Storage and Querying of Large Persistent Arrays

Arun C.S.

A Thesis Submitted to

Indian Institute of Technology Hyderabad

In Partial Fulfillment of the Requirements for

The Degree of Master of Technology

भारतीय प्रौधोगिकी संस्थान हैदराबाद
Indian Institute of Technology
Hyderabad

Department of Computer Science and Engineering

July 2011

Declaration

I declare that this written submission represents my ideas in my own words, and where ideas or words of others have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources that have thus not been properly cited, or from whom proper permission has not been taken when needed.
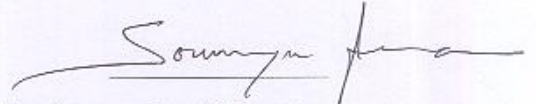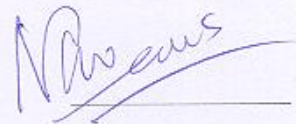
(Signature)
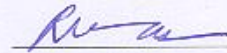
(Arun C.S.)

CS04 G001

(Roll No.)

Approval Sheet

This Thesis entitled Storage and Querying of Large Persistent Arrays by Arun C.S. is approved for the degree of Master of Technology from IIT Hyderabad

(Dr. Soumya Jana) Examiner
Dept. of Electrical Engineering
IITH

(Dr. Naveen Sivadasan) Examiner
Computer Science and Engineerng
IITH

(Dr. Ravindra Guravannavar) Adviser
Dept. of Computer Science and Engineerng
IITH

(Dr. Vinod Janardhanan) Chairman
Dept. of Chemical Engineering
IITH

Acknowledgements

Apart from my own efforts, the success of my project has largely depended on the encouragement and advice of many others. I would like to take this opportunity to express my gratitude to the faculty members who have been instrumental in the successful completion of this project.

I would like to show my greatest appreciation to Dr. Ravindra Guravannavar. I cant say "thank you" enough for his tremendous support and help. I felt motivated and encouraged every time I met him. Without his encouragement and guidance this project would not have materialized.

The guidance and support received from all the faculty members who contributed and who are contributing to this project, was vital for the success of the project. I am grateful for their constant support and help.

I would like to thank our Director Prof. U.B. Desai for his friendly administrative support in getting all our requirements done as quickly as possible.

Finally, I thank all my friends for their helping hand.

Dedication

To My Beloved Parents

Abstract

The scientific and analytical applications today are increasingly becoming data intensive. Many such applications deal with data that is multidimensional in nature. Traditionally, relational database systems have been used by many data intensive application, and relational paradigm has proved to be both natural and efficient. However, for multidimensional data, when the number of dimensions becomes large, relational databases are inefficient both in terms of storage and query response time. In this thesis, we explore linearised storage, and indexed and skiplist based retrieval on persistent arrays. The application programs are provided with a logical view of multidimensional array. The techniques have been implemented in a home-grown database management system called MuBase.

Contents

List of Figures

Chapter 1

Introduction

Most of today's database management systems use the relational model [3]. Relational model is a set based mathematical model. Relational model considers data as n-ary relation. An n-ary relation being a subset of cartesian product of n domains. In relational model, relations are represented as table. A table comprises of multiple columns and rows. Rows are called tuples and columns are called attributes. A relational database comprises of one or more relations(tables). Consider a bank database that consists of three relations: *Customer*, *Account* and *Depositor*. Database schema is used to describe structure of database. It is described in formal language. Schema of these relations are given below (Schema-1).

Schema-1: Bank database schema

```
Customer(customer_id, customer_name, age, sex)
Account(account_id, account_type, branch_id, balance)
Depositor(customer_id, account_id)
Branch(branch_id, branch_name)
```

The Customer relation contains four attributes: `customer_id`, `customer_name`, `age` and `sex`. The Account relation has four attributes: `account_id`, `account_type`, `branch_id and amount`. The Depositor relation shows relation between Customer table and Account table.

Operations on the relational model are described by *relational algebra*. Relational algebra defines a set of operators on relations. Five basic operations are present in the relational algebra: *selection*(σ), *projection*(Π), *union*($\cup$), *difference*($-$) and

Figure 1.1: Example: Relations in bank database

cartesian product($\times$). All other operations needed to manipulate data in the relational model can be expressed with these five basic operations. Some of these operations are given below.

1. Show āccount_id and amount of all customers

    $$\Pi_{account\_id,amount}(Account)$$

2. Find āccount_id of all customers those who have amount greater than 1000

    $$\Pi_{account\_id}(\sigma_{amount>1000}(Account))$$

3. Find āll account_id which is having saving-bank facility only

    $$\Pi_{account\_id}(\sigma_{account\_type='saving'}(Account)) - \Pi_{account\_id}(\sigma_{account\_type='current'}(Account))$$

4. Find dētails of all those customers who have saving-bank account

    $$\Pi_{customer\_id,customer\_name,account\_id,amount}(\sigma_{\substack{Customer.customer\_id=Depositor.customer\_id \\ \land Account.account\_id=Depositor.account\_id}}$$

3

$$_{\wedge Account.account\_type='saving'}(Customer \times Account \times Depositor))$$

Most of the commercial relational systems use a language called Standard Query Language(SQL). IBM introduced SQL in 1970s. Later in 1986, American National Standard Institute(ANSI) accepted it as standard. After that they revised it many times(SQL-89, SQL-92, SQL-99, SQL-2003, SQL2006, SQL-2008). Some of the commands use in SQL are SELECT, UPDATE, INSERT, DELETE. SQL equivalent of the above queries are given below. Even though SQL is not equivalent to relational algebra, it is powerful to express all query in commercial database system.

SQL query:

1. SELECT account_id, amount
   FROM Account

2. SELECT account_id
   FROM Account
   WHERE $amount > 1000$

3. SELECT account_id
   FROM Account
   WHERE $account\_type =' saving'$
   EXCEPT
   SELECT account_id
   FROM Account
   WHERE $account\_type =' current'$

4. SELECT customer_id, customer_name, account_id, amount
   FROM Customer, Account, Depositor
   WHERE Customer.customer_id=Depositor.customer_id
   AND Account.account_id=Depositor.account_id
   AND Account.account_type='saving'

Data used in scientific applications such as astronomy, oceanography is usually multidimensional. Let's consider data obtained from the Sloan Digital Sky Survey(SDSS) project [4][5]. It is a celestial scan project to obtain multi-colour images (2 dimensional data) of sky. SDSS uses a special telescope of 2.5m length. This 120 mega-pixel camera scans 1.5 square degree of sky at a time. In SDSS's 8 years of operation, it covered more than one quarter of the sky. Total data size up to seventh major data release is around 40TB: 15TB of images and 26TB of other data products, catalog, masks, JPEG images, etc. This data is available for scientific research and education purpose. Scientists make 3D model from those 2D images for detailed analysis of the sky. As a second example, consider an analytical application that keeps sale of all products over the year in all departments. Its schema is `sale_table(Dept_Name, Prod_Name, Day_of_Year, Sale)`(Figure 1.2). Most frequent query on this data is "*find sales of a product in a department over the year 2010*", i.e. aggregation of sales of a product in a department over the year 2010.
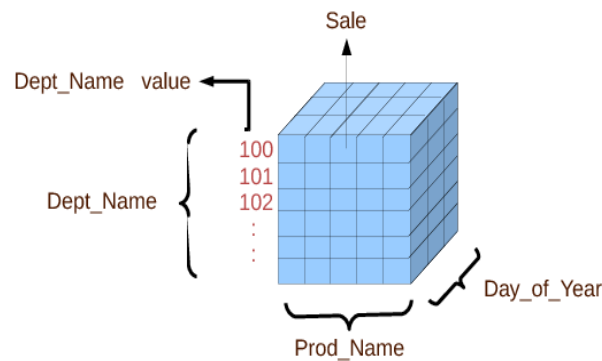

Figure 1.2: Multidimensional array

This thesis presents various approaches for storage and querying of large multidimensional array data.

Chapter 2

Database Support for Multidimensional Arrays

Scientific and analytical applications, such as the ones mentioned in the previous chapter, require arrays to be supported as first-class objects in the database. Although relational database systems do not provide direct support for multidimensional arrays, several approaches can be used to store multidimensional arrays in relational database management system(RDBMS). Some of these approaches are explained below.

2.1 Binary Large Object(BLOB)[1]

BLOBs were invented at DEC by Jim Starkey[1]. BLOB is a collection of unstructured binary data. The database system treats it as a single entity or object. That cannot be decomposed into relational schema. It is used to store multidimensional arrays. In BLOB, array contents are considered as stream of bytes. Storage manager does not have any idea about the individual elements; for them its only byte stream. Meaning of these elements is up-to application. BLOB data type consists of BLOB locator and BLOB value. Oracle 11g supports BLOB data type of size upto 128TB[6]. oracle 11g support two types of BLOB: internal BLOB and external BLOB.

Internal BLOB: Internal BLOB stores data within the database, either in-line in the table or in a separate table. If the data size smaller than 4KB the BLOB is stored in-line. Once it grows bigger, it automatically moves out of table. Internal BLOB is supported by BLOB data type in Oracle 11g[7].

External BLOB: External BLOB stores data outside the database, in operating system files. It is supported by BFILE data type [7].

Advantage of this method is that there is no limit on the size of array.
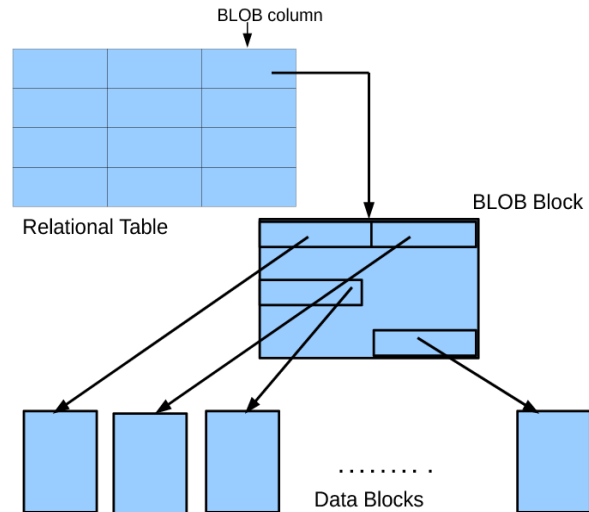
Figure 2.1: BLOB representation of multidimensional array

In BLOB, data can be accessed as entire BLOB-data only, i.e. for accessing of a measurement in the array, database has to read entire BLOB-block. This increases accessing time. This impaired the advantage of array ADT. End user can not use BLOB locator in a SELECT or WHERE clause of the SQL query.

2.2 Array as a relation

Array also can be implemented on a relation by considering each dimension as a column in a relational table. Then we store each measurement with its corresponding dimension values on table explicitly. For storing an array of n dimension require a relational table with n+1 columns. The following figure depict how an array stored in relational table.

In relational representation of array, arrays are represented by storing both array-index and measurement (array-value) together in relational table. This increases the storage space needed for storing. In order to access data, the relational database has to read 'array-index' columns in the table. This slows down data access. If arrays are considered as first-class object then no need of storing array-index, only measurement is stored in database. All measurements in an array are of same data type. So storage space needed can be reduced by encryption.

7

X	Y	Z	Value
100	a	1	100
100	b	2	150
.	c	.	.
.	.	.	.
101	a	1	175
101	c	2	180
.	.	.	.
102	c	2	200
.	.	.	.
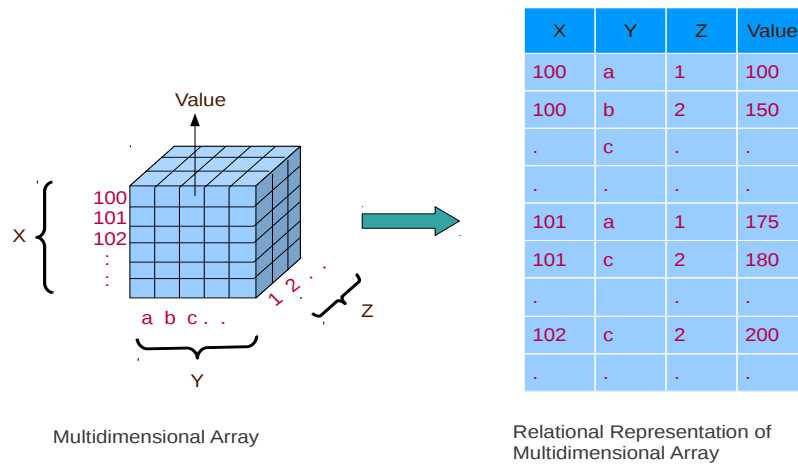
Figure 2.2: Relational representation of multidimensional array

2.3 Arrays in object-relational database systems

Object relational database Oracle-11g support array. Keyword VARRAY uses to create array data type. VARRAY is an ordered collection of elements. VARRAY is normally stored inline, i.e. in the same tablespace. If array is too large, orcle stores it in BLOB.

Array implementation in object relational database (ORDB) is against normalisation. Updation of an array element is very inefficient in ORDB. For example, commercial database system such as Oracle do not support piecewise updates on VARRAY columns. VARRAY columns can be inserted into or updated as an atomic unit. That is, if we want to update or delete individual element in the array, then we have to take whole element from table, change it, and update the table to include new array. This leads to inefficiency in array operations.

2.4 Native array support

One another way to represent multidimensional arrays in database is, treat array as it is, i.e., consider array as first-class object. The database systems, RasDaMan and RAM (discussed in Chapter 4) consider array as first class object. RasDaMan uses specialised storage for multidimensional array. RAM embedded array object to the existing relational model.

8

Chapter 3

Operations on Multidimensional Arrays

Let's consider another example from oceanographic research.The sensors placed under the sea are generating time series data, i.e. 3-dimensional data. Data from OLAP applications also have large dimensionality. For example sales details of products in departments have the schema: (Dept_Name, Prod_Name, Day_of_Year, Sale). One of the most common operations performed on the above schema is the entire calculation of sales over years or a particular year. For the better efficiency of this operation, keep the data in a multidimensional array.

A workshop had been conducted on the year 2008 in Asilamor, for finding out the common requirement of scientific applications. In this workshop, people from different areas, scientific(Astronomy, Biology, Particle Physics, Geoscientific), database, industrial, were present. They put forward requirements[8] from their field. Some of the array operations identified in this workshop are given below. These operation are mainly classified into 3 categories: Structural operations [9], Content-based operations[9] and Meta-data operations[9].

3.1 Structural operations

- Indexed Retrieval

- Sub-sample

- Structural Join or SJoin [8, 10]

- Aggregation on dimensionality

All the above operations perform on the dimension values of the database. They are data-agnostic [11], i.e., it does not affect the array element. Each structural operations are described below.

3.1.1 Indexed retrieval

Indexed retrieval(with array-index) is the basic operation, which performs on an array database. In array database, the array name and the array index are given to retrieve the information. Operation *array(A, i,j)* retrieve element from array A whose index is i and j. Here i is the higher order dimension and j is the lower order dimension.

3.1.2 Subsampling

A large number of queries used in SDSS applications are for retrieving an image in a certain area, that is sub-sampling of an image. Sub-sampling can not reduce array dimensionality, i.e dimensionality of resultant array is same as that of input array. Consider the query: array_image[x:2-6][y:1-5]. Here the value of x is ranging from 2 to 6 and the value of y is ranging from 1 to 5. In traditional database, need to sort the table primarily on (x, y) column then fetch the required columns. In an array, the array indices are in sequential order so that the database can fetch the required slice very easily.
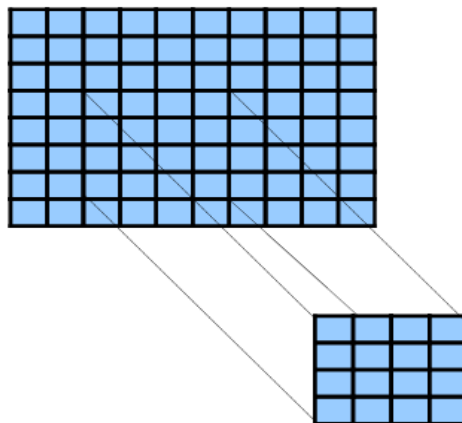

Figure 3.1: Subsampling

3.1.3 Structural Join or SJoin

Rakesh Agrawal, Ashish Gupta and Sunita Sarawagi introduced the structural join for those kind of operations in their research paper, "Modelling Multidimensional Databases"[10]. Later, Michael Stonebraker et. al. [8] defined a join operator: SJoin, that joins arrays over their dimensional values. Let's consider two arrays: array A of m dimensional and array B of n dimensional. $SJoin(A, B, d_1, d_2, ..., d_k)$ joins array A and B over the dimensions $d_1, d_2, ..., d_k$. The resultant array is of $(m + n - k)$ dimensions and values of each cells is either value of one of the arrays or output of an expression (arithmetic, logical, conditional or combination of these). The following example illustrates the concept of this operation. Let us consider three 2D images of a sea: one in x-y axis, one in y-z axis and third one in x-z axis (figure.3.2). Intensity of each pixel in images in x-y axis, y-z axis and x-z axis are stored in arrays A, B and C respectively. For detailed study of sea, the scientists wants to make it as 3D image. They can combine images in the 3 axes by using SJoin operator. The query for making 3D image is given below (Query-1). First SJoin joins array A and B and maintains both intensity value. This result is stored to an temporary array T. Then this temporary array SJoin with array C. 'IN' condition given in the SJoin operator gives the output if any of the right hand side value is equal to the left hand side value. That is the resultant array R contains intensity values of array C which are equivalent to any of the intensity values in the corresponding dimensions of T.

Query-1: Query to make 3D image from three 2D image.
$$T = SJoin(A, B, (A.intensity, B.intensity))$$
$$R = SJoin(T, R, R.intensity\ IN\ T.intensity$$

3.1.4 Aggregation

Aggregation is one of the fundamental operations in OLAP applications. Let us consider a query on array given in Figure 1.2: *Find total sales of a product in a department over the year 2010.* This is one of the aggregate operations over array measurements. sum(array_sales[Dept_Name:department1] [Prod_Name:product1] [Day_of_Year:1-1-2010,31-12-2010]). Here department is *department1*, product is *product1* and value of Day_of_Year ranging from 1-1-2010 to 31-12-2010. To do this efficiently in traditional database, the table to be sorted on (Dept_Name, Prod_Name)
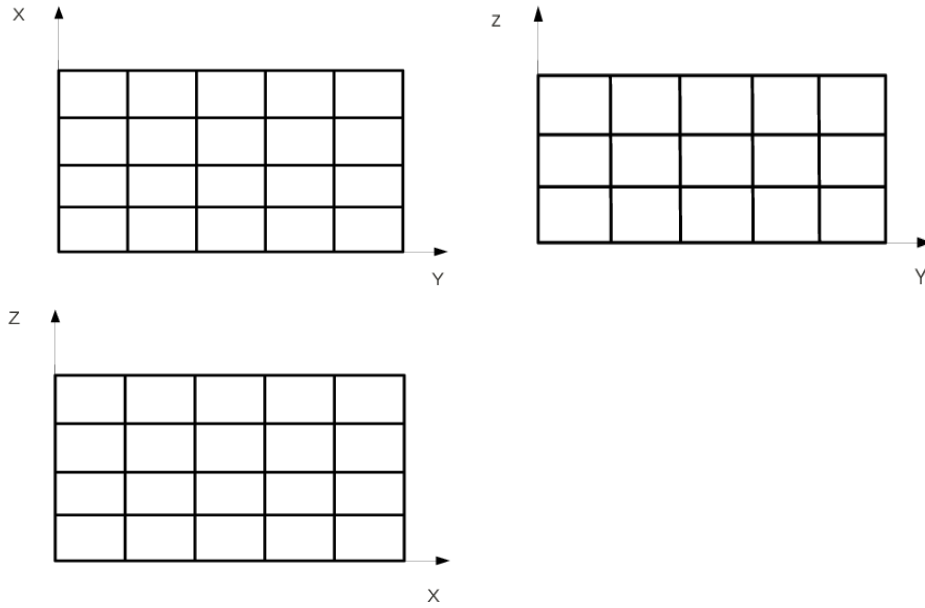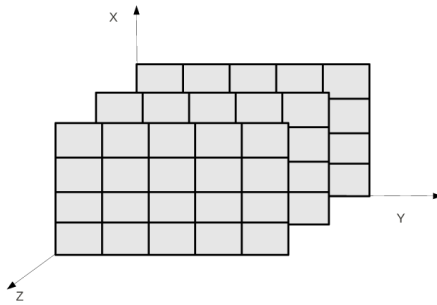
Figure 3.2: Images on 3 axes

Figure 3.3: Structural join on arrays A, B and C

columns. But if we maintain the same data in an array with a hierarchy of $Dept\_Name \rightarrow Prod\_name \rightarrow Day\_of\_Year \rightarrow Sale$ then the aggregation can be done fast.

3.2 Content-based operations

- Filter(constraints on element value)

- Content-based Join or CJoin [11]

3.2.1 Filter

Filter is an operation that separates a part of data from array for analysis. This operation is equivalent to $selection(\sigma)$ in relational model. For example $filter(A, >, 10)$ operation outputs an array which is of same size of array A. The resultant array contains elements whose value is greater than 10.

3.2.2 Content-based join

CJoin is used for joining two arrays based on their measurement values. This operation is primarily introduced in the thesis "Requirements for Science Data Bases and SciDB", CIDR 2009 Conference [8]. CJoin of an m dimensional and an n dimensional array gives an array of m+n dimensions.
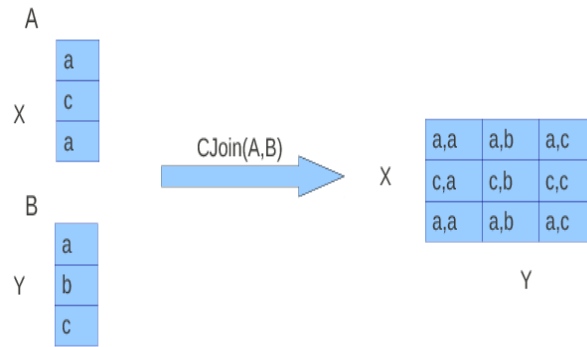
Figure 3.4: Cjoin

3.3 Meta-data operations

- add-dim/rem-dim (Add/Remove Dimensions)

- Reshape

3.3.1 Add/remove dimension

These operations are performing on the dimensional space of the array. The operation add-dim adds one or more new dimension(s) to an array. The operation $add - dim(A, x[1 : 10], y[1 : 20], high/lower)$ adds 2 more dimensions to array A. Here x and y having dimensional space of 1 to 10 and 1 to 20 respectively. Argument high/lower indicates hierarchy of dimensions to be added. High adds dimensions in

the higher and lower adds in the lower hierarchy. The operation rem-dim reduces the dimensionality of the array. $rem - dim(A, p = 1, q = 25)$ operation reduces the number of dimensions of array A by removing the dimensions of p and q other than 1 and 25 respectively, i.e. the resultant array contains value from the slice which has value $p = 1$ and $q = 25$. This array can be accessed without specifying p and q.

3.3.2 Reshape

The operation reshape changes the number of dimension(s) of an array without changes number of elements in the array. Let's consider an array C of 3-dimensions,($4 \times 5 \times 2$). Reshape(C,[i:20],[j:2]) operation change the array C to a 2-dimensions. i and j are it's dimensions. Dimensional value of i varies from 1 to 20, and of j are 1 and 2. Total number of cells, before and after reshape operation, are same.

Chapter 4

Known Approaches for Native Array Support

4.1   RasDamMan(Raster Data Management)

RasDaMan[2] is a research project sponsored by European Community to develop multidimensional database. RasDaMan enables storage of multi-dimensional raster ("array") data of unlimited size in a standard database for retrieval through its declarative, optimizing query language[12]. The RasDaMan array engine can be coupled with many different database systems and offers highly effective hardware and software optimizations[12]. This RasDaMan system is implemented in several projects. The EarthServer project is one of these project. The EarthServer (European Scalable Earth Science Service Environment) project aims at open access and ad-hoc analytics on massive Earth Science data, based on the OGC geo service standards Web Coverage Service (WCS) and Web Coverage Processing Service (WCPS)[12]. RasDaMan separates logical and physical schemas.

They presented RasQL(Raster Query Language) to interface with database. RasQL comprises of RasDL(Raster Definition Language) and RasML(Raster Manipulation Language). RasQL supports multidimensional operations like slicing, updation, aggregations, etc. The operation set is based on RasDaMan Array Algebra[13] which allows for declarative expression of operations up to the complexity of the Discrete Fourier Transform[2]. Arrays are represented in Array Algebra as functions mapping n-dimensional points (i.e., vectors) from discrete Euclidean space to values [13]. RasDaMan allows set based operations over array. These operations have to be second order which apply in cell wise manner. Array expressions are embedded into standard

SQL-92 in the array query language RasQL. Essentially, the algebra in RasDaMan consists of three operations: [13].

- Trimming (rectangular cutout) and section (extraction of a lower-dimensional hyperplane)

- Induced operations which apply cell operations simultaneously to the whole array

- generalized array aggregation

The RasDaMan uses a client/server architecture 4.1. Its API consists of array-extended SQL-92, RasQL, ODMG conformant C++ API. The client side sends queries to the server through communication layer. Server side contains four main modules: server communication layer, query evaluator, metadata manager and storage manager. The server side passes query from the client to query evaluator. Then the query evaluator parses the query and builds parse tree. Query optimisation and tiling (discussed later) takes place in next stage. Storage manager contains information about physical storage.
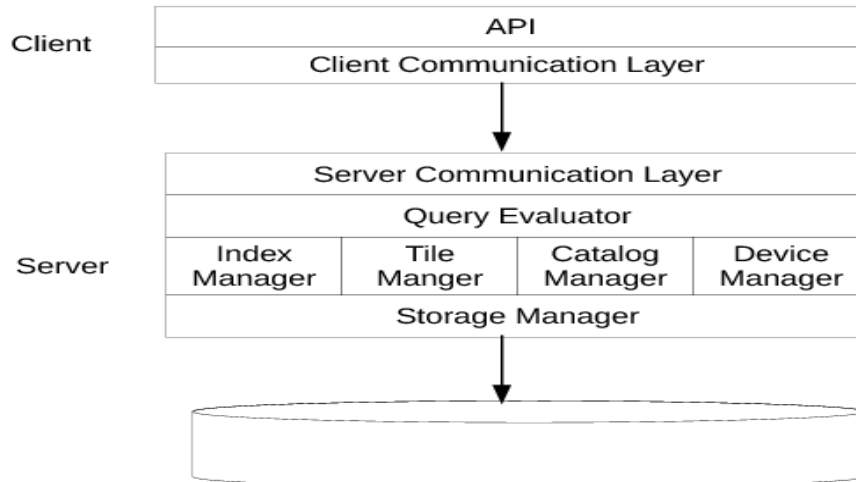
Figure 4.1: Architecture of RasDaMan[2]

Storage manager in RasDaMan supports efficient paging methods for accessing data. RasDaMan uses three different storage strategies: linear subdivision, aligned tiling, arbitrary subdivision. Pictorial representation of these strategies are given in Figure 4.2. Large outer blocks represent array and small shaded blocks represent portion of array which is to be accessed by application. Let's assume required

page size is equivalent to one disk page. Each cell in the large blocks are subdivision of array which are stored in the disk block. These storage strategies use different subdivision method.

In linear subdivision, arrays are divided into small linear blocks. Even though the required portion of array is equivalent to one disk block, database has to access six pages from disk to get the required portion. Aligned tiling divides arrays into small tiles, each of these tiles is stored as linearised array. Size of all these tiles are same and it is equivalent to one disk page. To read the required portion, database has to access four disk pages. Aligned tiling gives better performance than linearised tiling. Third strategy, arbitrary rectangular tiles, divides the multidimensional arrays into arbitrary multidimensional tiles. We can divide an array depending on frequent queries. Arbitrary subdivision increases the locality reference, so we get better performance in sub-sampling queries. This reduces the number of blocks needed to access the data. In the above example, only three disk pages is needed to satisfy requirement. First two approaches are special case of arbitrary subdivision, i.e., database can implement linear subdivision and aligned tiling with arbitrary subdivision. Query performance can be optimized by arbitrary subdivision. But it requires information about user querying.
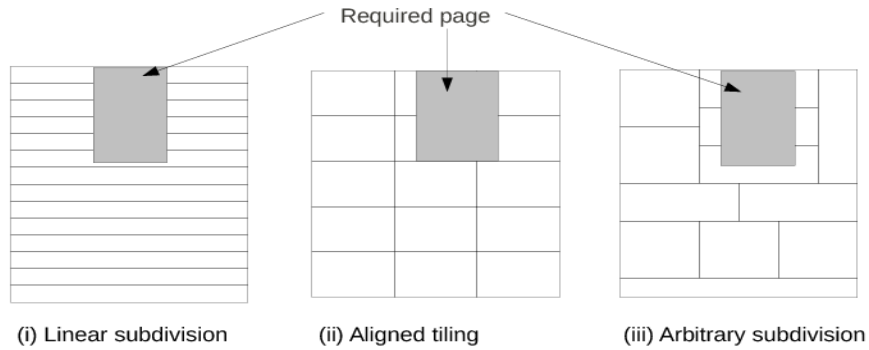

Figure 4.2: Different storage strategies in RasDaMan

Efficient server based query evaluation is enabled by an intelligent optimizer and a streamlined storage architecture based on flexible array tiling and compression [2].

4.2 RAM

The set based data model may no longer suffice for tasks like multimedia analysis. Alex R.van Ballegooij has introduced a prototype system as part of his research: RAM, a Multidimensional Array DBMS[14]. The main issue to be addressed in the RAM is the actual storage and manipulation of array structures in a relational database environment. The RAM adds array support to an existing database system. It uses separate front-end for array specific queries. This front-end translates array specific queries to an intermediate array-algebra before transforming to final relational domain, i.e., it maps arrays to relational model. RAM compresses the multiple index columns into single column by enumerating the array index into row major order (Figure 4.3).
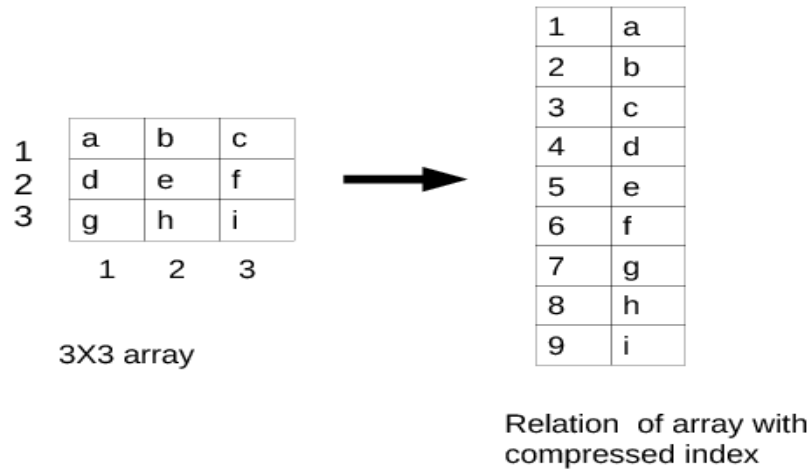
Figure 4.3: Representation of array in RAM

In RAM, arrays are defined as a many to one function over the array index[14]. RAM embedded two more component to the existing database to support array operations: methods to extract values from arrays and methods to construct arrays. There is no update option in query language; i.e., once created, we can not alter that array. The basic array operations implemented in RAM are given below:

- *const(S, c)*[14]: The const operator creates a new array of a given shape S filled with a constant value c. $const([3,4], o)$ creates an array of dimensions 3×4 and initialise array with zero.

- *grid(S, j)*[14]: The grid operator creates a new array of a given shape S filled with values taken from its index values at j^{th} position. `grid([2,2], 1)` creates

an array of 2×2 and initialize with array index of each row, i.e., resultant array is:

Table 4.1: Resultant array of operation $grid([2, 2], 1)$

	1	**2**
1	0	0
2	1	1

- *Aligned Array*[14]: Aligned arrays are arrays with identical shape representing related data: in these arrays elements with corresponding index-vectors are related. Using aligned arrays, multiple arrays can be used to represent a single array with tuple-elements.

- *map(f, $A_1, ..., A_k$)*[14]: The map operator creates a new array of which each element is the result of applying a given function to aligned elements in a set of arrays. For example $map(+, A, B)$ gives an array of which each element is the sum of corresponding elements in array A and B.

- *choice(C, A, B)*[14]: The choice operator creates a new array of which each element is $choice(C, A, B) = [if\ (C_i)\ then\ A_i\ else\ B_i\ |\ i < S_C]$

Table 4.2: Array C

0	1
0	1

Table 4.3: Array A

a	b
c	d

Table 4.4: Array B

e	f
g	h

Table 4.5: Resultant array

e	b
g	d

- *aggregate(g, j,A)*[14]: The aggregate operator applies an aggregation function over the first j axes of an array.

  `aggregate(f, j, a)` $= \left[f([A_i] | i_0, i_1, \ldots\ldots, i_{j-1}) \,\middle|\, i_j, i_{j+1}, \ldots\ldots, i_{n-1} \right]$

The RAM database is under development. The complete set of operations are unavailable. So we only considered the storage structure of the RAM. The above operations are mentioned here to give a brief idea about some of the operations possible over the array. Relational mapping of array index in RAM reduces the storage space needed as compared to the relational representation in Section 2.2. Other benefit of RAM approach is the time saved by not recreating functionality readily available. RAM reused benefits of existing query optimizers, transaction control and integrity. These areas in relational model are very efficient. In RAM, arrays are still mapped to relations only. This causes some performance degradation.

Chapter 5

Skiplist and Index Based Arrays

Before consider the design details, we can look into underlying disk storage structure. The disk structure is the layout of Tracks, Cylinders and Sectors on a platter within a hard drive. Tracks are concentric circular paths written on each side of a platter. The tracks are identified by number, starting with track zero at the outer edge. Tracks are divided into smaller areas or sectors, which are used to store a fixed amount of data. Sectors are usually formatted to contain 512 bytes of data (there are 8 bits in a byte). A cylinder is comprised of a set of tracks that lie at the same distance from the spindle on all sides of all the platters. The factors that limit the time to access the data on a hard disk drive (access time) are mostly related to the mechanical nature of the rotating disks and moving heads. Read/write head takes some amount of time to reach to the track of the disk that contains data. This time is called seek time. Also some amount of time is taken to rotate the disk to bring the required disk sector under the read-write head. This time is called rotational latency. Access time is the summation of seek time and rotational latency. To reduce the rotational latency, disk sectors are usually allocated as blocks. Blocks are logical division of hard disk. It refers a group of sectors. Most of the file-system uses block size of 4KB.

The arrays in database are large in size, usually in giga bytes or tera bytes range. It needs number of disk block to store entire array data. But it is very difficult to get that many continuous blocks. Array data blocks are scattered around the disk. These scattered blocks are linked together. So database cannot directly access across these block, it requires traverse through the blocks. The Database uses extent based allocation for efficient access. Extent is a contiguous disk blocks. Then these extents are linked together as in blocks. Extent helps to maintain more number of array data in contiguous fashion.

To access data from array, we have to traverse through extents. This

traverse increases time needs to access array data. B+ tree indexing and skiplist are fast searching techniques used to reduce the accessing time. They are described below (Section 5.1 and 5.2).

5.1   Index based arrays

B+ tree is one kind of tree (Figure 5.1). It stores array-index in way that to make array access fast. B+ tree stores array data-block address and array-index for efficient retrieval. B+ tree has two kind of nodes: Leaf node and non-leaf node (Figure 5.2). Leaf node contains n pairs of data-block address and array-index. The array-index is the last array-index in the array data-block. Total number of entries in the leaf nodes are equal to the number of extents. Non-leaf node contains n array-index value and n+1 data-block addresses. Array-index in the non-leaf node is the last array-index in the child node.

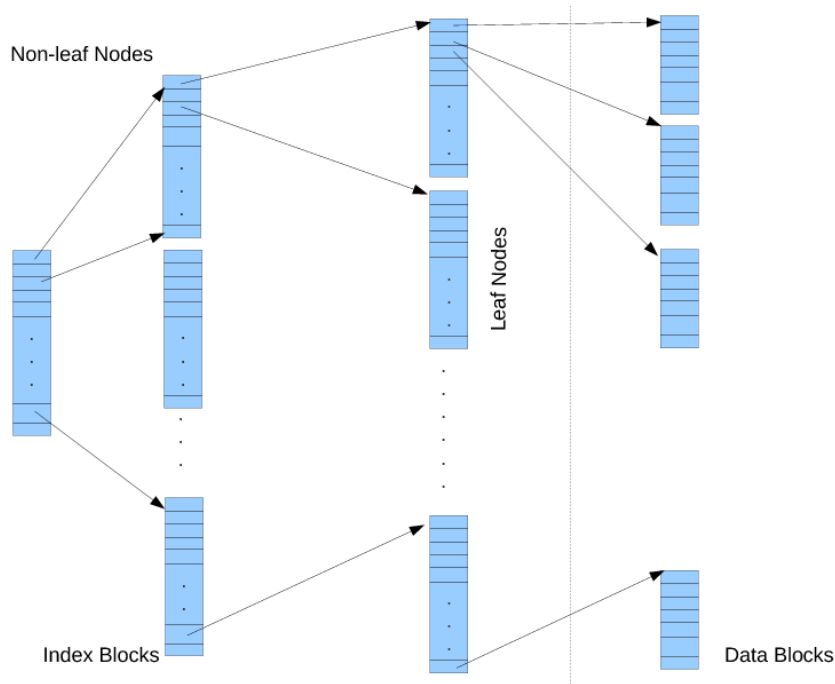

Figure 5.1: Index based array

5.2   Skiplist based arrays

Skiplist is a list of nodes which is linked together. Skiplist uses probabilistic balancing rather than strictly enforced balancing as in B+ tree indexing. Each node in the
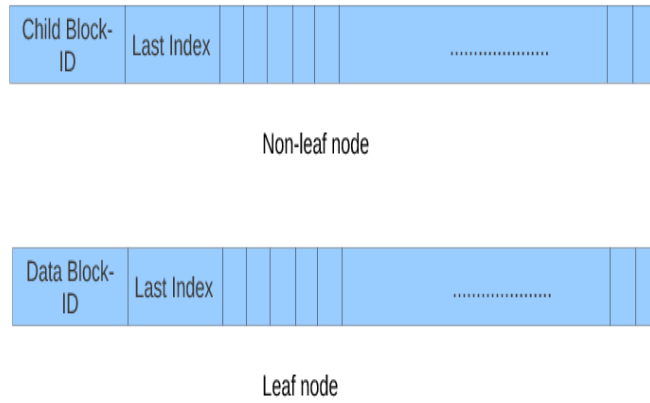
Figure 5.2: Structure of tree node

skiplist contains last array-index in the each array-data block and array-block address, and link field to the next node in the skiplist. These nodes are stored in sorted order of array-index. Number of link fields vary from node to node. Pictorial representation of skiplist is given in Figure.5.3.

Figure 5.3: Skiplist based arrays

In our database implementation, arrays are considered as first-class objects, i.e., database treats array as it is. The database architecture is given in figure 6.1. For fast retrieval of array measurement, this database uses two kinds of techniques: B+ tree indexing and skip list. Details of index based access and skiplist based access are given in the next chapter.

Chapter 6
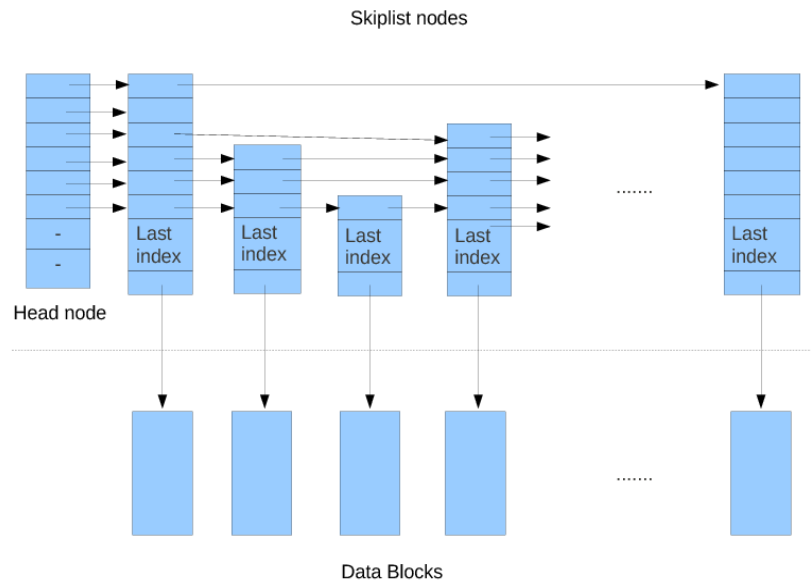
Implementation in MuBase

Before getting into the design details, we introduce MuBase, a database system on the lines of MiniRel, which we are developing at IIT Hyderabad. It has three main modules: Storage Manager, Buffer Manager and indexing layer. Details of each module will be discussed in the following sections.

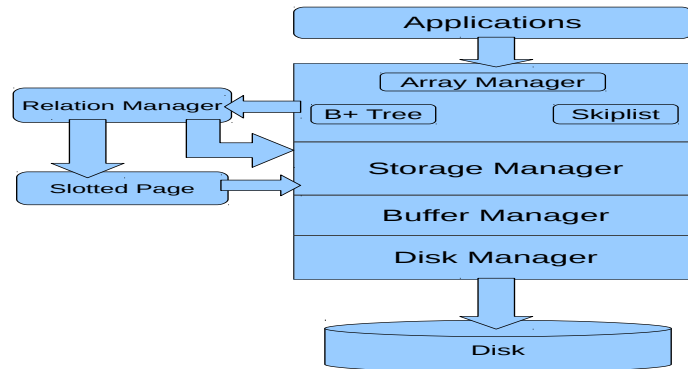Figure 6.1: MuBase system Architecture

6.1   Storage Manager in MuBase

Storage Manager handles allocation and deallocation of disk blocks. It also maintains free list. Free-list is the list of free blocks in the database which are linked together. Storage structure of the MuBase is given in Figure 6.2. First block in the database(i.e., Block 0) contains three fields: free list, relational metadata table

and column metadata table. Initially all disk blocks are linked one after other and
block Id of the first block is stored in free list field of block 0. Whenever a request
for n disk block comes, the storage manager search in the free list for n contiguous
block (or an extent of size n). If n contiguous blocks are available then allocate it
and return starting block address, otherwise return -1. Storage Manager also keeps
track of all allocated blocks. 'relational metadata table' and 'column metadata table'
fields in the block 0 maintain first block address of RELATIONAL_METADATA and
COLUMN_METADATA tables respectively. Details of RELATIONAL_METADATA
and COLUMN_METADATA tables are given in Section 6.4
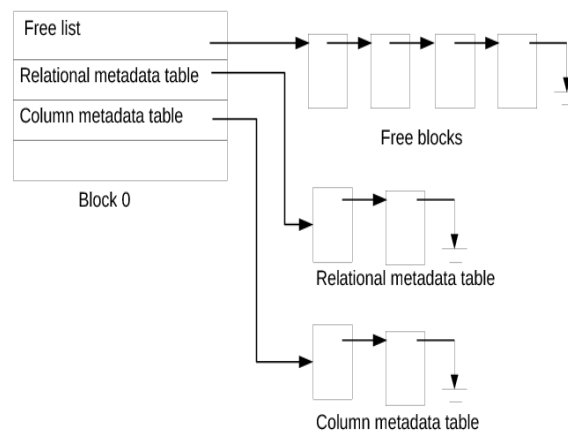
Figure 6.2: Storage structure of MuBase

6.2   Buffer Manager in MuBase

Buffer Manager handles buffering of pages needed for manipulation in the primary
memory. It uses LRU(Least Recently Used) replacement policy for page replacement.
LRU policy replaces the page which is least recently used with the newly coming
page, when there is no space for new page. LRU replacement is implemented with
FIFO(First In First Out) queue. Buffer Manager has two functions: pin-page and
unpin-page. Pin-page reads new page to the memory and tracks details about page
like it is loaded to which frame and number of process holding that page. Unpin-page
adds unused page to the head of queue. Then whenever the page replacement is
needed it replaces page from the tail of the queue.

6.3   Indexing and skiplist layer in MuBase

In order to makes array access fast, the database uses either B+ tree indexing or skiplist. So these modules are also incorporated into Array Manager(for details about Array Manager refer to Section 6.4). Structure of B+ tree nodes are given in Figure 5.2. Theses nodes are stored in disk blocks. Address of the root node(block) is stored in disk blocks. Address of the root node(block) is stored in the BPLUS_METADATA table. This table also contains other meta informations: array name, tree id and tree name. Schema of BPLUS_METADATA are given in Schema-2, Section6.4. The second method used to access array fast is skiplist. This is not implemented in the current implementation. We can easily incorporate this module to the existing system. Schema of skiplist metadata table is given in Schema-2, Section6.4.

6.4   Supporting arrays in MuBase

Two layers on top of MuBase, Array Manager and Relation Manager, support array related operations. Relation Manager maintains all metadata, i.e., data about arrays, B+ tree index and skiplist. Relation Manager stores these metadata in system tables such as ARRAY_METADATA, BPLUS_METADATA, SKIPLIST_METADATA tables. Details of these system tables are stored in RELATIONAL_METADATA and COLUMN_METADATA tables. RELATIONAL_METADATA table contains details of metadata tables such as object id, relation name, number of attributes(or columns) in the metadata table and starting block address. COLUMN_METADATA table contains details of each attributes in metadata table such as attribute name, attribute type and attribute size. RELATIONAL_METADATA and COLUMN_METADATA tables are stored in fixed length slotted pages. Its structures are given in Figure 6.3 and Figure 6.4. System tables are stored in variable length slotted pages(Figure 6.5).

Number of entries	Next block	Object ID
Relation name	# Attributes	Starting block ID
Object ID		....
....		
....	# Attributes	Starting block ID

Figure 6.3: Slotted page organisation of `RELATIONAL_METADATA` table

Number of entries	Next block	Object ID
Relation name	Column name	Column index
Column type	Object ID	....
....		
....	Column index	Column type

Figure 6.4: Slotted page organisation of COLUMN_METADATA table

Number of entries	End of free space	Next block
Pointer to slot 1	Length of slot 1	Pointer to slot 2
Length of slot 2	......	
....		
....	Slot 2 (metadata)	Slot 1 (metadata)

Figure 6.5: Slotted page organisation of system tables tables

Array Manager supports basic operations on arrays such as create array, read array, write array, populate array and update array. Create array operation in the Array Manager module create an array of given dimensions. For example create(A, 10, 10) create an array A of dimensions 10×10. The Array Manager linearizes this array, i.e., it map multidimensional array to single dimension space, and it calls the Storage Manager to allocate the required space(i.e., required number of extents). Then these linearised arrays are stored in array-blocks. Each array-block maintains two more fields, last linear-index in that block and pointer to next array-block, to make array operations easier. Metadata of arrays are stored in ARRAY_METADATA and ARRAY_DIMENSION_METADATA tables. ARRAY_METADATA table contains array id, array name, dimensionality, element type, starting block address. ARRAY_DIMENSION_METADATA contains details of each dimensions such as dimension index(i.e. hierarchical order) and dimension size. These metadata are stored in variable-sized slotted page. Schema of ARRAY_METADATA and AR-RAY_DIMENSION_METADATA are given below (Schema-2, Section6.4). Relation Manager handles reading and writing of these metadata.

In case of read operation, Array Manager uses B+ tree index to speed up the array element access. Array Manager converts the required element's array indices to linear space. Then Array Manager calls B+ tree module to find the extent

27

to which required array element is stored. Address of starting block of B+ tree(i.e., root block address) is maintained in the BPLUS_METADATA table. The database performs the read, write and update operations in similar fashion.

verbatim

Schema-2: system tables

```
ARRAY_METADATA(ARRAY_ID, ARRAY_NAME, DIMENSIONALITY, ELEMENT_TYPE, START_BLOCK)
ARRAY_DIMENSION_METADATA(ARRAY_ID, DIMENSION_INDEX, DIMENSION_SIZE)
BPLUS_METADATA(ARRAY_NAME, TREE_ID, TREE_NAME, START_BLOCK)
SKIPLIST_METADATA(ARRAY_NAME, SKIPLIST_ID, SKIPLIST_NAME, START_BLOCK)
```

6.5   Implementation status

We represented array as a first-class object and implemented underlying storage for array. Storage Manger linearises the multidimensional array and then store into disk block. We implemented basic functionality such as create array, retrieval and update of array data. Array Manager, top layer in the database system, supports array related operations. Array of any number of dimensions can be created with methods in the Array Manager. Retrieval and update of array can be done through B+ tree indexing. This gives a fast access to the array data. This implementation also supports linearised access. It gives better performance for queries which are accessing range of elements from multidimensional array.

Chapter 7

Summary and Future Work

In this thesis we considered issues related to disk based storage of large multi-dimensional arrays, which commonly arise in scientific and analytical applications. Traditional approaches like storing them as binary-large-objects or as relations have several limitations. In this thesis we presented approaches for supporting arrays as first-class objects in a database management system. We presented indexing schemes for fast retrieval of array elements when storage is allocated in blocks or extents. We have partially implemented the techniques in MuBase, a database system which we are developing at IIT Hyderabad.

Our current implementation of arrays in MuBase supports creation of arrays with specified dimensions, index based storing and retrieval of values. B+ tree index support is provided for fast index based retrieval. Future work includes completing the array support in MuBase with all the operations discussed in this thesis, and comparing the performance of different alternatives like B+ tree indexing and skiplists. Future work can also explore new requirements arising in areas like image and video processing, which require to manage large arrays

References

[1] M. Shapiro and E. Miller. Managing Databases with Binary Large Objects. *16th IEEE Mass Storage Systems Symposium and 7th NASA Goddard Conference on Mass Storage Systems and Technologies (1999)* 185–193.

[2] P. Baumann, P. Furtado, R. Ritsch, and N. Widmann. The RasDaMan approach to multidimensional database management. *Proceedings of the ACM symposium on Applied computing (1997)* 166–173.

[3] E. F. Codd. A relational model of data for large shared data banks. *Magazine: Communications of the ACM* volume 13, (1970) 339–346.

[4] A. S. Szalay, J. Gray, A. R. Thakar, P. Z. Kunszt, T. Malik, J. Raddick, C. Stoughton, and J. va ndenBerg. The SDSS SkyServer- Public Access to the Sloan Digital Sky Server Data. *ACM SIGMOD proceedings (2002)* 570–581.

[5] SDSS home page. http://www.sdss.org.

[6] S. Arkhipentov and D. Golubev. Oracle Express OLAP, ,ISBN:1584500840. Oracle Books, 2001.

[7] Oracle Documentation: Oracle Database Concepts 11g Release 1 (11.1). Oracle Books.

[8] J. Becla and K.-T. Lim. Report from the SciDB Workshop. *Data Science* volume 7, (2008) 88–95.

[9] P. Cudre-Mauroux, H. Kimura, K.-T. Lim, J. Rogers, R. Simakov, E. Soroush, P. Velikhov, D. L. Wang, M. Balazinska, J. Becla, D. DeWitt, B. Heath, D. Maier, S. Madden, J. Patel, M. Stonebraker, and S. Zdonik. A Demonstration of SciDB: A ScienceOriented DBMS. *Proceedings of the VLDB Endowment (2009)* .

[10] R. Agrawal, A. Gupta, and S. Sarawagi. Modeling Multidimensional Databases. *Proceedings of ICDE (1997)* 232–243.

[11] M. Stonebraker, J. Becla, D. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. Zdonik. Requirements for Science Data Bases and SciDB. *CIDR Conference (2009)* .

[12] Rasdaman home page. http://rasdaman.eecs.jacobsuniversity.de/trac/rasdaman.

[13] P. B. Angelica Garcia Gutierrez. The RasDaMan Array Algebra. *RasDaMan Technical Report No.7* .

[14] A. R. van Ballegooij. RAM: A Multidimensional Array DBMS. *EDBT 2004 Workshops LNCS 3268* 154165.