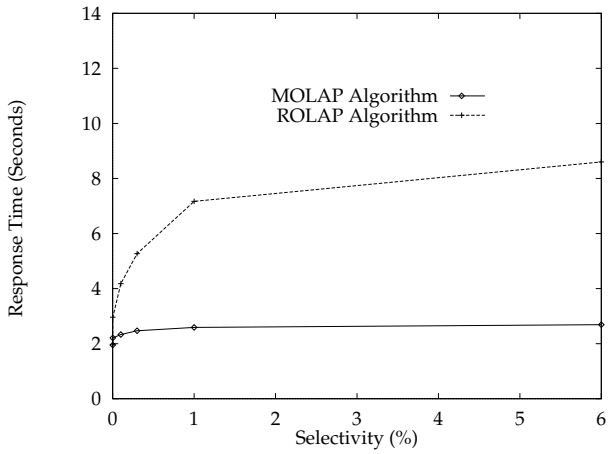


Figure 10: Query 3 on the 40x40x40x100 array

# 6 Conclusion

In this paper, we have used an implementation of an OLAP Array Abstract Data Type to show that multi-dimensional arrays can be more efficient both in terms of storage space and performance than relational tables for multi-dimensional OLAP data sets. This performance comparison was done in the context of the Paradise object-relational database system. We conclude that for a wide variety of queries multi-dimensional arrays are surprisingly efficient, when compared with relational tables, for storing and operating on relatively sparse OLAP data sets. The exception we observed was that by using bitmap indices over a highly optimized file structure, the more standard relational algorithm outpeformed the OLAP Array algorithm for selectivities less than 0.00024. These results suggest that the support for arrays in object-relational database technology can be used to improve system performance for OLAP-style multi-dimensional workloads.

Clearly, there are many issues that need to be resolved before the array ADT can be used to build real OLAP server. Currently we are focusing on three of these issues, including the query language interface to the array ADT, efficient support for updates to the array ADT, and steps toward parallelizing the array ADT.

# 7 Acknowledgements

We would like to thank Navin Kabra, Jignesh Patel, and Jiebing Yu for their help with the implementation of the ROLAP consolidation algorithms. Nancy Hall gave helpful suggestions in implementing the fact file and bitmap indices. Also, Prasad Deshpande and Amit Shukla reviewed the paper and provided us good suggestions.

## 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 Applications," 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**"

[OR95] Nigel Pendse and Richard Creeth *The OLAP Report - Succeeding with On-Line Analytical Processing*, Business Intelligence, 1995.

[OQ97] Patrick O'Neil and Dallan Quass. "Improved Query Performance with Variant Indexes." Proc. of the 1997 SIGMOD Conference, May, 1997.

[Su96] Prakash Sundaresan. "Data Warehousing Features in Informix OnLine XPS." Presentation at the Fourth International PDIS Conference, December 18-20, 1996, Miami Beach, Florida.

[Ki95] Ralph Kimball. "Data Warehousing Toolkit." John Wiley & Sons, 1995.

[RQZN] Karthikeyan Ramasamy, Qi Jin, Yihong Zhao, and Jeffrey F. Naughton. "Bit-Map Indices: Implementation Issues and Performance Results." Working paper.

[Informix] Informix Corporation. *INFORMIX_Online Extented Parallel Server: A New Generation of Decision Support Indexing.* White Paper, Informix Corporation.

[Redbrick] Red Brick Systems. *Star Schemas and STARjoin Technology.* White Paper, Red Brick Systems. "**http://www.redbrick.com/rbs/whpapers.html**"

[SybaseIQ] Sybase Inc. *Sybase IQ – Optimizing Interactive Performance for the Data Warehouse.* White Paper, Sybase Inc. "**http://www.sybase.com/products/dataware/iqwpaper.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**"

[St96] Michael Stonebraker. "Object-Relational Database Systems – The Next Wave." Morgan Kaufmann Publishers, 1996.

[SS94] Sunita Sarawagi, Michael Stonebraker, "Efficient Organization of Large Multi-Dimensional 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.

[Ki95] Won Kim. "Modern Database Systems: The Object Model, Interoperability, and Beyond." ACM Press and Addison-Wesley 1995

[ZDN97] Yihong Zhao, Prasad M Deshpande, Jeffrey F. Naughton. "An Array-Based Algorithm for Simultaneous Multi-Dimensional Aggregates." Proc. of the 1997 SIGMOD Conference, May, 1997.