Thesis No: CSER-M-17-02

A Scalable Storage System for Structured Data

By

Mehnuma Tabassum Omar

Department of Computer Science and Engineering
Khulna University of Engineering & Technology
Khulna 9203, Bangladesh

December, 2017

A Scalable Storage System for Structured Data

By

Mehnuma Tabassum Omar

Roll No: 1307505

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering

Department of Computer Science and Engineering

Khulna University of Engineering & Technology

Khulna 9203, Bangladesh

December, 2017

Declaration

This is to certify that the thesis work entitled "A Scalable Storage System for Structured Data" has been carried out by Mehnuma Tabassum Omar in the Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh. The above thesis work or any part of this work has not been submitted anywhere for the award of any degree or diploma.

Signature of Supervisor

Signature of Candidate

Approval

This is to certify that the thesis work submitted by Mehnuma Tabassum Omar entitled "A Scalable Storage System for Structured Data" has been approved by the board of examiners for the partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering in the Department of Computer Science and Engineering, Khulna University of Engineering & Technology, Khulna, Bangladesh in December, 2017.

BOARD OF EXAMINERS

1. 12.12.17
Dr. K. M. Azharul Hasan — Chairman (Supervisor)
Professor, Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

2. 12-12-17
Head of the Department — Member
Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

3. 12/12/2017
Dr. M. M. A. Hashem — Member
Professor, Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

4. 12.12.17
Dr. Kazi Md. Rokibul Alam — Member
Professor, Department of Computer Science and Engineering
Khulna University of Engineering & Technology, Khulna

5. 12.12.17
Dr. Md. Anisur Rahman — Member (External)
Professor, Computer Science and Engineering Discipline
Khulna University

Acknowledgment

All the praise to the almighty Allah, whose blessing and mercy succeeded me to complete this thesis work fairly. I gratefully acknowledge the valuable suggestions, advice and sincere co-operation of Dr. K. M. Azharul Hasan, Professor, Department of Computer Science and Engineering, Khulna University of Engineering & Technology, under whose supervision this work was carried out. His open-minded way of thinking, encouragement and trust makes me feel confident to go through different research ideas. From him, I have learned that scientific endeavor means much more than conceiving nice algorithm sand to have a much broader view at problems from different perspectives. I would like to convey my heartily ovation to all the faculty members, officials and staffs of the Department of Computer Science and Engineering as they have always extended their co-operation to complete this work. I would also like to thank my parents for their wise counsel. Last but not least, I wish to thank my friends for their constant support.

Author

Abstract

Array based storage system is a key choice of many featured applications such as scientific, engineering, and financial computing applications; for their easy maintenance. However, the lack of scalability of the conventional approaches degrades with the dynamic size of data sets as they entail reallocation in order to preserve expanded data velocity. To maintain the velocity of data, the storage system must be scalable enough by allowing subjective expansion on the boundary of array dimension. Again, for an array based storage system, if the number of dimension and length of each dimension of the array is very high then the required address space overflows and hence it is impossible to allocate such a big array. We demonstrate a dynamic scalable array storage scheme namely Scalable Array Indexing (SAI) that can be an efficient choice of large volume dynamic data management by removing the problems of the existing ones. The SAI converts an n dimensional array to 2 dimensions. Traditionally, the dynamic array models need indices for each dimensions. Since, SAI is a 2 dimensional dynamic model it reduces the index overhead significantly and compromises relatively faster data accessing. We also propose another scalable structure based on the SAI scheme to increase storage utilization. We named the structure as Segment based Scalable Array Indexing (SSAI). Using our SSAI structure, we also offer an efficient encoding with good comparison ratio and range of usability. All the operations are presented with sufficient theoretical analysis and experimental results.

Contents

		PAGE
	Title Page	i
	Declaration	ii
	Approval	iii
	Acknowledgment	iv
	Abstract	v
	Contents	vi
	List of Abbreviations	viii
	List of Tables	ix
	List of Figures	x
CHAPTER I	**Introduction**	**1**
	1.1 Introduction	1
	1.2 Problem Statement	2
	1.3 Objectives	4
	1.4 Scope	4
	1.5 Contribution	5
	1.6 Organization of the Thesis	5
CHAPTER II	**Literature Review**	**6**
	2.1 Introduction	6
	2.2 The Realization of Multidimensional Array Structure	6
	2.2.1 Conventional Multidimensional Array	6
	2.2.2 Traditional Extendible Array	7
	2.2.3 Axial VectorExtendible Array	9
	2.2.4 Extendible Karnaugh Array	10
	2.2.5 Generalized Two-dimensional Array	12
	2.2.6 Rasdaman	13
	2.2.7 MonetDB	14
	2.2.8 Other Structures	15
	2.3 Encoding Schemes for High Dimensional Data	16
	2.3.1 Chunk-Offset Encoding	16
	2.3.2 History-Offset Encoding	17
	2.3.3 Segment-Offset Encoding	19
	2.3.4 History Pattern Encoding	19
	2.3.5 Integer-Key Encoding	21
	2.3.6 Other Schemes	22
	2. 4 Discussion	23

		PAGE
CHAPTER III	**Scalable Storage Systems for Higher Order Index Array**	**24**
	3.1 Introduction	24
	3.2 Realization of a Scalable Array Indexing (SAI)	25
	3.2.1 Dimension Conversion	25
	3.2.2 Scalable Indexing	27
	3.3 Operations on a SAI System	28
	3.3.1 Construction and Extension	28
	3.3.2 Dimension Transformation	29
	3.3.3 Point Query	32
	3.3.4 Range Query	33
	3.4 Realization of a Segment based Scalable Array Indexing (SSAI)	36
	3.4.1 Segmentation	36
	3.5 Operations on a SSAI System	37
	3.5.1 Construction and Extension	37
	3.5.2 Point Query	38
	3.6 2 Dimensional Key Value Encoding (2DKVE)	40
	3.6.1 Encoding	40
	3.6.2 Data Access	41
	3.6.3 Decoding	41
	3.7 Conclusion	42
CHAPTER IV	**Experimental Analysis**	**43**
	4.1 Experimental Setup	43
	4.2 Performance Analysis of the Structure	44
	4.2.1 Index Overhead	44
	4.2.2 Construction Cost	46
	4.2.3 Extension Cost	48
	4.2.4 Retrieval Cost	52
	4.2.5 Storage Utilization	54
	4.3 Performance Analysis of the Encoding	56
	4.3.1 Index Overhead	57
	4.3.2 Range of Usability	58
	4.3.3 Storage Cost	61
	4.4 Discussion	64
CHAPTER V	**Conclusion**	**65**
	5.1 Summary	65
	5.2 Future Scope of Work	66
References		**67**

LIST OF ABBREVIATIONS

Abbreviation	Description
CCS	Compressed Column Storage
CMA	Conventional Multidimensional Array
COE	Chunk-Offset Encoding
CRS	Compressed Row Storage
EA1	Traditional Extendible Array
EA2	Axial Vector Extendible Array
EaCRS	Extendible CRS
ECCS	Extendible CCS
EKA	Extendible Karnaugh Array
EKMR	Extended Karnaugh Map Representation
G2A	Generalized Two-dimensional Array
HOE	History-Offset Encoding
HPE	History-Pattern Encoding
IKE	Integer-Key Encoding
SA	Subarray
SAI	Scalable Array Indexing
SSAI	Segment based Scalable Array Indexing
SOE	Segment-Offset Encoding
2DKVE	Two Dimensional Key value Encoding

LIST OF TABLES

Table No.	Title	Page
4.1	Parameters for Constructed Prototypes	43
4.2	Analytical Index Overhead for Constructed Prototypes	45
4.3	Analytical Construction Cost for Constructed Prototypes	47
4.4	Analytical Extension Cost for Static (CMA) and Dynamic (EA)	49
4.5	Analytical Extension Cost for Dynamic (EA) Prototypes	49
4.6	Analytical Result of maximum length of the compared Prototypes	54
4.7	Analytical Index Overhead of Encoding Schemes	57
4.8	Analytical Compression Ratio of Encoding Schemes	59
4.9	Analytical Usable Length of Encoding Schemes	59
4.10	Analytical Storage Cost Values for Encoding Scheme	62

LIST OF FIGURES

Figure No.	Title	Page
2.1	A Three Dimensional CMA of Size [3×4×5].	7
2.2	A Three dimensional Traditional Extendible Array of Size [5×6×4].	8
2.3	A 3-dimensional Axial Vector Extendible Array of size [5×5×3].	9
2.4	Realization of Boolean function using K-map.	11
2.5	Logical extension of a 4-dimensional EKA.	11
2.6	An Extension Realization of EKA(4)	12
2.7	CMA(3) and It's Equivalent G2A.	13
2.8	Basic Structure of a Rasdaman Array Model.	13
2.9	Basic Structure of a MonetDB Array Model.	14
2.10	A 3-dimensional Array Partitioned into Chunks.	16
2.11	A 3-dimensional Array Stored As Chunk-Offset Encoding.	17
2.12	A 3-dimensional Extendible Array Encoded as History-Offset Encoding.	18
2.13	A 4-dimensional EKA Encoded as Segment-Offset Encoding.	19
2.14	A 2-dimensional Representation of History Pattern Encoding.	20
2.15	Memory Layout of B+ Tree Node of IKE.	21
2.16	The Keylist of Fixed Length Integer Keys.	21
2.17	The KeyList of Variable Length Integer Keys	22
3.1	Transformation of a CMA[2, 2, 2, 2, 2] to a SAI[8,4]	26
3.2	Construction and Extension of a SAI System.	28
3.3	Realization of a SAI[27,9].	29
3.4	Range Key Operation on SAI.	35
3.5	Segmentation of a SAI to a SSAI.	36
3.6	Construction and Extension of a SSAI System.	37
3.7	Realization of a SSAI[18,9].	38

Figure No.	Title	Page
3.8	A Sparse Representation of a SSAI[18][9].	39
3.9	A Realization of a 2DKVE System	40
3.10	Key Structure of a 2DKVE System.	40
4.1	Analytical Result of Index Overhead	45
4.2	Experimental Result of Index Overhead	46
4.3	Analytical Result of Construction Cost	47
4.4	Experimental Result of Construction Cost	48
4.5	Analytical Result of Extension Cost	49
4.6	Experimental Result of Extension Cost	51
4.7	Experimental Result of Retrieval Cost	54
4.8	Analytical Result of Storage Utilization	55
4.9	Experimental Result of Storage Utilization	56
4.10	Analytical Result of Index Overhead for Encoding Schemes	58
4.11	Experimental Result of Index Overhead for Encoding Schemes	58
4.12	Analytical Result of Range of Usabilities of Encoding Schemes	60
4.13	Experimental Result of Range of Usabilities of Encoding Schemes	61
4.14	Analytical Result of Storage Cost of Encoding Schemes	63
4.15	Experimental Result of Storage Cost of Encoding Schemes	64

CHAPTER I
Introduction

1.1 Introduction

Arrays are the most popular data structures for their outstanding features in rational storing and swift processing. The high-volume arbitrary dimensionality feature of arrays make it diversely perceptible in researches, like medical imaging, geographic information system, environmental and astronomical surveillances, or high precision prototypes of physical consequence [1]. For Big Data applications, the array structures like Conventional Multidimensional Array (CMA) [2] model can lead other structures like Rasdaman [3], MonetDB [4], SQL based query language such as SciQL [5], NoSQL and NewSQL [6], parallel programming model like GPU based architecture [7], distributed optimization [8] in terms of data storage or retrieval or both [9, 10]. Array based storage system is the key choice of various featured applications for their easy maintenance, but the lack of scalability of the conventional approaches degrades with the dynamic size of data sets as they entail reallocation in order to preserve expanded data velocity that means the structure is not dynamically scalable. To maintain the velocity of data, the storage system must be scalable enough by allowing subjective expansion on the boundary of array dimension. The range in which the linearized array elements map is called address space which depends on the length and/or number of dimension of array. For an array based storage system, if the number of dimension and/or length of each dimension of the array is very high then the required address space overflows quickly and hence it is impossible to allocate such a big array in the memory. The index array [11-13] offers a dynamic storage structure for preserving expanded data velocity by employing indices for each dimension. Indexing of array is a process of monitoring location of data record by assigning a key with them for the corresponding system for assisting in fast query processing [14]. Although the extendible array models are scalable enough but it requires indices for each of the dimension. Hence, the model impose high overhead to the data storage. Another problem is that, along with the rise in dimensionality, the effort in computing index, cache miss rate and data representation complexity rises [2, 15]. The traditional approaches on

algorithms and computation are inappropriate for data models having large dimensionality especially for data warehouse or big data [16]. Therefore, the traditional approaches are unable to index structured big data proficiently.

In this research work, we are going to propose scalable array storage structures that convert the n dimensions of the array into 2 dimensions; hence it involves only 2 indices. Using these 2 indices, we also offer a lossless encoding structure which ensures lower encoding cost, lower indexing cost and higher data locality.

1.2 Problem Statement

The Conventional Multidimensional Array (CMA) is a well-known array structure chosen by various applications for retrieving the array element by evaluating addressing function directly, but it has following limitations:

i. Static allocation as the data length and dimensionality is predefined and it is not dynamically scalable.
ii. Inability to represent or visualize the large volume and large dimensionality of data.
iii. Address space overflow for large value of data length or dimensionality (or both) even though resource is available.
iv. Inability to attain useful information from the huge volume applications which are generally sparsed.

Let, A[l_1] [l_2]… [l_n] be an n dimensional CMA of size [l_1, l_2,…,l_n]. Here l_1, l_2,…,l_n is the length of each dimension $d_1, d_2, \ldots, d_n$ respectively. Then the total address space required for an array would be $S_T = \prod_{1 \le i \le n} l_i = l^n$ (if $l_i = l^n$ for all i). If the elements of the CMA occupy K bytes in memory, then the allocation volume would be $V_T = S_T \times K = l^n \times K$. The total address space or array volume V increases exponentially if the length of each dimension l_i or the number of dimension n (or both) increases. As a result, it accelerates to exceed the machine word size even though the system is highly configured such as 64 bit machine. The Index Array model [11-13] solves the limitation (i) above by dynamically allocating memory during run time as form of subarrays. The subarrays are $n-1$ dimensional and hence it can delay the overflow compared to CMA with an address space allocation of size $S_E = \prod_{1 \le i \le n-1} l_i = l^{n-1}$ and volume $V_E = S_E \times K = l^{n-1} \times K$. The Index Array model is good compared to the CMA, but it cannot meet the expected demand

of memory utilization as per the demand of data velocity especially for "Big Data" applications [17]. Again, for an n dimensional array, the indexing requirement of an indexed based model is also n dimensional which reduces the capacity of storage utilization. Another concern is problem (iv) which can decrease the efficiency of large volume applications. Data encoding can be a proficient way to lessen this unintended cost of the system on the basis that the potential volume of data is not always interesting. It is a process to reserve only those data cells which are densed and significant as well. In order ensure data accuracy of an encoding structure, it is crucial to employ some data decoding structure along with the encoding structure to provide lossless information. Therefore, the process of ensuring data accuracy should be a two-way scheduling. The first scheduling generates an encoded tuple for the compressed array that resembles a memory location of the actual array. This scheduling is named as Data Encoding. And the rest one is named as Data Decoding which generates a memory location of the actual array from an encoded tuple of the compressed array.

Here, we propose a scalable index array system namely Scalable Array Indexing (SAI) which represent an n dimensional array by 2 dimensions (towards column and row direction) only. As n dimensional array requires n dimensional indexing, hence the proposed structure requires 2 indices only. But the SAI structure also suffers from address space overflow. For this reason, we also provide another structure which can enhance the performance of a SAI structure named Segment based Scalable Array Indexing (SSAI). Likewise, SAI, the SSAI structure also converts the n dimensions (nD) into 2 dimensions (2D). The only difference is that here the allocation is divided into segments. In our experiment, we have found that the SSAI does not overflow the address space and can utilize the available resource of the system. On the contrary, the existing indexed array models along with the SAI structures overflow the address space. Hence, the proposed SSAI structure has more memory utilization than the other structures. Using these 2 indices of a SSAI structure, we have also recommended a lossless data encoding scheme named as 2 Dimensional Key Value Encoding which can outperforms the other schemes as it requires only 2 indices to encode n dimensional sparse data. The SSAI structure can be applied to scalable array database [18], distributed array storage [3], parallel and distributed database [19, 20] and big data storage [21].

1.3 Objectives

The traditional multidimensional approaches are unable to index big data proficiently. To cope with this situation, the data scientists have appreciated higher dimensional data linearization. The linearization is well sufficient as per secondary memory. However, the linearization process not only rises the retrieval time and operation cost but also reduces the ability for parallelization. Again, the size of data gradually expands in scale of terabytes and petabytes. To contract with this event, random extension on the bound of array dimension is entailed as typical multidimensional array structures, are incapable of managing (extend or shrink) their bounds devoid of rearranging existing data [11]. Extendible Array resolves this challenge, but consumes high memory for indexing as per dimension value increases.

Therefore, main objectives of this research topic are –

- To propose a dynamic multidimensional array structure by dimension conversion.
- To reduce the indexing cost of an Index Array model by using two indices only for n dimensional structure.
- To find an efficient solution for the problems of the existing static structure like CMA [2] and also for the dynamic structures like Extendible Array [13] and Extendible Array [11].
- To offer an encoding scheme for the proposed structure.
- To analyze the performance and usability of the proposed encoding scheme.

1.4 Scope

The proposed scalable structures: Scalable Array Indexing (SAI) structure and Segment based Scalable Array Indexing (SSAI); are a new representation of scalable multi-dimensional array model. The important scopes are:

- The number of dimension is increased up to 16.
- The length of dimension is increased up to 648.
- The scaling operation is done one by one and up to 646.
- The machine limits are: Intel(R) Xeon(R) E5620 @ 2.40GHz processor with 8 processors, 32 GB RAM, 1406 MB cache memory and 1.3TB usable HDD.

- The program is written in C and compiled in gcc compiler on debian squeeze 6.0.5 operating system.
- The data limit is 64 bit integer only.

1.5 Contribution

The major contributions of this research topic are –

- To offer a way to represent *n* dimensional array to a feasible one through 2 dimensional representation which aid in easy visualization of large *n* dimensional array.
- To make the proposed 2 dimensional array dynamic which manages better storage utilization by removing reallocation of static structure.
- To decrease indexing cost of dynamic array model by utilizing 2 dimensional indexing of 2 dimensional proposed model.
- To offer an efficient information retrieval paradigm by utilizing proposed 2 dimensional dynamic array representation.
- To delay address space overflow by segmenting the proposed structure which increases the storage utilization.
- To offer an efficient encoding scheme using 2 dimensional indexes of proposed 2 dimensional structure which requires less encoding cost and higher range of usability.

1.6 Organization of the Thesis

The thesis is organized in six chapters as follows:

- **Chapter II** presents Literature Review of the similar domains and finds some limitations of the existing works.
- **Chapter III** proposes the scalable array models by dimension conversion. The chapter also describes different operations and algorithm with examples.
- **Chapter IV** shows the experimental outcomes of different array operations over the SAI and SSAI structure and also the usuability of the 2DKVE scheme.
- **Chapter VI** exhibits the future direction of the proposed model and outlines the conclusion.

CHAPTER II

Literature Review

2.1 Introduction

The multidimensional array structures are becoming an important data structures for storing large scale, composite and higher order data; e.g., in Big Data. Several appliances encompassing accumulation of climate information by sensors, gathering digital multimedia records, transaction documents procuring, and GPS signals commencing cell phones, are frequently using Big data in order to expound their extent of data which leads to statistics of substantial volumes [21]. The multidimensional array yet dictates such applications [22, 23]. Hence, several array models have been examined in order to verify their tremendous features.

2.2 The Realization of Multidimensional Array Structure

The array Computational paradigm is prevalent in most sciences and it has drawn attention from the database research community for many years. Some of the multidimensional array structures are given below:

2.2.1 Conventional Multidimensional Array

A Conventional Multidimensional Array (CMA) [2] or simply Array $A[l_1,l_2,\ldots,l_n]$ is an association between n-tuples of integer indices $<x_1, x_2,\ldots,x_n>$. Consider an n dimensional Conventional Multidimensional Array (CMA(n)). Let, the size of a CMA(n) or $A[l_1]$ $[l_2]\ldots$ $[l_n]$ is $[l_1, l_2,\ldots,l_n]$. Then $<x_1, x_2,\ldots,x_n>$ be the Real n dimensional Index; where $l_1, l_2,\ldots,l_n$ is the length of each dimension $d_1,d_2,\ldots, d_n$ respectively and $x_i = 0,1,2, \ldots, (l_i\text{-}1)$ $(0 \leq i \leq n)$, where l_i is the length of dimension i. The domain from which the elements are chosen is immaterial and we make the assumption that only one memory location need be assigned to each n-tuples. Each array may be visualized as the lattice points in a rectangular region of n-space. The set of continuous memory locations into which the array maps is denoted

by $A[0{:}D]$ where $D = (\prod_{i=1}^{n} l_i) - 1$ .Any element in the multidimensional array is determined by an addressing function as follows,

$$f(x_n, x_{n-1}, x_{n-2},...,x_2, x_1) = l_1 l_2..l_{n-1} x_n + l_1 l_2..l_{n-2} x_{n-1} + ... + l_1 x_2 + x_1 \quad (2.1)$$

Conventional storage of multidimensional arrays is done by linearization. In the two dimensional case, the linearization may be done by rows or by columns. But in general, for n-dimensional array there are $n!$ possible linearization orders according to the possible ordering of the dimensions. An illustration of 3 dimensional TMA of dimension length 3×4×5 is given in Figure 2.1. In the CMA scheme, a three dimensional array of size 3×4×5 can be viewed as three 4×5 two-dimensional arrays. Here, l_1=5, l_2=4, l_3=3. Hence, (see Eq. 2.1) it can be shown that the index $< x_3, x_2, x_1 > = < 0, 2, 3 >$ maps to the memory position $< 0, 2, 3 > = 0 \times 5 \times 4 + 5 \times 2 + 3 = 13$.

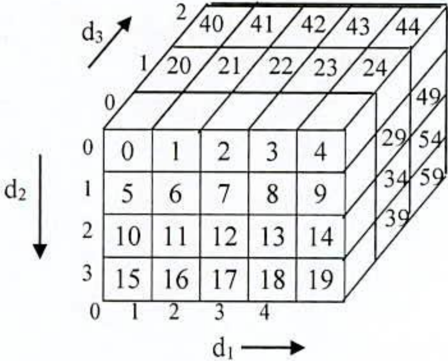


Figure 2.1: A Three Dimensional CMA of Size [3×4×5].

2.2.2 Traditional Extendible Array

The Traditional Extendible Array [13, 24] or simply Extendible Array is another representation of multidimensional array. It has the property that the indices of the respective dimensions can be arbitrarily extended without reorganizing previously allocated elements. Following is a short description of a Traditional Extendible Array.

Subarray (SA).The memory allocation of an Extendible Array is done by allocation a collection of memory called subarray (SA). A subarray $SA[l_1,l_2,...,l_{n-1}]$ of an n dimensional Array $A[l_1,l_2,...,l_n]$is an association between $(n-1)$ tuples of integer indices $<x_1, x_2,...,x_n>$ and $x_i = 0,1,2, ..., (l_i-1)$ $(0 \le i \le n-1)$. The set of continuous memory locations into which the array maps is denoted by $SA[0:D]$ where $D = (\prod_{i=1}^{n-1} l_i) - 1$ .For an

extension along dimension i of the nD array the SA would be $(n-1)$D and the SA size, sz is calculated as follows:

$$sz = \prod_{j=1}^{n} l_j \ (i \neq j) \tag{2.2}$$

Auxiliary Table. A Traditional Extendible Array manages its scalability by using three types of auxiliary tables. For each dimension these tables exist. These are required for monitoring dynamic extensions and also fast data retrieval. The Extendible Array can be extended in any direction in any dimension only by the cost of these three auxiliary tables. The auxiliary tables are as follows:

- History Table: It contains the construction or extension history of an Extendible Array.
- Address Table: It contains the first address of the subarray of an Extendible Array.
- Coefficient Table: The table is required for storing the coefficients ($l_1 l_2 .. l_{n-1}, l_1 l_2 .. l_{n-2}, ..., l_1$) of the addressing function (see Eq. 2.1). Coefficient table holds the coefficient of the n-1 dimensional SA and it is $n-2$ dimensional.

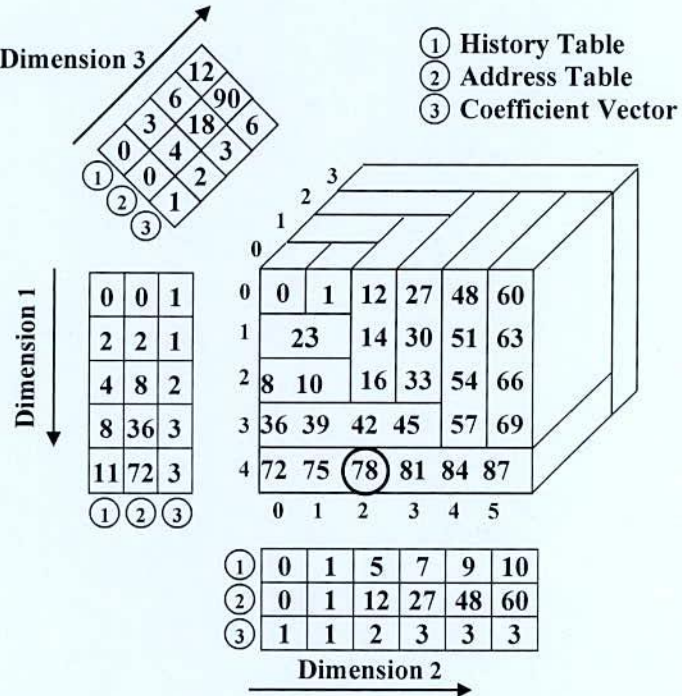


Figure 2.2: A Three dimensional Traditional Extendible Array of Size [5×6×4].

The accessing of the elements of an Extendible array is done by using these three kinds of auxiliary tables, the address of an array element can be computed as follows. Consider the element ⟨4, 2, 0⟩ in Fig. 2.2. Compare $H_1[4] = 11$, $H_2[2] = 5$ and $H_3[0] = 0$. Since $H_1[4] > H_2[2]$, $H_1[4] > H_3[0]$, it can be proved that the element ⟨4, 2, 0⟩ is involved in the extended subarray S beginning from the address $L_1[4] = 72$. From the coefficient vector of $C_1[4] = 3$, the offset of element ⟨4, 2, 0⟩ from the first address of S is computed by 3×2+0 = 6, the address of the element is determined as 72 + 6 = 78.

2.2.3 Axial Vector Extendible Array

The Axial Vector Extendible Array [11] is another representation of Traditional Extendible Array (sec. 2.2.2). Here, the mapping function or the addressing function mentioned in Eq. 2.1 has been reorganized as the conventional array mapping function allows extendibility in only one dimension 0 (in the case of row major). They renamed the auxiliary table as axial vector that includes *<starting index of the dimension, starting address of the subarray, multiplicative coefficients, memory pointers>*.

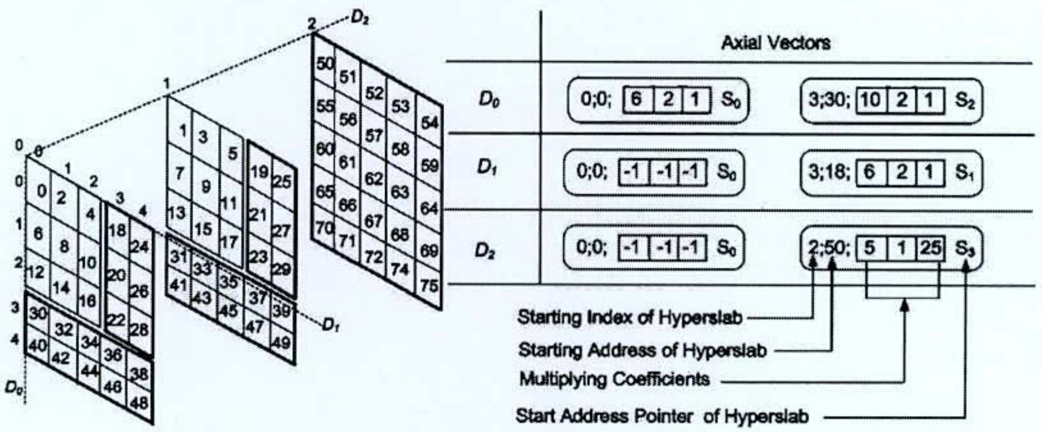


Figure 2.3: A 3-dimensionalAxial Vector Extendible Array of size [5×5×3].

Suppose that in a n dimensional extendible array A[l_0] [l_1]… [l_n], dimension d_i is extended by λ_i, then then the index range increases froml_i to $l_i + \lambda_i$. The idea is to allocate an-1 dimensional block of array elements or subarray so that addresses are computed as displacement from the location of element$A < 0, 0, 0, \ldots, l_i, \ldots, 0 >$. The desired mapping function that computes the address of an element $(x_0, x_1, x_2, \ldots x_n)$ during allocation is given by:

$$f(x_0, x_1, x_2, \ldots x_n) = Z_{l_i} + (x_i - l_i)C_i + \sum_{\substack{j=0 \\ j \neq i}}^{n-1} x_j C_j \qquad (2.3)$$

Where,

$$C_i = \sum_{\substack{j=0 \\ j\neq i}}^{n-1} l_j \, , C_j = \sum_{\substack{r=j+1 \\ r\neq i}}^{n-1} l_r$$

Here, Z_{l_i}denotes the maximum starting address of the subarray that is adjoined. i denotes the dimension that was extended. l_i denotes the bound of the index range before the expansion. The starting address of such a contiguous sequence of locations is what is stored in the subarray. The values $v[2]; v[3]; \ldots v[n-1]$ are the respective multiplicative coefficients and $v[0]$ and $v[1]$ are the starting index and the starting linear address of hyperslab respectively. The value -1 denotes null entries. To compute the address of a given n dimensional index $(x_0, x_1, x_2, \ldots x_n)$ the subarray that contains the element needs to be determined first. The subarray whose first elements ar e$(x_0, 0, \ldots, 0)$, $(0, x_1, 0 \ldots, 0)\ldots$ $(0, 0, \ldots, x_n)$ give the candidate subarray that should contain the element whose index is $(x_0, x_1, x_2, \ldots x_n)$. The element of $(x_0, x_1, x_2, \ldots x_n)$ always belongs to the subarray with the maximum starting address of the candidate subarrays. This is determined by comparing the starting addresses of the corresponding elements of the axial-vectors. Let the vector of records of dimension j be denoted by $\Gamma_j[]$. The starting addresses of the axial-vectors are given by $\Gamma_j[x_j], 0 \leq j < k$. Fig. 2.3 shows the extension of a three dimensional array A of initial size [5×5×3], and corresponding axial vectors. For example, suppose we desire the linear address of the element A[3,3,1]. $Z_{l_i} = max(\Gamma_0[3]v[1], \Gamma_1[3]v[1], \Gamma_2[2]v[1]) = max(30, 18, 0) = 30$. Thus, $i = 0 Z_{l_i} = Z_{l_0} = 30$ and $l_0 = 3$. Now, using Eq. 2.2 we have, $f(3, 3, 1) = 30 + 10 \times (3-3) + 2 \times 3 + 1 \times 1 = 30 + 0 + 6 + 1 = 37$.

2.2.4 Extendible Karnaugh Array

The Extendible Karnaugh Array (EKA) [12] is a multidimensional Extendible Array model that utilizes the concepts of Karnaugh Map (K-map) [25]. The K-map is a well-known depiction employed in Boolean expression minimization typically assisted by mapping values for each potential combinations. Fig. 2.4 (a) depicts a 4 variable K-map representation of a Boolean function (2^4 combinations). The row is denoted by the pair (w, x) and the column is denoted by the pair variables (y, z). The row and the column indicate the potential combinations of a Boolean function in a form of two dimensional array. The row pair and column pair of a K-map are re-expressed as row dimension and column dimension respectively of an EKA. Here, the indices of the row dimension are adjacent to

each other and the indices of the column dimension are adjacent to each other. The adjacent dimensions of row is denoted as *adj*(*z*) = *y* or *adj*(*y*) = *z* and adjacent dimensions of column is denoted as *adj*(*w*) = *x* or *adj*(*x*) = *w* and. The EKA representation of Fig. 2.4(a) is shown in Fig. 2.4(b). The EKA uses the same auxiliary table as EA1 (sec. 2.2.2) except now the address table store the first address of the first segment of a SA as the SA of an EKA is divided into some segments.

Consider the array in Fig. 2.5(a), the dimensions are d_1, d_2, d_3 and d_4 and the size of the array is $[l_1, l_2, l_3, l_4]$ and subscripts varies from 0 to l_i-1. In the current example $l_i = 2$. The dimension (d_1, d_3) and (d_2, d_4) are adjacent dimensions respectively. The logical extension in d_1 is shown in Fig. 2.5(b). The size of the extended subarray which is allocated dynamically is determined by $[l_2, l_3, l_4]$ (i.e. 3 other dimensions). The number of segments is the length of the adjacent dimension, adj(d_1) = d_3 ; In this case it is $l_3 = 2$. The size of each segmented subarray extended along dimension d_1 is determined by $[l_2, l_4]$. After extending along dimension d_1, the length of that dimension is incremented by 1. For each extension the corresponding auxiliary tables are maintained accordingly. Fig. 2.5(c), 2.5(d) and 2.5(e) shows the extension realization along dimension d_2, d_3 and d_4 respectively.Fig.2.6shows the extension realization along with the auxiliary table values of the realization Fig. 2.5(e).

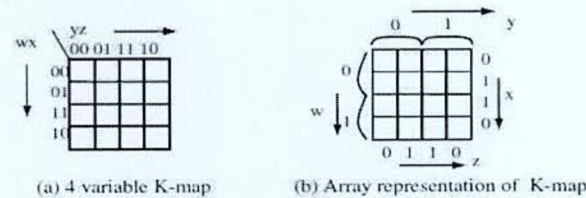


Figure 2.4: Realization of Boolean function using K-map.

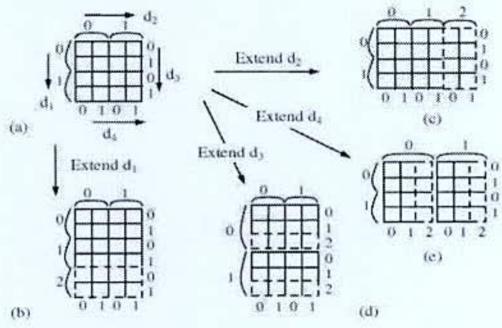


Figure 2.5: Logical extension of a 4-dimensional EKA.

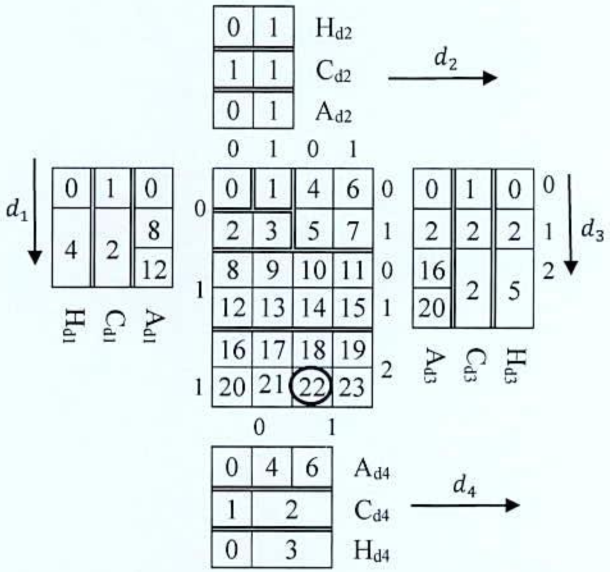


Figure 2.6: An Extension Realization of EKA(4)

Let the value to be retrieved is indicated by the subscript $\langle x_1, x_2, x_3, x_4\rangle$. The maximum history value among the subscripts h_{max}= max($H_{d1}[x_1]$, $H_{d2}[x_2]$, $H_{d3}[x_3]$, $H_{d4}[x_4]$) and the dimension (say d_{max}) that corresponds to history value h_{max} is determined. h_{max} is the subarray that contains the desired element. The adjacent dimension $adj(d_{max}) = d_{adj}$ (say) and its subscript x_{adj} is found. Now the first address of the segment is found from $A_{dmax}[x_{max}][x_{adj}]$. The *offset* from the first address is computed using the addressing function (see Eq. 2.1); the coefficient vectors are stored in C_{dmax}. Then adding the *offset* with the first address, the desired array cell (x_1, x_2, x_3, x_4) is found. Let (x_1, x_2, x_3, x_4) =(1, 0, 2, 1) is given (see Fig.2.6). Here h_{max}= max($H_{d1}[1]$, $H_{d2}[0]$, $H_{d3}[2]$, $H_{d4}[1]$)= max(4, 0, 5, 3) =5, and dimension corresponding to h_{max} i.e. $d_{max} = d_3$ whose subscript $x_{max} = 2$ and $adj(d_{max}) = adj(d_3) = d_1 = d_{adj}$ and $x_{adj} = 1$. So the *firstAddress* = $A_{d3}[2][1]$ = 20, and offset is calculated using the coefficient vector stored in coefficient table C_{d3} which is 2. Here, *offset* = $C_{d3}[2]*x_4 + x_2 = 2*1 + 0 = 2$. Finally adding the offset with the first address the desired location 20 + 2 = 22 is found and circled in Fig. 2.6.

2.2.5 Generalized Two-dimensional Array

The Generalized Two-dimensional Array (G2A) [26] represents an algorithm to represent an n dimensional (*n*D) array by a 2 dimensional (2D) array. The *n*D array is converted to a 2D array. Hence the indexes of the *n*D array are also converted to 2D array. $\left\lceil \frac{n}{2} \right\rceil$ subscripts are converted to row direction and the rest $\frac{n}{2}$ columns to column direction. Hence an *n*D

array can be drawn in a 2D plane to visualize the data. In G2A, the 3 dimensions d_1, d_2, d_3 are converted to 2 dimensions where d_1, d_3 are for row and d_2 are for the column. Fig. 2.7 shows the G2A,A$'[l'_1][l'_2]$ for a TMA(3) A[2][3][4] where $l'_1 = l_1 \times l_3 = 8$ and $l'_2 = l_2 = 3$.For example an element A[1][1][2] of TMA(3) is equivalent G2A is A$'[x'_1][x'_2]$ where $x'_1 = x_1 l_3 + x_3 = 1 \times 4 + 2 = 6$ and $x'_2 = x_2 = 1$. For backward mapping, if an element in G2A is A$'[x'_1][x'_2]$ is known then it's equivalent TMA(3) becomes A[x_1][x_2][x_3] where $x_3 = x'_1 \% \, l_3 = 6 \% 4 = 2$, $x_1 = \frac{x'_1}{l_3} = 6/4 = 1$ and $x_2 = x'_2 = 1$. For example, A$'$[6][1] is equivalent to A[1][1][2]. Here % indicates the 'modulus' operation and / indicates 'division' operation.

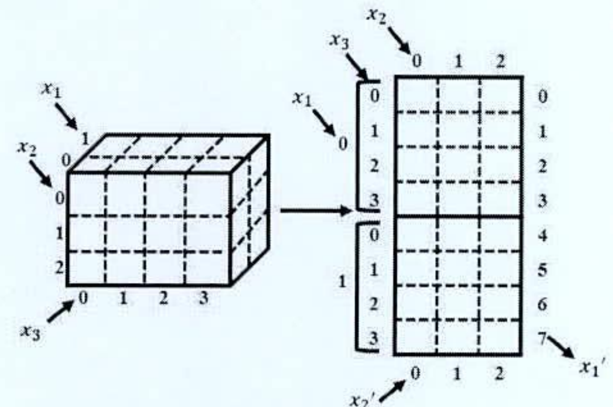


Figure 2.7: CMA(3) and It's Equivalent G2A.

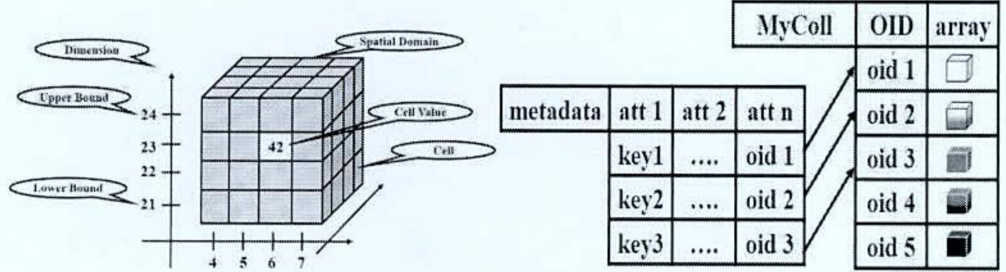


Figure 2.8: Basic Structure of a Rasdaman Array Model.

2.2.6 Rasdaman

Rasdaman ("raster data manager") [3, 27] is an Array DBMS, that is: a Database Management System which adds capabilities for storage and retrieval of massive multi-dimensional arrays, such as sensor, image, and statistics data. A frequently used synonym to arrays is raster data, such as in 2D raster graphics; this actually has motivated the name Rasdaman. However, Rasdaman has no limitation in the number of dimensions - it can serve, for example, 1D measurement data, 2D satellite imagery, 3D x/y/t image time series

and x/y/z exploration data, 4D ocean and climate data, and even beyond spatiotemporal dimensions. The Rasdaman conceptual model centers around the notion of a multidimensional array of arbitrary dimension, extent in each dimension – whereby each lower and upper bound can be fixed or variable –, and base type. Usually such an array will be an attribute of some other object, e.g., the "raw data" accompanied by "registration data" within an image. In Rasdaman databases, arrays are grouped into collections. All elements of a collection share the same array type definition. Collections form the basis for array handling, just as tables do in relational database technology. All operations applied to a collection are applied in term to each of the array in the collection. A collection is essentially equivalent to a relational table with two columns: one holds the array values, the other holds a unique ID for each array object. Fig. 2.8 shows the basic structure of a Rasdaman Array Model.

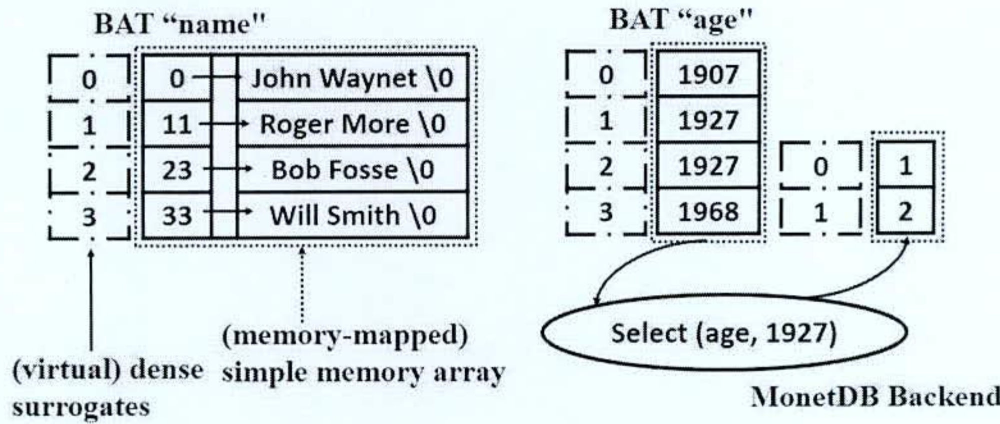


Figure 2.9: Basic Structure of a MonetDB Array Model.

2.2.7 MonetDB

MonetDB [28, 29] is column-oriented database management system which was designed to provide high performance on complex queries against large databases, such as combining tables with hundreds of columns and millions of rows. MonetDB has been applied in high-performance applications for online analytical processing, data mining, geographic information system (GIS). Different from traditional database systems, MonetDB does not store all attributes of each relational tuple (together in one record), and instead treats a relational table as vertical fragmentations. Thus, MonetDB stores each column of the table in a separate (surrogate, value) table, called a BAT (Binary Association Table). The left column, called head column, is surrogate or OID (object-

identifier), and only the right column stores the actual attribute values (called tail). As a result, a relation table consisting of k attributes then is represented by *k* BATs. With the help of the system generated OID, MonetDB needs to lookup the *k* BATs in order to reconstruct the tuple. In order to perform tuple reconstructions from the k BATs, MonetDB adopts a tuple-order alignment across all base columns. That is, each attribute value belonging to a tuple *t*is stored in the same position of the associated BAT. Next, to represent the tail column, MonetDB considers two cases. (i) For fixed-width data type (e.g., integer, decimal and floating point numbers), MonetDB uses a C-type array. (ii) For variable-width data types (e.g., strings), MonetDB adopts a dictionary encoding where the distinct values are then store in Blob and the BAT only stores an integer pointer to the Blob position. The BATs "name" and "age"'. Fig. 2.9 illustrate the BATs with variable-width and fixed-width types of tails, respectively. When the data is loaded from disk to main memory, MonetDB uses the exactly same data structure to represent such data on disk and in main memory. In addition, MonetDB adopts a late tuple reconstruction to save the main memory size. That is, during the entire query evaluation, all intermediate data are still the column format (i.e., the integer format instead of the actual values), and the tuples with actual values are finally reconstructed before sending the tuples to the client. In this approach a tree-based index is used to keep track of the growth of the array in any dimension and even allow adding of new dimensions. An extension of a *k*-dimensional array A along dimension *i* is viewed as appending a *k* dimensional subarray A^S to it along the *i*th dimension. The ranges of A^S are identical to those of A along each dimension except for dimension *i* whose range depends on the size of the extension. The length l_i, of dimension *i* is called as the range of dimension *i*.

2.2.8 Other Structures

Several array models have also been examined in order to verify their tremendous features of array. [30] shows a Rasdaman Array based query processor. Another Rasdaman array database that offers scalability is mentioned in [18]. A MonetDB structure for managing an information retrieval system has been anticipated by means of raw speed, light-weight data compression, and distributed execution in [31]. Another query processor based on column-oriented in-memory storage is mentioned in [32]. Array structure can also be used in scalable distributed system like Geoscientific Array mentioned in [33], NoSQL and

NewSQL [6] or distributed programming model like [8]. An array based parallel processing optimization has been described in [34].

2.3 Encoding Schemes for High Dimensional Data

Multidimensional array is the basic data structure used in many scientific or business applications where large volume is a main concern. But in many cases, it becomes crucial to attain useful information from the huge volume which are generally sparse din nature – i.e. many of the array cells contain null values and consume unnecessary space. So it is important to device a technique, "Encoding", to store deal with such array cells. Some common encoding schemes are reviewed here.

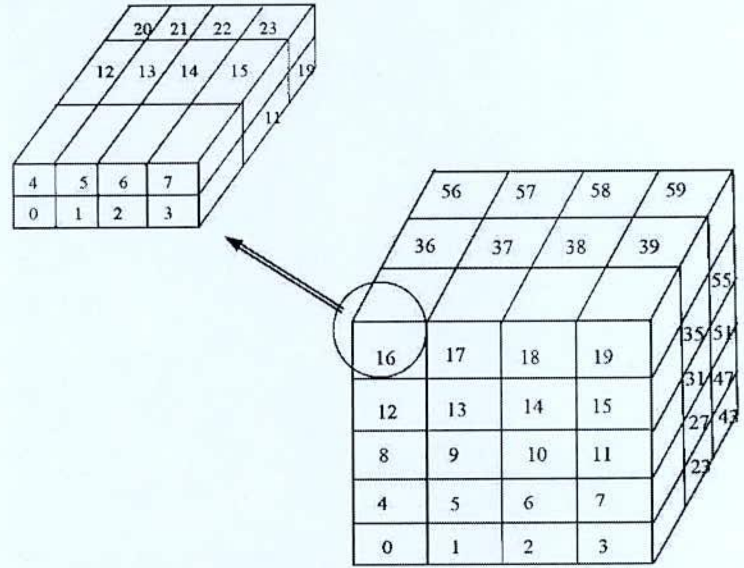


Figure 2.10: A 3-dimensional Array Partitioned into Chunks.

2.3.1. Chunk-Offset Encoding

To address the problems faced by applications that do not perform well with traditionally ordered arrays on disk, The data management libraries that support storage of multidimensional arrays on disk with the elements arranged in subarray chunks rather than in the traditional ordering is important. This allows efficient assembly of subarrays in multiple dimensions. In this scheme the large multidimensional arrays are broken into chunks for storage and processing. Consider an n dimensional array A, whose dimensionality is $d_1 \times d_2 \times d_3 \times \ldots \times d_n$, the chunks can be formed by breaking each d_i into several ranges. Within A, two positions are in the same chunk if and only if, in every

dimension, they fall within the same range. Fig. 2.10 shows a 3 dimensional array divided into sixty chunks (4×5×3) that are numbered in row-major fashion. Chunk 16 is itself 4×2×3 array whose 24 cells are numbered in row-major order and are stored contiguously. In chunk-offset encoding (COE) [35, 36], for each valid array entry, a pair (*OffsetInChunk*, *data*) is stored. The offset inside the chunk (*OffsetInChunk*) can be computed using the multidimensional array linearization function described before (see Eq. 2.1). Fig. 2.11(a) shows a 3 dimensional array partitioned into 36 chunks each of which is 3×3×3 (Fig. 2.11(b)). The details of a chunk with 8 data values and offset within the chunk are shown in Fig. 2.11(c), and Fig. 2.11(d) displays memory or disk arrangement of that chunk. Note that the chunks which have no nonempty elements are not physically allocated in the secondary storage.

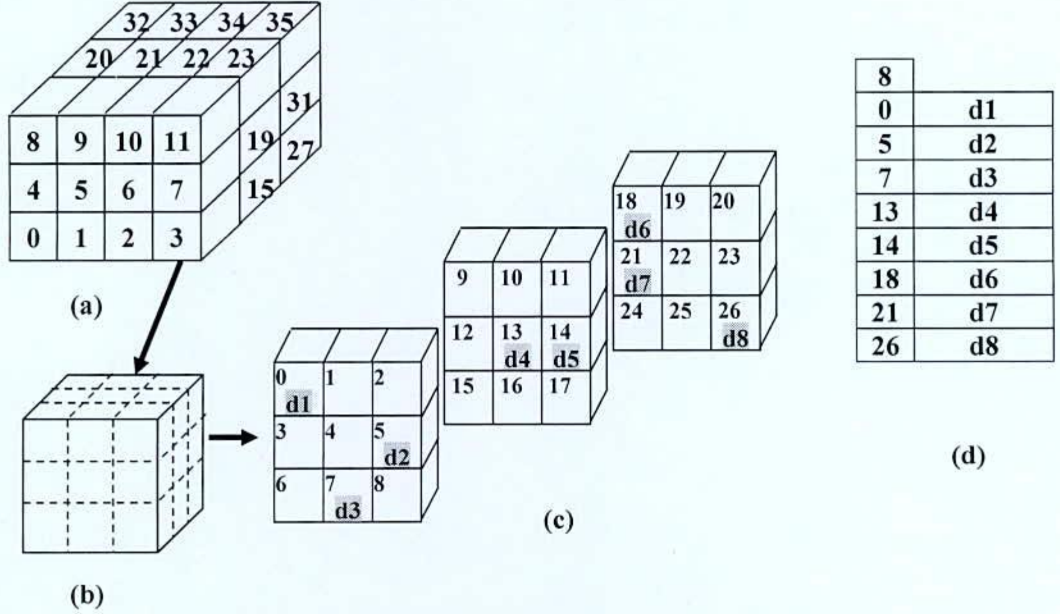


Figure 2.11: A 3-dimensional Array Stored As Chunk-Offset Encoding.

2.3.2. History-Offset Encoding

The History Offset Encoding (HOE) [16] scheme is based on Extendible Array (as sec. 2.2.2). In this technique, an element is specified using the pair of history value (h) and offset value (o) of the extendible array. Since a history value is unique in extendible array and has one to one correspondence with the corresponding subarray, the subarray including the specified element of an extendible array can be referred to uniquely by its corresponding history value h. Moreover, the offset value (i.e., logical location) of the

element in the subarray can be computed by using the addressing function and this is also unique in the subarray. Therefore, each element of an *n*-dimensional extendible array can be referenced by specifying the pair (history value, offset value). Like Chunk-offset compression, the extended sparse subarray elements are stored in memory in sorted fashion. Fig. 2.12 applies the HOE encoding on a 3 dimensional Extendible Array as mentioned in Fig. 2.2. The scheme omits the sparsed data and replaces the densed data with a tuple *t* = *<h, o>, Data>*, where offset is calculated from the addressing function (Eq. 2.1). For example, to store third data 42 in the SA number 8, the required tuple for HOE is *t* = *<8,2>,42>*.

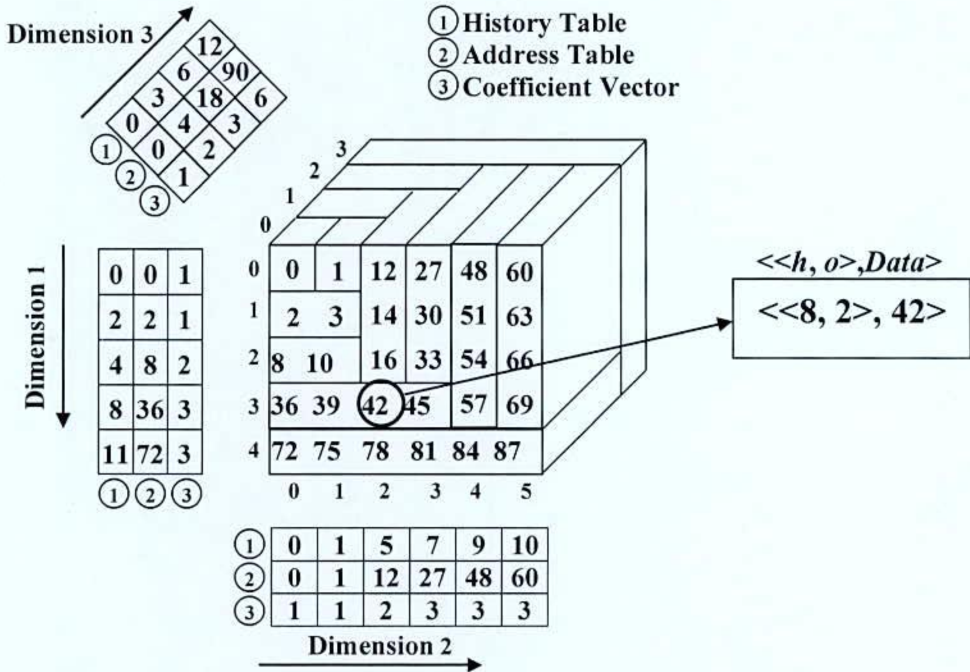


Figure 2.12: A 3-dimensional Extendible Array Encoded as History-Offset Encoding.

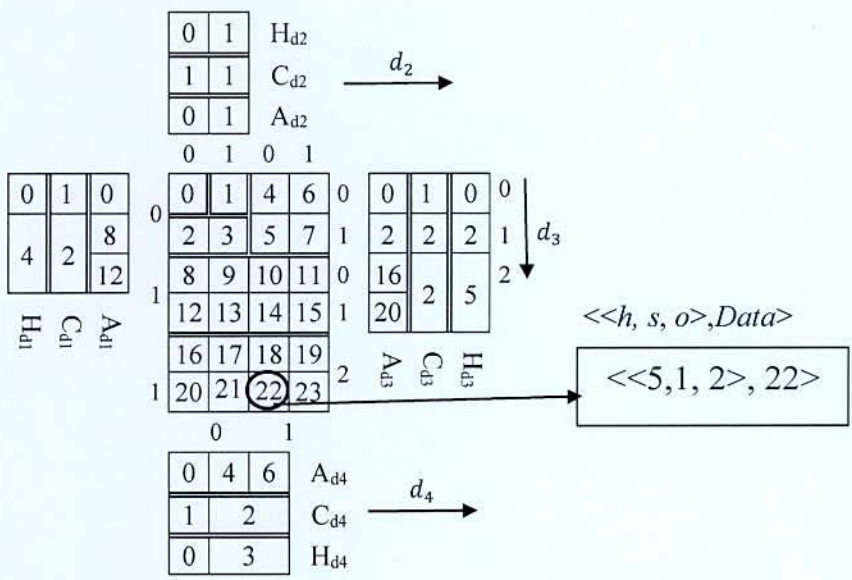


Figure 2.13: A 4-dimensional EKA Encoded as Segment-Offset Encoding.

2.3.3. Segment-Offset Encoding

The Segment Oriented (SOE) [37] Encoding scheme is based on EKA (as sec. 2.2.3). In this technique, an element is specified using the tuple of history value (*h*), segment number (*s*), and the offset (*o*) of the segment of the SA. The segment number is unique inside a SA. Here, history value is required to identify the SA, the segment number is required to identify the segment and the offset of segment is required to point cell position of the segment of the SA. Fig. 2.13 applies the SOE encoding on a 4 dimensional EKA as mentioned in Fig. 2.5. The scheme omits the sparsed data and replaces the densed data with a tuple t = <*h*, *s*, *o*>, *Data*>, where offset is calculated from the addressing function (see Eq. 2.1).

2.3.4. History-Pattern Encoding

The History Pattern Encoding (HPE) [38, 39] is a variant of the History Offset Encoding (as sec 2.3.2). Many of the tuple encoding schemes, including history-offset encoding, use the addressing function (Eq. 2.1) of a multidimensional array to compute the position. However, there are two problems inherent in such encodings. First, the size of an encoded result may exceed the machine word size (typically 64 bits) for large-scale datasets. Second, the time cost of encoding/decoding in tuple retrieval may be high; more

specifically, such operations require multiplication and division to compute the addressing function, and these arithmetic operations are expensive. To resolve these two problems without performance degradation the History Pattern Encoding (HPE) has been introduced. The scheme encodes an *n* dimensional tuple into a pair of scalar values <*history value*, *pattern*> even if *n* is sufficiently large. An encoded tuple can be a variable length record; the history value represents the extended subarray in which the tuple is included and also represents the bit size of the pattern. Additionally, the scheme does not employ the addressing function, hence avoiding multiply and divide instructions. Instead, it encodes and decodes tuples using only shift and and/or register instructions.

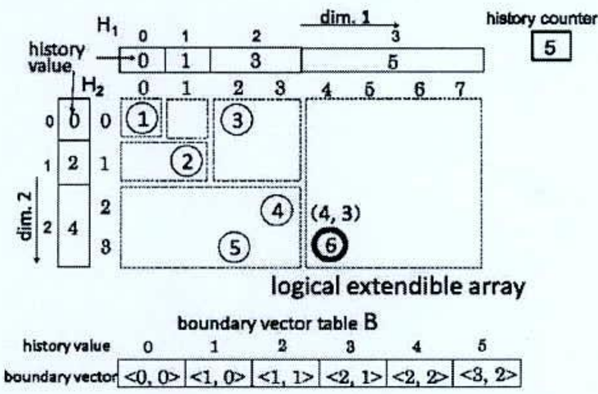


Figure 2.14: A 2-dimensional Representation of History Pattern Encoding.

Fig. 2.14 shows a 2 dimensional representation of a History Pattern Encoding. An *n* dimensional coordinate $X = (x_1, x_2, x_3, \dots x_n)$ can be encoded to the pair $< h, p >$ of history value *h* and bit pattern *p*. The history value *h* is determined as the maximum value in $\{H_k[b(x_k)] | 1 \leq k \leq n\}$, where$b(x_k)$ is the bit size of the subscript x_k in *X*. For each history value *h*, the boundary vector in B[*h*] gives the bit pattern size of each subscript in X. According to this boundary vector, the coordinate bit pattern *p* can be obtained by concatenating the subscript bit pattern of each dimension in descending order (from the lower to the higher bits of *p*). The storage unit for *p* can be one word length, i.e., 64 bits. Let, <*h*, *p*>= (4,3). $H_1[b(4)] = H_1[b(100_{(2)})] = H_1[3] = 5$ and $H_2[b(3)] = H_1[b(11_{(2)})] = H_2[2] = 4$. Since, $H_1[b(4)] > H_2[b(3)]$, h is $H_1[3] = 5$. So element (4,3) is known to be included in the subarray on dimension 1 at history value 5. Therefore, the boundary vector to be used is <3, 2> in B[5]. In (4, 3) to be encoded, the subscript 4 of the first dimension and the subscript 3 of the second dimension form the upper 3 bits and lower 2 bits of *p*, respectively. Therefore, *p* becomes $10011_{(2)} = 19$. Eventually, the

element (4, 3) is encoded to <5, 19>. Generally, the bit size of history value h is rather small compared to that of pattern p; if the storage size for the pair is assumed to be 16 bits, typically the upper 4 bits are for h, and the lower 12 bits are for p. Conversely, to decode the encoded pair <h, p> to the original n dimensional coordinate$X = (x_1, x_2, x_3, \dots x_n)$, the boundary vector in B[h] is known. Then, the subscript value of each dimension is sliced out from p according to the boundary vector. For example, consider the encoded pair $< h, p > = < 5, 19 >$. The boundary vector B[h] is $< 3, 2 >$ so $p = 10011_2$ can be divided into $100_2$and 11_2. Therefor, $< 5, 19 >$ can be decoded to the coordinate (4, 3).

2.3.5. Integer-Key Encoding

Integer-Key Encoding (IKE) [40] is an encoding scheme of integer keys in a B^+ tree index. They mainly focused in encoding 32 bit unsigned integers. Here, integers are differentially coded prior to encoding so that most of them are small. That is, starting from an array of integers$x_1, x_2, x_3, \dots$, they encoded the integers $x_1, x_2 - x_1, x_3 - x_2, \dots$. During decoding, given the differences $\delta_1 = x_1$, $\delta_2 = x_2 - x_1$, $\delta_3 = x_3 - x_2$, ... we need to reconstruct$x_1, x_2, x_3, \dots$. This operation requires the computation of a prefix sum$(\delta_1, \delta_1 + \delta_2, \delta_1 + \delta_2 + \delta_3, \dots)$. The B^+ tree node (also called a page) of IKE stores keys and values separately from each other. The actual in-memory layout is described in Fig. 2.15. Each node has a header structure of 32 bytes containing flags, a key counter, and pointers to the left and right siblings and to the child node. This header is followed by the KeyList (where we store the key data) and the RecordList (where we store the value's data).The RecordList of an internal node stores 64 bit pointers to child nodes, whereas the RecordList of a leaf node stores values or 64 bit pointers to external blobs if the values are too large. Fixed-length keys (Fig. 2.16) are always stored sequentially and without overhead. Variable-length keys (Fig. 2.17) use a small in-node index to manage the keys. Long keys are stored in separate blobs; the B^+ tree node then points to this blob.

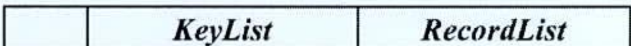


Figure 2.15: Memory Layout of B^+ Tree NodeOfIKE

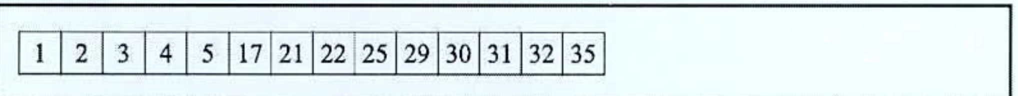


Figure 2.16: The Keylist of Fixed Length Integer Keys

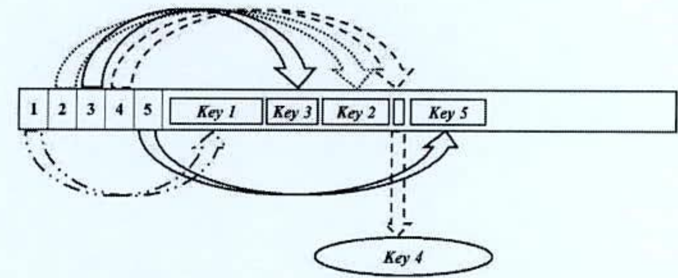


Figure 2.17: The KeyList of Variable Length Integer Keys

2.3.6. Other Schemes

Several encoding schemes have also been examined in the field data sparsity handling. Compressed Row Storage (CRS) and Compressed Column Storage (CCS) [36, 41] are used due to their simplicity and purity with a weak dependence relationship between array elements in a sparse array. It uses two one dimensional integer arrays RO and CO to compress all of the nonzero array elements along the rows (columns for CCS) of the sparse array. Array RO stores information about the non-zero array elements of each row and CO stores the column (row for CCS) indices of those elements (for two dimensional arrays). For higher dimensional sparse arrays more one dimensional integer arrays are needed. Hence compression ratio and range of usability become impractical for higher dimensional arrays. For an n dimensional extendible array, the EaCRS scheme requires $n - 1$ auxiliary arrays for each of the $(n - 1)$ dimensional subarray to compress it. Hence the compression ratio is not good enough for higher number of dimensions. A compression scheme, namely ECRS/ECCS for array model EKMR [15] is presented in [42]. The scheme is based on CRS/CCS [37, 42], and applied on EKMR. The EKMR represents n dimensional arrays by a set of two dimensional arrays. When applying the CRS/CCS scheme on EKMR the number of auxiliary arrays is always less. Hence compression ratio and range of usability become efficient. But the CRS/CCS and ECRS/ECCS schemes are applicable for statically allocated arrays. Encoding scheme based on history-offset parameter can also be obtained in [43 - 45]. Most of the index model mentioned above demand n dimensional indexing and requires n dimensional indices for run time calculation of cell position or cell offset.

2.4 Discussion

All the array models presented in this chapter have some pros and cons. Although the CMA is good for random accessing, it suffers from dynamic extension. The Traditional Extendible Array [13, 25], EKA [12], Axial Vector Extendible Array [11] are good for dynamic extension. But they all have a concept of SA which is always $(n-1)$ dimensional and requires n dimensional indexing. For large value of length for each dimension or for large number of dimension value of offset grows exponentially and overflows the address space.

Typical encoding schemes have some limitations in compressing data. The scheme Compressed Row Storage (CRS) [41] or Chunk Offset Encoding [35, 36] are effective for encoding large sparse arrays. But still they cannot be applied on extendible databases. The dynamic models like HOE [16], SOE [37], HPE [38, 39] etc. can improve the performance of an encoding scheme compared to static models, but they require handling the dimension value n.

In this circumstances, we propose a dynamic scalable array model which will outperform the static models like CMA as well as dynamic models like Traditional Extendible Array or Axial Vector Extendible Array. We also provide an encoding scheme based on our scalable structure. The detail of the proposed structure and encoding scheme is presented in the next chapter.

CHAPTER III

Scalable Storage Systems for Higher Order Index Array

3.1. Introduction

Array based storage and retrieval systems are demanded in many high dimensional systems like Big data for their easy maintenance. However, the lack of scalability of the conventional approaches degrades with the dynamic size of data sets as they entail reallocation in order to preserve expanded data velocity. To maintain the velocity of data, the storage system must be scalable enough by allowing subjective expansion on the boundary of array dimension. The index array offers a dynamic storage scheme for preserving expanded data velocity by employing indices for each dimension. Again, for an array based storage system, if the number of dimension and length of each dimension of the array is very high then the required address space overflows and hence it is impossible to allocate such a big array in the memory. The Index Array model is good compared to the CMA, but it cannot meet the expected memory utilization as per the demand of data velocity especially for "Big Data" applications [17] and has to face the following problems:

i. To represent the large dimensionality of data.

ii. To lessen indexing cost as it requires indices for each dimension for preserving expanded data velocity.

iii. To avoid address space overflow even though resource is available.

iv. To conquer significant information from the large volume which has data sparsity.

Problem (i) is an issue as easy representation makes data more meaningful for computer analysis and user interpretation. Nevertheless, an improper data representation will reduce the value of the original data and may even obstruct effective data analysis. Efficient data representation shall reflect data structure, class, and type, as well as integrated technologies, so as to enable efficient operations on different datasets. [23]. Though indexing process converts a static CMA to a dynamic Index Array by adding scalability, but for problem (ii) it has exponentially increasing indexing cost. The increase in number of dimension of the

array causes increase in indexing cost and thus reduces the performance of an Index Array. The problem (iii) involves address space overflow. The range in which the linearized array elements map is called address space – which depends on the length or number of dimension of array. For an array based storage system, if the number of dimension and length of each dimension of the array is very high then the required address space overflows and hence it is impossible to allocate such a big array in the memory. In case of address space, the Index Array model is good compared to the CMA, but it cannot fully utilize the available resources because of address space overflow. Again, we have problem (iv) which can decreases the storage utilization of large volume applications.

Thus special computing techniques through comprehensive research to handle large scale higher dimensional data efficiently and effectively are cramming needs to data scientists. It emphasizes the new organization and implementation schemes on higher dimensional data. In this chapter we have explained two new scalable index structures that can enhance the capabilities of an Index Array. The first one is named as Scalable Array Indexing (SAI) that transform the n dimensions of an array into 2 dimension which reduces indexing cost and improves data locality of an Indexed Array. However, like an Index Array it also suffers from address space overflow. Hence, we modify SAI structure and renamed as Segment based Scalable Array Indexing (SSAI) which segmentify the subarrays (SAs). We also provide an encoding scheme based on SSAI and named as 2 Dimensional Key Value Encoding (2DKVE).

3.2. Realization of a Scalable Array Indexing (SAI)

The SAI structure converts an n dimensional array with row-column abstraction [27]. Odd dimensions contribute along row direction and even dimensions along column direction which gives lower cost of index computation and higher data locality. It is a permutation on higher dimensional array to fit into a new 2 dimensional array. Thus the length and indices of new 2 dimensional array is determined based on n dimensional arrays' length and indices.

3.2.1. Dimension Conversion

Consider an n dimensional Conventional Multidimensional Array (CMA(n)). Let, the size of a CMA(n) or A[l_1] [l_2]… [l_n] is [$l_1, l_2, \ldots, l_n$]. Then $<x_1, x_2, \ldots, x_n>$ be the Real n dimensional Index and denoted as RnI; where $l_1, l_2, \ldots, l_n$ is the length of each dimension $d_1, d_2, \ldots, d_n$

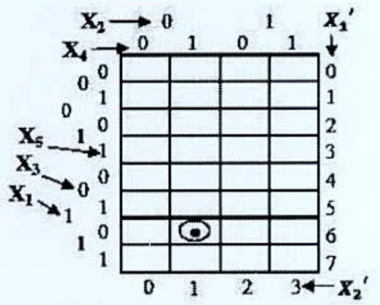


Fig. 3.1: Transformation of a CMA[2, 2, 2, 2, 2] to a SAI[8,4]

respectively and $x_i = 0,1,2, \ldots, (l_i-1)$ $(0 \leq i \leq n)$. Among the n dimensions of CMA(n), $\left\lceil \frac{n}{2} \right\rceil$ number of odd dimensions fit along row direction and the rest $\frac{n}{2}$ number of even dimensions along column direction. We convert the n dimensions of a CMA(n) into two dimensions of a new structure named as Converted 2 dimensional Array (C2A) $A'[l_1'][l_2']$ of size $[l_1', l_2']$ and the Converted 2 dimensional Index denoted as C2I becomes $<x_1', x_2'>$ where l_1' and l_2' are the length of dimension d_1' (row dimension) and d_2'(column dimension) respectively; $x_1' = 0, 1, 2,.., (l_1'-1)$ and $x_2' = 0, 1, 2,.., (l_2'-1)$. The mapping function that converts $<x_1, x_2, \ldots, x_n>$ to $<x_1', x_2'>$ is as follows:

$$x_1' = x_1 l_3 l_5 \ldots l_{n-3} l_r + x_3 l_5 \ldots l_{n-3} l_r + \ldots + x_r$$
$$x_2' = x_2 l_4 l_6 \ldots l_{n-3} l_c + x_4 l_6 \ldots l_{n-3} l_c + \ldots + x_c \quad (3.1)$$

Where

$$r = \begin{cases} n-1, \text{if } n \text{ is even} \\ n, \quad \text{if } n \text{ is odd} \end{cases} \text{ and } c = \begin{cases} n-1, \text{if } n \text{ is odd} \\ n, \quad \text{if } n \text{ is even} \end{cases}$$

Hence the index computing function of CMA(n) becomes

$$f(x_1', x_2') = \begin{cases} x_1' \times l_2' + x_2', \text{ if } d_1' \text{holds the SA} \\ x_2' \times l_1' + x_1', \text{ if } d_2' \text{holds the SA} \end{cases} \quad (3.2)$$

Where,

$$l_1' = l_1 \times l_3 \times \ldots \times l_r$$
$$l_2' = l_2 \times l_4 \times \ldots \times l_c$$

Example 3.1. Fig. 3.1 shows a SAI of a CMA(5) of size [2, 2, 2, 2, 2]. The RnI index < 1, 0, 1, 1, 0> is converted to C2I by <6, 1> and $l_1' = 8$ and $l_2' = 4$. The dynamic extension for any dimension of the CMA(n) causes corresponding extension on row or column direction of C2A.

3.2.2. Scalable Indexing

Scalability is an important property to store present and future dataset for big data storage [22]. The index array model namely extendible array inherits scalability through a process called indexing which preserves the dynamic extension of array [11-13]. Indexing is a process of monitoring location of data record by assigning a key with them for the corresponding system. The process assist in fast query processing [11]. The indexing of a SAI system is done by introducing five types of *Supplementary Tables* (ST) which help the SAI in managing the scalability as well as faster accessing of the structure.

- *History Table* (HT): The HT table stores a unique number to monitor the construction history of the SA.
- The *Index Table* (IT): The dynamic extension can occur in any dimension of the CMA(n). The start index of the corresponding extended dimension is stored in index table.
- *Extend Dimension* (EDT): The SAI structure is a compressed dimension representation of an n dimensional array. Hence, to track the current extended dimension EDT is required. It tracks the scalable direction by assigning value 1 to n.
- *Multiplicative Coefficient Table* (MCT): The MCT stores the co-efficient of the addressing function. As the new SAI is a 2 dimensional structure, hence MCT stores the co-efficient of the new index x_1'or x_2'(Eq.1).
- *Address Table* (AT): The first address of the dynamically allocated subarray is saved in AT. This is mainly useful when the allocated memory is not consecutive. For consecutive memory allocation we can avoid AT.

The supplementary tables are the indices of the structure. For each of the 2 dimensions the indices are necessary. Let the indices are ST_1 (for row direction) and ST_2 (for column direction). Each of the index entry requires above tupple namely *<history value, index, first address, coefficient vector, extended dimension>*. Extendible Array [13] uses 3 tupples and Extendible Array [11] uses 4 tupples respectively. But they need n indices to be placed for each dimension. By increasing one entry in the tupple we reduced the total number of indices to 2 only.

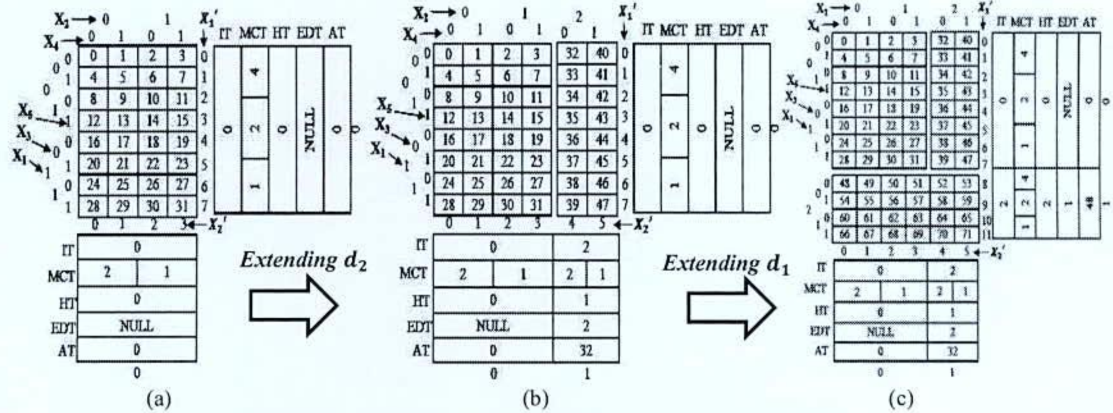


Fig. 3.2: Construction and Extension of a SAI System

3.3. Operations on a SAI System

3.3.1. Construction and Extension

To construct a SAI, the RnI of a CMA is converted to its corresponding C2I pf a C2A (as sec 3.2.1). The supplementary tables of both dimensions (namely ST_1 and ST_2) are maintained. The first address of the allocated memory is preserved in AT. The extension history i.e the value of history counter is initialized and saved in history table, HT. The initial index values of the both dimensions are stored in IT. The coefficients of the odd dimensions are stored in the ST_1.MCT and the coefficients of the even dimensions are stored in the ST_2. MCT. The EDT holds null as no extension is held yet.

Example 3.2: Consider the transformation of a CMA(5) of size[2,2,2,2,2] in Fig. 3.2(a). The row dimension is constructed from [l_1, l_3, l_5] and the column dimension is constructed from [l_2, l_4]. Initially, set ST_1[0].IT = 0 , ST_1[0].HT = 0 and ST_1[0].EDT = NULL. The cell values also represent their cell position in the actual array. Hence, set ST_1[0].AT = 0. MCT values are initialized to ST_1[0].MCT[0]= $l_3 \times l_5$=4 , ST_1[1].MCT[1]= l_5 =2 and ST_1[1].MCT[2]=1 for x_1' and ST_2[0].MCT[0]=l_4=2 and ST_2[0].MCT[1]=1 for x_2'. ST_2[0].IT=0, ST_2[0].HT=0, ST_2[0].EDT=NULL and ST_2[0].AT=0. And finally A[2][2][2][2][2] of CMA is converted to a A'[8][4] of SAI.

The dynamic extension along any arbitrary direction d_i' of the SAI is done by allotting a block of memory or SA (Eq. 2.2).

Example 3.3. Let, the structure in Fig. 3.2(a) has been expanded in dimension d_2 shown in Fig. 3.2(b). In this case, the size of the CMA(5) becomes [2,3,2,2,2]. Then a block of

memory size $(2 \times 2 \times 2 \times 2)$ or 16 (see Eq. 2.2) is allotted dynamically. As the dynamic extension is done along d_2 that contributes for x_2', hence ST_2 is maintained. The first address of the memory block (i.e 32) is stored in $ST_2[1]$.AT. The history counter is incremented and stored in $ST_2[1]$.HT. The extended index of d_2 (i.e 2) is stored in $ST_2[1]$.IT. The value of the extended dimension (i.e 2) is hold by $ST_2[1]$.EDT. Finally the multiplicative coefficient l_4 and 1 is stored in $ST_2[1]$.MCT[0] and $ST_2[1]$.MCT[1] respectively. Fig. 3.2(c) depicts an extension along d_1. The supplementary tables are maintained for ST_1. Finally Fig. 3.3 shows the SAI after extending on d_4, d_3 and d_5 respectively.

3.3.2. Dimension Transformation

The operation of dimension transformation has been divided into two parts. The first part will elaborate the transformation from *n* dimension to 2 dimension named as *Forward Transformation*. And the second part will elaborate the transformation from 2 dimension to *n* dimension named as *Backward Transformation*.

Forward Transformation. Let the subscript of CMA $(x_1, x_2, x_3, x_4, \ldots x_n)$ is to be transformed into 2 dimension of SAI or $< x_1', x_2' >$. The subscripts that contribute to row and column direction are calculated (section 3.1). Let, the row subscripts be $(x_1, x_3, x_5, \ldots x_r)$ and even subscript be $(x_2, x_4, x_6, \ldots x_c)$. Let $max()$ returns the maximum value

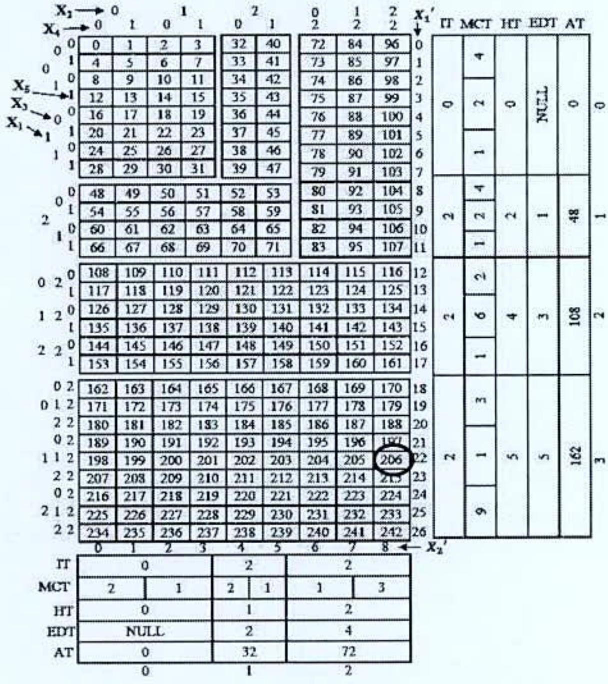


Fig. 3.3: Realization of a SAI[27,9]

and $fmax()$ returns the count of the maximum value. Now find the following

$x_\alpha = max(x_1, x_3, \ldots x_r)$ and $m_r = fmax(x_1, x_3, \ldots x_r)$

And

$x_\beta = max(x_2, x_4, \ldots x_c)$ and $m_c = fmax(x_2, x_4, \ldots x_c)$

Where x_α is the maximum subscript in the row direction and m_r is the count of the subscript that have maximum subscript value. We need to find i and j subscript from ST_1 and ST_2 respectively to locate the candidate subarray that contains the element. To find i from ST_1, there can be two cases based on m_r

Case 1: if $m_r = 1$, find i such that $ST_1[i]$.IT=x_α and $ST_1[i]$.EDT = α

Case 2: if $m_r > 1$, $m_r = a$ (say). Let $i_1, i_2, \ldots, i_a$ contains x_α. Among $i_1, i_2, \ldots, i_a$ find $u \in (i_1, i_2, \ldots, i_k)$ $(k \leq a)$ such that $ST_1[u]$.IT=x_α and $ST_1[u]$.EDT = α.

Now from $i_1, i_2, \ldots, i_k$ find $h_{max} = \max(ST_1[i_1].HT, ST_1[i_2].HT, \ldots, ST_1[i_k].HT)$. Find i such that $h_{max} = ST_1[i].HT$

To find j from ST_2, there can be similar cases based on m_c

Case 1: if $m_c = 1$, find j such that $ST_2[j]$.IT=x_β and $ST_2[j]$.EDT = β

Case 2: if $m_c > 1$, $m_c = b$ (say). Let $j_1, j_2, \ldots, j_b$ contains x_β. Among $j_1, j_2, \ldots, j_b$ find $u \in (j_1, j_2, \ldots, j_k)$, $(k \leq b)$ such that $ST_2[u]$.IT=x_β and $ST_2[u]$.EDT = β. Now from $j_1, j_2, \ldots, j_k$ find $h_{max} = \max(ST_2[j_1].HT, ST_2[j_2].HT, \ldots, ST_2[j_k].HT)$ Find j such that $h_{max} = ST_2[j].HT$

Using i and j, x_1', x_2' can be re-calculated using Eq. 3.1 as follows:

$$x_1' = x_1 ST_1[i].MCT[0] + x_3 ST_1[i].MCT[1] + .. + x_r ST_1[i].MCT\left[\left\lceil\frac{n}{2}\right\rceil - 1\right]$$

$$x_2' = x_2 ST_2[j].MCT[0] + x_4 ST_2[j].MCT[1] + .. + x_c ST_2[j].MCT\left[\frac{n}{2} - 1\right]$$

Where, MCT[0] is the first multiplicative coefficient and so on (as Fig. 3.3).

Example 3.4. Consider an input (x_1, x_2, x_3, x_4, x_5) = (1, 2, 1, 2, 2) is to be retrieved from Fig. 3.3. So, row index is (x_1, x_3, x_5) = (1, 1, 2) and column index is (x_2, x_4) = (2, 2). For row index we have maximum index value X_α = max(x_1, x_3, x_5) = max(1, 1, 2) = 2 and $fmax(x_1, x_3, x_5) = m = 1$. So, we select ST_1 index i = 3 (as $ST_1[3]$.IT = 2 and

$\mathrm{ST_1[3].EDT} = 5$)(i.e case 1). For column index, we have $X_\beta = \max(x_2, x_4) = \max(2, 2) = 2$ and $fmax(2,2) = \mathrm{m} = 2$. Here, we select $\mathrm{ST_2}$ index $j_1 = 1$ and $j_2 = 2$ as $\mathrm{ST_2[1].IT} = 2$ and $\mathrm{ST_2[1].EDT} = 2$, $\mathrm{ST_2[2].IT} = 2$ and $\mathrm{ST_2[2].EDT} = 4$. Now, as $\mathrm{ST_2[1].HT} < \mathrm{ST_2[2].HT}$ or $(1 < 3)$, hence $j = 2$ for $\mathrm{ST_2}$. And the converted 2D indices x_1', x_2' are as follows

$$x_1' = \mathrm{x_1} \times \mathrm{ST_1[3].MCT[0]} + \mathrm{x_3} \times \mathrm{ST_1[3].MCT[1]} + \mathrm{x_5} \times \mathrm{ST_1[3].MCT[2]}$$
$$= 1 \times 3 + 1 \times 1 + 2 \times 9 = 22$$

$$x_2' = \mathrm{x_2} \times \mathrm{ST_2[2].MCT[0]} + \mathrm{x_4} \times \mathrm{ST_2[2].MCT[1]} = 2 \times 1 + 2 \times 3 = 8$$

So, 5D (1, 2, 1, 2, 2) is equivalent to 2D (22, 8).

Backward Transformation. Let the subscript of SAI $< x_1', x_2' >$ is to be transformed into n dimension of CMA or $(x_1, x_2, x_3, x_4, \ldots x_n)$. Let i and j are the indices of $\mathrm{ST_1}$ on $\mathrm{ST_2}$ for the selected SA respectively. To determine the value of i and j, find largest i and j such that

$$x_1' \geq \mathrm{ST_1}[i].\mathrm{IT} \times \mathrm{ST_1}[i].MCT_{max}$$
$$x_2' \geq \mathrm{ST_2}[j].\mathrm{IT} \times \mathrm{ST_2}[j].MCT_{max} \tag{3.3}$$

Now, let there are p number of entries in $ST_1[i].MCT$ and q number of entries in $ST_2[j].MCT$. Thus the row indices $(x_1, x_3, x_5, \ldots, x_r)$ and column indices $(x_2, x_4, x_6, \ldots x_c)$ can be calculated using the following equation:

$$x_o = \frac{\left(\left(\left(x_1'\%MC_{1max}\right)\%MC_{2max}\right)..\%MC_{(p-1)max}\right)}{MC_o}$$
$$x_e = \frac{\left(\left(\left(x_2'\%MC_{1max}\right)\%MC_{2max}\right)..\%MC_{(q-1)max}\right)}{MC_e} \tag{3.4}$$

Here, x_o represents the odd indices $(x_1, x_3, x_5, \ldots x_r)$ and x_e represents the even indices $(x_2, x_4, x_6, \ldots x_c)$. MC_{1max} is the first largest Multiplicative Coefficient, MC_{2max} is the second largest Multiplicative Coefficient and $MC_{(p-1)max}$ is the last largest Multiplicative Coefficient before x_o's coefficient, where MC_o is the Multiplicative Coefficient of x_o.

Example 3.5. Let $<x_1', x_2'>$ be $<22, 8>$. As $x_1' \geq \mathrm{ST_1[3].IT} \times \mathrm{ST_1[3]}.MCT_{max}$ or $22 \geq 2 \times 9$, hence $i = 3$ and $x_2' \geq \mathrm{ST_2[2].IT} \times \mathrm{ST_2[2]}.MCT_{max}$ or $8 \geq 2 \times 3$, hence $j = 2$. Now, we have *three* entries in $ST_1[3].MCT$. So, using Eq. 8 the row indices are as follows:

$$x_1 = (x_1'\%MC_{1max})/MC_1 = (22\%9)/3 = 1$$
$$x_3 = \left((x_1'\%MC_{1max})\%MC_{2max}\right)/MC_3 = \left((22\%9)\%3\right)/1 = 1$$

$$x_5 = x_1'/MC_5 = 22/9 = 2$$

We have two entries in $ST_2[1].MCT$, hence using Eq. 8 the column indices are as follows:

$$x_2 = (x_2'\%MC_{1max})/MC_2 = (8\%3)/1 = 2$$

$$x_4 = x_2'/MC_4 = 8/3 = 2$$

Finally $< x_1', x_2' > = <22, 8>$ maps to $(x_1, x_2, x_3, x_4, x_5) = (1, 2, 1, 2, 2)$.

3.3.3. Point Query

Point query is a form of data query, where all the subscripts of all the domains are known. In our proposed model the input of a *Point Query* is an *n* dimensional index RnI of form $(x_1, x_2, x_3, x_4, \ldots x_n)$ and output is an array cell value (VALUE) representing a memory cell (CELL).

The first task of a point query is to generate the 2 dimensional index C2I form the given *n* dimensional index RnI (see sec 3.2.3.2). Using *i* and *j* of supplementary table ST_1 and ST_2 (respectively), find H_{max} =($ST_1[i]$.HT, $ST_2[j]$.HT). If H_{max} = $ST_1[i]$.HT, then d_1' is the SA direction that contains the desired element. The SAs can store consecutive memory block or non-consecutive memory block. If the SAs are in consecutive memory, then the value can be calculated using Eq. 3.2 as follows

$$\text{VALUE} = f(x_1', x_2') = \begin{cases} x_1' \times l_2' + x_2', & \text{if } d_1' \text{holds the SA} \\ x_2' \times l_1' + x_1', & \text{if } d_2' \text{holds the SA} \end{cases}$$

If the SAs are non-consecutive, then, the required cell position CELL in the candidate SA is

$$\text{CELL} = f(x_1', x_2') - \begin{cases} \text{ST}_1[i].\text{AT, when SA exists on } d_1' \\ \text{ST}_2[j].\text{AT, when SA exists on } d_2' \end{cases} \quad (3.5)$$

And the required cell value is

$$\text{VALUE} = \text{CELL} + \begin{cases} \text{ST}_1[i].\text{AT, when SA exists on } d_1' \\ \text{ST}_2[j].\text{AT, when SA exists on } d_2' \end{cases} \quad (3.6)$$

Example 3.6. Consider an RnI input $(x_1, x_2, x_3, x_4, x_5) = (1, 2, 1, 2, 2)$ is to be retrieved from Fig.3.3. The corresponding C2I is (22, 8). Here, $i = 3$ and $j = 2$. Now, as $\text{ST}_2[2].\text{HT} < \text{ST}_1[3].\text{HT}$ or $(3 < 5)$, hence d_1' holds the SA. If the SAs are consecutive then

$$\text{VALUE} = x_1' \times l_2' + x_2' = 22 \times 9 + 8 = 206$$

And if the SAs are not consecutive then

$$\text{CELL} = f(x_1', x_2') - \text{ST}_1[i].\text{AT} = x_1' \times l_2' + x_2' - \text{ST}_1[i].\text{AT} = 22 \times 9 + 8 - 162 = 44$$

And resultant cell value is

$$\text{VALUE} = \text{ST}_1[i].\text{AT} + \text{CELL} = 162 + 44 = 206$$

The resulted cell is marked in Fig. 3.3.

3.3.4. Range Query

A range key query [51, 52] has a single predicate of the form (column subscript < value) or (column subscript > value) or (column subscript between value1 and value2). On the other hand, for a single key query predicate has the form (column subscript = value). So we can say that single key query is a special case of range key query with only a single range subscript. The rest subscripts are denoted by the sign '*' or don't care situation. In our proposed 2D model, we have two types of dimensions. The first one is named as major dimension if the first SA selected by the given key corresponds to the same dimension as the SA dimension. And if the first SA selected by the given key corresponds to the opposite dimension as the SA dimension, then it is called minor dimension. The required code segments for the single range key query is as follows:

```
for( i = START; i <= END; i = i + step )
{
      for( j = 0; j < total_data; j++ )
        {
           if( major dimension )
                 pos = i + j;
           else
                 pos = i + TARGET[j];
           retrieve → SA[pos];
        }
   }
```

The first task of the query is to find the first SA and corresponding history that contains the key (see se. 3.3.3). Then the SA is loaded from disk to main memory. Here, `START` is the position of the SA that initiates the query and `END` $(sz - 1)$ is the position of the SA that terminates the query on that SA. The `step` is the step size for the required search. The `target_cells` is the number of target cells or offsets for the required key, `target_indices` are the

number of major indices selected for the key and `TARGET` holds the selected major indices for using on minor dimension. The number of successive data block to retrieve is `NOD` and the total number of data to retrieve is `total_data = NOD × target_cells`. The `pos` generates the offset of the SA. The task of the function `retrieve` is to generate the cell value `SA[pos]` of the SA. The key is k on the index position x_α with length l_α and multiplicative co-efficient MCT_α and maximum multiplicative coefficient is MCT_{max}.

Single Range Query on Major Dimension. For major dimension, if d_1' holds the SA then all the x_2' of is the target cell or `target_cells` $= x_2'$ where $0 \leq x_2' < l_2'$ otherwise `target_cells` $= x_1'$ where $0 \leq x_1' < l_1'$ (see Eq. 3.2). Let, the major dimension is on d_1'. Now, there arise the following two cases:

Case 1. If $x_\alpha = ST_1[i].EDT$ and $k = ST_1[i].IT$, then the whole size of the SA or *sz* (Eq. 2.2) is retrieved and `NOD = step = sz`. Hence, `START = 0` and `END = sz - 1`. `target_indices` $= MCT_\alpha$, start index of the SA is `SI` $= ST_1[i].IT \times MCT_\alpha$ and end index of the SA is `EI = SI +` MCT_α`-1`. Hence, `SI` $\leq$ `TARGET` $\leq$ `EI`.

Case 2. If $x_\alpha \neq ST_1[i].EDT$, then `NOD` $= MCT_\alpha$, `total_data = NOD ×` `target_cells` $= MCT_\alpha \times l_2'$, `step = total_data` $\times l_\alpha$, `START` $= k \times$ `total_data` and `END = sz - 1`, `SI` $= ST_1[i].IT \times MCT_{max} + k \times MCT_\alpha$, `EI = SI +` MCT_α`-1`. Hence, `SI` $\leq$ `TARGET` $\leq$ `EI`. On each `step` the `SI` (`SI = EI + NOD + 1`) and thus `EI` is updated until reach `END`. Hence, `TARGET` is updated.

Single Range Query on Minor Dimension. For minor dimension, all the selected target indices from the first major dimension to the last will be the candidate target cells. That means, if the first major dimension is on history h_1, minor dimension is on h_{minor} and the last major dimension is on history $h_2 < h_{minor}$. Then all the target indices from h_1 to h_2 will be target cells for minor dimension. Here, `NOD = 1`, `START = 0` and `END = sz - 1`, and `step` $= l_{major}' = l_1'$, where l_{major}' is the length of major dimension, `target_cells = target_indices`.

Example 3.7. Let, in Fig. 3.4, the data to retrieve is (*, 1, *,*,*). Here, $k = 1$, $\alpha = 3$. The requested query has two major candidates (case. 2) in the first SA or SA1 ($h = 0$); two minor candidates in the second SA, SA2 ($h = 1$) and one major candidate in the third SA, SA3 ($h = 2$). For SA1: $l_\alpha = 2, l_{minor}' = 4, MCT_\alpha = 2$, `target_cells` $= l_{minor}' = l_2' =$

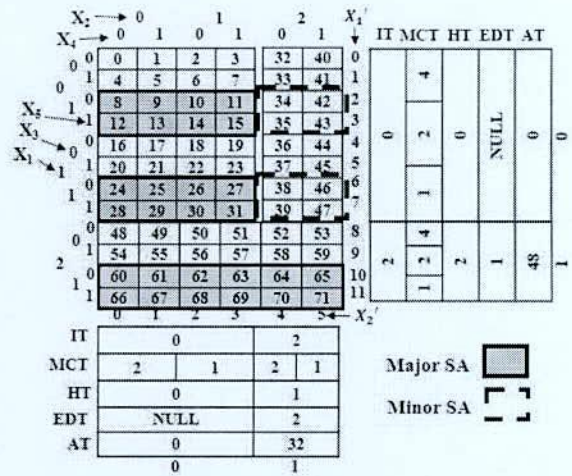


Fig. 3.4: Range Key Operation on SAI

4, `NOD` = $MCT_\alpha = 2 = 2$, `total_data` = `NOD` × `target_cells` = $2 \times 4 = 8$, `step` = `total_data` $\times l_\alpha = 8 \times 2 = 16$, $END = sz - 1 = 31$, now `i` = `START` = $k \times$ `total_data` = 1×8 = 8, retrieve successive positions from SA[i] = SA[0] = 8 to SA[i + totaldata -1] = SA[7] = 15, `SI` = $ST_1[i].IT \times MCT_{max} + k \times MCT_\alpha = 0 \times 4 + 1 \times 2 = 2$, `EI` = `SI` + MCT_α-1 = 2 + 2 – 1 = 3 and `TARGET` = {2, 3}; now `i` = `i` + `step` = $8 + 16$ = 24, retrieve successive positions from SA[i] = 24 to SA[totaldata -1] = 32, `SI` = `EI` + `NOD` + 1 = 3 + 2 + 1 = 6, `EI` = `SI` + MCT_α-1 = 6 + 2 – 1 = 7 and `TARGET` = {2, 3, 6, 7}, `target_indices` = 4; For SA2: `START` = 0, `END` = sz – 1 = 16 – 1 = 15, `NOD` = 1, `step` = $l_{major}' = l_1' = 8$ and `target_cells` = `target_indices` = 4, `total_data` = `NOD` × `target_cells` = 1 × 4 = 4, now `i` = `START` = 0 and retrieve positions from SA[i + 2] = SA[2] = 34, SA[i + 3] = SA[3] = 35, SA[i + 6] = SA[6] = 38 and SA[i + 7] = SA[7] = 39; now `i` = `START` + `step` = 0 + 8 = 8 and retrieve positions from SA[i + 2] = SA[10] = 42, SA[i + 3] = SA[11] = 35, SA[i + 6] = SA[14] = 38 and SA[i + 7] = SA[15] = 39. For SA3: $l_\alpha = 2, l_{minor}' = 6, MCT_\alpha = 2$, `target_cells` = $l_{minor}' = l_2' = 6$, `NOD` = $MCT_\alpha = 2 = 2$, `total_data` = `NOD` × `target_cells` = $2 \times 6 = 12$, `step` = `total_data` $\times l_\alpha = 12 \times 2 = 24$, $END = sz - 1 = 24$, now `i` = `START` = $k \times$ `total_data` = 1×12 = 12, retrieve successive positions from SA[i] = SA[12] = 60 to SA[i + totaldata -1] = SA[23]= 71, `SI` = $ST_1[i].IT \times MCT_{max} + k \times MCT_\alpha = 2 \times 4 + 1 \times 2 = 10$, `EI` = `SI` + MCT_α-1 = 10 + 2 – 1 = 11 and `target_indices` = 6, `TARGET` = {2, 3, 6, 7, 10, 11}.

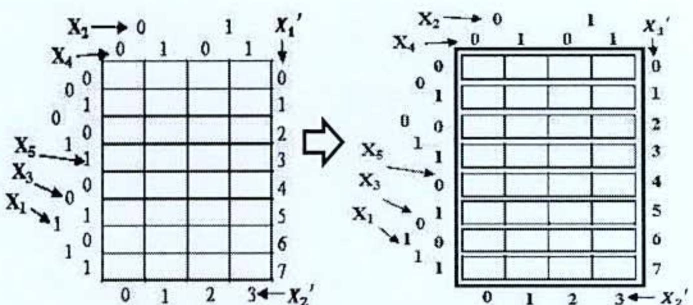


Fig. 3.5: Segmentation of a SAI to a SSAI

3.4. Realization of a Segment based Scalable Array Indexing (SSAI)

The Segment based Scalable Array Indexing (SSAI) is a segmentation of the SA of a SAI system. That means the SSAI is a segment based scalable array storage that also transforms an *n* dimensional (*n*D) array into 2 dimensional (2D) array (as sec 3.2.1). The SSAI replaces the SA block memory allocation of a SAI system by small segments. Thus, it not only delivers lower index computation cost and higher data locality but also delay the address space overflow which provides high storage utilization. For scalable indexing of the SSAI scheme, the same supplementary tables (as sec 3.2.2) as SAI are used except for AT entry which stores the first address of the first segment of dynamically allocated SA.

3.4.1. Segmentation

The SSAI divides the SA into a collection of segments. Since the SSAI is a 2D structure, the SA is 1D, hence the segment size becomes the length of opposite dimension of SSAI. For example, if the row dimension d_1' holds the SA then the size of the segment is the length of column dimension or l_2'. If the SA size is *sz* (Eq. 2.2) and segment size is *SG_SZ*, then the number of segment (*nos*) in a SA is calculated using the following equation

$$nos = \frac{sz}{SG_SZ},\ SG_SZ = \begin{cases} l_2', \text{when SA exists on } d_1' \\ l_1', \text{when SA exists on } d_2' \end{cases} \tag{3.7}$$

Example 3.8. Fig. 3.5 shows the segmentation of Fig. 3.1. The SA size is 32 and d_1' holds the SA. As l_2'= 4, the *nos* is $\frac{32}{4}$ or 8. So, 8 segments of size 4 have been allocated for the current SA as mentioned in Fig. 3.5.

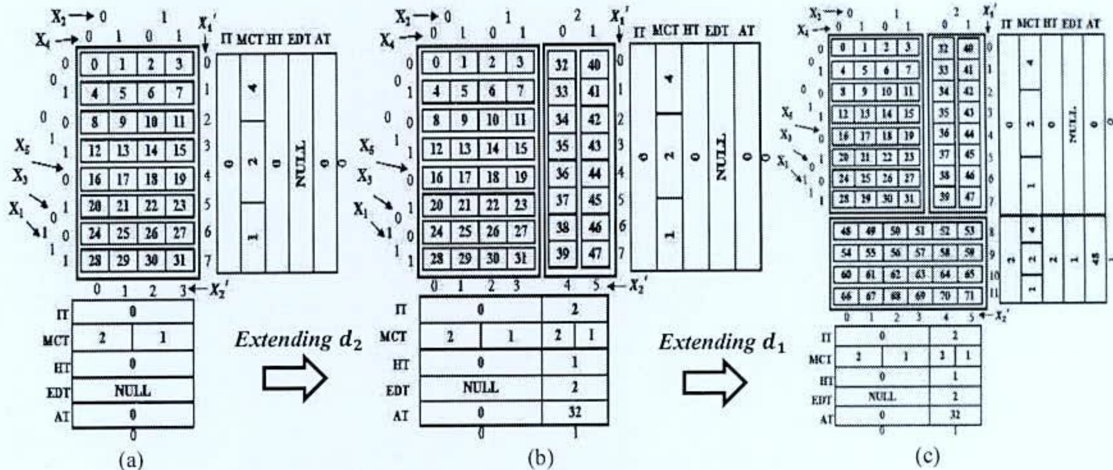


Fig. 3.6: Construction and Extension of a SSAI System

3.5. Operations on a SSAI System

The SSAI structure is a modify version of SAI structure. So, the operations of SSAI slightly differ from the operations of SAI. For example, the dimension transformation and range key query of a SSAI structure is same as a SAI structure. The rest operations are slightly different of SAI structure as SSAI structure offers segmented SA.

3.5.1. Construction and Extension

The construction and extension operation of a SSAI is same as SAI (as sec. 3.2.3.1) except the memory allocation which is done by segmentation.

Example 3.9. Consider the SSAI in Fig. 3.6(a). Here 8 segments are determined each of size $=l_2' = l_2 \times l_4 = 4$ (as sec 3.3.1). Store 1st segment's cell position to ST1[0].AT = 0. Initially, set ST1[0].IT = 0, ST1[0].HT = 0 and ST1[0].EDT = NULL. MCT values are initialized to ST1[0].MCT[0] = $l_3 \times l_5 = 4$, ST1[0].MCT[1] = $l_5 = 2$ and ST1[0].MCT[2] = 1 for d_1' and ST2[0].MCT[0] = $l_4 = 2$ and ST2[0].MCT[1] = 1 for d_2'. Store ST2[0].IT = 0, ST2[0].HT = 0, ST2[0].EDT = NULL and ST2[0].AT = 0.

The dynamic extension along any arbitrary direction d_i' of the SSAI is done by allotting one segment at a time.

Example 3.10. Let, the SSAI in Fig. 3.6(b) demands an extension in direction d_2. The extended SA size is (2^4) or 16. As the extension corresponds to d_2', hence the segment size is 8 (i.e l_1') and number of segment, *nos* is 2 (i.e $\frac{16}{8}$). In this extension ST2 will be preserved.

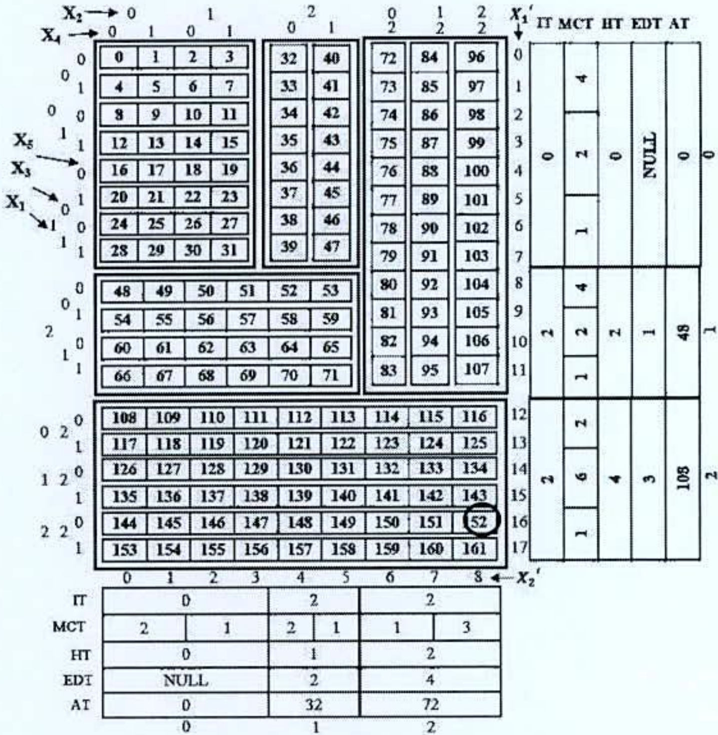


Fig. 3.7: Realization of a SSAI[18,9]

The first address of the 1st segment (i.e 32) is stored in $ST_2[1].AT$. The history counter is incremented and stored in $ST_2[1].HT$. The extended start length of d_2 is stored in $ST_2[1].IT$. The value of the extended dimension (i.e 2) is hold by $ST_2[1].EDT$. Finally the multiplicative coefficients are stored as $ST_2[1].MCT[0] = l_4$ (for x_2') and $ST_2[1].MCT[1] = 1$ (for x_4'). Hence a SSAI[8][4] has been extended to SSAI′[8][6]. Fig. 3.6(c) shows the SSAI after extending the SSAI on d_1. Fig. 3.7 shows the SSAI after extending on d_4 and d_3 respectively.

3.5.2. Point Query

Like a SAI, the first task of the point query of a SSAI structure is to calculate the C2I index. Afterwards the candidate SA is determined using the supplementary index ST_1 and ST_2 (as sec 3.2.3.3). Now, calculate the start SSAI index, $sx'$of the subarray and find the largest multiplicative coefficient MCT_{max} as the extended direction holds the largest coefficient of the SA. Then,

$$sx' = \begin{cases} \mathrm{ST_1}[i].\mathrm{IT} \times \mathrm{ST_1}[i].MCT_{max}, \text{when SA exists on } {d_1}' \\ \mathrm{ST_2}[j].\mathrm{IT} \times \mathrm{ST_2}[j].MCT_{max}, \text{when SA exists on } {d_2}' \end{cases} \tag{3.8}$$

Now, calculate the candidate segment number, SN using following equation

$$\text{SN} = \begin{cases} \text{x}_1' - sx', \text{when SA exists on } d_1' \\ \text{x}_2' - sx', \text{when SA exists on } d_2' \end{cases} \tag{3.9}$$

And the required segment's first cell address, SFA is

$$\text{SFA} = \begin{cases} \text{ST}_1[i].\text{AT}[0] + \text{SN} \times l_2', \text{when SA exists on } \text{d}_1' \\ \text{ST}_2[j].\text{AT}[0] + \text{SN} \times l_1', \text{when SA exists on } d_2' \end{cases} \tag{3.10}$$

And the required cell value, VALUE is

$$\text{VALUE} = \begin{cases} \text{SFA} + \text{x}_2', \text{when SA exists on } d_1' \\ \text{SFA} + \text{x}_1', \text{when SA exists on } d_2' \end{cases} \tag{3.11}$$

Example 3.11. Consider an input $(x_1, x_2, x_3, x_4,\ldots x_n) = (1, 2, 2, 2, 1)$ is to be retrieved from Fig. 3.7. Now $i = 2$ for ST_1, $j = 2$ for ST_2, $< x_1', x_2' > = < 15, 8 >$ and d_1' holds the SA. Hence start SAI index sx' of the subarray is

$$sx' = \text{ST}_1[i].\text{IT} \times \text{ST}_1[i].MCT_{max} = 2 \times 6 = 12$$

The candidate segment number, SN is

$$\text{SN} = x_1' - sx' = 15 - 12 = 3$$

And the required segment's first cell address, SFA is

$$\text{SFA} = \text{ST}_1[i].\text{AT}[0] + \text{SN} \times l_2' = 108 + 3 \times 9 = 135$$

And the required cell value (marked in Fig. 3.7) is

$$\text{VALUE} = \text{SFA} + x_2' = 135 + 8 = 143$$

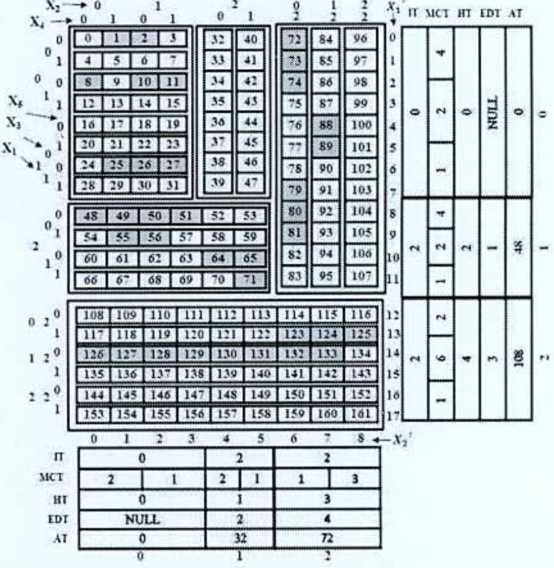


Figure 3.8: A Sparse Representation of a SSAI[18][9]

3.6. 2 Dimensional Key Value Encoding (2DKVE)

The History Pattern Encoding scheme [42, 43] eliminates the drawbacks of runtime calculation (multiplication, division) of large cell positions of History Offset Encoding scheme [16], but it is an n dimensional representation. The 2 Dimensional Key Value Encoding is an encoding scheme that encodes n dimensional data into a key that uses only 2 dimensional indices of SSAI structure. Let the shaded cells in Fig. 3.8 represent non-empty cells. The 2DKVE representation of the SSAI system of Fig. 3.8 is depicted in Fig. 3.9. The structure eliminates the 2D Address Table (AT) entry from the supplementary table and creates a new individual 1D First Address Table (FAT) that contains the first non-empty address of a SA, if the SA is not empty. Otherwise it will store the negation location value of the next SA. Then the last location of the SA can be found by the successive value (absolute) difference from the FAT table if it is not the last SA in the structure (otherwise the last position of the memory will be considered as the last location of the SA). The index of the FAT table is labeled by the history of the SSAI system.

3.6.1 Encoding

The 2DKVE encodes a cell of the array by the pair $< x_1', x_2' >$ instead of the pair $< x_1, x_2, x_3, \ldots, x_n >$. Hence the size of the encoding key becomes fixed irrespective of the value of n. We generate a single *key* for the encoded pair $< x_1', x_2' >$. In this encoding scheme, the

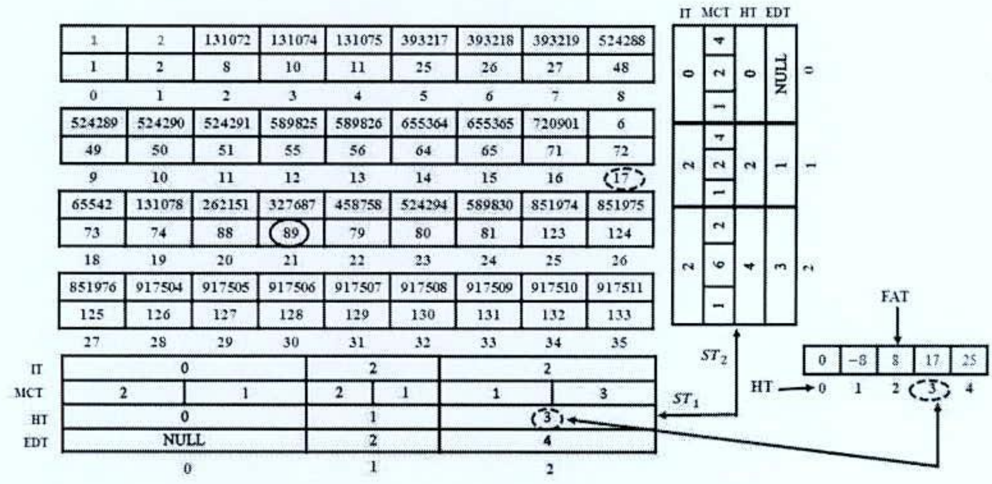


Figure 3.9: A Realization of a 2DKVE System

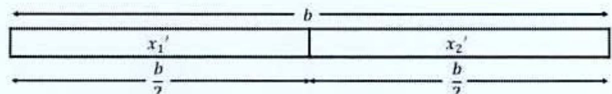


Figure 3.10: Key Structure of a 2DKVE System

entries of the supplementary tables are same as SAI, except the AT entry which is replaced by the individual 1D FAT table. The FAT stores the location of the first nonempty *key* of the SA. The FAT stores NULL if the whole SA is empty. Let the *key* contains *b* bits. Among *b* bits the x_1' is stored in the most significant *b* bits (MSB) and x_2' is stored in the least significant *b* bits (LSB) as shown in Fig. 3.10. Hence the *keys* are stored in the order of x_1'. The x_1' is inserted to the *key* and successive left shift operation is applied to move it to MSB. Afterwards the x_2' is added with the *key*. Finally the < *key*, *value* > pair is stored where *value* is the actual data in the SSAI.

Example 3.12. Consider an R*nI* index $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 2, 1)$. The corresponding R2I is $<x_1', x_2'>$ = (5, 7). Let the *key* comprises b = 64 bits. Then MSB 32 bits are $00...000101 = 327680$ and LSB 32 bits are $00...000111 = 7$ and *key* = 327687. Then the encoding value is <*key, value*> = (327687, 89) (as sec Fig. 3.9).

3.6.2 Data Access

To access an item from a 2DKVE it is necessary to determine $<x_1', x_2'>$ and H_{max} as described in sec 3.5.2. Now, find the supplementary table index *i* that contains H_{max}. The candidate SA can be found from FAT[H_{max}]. Now generate the *key* from $<x_1', x_2'>$ which is to be accessed. Then the subarray is loaded from disk to memory. Since the keys are stored in order of x_1', the binary search is performed to find the *key* and the corresponding *value* is the desired array cell.

Example 3.13. Consider an RnI index $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 2, 1)$. The value of $<x_1', x_2'>$, H_{max} and *key* are <5, 7>, 3 and 327687 respectively (as Example 3.12). The location of the first cell of the SA is FAT[3] = 17. Now load the SA from disk to memory. And the binary search to find the value of *key* or 327687 shows that the desired *value* is 89 as encircled in Fig. 3.9.

3.6.3 Decoding

The aim of *Decoding* is to retrieve an RnI from a given <*key*, *value*> of 2DKVE. By successive right shift the $<x_1', x_2'>$ is determined. Then binary search is performed in the mapping table ST_1 and ST_2 to find their index *i* and *j* (respectively) using the conditions mentioned in Eq. 3.3. Now find $H_{max} = max(ST_1[i].HT, ST_2[j].HT)$. If FAT[$H_{max}$] = NULL, then the SA is empty. Otherwise find the RnI $(x_1, x_2, x_3, x_4, ...x_n)$ using Eq. 3.4.

Example 3.14. Let *KEY* = 327687. If the key comprises 64 bits, then $x_1' = 327687 = 5$ and $x_2' = 7$. Now, the value of *i* and *j* is 0 and 2 respectively (Eq. 3.3). And $H_{max} = 3$. As, FAT[2] ! = NULL, hence the segment is not empty. We have *three* entries in $ST_1[0].MCT$. So, using Eq. 3.4 the odd indices are as follows:

$$x_1 = x_1'/MC_3 = 5/4 = 1$$

$$x_3 = (x_1'\%MC_{1max})/MC_1 = (5\%4)/2 = 0$$

$$x_5 = \left((x_1'\%MC_{1max})\%MC_{2max}\right)/MC_1 = \left((5\%4)\%2\right)/1 = 1$$

We have *two* entries in $ST_2[2].MCT$, hence using Eq. 11 the even indices are as follows:

$$x_2 = (x_2'\%MC_{1max})/MC_2 = (7\%3)/1 = 1$$

$$x_4 = x_2'/MC_4 = 7/3 = 2$$

So, *KEY* = 327687 maps to $< x_1', x_2' > = <5, 7>$ which maps to $(x_1, x_2, x_3, x_4, x_5) = (1, 1, 0, 2, 1)$.

3.4. Conclusion

In this chapter we elaborate our proposed models with the model structure and operations. We effectively convert the *n* dimensions of the array into 2 dimensions which helps in large dimensional data representation. We use the concept of segment to delay the address space overflow and using this concept we also provide an encoding scheme which can increase the storage utilization by efficiently employing only 2 indices of 2 dimensions.

CHAPTER IV

Results and Analysis

4.1 Experimental Setup

In this chapter, we present the experimental results along with the theoretical analyses of the proposed schemes. We have compared the proposed schemes with static CMA and dynamic Extendible Array [11] and Extendible Array [13]. In the following we rename the Extendible Array model [13] by EA1 and [11] by EA2. To analyze the performances of the proposed structures we develop a prototype system in a machine having Intel(R) Xeon(R) E5620 @ 2.40GHz processor with 8 processors, 32 GB RAM, 1406 MB cache memory and 1.3TB usable HDD. The actual array was placed in the secondary storage. The program is written in C and compiled in gcc compiler on debian squeeze 6.0.5 operating system with the parameter values shown in Table. 4.1. In all performance analysis, we have considered the index table to be stored in secondary memory. All lengths or sizes of storage areas are in bytes. The analyses are also represented as a function of bytes.

Table 4.1. Parameters for Constructed Prototypes

Parameter	Description
n	No. of dimension in array
l_i	Length of dimension $i(1 \leq i \leq n)$, let $l_1 = l_2 = \cdots = l_n = l$
V	The array volume= $\sum_{i=1}^{n} l_i = l^n$
hc	Total no. of extension or maximum history value
γ	Size of an index
α	Size of an array cell
β	Size of a key
N	Total no. of non-empty cells in actual array
ρ	Data density of actual array $= \frac{N}{V} = \frac{N}{l^n}, 0 \leq \rho \leq 1$

4.2 Performance Analysis of the Structure

4.2.1. Index Overhead (ϒ)

a) Theoretical Analysis.

Let the number of indices is denoted by *noi*. The no. of index in EA1 scheme or $noi_{EA1} = 3$ (<*history value, first address, coefficient vector*>). In EA2 the no. of index is $noi_{EA2} = 4$ (<*initial index, start address, coefficient vector, start address pointer*>). The no. of index in SAI scheme or $noi_{SAI} = 5$ (<*history value, initial index, first address, coefficient vector, extended dimension*>). If the totla size of index is *tsi* = $noi \times n \times \gamma$ and total index overhead is $\Upsilon = \left(tsi \times (hc + 1)\right)$. Then the index overhead of SAI (or SSAI), EA1 and EA2 is as follows:

i. No. of index in SAI (or SSAI) scheme, $noi_{SAI} = 5$
   Total size of index in SAI (or SSAI) scheme, $tsi_{SAI} = 5 \times 2 \times \gamma = 10\gamma$
   Total index overhead: $\Upsilon_{SAI} = \left(tsi \times (hc + 1)\right) = 10\gamma \times (hc + 1)$
ii. No. of index in EA1 scheme, $noi_{EA1} = 3$
   Total size of index in EA1 scheme, $tsi_{EA1} = 3 \times n \times \gamma = 3n\gamma$
   Total index overhead: $\Upsilon_{EA1} = \left(tsi \times (hc + 1)\right) = 3n\gamma \times (hc + 1)$
iii. No. of index in EA2 scheme, $noi_{EA2} = 4$
   Total size of index in EA2 scheme, $tsi_{EA2} = 4 \times n \times \gamma = 3n\gamma$
   Total index overhead: $\Upsilon_{EA2} = \left(tsi \times (hc + 1)\right) = 4nC \times (hc + 1)$

Consider index size $\gamma = 8$. Let $l = 4$. Now consider two cases CASE 1 and CASE 2. In first case or CASE 1, vary $n = 4{\sim}6$ and in second case or CASE 2, vary $hc = 0{\sim}2$. Then, the index overheads of the structures can be calculated as mentioned in Table. 4.2. The first case is mentioned in Fig. 4.1(a). Here, it can be seen that the SAI scheme requires smallest and constant index overhead which does not depend on the value of *n*. The second case is depicted in Fig. 4.1(b). Here, in accordance with the dynamic extensions, the index overhead of SAI scheme increases but still consumes lowest overhead than the other schemes.

b) Experimental Analysis.

The CMA is a static structure. It does not require any indexing. The EA1 and EA2 demand indices for each of the *n* dimensions. For EA1, the value of each index entry is a 3 tupple

Table 4.2. Analytical Index Overhead for Constructed Prototypes

γ	CASE	hc	n	Υ_{SAI} $10\gamma \times (hc+1)$	Υ_{EA1} $3n\gamma \times (hc+1)$	Υ_{EA2} $4n\gamma \times (hc+1)$
8	1	0	4	80	96	128
			5	80	120	160
			6	80	144	192
	2	0	4	80	96	128
		1		160	192	256
		2		240	288	384

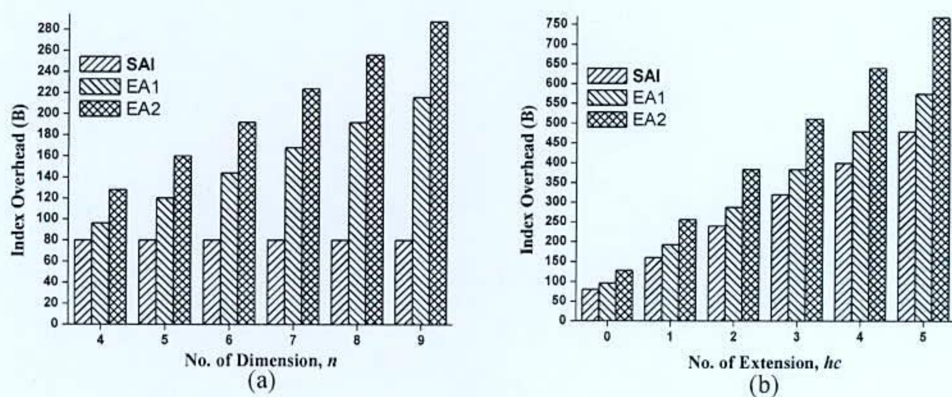


Figure 4.1: Analytical Result of Index Overhead

<history value, first address, coefficient vector>. For EA2, the value of each index entry is a 4 tupple *<initial index, start address, coefficient vector, start address pointer>*. Where coefficient vector has $(n-2)$ entries. Both the EA1 and EA2 needs n indices to be placed. Hence the index overhead increases with the increasing n. However in SAI, the index entry is a 5 tupple *<history value, initial index, first address, coefficient vector, extended dimension>*. But the total number of indices are 2 irrespective of the value of n. Hence, index overhead for index is very small in SAI. Fig. 4.2(a) shows the index overhead for SAI, EA1 and EA2 for $l = 4$, $hc = 0$ and varying n. As n increses, the SAI shows constant and small index overhead, the EA2 shows more overhead than EA1 since it has more entry for index than EA1. Fig. 4.2(b) shows the index overhead for SAI, EA1 and EA2 for $n = 4$ and varying hc. As hc increses, the SAI shows increasing but small overhead compared to EA1 and EA2. The SSAI structure has same index cost as the SAI structure.

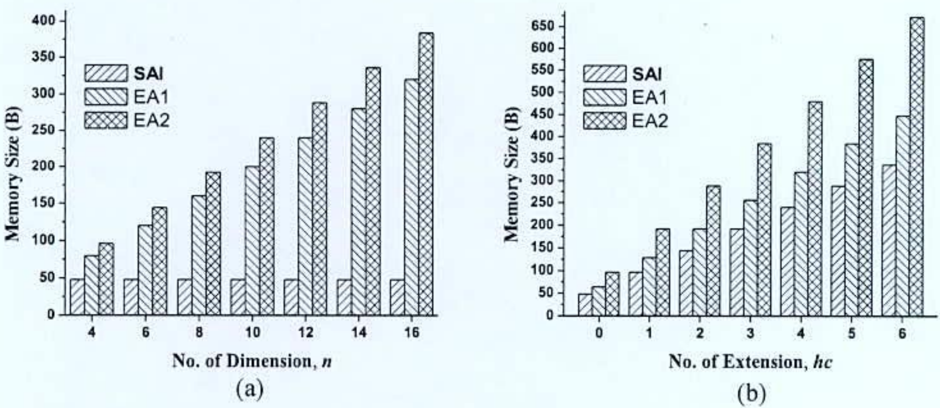


Figure 4.2: Experimental Result of Index Overhead

4.2.2. Construction Cost (C)

a) Theoritical Analysis.

The construction cost involves the cost of allocating and storing data volume, cost of allocating indexing and cost of indexing (as sec 4.2.1). If cost of allocating indexing is τ, then the construction cost of the schemes are follows:

i. Constructin Cost of CMA, $C_{CMA} = \alpha V = \alpha l^n$
ii. Constructin Cost of EA1, $C_{EA1} = \alpha V + n \times \tau \times \Upsilon_{EA1} = n \times \tau \times 3n\gamma(hc+1) + \alpha V = 3n\gamma + \alpha l^n$
iii. Constructin Cost of EA2, $C_{EA2} = \alpha V + n \times \tau \times \Upsilon_{EA2} = n \times \tau \times 4n\gamma(hc+1) + \alpha V = 4n\gamma + \alpha l^n$
iv. Constructin Cost of SAI (or SSAI), $C_{SAI} = \alpha V + 2 \times \tau \times \Upsilon_{SAI} = 2 \times \tau \times 10\gamma(hc+1) + \alpha V = 2 \times \tau \times 10\gamma + \alpha l^n$

For all the above structures, the construction time directly depends on the value of l and n or the volume (V) of the structure. But for a dynamic structure, additional cost is required for index overhead. Consider index size $\alpha = \gamma = 8$. Let $l = 4$, $\tau = 1000$ and . Then, the construction cost of the structures for $n = 4{\sim}6$ can be calculated as mentioned in Table. 4.3. The comparison of construction cost is shown in Fig. 4.3.The CMA requires lowest construction time as it does not consume any index overhead. Among the three dynamic structures, as SAI (or SSAI) consumes lowest index overhead and it requires only 2 dimensional index initialization. Hence it has the lowest construction cost amongst others.

Table 4.3. Analytical Construction Cost for Constructed Prototypes

l	$\alpha = \gamma$	τ	n	C_{CMA} αl^n	C_{SAI} $2 \times \tau \times 10\gamma + \alpha l^n$	C_{EA1} $n \times \tau \times 3n\gamma + \alpha l^n$	C_{EA2} $n \times \tau \times 4n\gamma + \alpha l^n$
4	8	1000	4	2048	162048	386048	514048
			5	8192	168192	488192	648192
			6	32768	192768	608768	800768

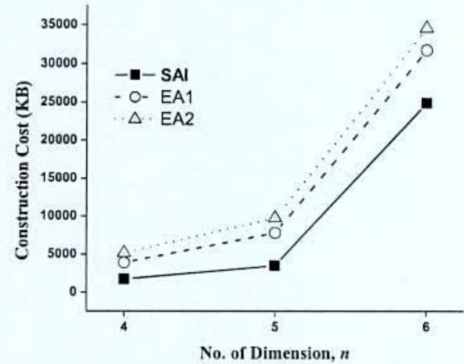
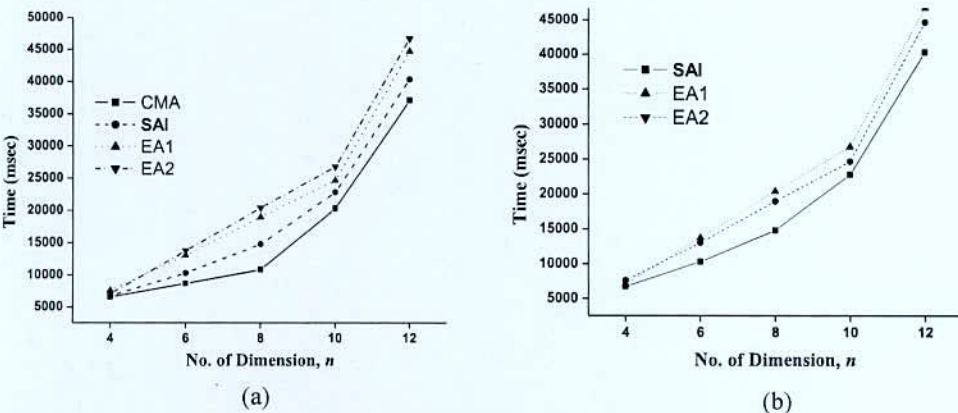


Figure 4.3: Analytical Result of Construction Cost

b) Experimental Analysis.

As the CMA does not maintain any index, the construction of CMA only involves the cost to allocate and store data. Again, for a dynamic structure like EA1or EA2 or SAI (or SSAI), the construction not only involves the time to allocate and store data but also to initialize indexing. Hence, the CMA takes smallest cost for initial construction compared to the dynamic models. Again the dynamic models differ their construction cost from CMA by their indexing cost. Among the dynamic array models, as the SAI requires two dimensional indexing, hence it has smallest cost (except CMA) for initial construction compared to theother dynamic models like EA1 and EA2 as n dimensional indexing is required for n dimensional Indexed Array. Fig. 4.4(a) shows the construction cost of the CMA along withthe dynamic schemes. Fig. 4.4(b) shows the initial construction cost of the dynamic models EA1, EA2 and SAI. Hence, the performance of SAI scheme has been validated with theoretically and experimentally.



(a) (b)

Figure 4.4: Experimental Result of Construction Cost

4.2.3. Extension Cost (EC)

a) Theoritical Analysis.

The CMA is a static structure. It requires reorganization of the array and rewrites both existing and new data elements. The existing elements of the initial array (e_1) need to be tackled and recalculate the new offsets (e_2) due to the extension for CMA.

Hence the extension cost of a CMA is

$$EC_{CMA} = e_1 + e_2$$

The cost of tackling the existing array elements,

$$e_1 = V = \sum_{i=1}^{n} l_i = l^n.$$

If a CMA is extended by 1 then a new CMA of length l +1 is to be reallocated and reallocation cost becomes

$$e_2 = (l + 1) \times l^{n-1}$$

So, Total extension cost for CMA(n),

$$EC_{CMA} = e_1 + e_2 = l^n + (l + 1) \times l^{n-1}$$

For a dynamic model, to compare with the static structure lets ignore the indexing cost. As a dynamic model does not require reallocation, the cost only depends on the new extended data size allocation or SA allocation. If an EA is extended by 1 then the SA length l^{n-1} is to be allocated. If, the SA allocation cost is SC, then the extension cost is follows:

$$EC_{EA} = \text{SC} = l^{n-1}$$

Hence, the extension cost gain (ECG) of a dynamic model compared to static CMA is as follows:

$$ECG = EC_{CMA} - EC_{EA} = l^n + (l+1) \times l^{n-1} - l^{n-1} = 2l^n$$

The extension cost of CMA and EA for $n = 8, 10, 12$ is shown in Table 4.4. The performances of the EA1, EA2 and SAI with respect to CMA are shown in Fig. 4.5(a).

Table 4.4. Analytical Extension Cost for Static (CMA) and Dynamic (EA)

l	n	Volume $V = l^n$	EC_{CMA} $V + (l+1) \times l^{n-1}$	EC_{EA} l^{n-1}	ECG $EC_{CMA} - EC_{EA} = 2\,l^n$
2	4	16	40	8	32
	6	64	160	32	128
	8	256	640	128	512

Table 4.5. Analytical Extension Cost for Dynamic (EA) Prototypes

l	γ	n	SC l^{n-1}	EC_{EA1} $SC + n\gamma$	EC_{EA2} $SC + (3+n) \times \gamma$	EC_{SAI} $SC + \left(4 + \left\lceil \frac{n}{2} \right\rceil\right) \times \gamma$
2	8	8	128	192	216	192
		10	512	592	616	584
		12	2048	2144	2168	2096

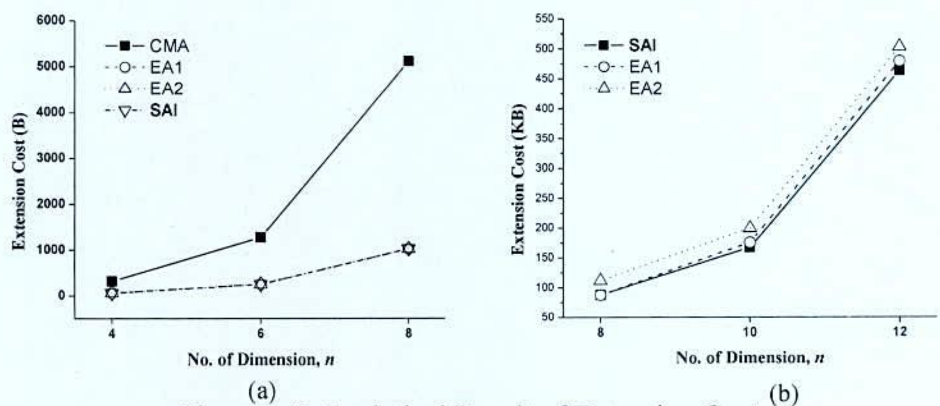


Figure 4.5: Analytical Result of Extension Cost

For dynamic models, the extension cost varies with respect to indexing cost. To extend a single dimension with a single unit (one *hc*) only one dimension needs indexing. If the indexing cost is I_{EA}, then the extension cost can be re-write as follows:

$$EC_{EA} = SC + \mathrm{I}_{EA}$$

But, the indexing of coefficients is different in different models. For EA1, the coefficient is $n-2$ dimensional. For EA2, the coefficient is n dimensional and for SAI (or SSAI), the coefficient is $\left\lceil \frac{n}{2} \right\rceil$ dimensional.

For EA1, the no. of index is 3 and indexing cost for an extension is as follows:

$$\mathrm{I}_{EA1} = 2\gamma + (n-2) \times \gamma = n\gamma$$

And

$$EC_{EA1} = SC + \mathrm{I}_{EA1} = l^{n-1} + n\gamma$$

For EA2, the no. of index is 4 and indexing cost for an extension is as follows:

$$\mathrm{I}_{EA2} = 3\gamma + n \times \gamma = (3+n) \times \gamma$$

And

$$EC_{EA2} = SC + \mathrm{I}_{EA2} = l^{n-1} + (3+n) \times \gamma$$

For SAI, the no. of index is 5 and indexing cost for an extension is as follows:

$$\mathrm{I}_{SAI} = 4\gamma + \left\lceil \frac{n}{2} \right\rceil \times \gamma = \left(4 + \left\lceil \frac{n}{2} \right\rceil\right) \times \gamma$$

And

$$EC_{EA2} = SC + \mathrm{I}_{SAI} = l^{n-1} + \left(4 + \left\lceil \frac{n}{2} \right\rceil\right) \times \gamma$$

The extension cost of EA1, EA2 and SAI for $n = 8, 10, 12$ is shown in Table 4.5 and the performances are shown in Fig. 4.5(b). From the above figure it can be seen that the SAI (or SSAI) outperfoms the other dynamic models.

b) Experimental Analysis.

The CMA is a static structure. CMA requires reallocation of previously stored data if we want to resize or extend it. For an index based array models, there is no need for reallocation. Hence, the extension cost for CMA is always higher than other dynamic indexed array models as mentioned in Fig. 4.6(a). The comparison of indexed based models is mentioned in Fig. 4.6(b). For an index based model, the extension cost involves allocation of SA (instead of reallocation) and updating auxiliary indexing information. For a single extension, among the n dimensions (for EA) or 2 dimensions (for SAI or SSAI) only one auxiliary table is updated. But, the coefficient allocation depends on the value of n. for, EA1, EA2 and SAI the co-efficient is $(\boldsymbol{n}-\mathbf{2})$, $\boldsymbol{n}$ and $\left\lceil\frac{n}{2}\right\rceil$ dimensional respectively. In theoretical analysis we have considered that all indexing parameters are of same size $(\boldsymbol{\gamma})$. But practically the size (say t_1) of coefficients and address are same and larger than the the size (say t_2) of parameters like history, initial index, extended dimension. Hence, for EA1 the indexing size is $\boldsymbol{t_2}+(\boldsymbol{n}-\mathbf{2})\boldsymbol{t_1}+\boldsymbol{t_1}$ or $\boldsymbol{t_2}+(\boldsymbol{n}-\mathbf{1})\boldsymbol{t_1}$ and for EA2 the indexing size is $\mathbf{2}\boldsymbol{t_2}+\boldsymbol{n}\boldsymbol{t_1}+\boldsymbol{t_1}$ or $\mathbf{2}\boldsymbol{t_2}+(\boldsymbol{n}+\mathbf{1})\boldsymbol{t_1}$. And the difference between the indexing of EA1 and EA2 is $\boldsymbol{t_2}-\boldsymbol{t_1}$. For this reason, the extension cost of EA1 and EA2 differs slightly. When the value of n is small the extension time is almost similar in case of SAI (or SSAI) compared to other models. However, with the increase in n, the extension cost of EA1 and EA2 increases due to n dimensional and $(\boldsymbol{n}-\mathbf{2})$ dimensional coefficient maintenance respectively.

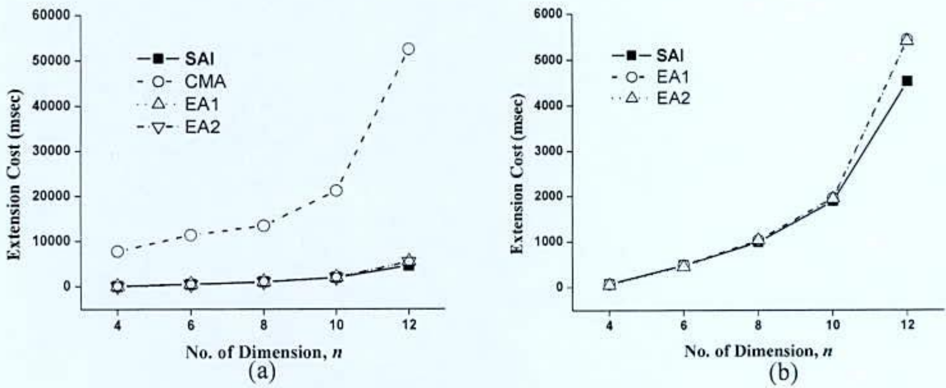


Figure 4.6: Experimental Result of Extension Cost

4.2.4. Retrieval Cost (RC)

a) Theoritical Analysis.

Let, n = 4, $l_1 = l_2 = \cdots = l_n = l = 2$. The form of an input for a single range key is $< k_1, k_2, k_3, u_4 >$, wheren k_i is the known index on dimension i and $0 \leq k_i \leq l-1, 1 \leq i \leq n$ and u_j is the unknown index on dimension j and $0 \leq u_j \leq l-1$, $1 \leq j \leq n, j \neq i$. Let, the difference between two successive selected block read for a query is s_j and s_{max} is the maximum difference between two successive blocks. The two blocks are consecutive if $s_j = 0$.

The addressing function of the CMA (using Eq. 2.1) can be rewrite as follows:

$$f(x_4, x_3, x_2, x_1) = x_4 \times l^3 + x_3 \times l^2 + x_2 \times l + x_1 = 8x_4 + 4x_3 + 2x_2 + x_1$$

Now, consider an input $< 1, 1, 1, * >$, where '*' means all. Then, we have values as 14 and 15 ($s_1 = 15 - 14 = 1$). If the input is $< 1, 1, *, 1 >$, then the values are 13 and 15 ($s_2 = 15 - 13 = 2 = l$). When the input is $< 1, *, 1, 1 >$, then the values are 11 and 15 ($s_3 = 15 - 11 = 4 = l^2$). And if the input is $< *, 1, 1, 1 >$, then the values are 7 and 15 ($s_4 = 15 - 7 = 8 = l^3$). Here, $s_{max} = s_4 = l^3$. Hence, if the number dimension is n, then the value of s_{max} is l^{n-1}.

For a dynamic model like EA, as the SA is $n-1$ dimensional, hence the addressing function for an extension along dimension d_4 can be rewrite as follows:

$$f(x_3, x_2, x_1) = x_3 \times l^2 + x_2 \times l + x_1 = 4x_3 + 2x_2 + x_1$$

Like CMA, the value of s_{max} can be calculated as $s_{max} = s_3 = l^2$ when n = 4 or $s_{max} = s_3 = l^{n-2}$ when n = n, which is smaller than the CMA. But for a dynamic model, the retrieval of an input requires to locate the SA by searching n dimensional supplementary tables. In contrast, for a static model, there is no requirement of a SA searching. It only genates maximum of l^{n-1} locations for a given query. If for a given input, a dynamic model needs to locate three SA each of which requires t unit of time, then the total time for generating the required cells is $l^{n-2} + 3t$. Hence, the static model degrades the performances of a dynamic model.

The proposed SAI is a 2 dimensional dynamic model. As $l = 2$, hence $l_2' = l_2 \times l_4 \times ... \times l_{\frac{n}{2}} = l^{\frac{n}{2}} = 2^{\frac{4}{2}} = 4$, $l_1' = l_1 \times l_3 \times ... \times l_{\lceil \frac{n}{2} \rceil} = l^{\lceil \frac{n}{2} \rceil} = 2^{\frac{4}{2}} = 4$, $0 \le k_i \le l_i' - 1, 1 \le i \le 2, 0 \le u_j \le l_j' - 1, 1 \le j \le 2, j \ne i$ and addressing function for SA on $d_1'$is as follows:

$$f(x_1', x_2') = x_1' \times l_2' + x_2' = 4x_1' + x_2'$$

Now, consider an input $< 1,* >$. Then the required values are 4, 5, 6, 7 ($s_2' = 5 - 4 = 1$). If the input is $<*, 1 >$, then the requied values are 1, 5, 9, 13 ($s_1' = 5 - 1 = 4 = l_2'$). Hence, $s_{max} = l_2' = l^{\frac{n}{2}}$. Like a dynamic model, the SAI structure also requires supplementary table searching, but the table is 2 dimensional. Hence, the proposed SAI outperforms both the static model and dynamic model.

b) Experimental Analysis.

Fig. 4.7(a) compares the single range retrieval performances of the compared models of volume $29, 30, 52, 74$ GB for $n = 4, 6, 8, 10$ respectively. A CMA model searches the given input among n dimensional index. If we exclude the given input's dimension, then it will require $n - 1$ dimensional index to generate the resultant cells and requires $n - 1$ loops. Furthermore, EA1 and EA2 search a SA of $(n - 1)$ dimension. To find a subscript of dimension $n - 1$ of size $[l_{n-1}, ..., l_1]$ they need to calculate array indices of dimension $n - 1$ (ex. A[$x_1, x_2, ..., x_{n-1}$]). If we exclude the given input's dimension, then the indices calculation reduces from $(n - 1)$ to $(n - 2)$. Hence to calculate such $(n - 2)$ dimensional indices they will need $(n - 2)$ loops which is smaller than CMA. But for locating a SA, the dynamic models require n dimensional supplementary table searching which decreases the performances of dynamic models compared to static model. Again, as the EA1 requires less indexing than EA2. Hence the EA1 outperforms the EA2. On the other hand, the SAI needs 2 loops only to calculate array indices of dimension 2 by $f(\mathrm{x}_1', \mathrm{x}_2') = \mathrm{x}_1' \times l_2' + \mathrm{x}_2'$(Eq. 3.2). Thus, it outperforms the dynamic models and at the same time static model. Fig. 4.7(b) shows the performances of large length data retrieval where we omit the performance of CMA as it does not support such large length. To compare with the dynamic models, we choose data size 103, 114 and 153 GB for 12, 14 and 16 dimension respectively. The retrieval performance of SSAI is same as SAI.

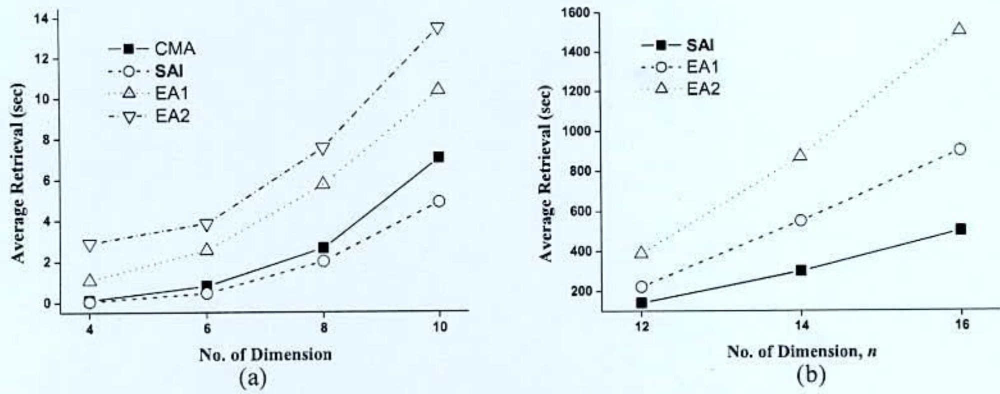


Figure 4.7: Experimental Result of Retrieval Cost

4.2.5. Storage Utilization (SU)

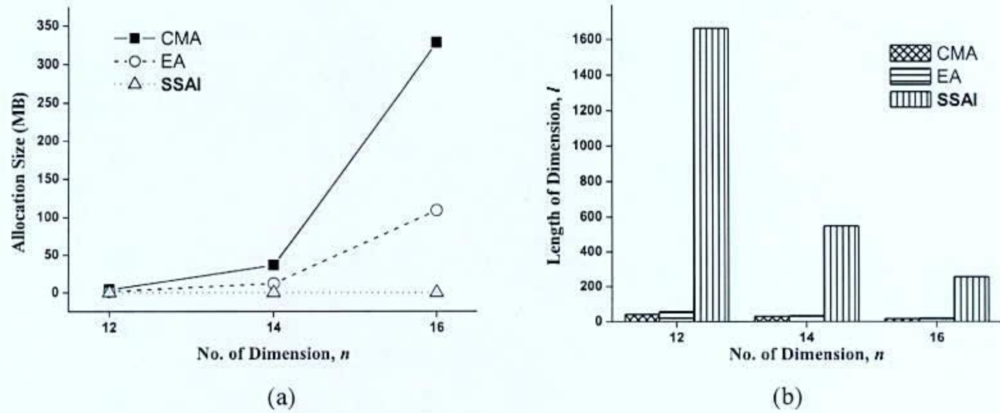
a) Theoritical Analysis.

If the SA size is *sz* and segment size of SSAI is *sgz* then the allocation requirements of the compared protoypes are as follows:

i. Allocation requirement of CMA, $AR_{CMA} = V = l^n$
ii. Allocation requirement of EA, $AR_{EA} = sz = l^{n-1}$
iii. Allocation requirement of SAI, $AR_{SAI} = sz = l^{n-1}$
iv. Allocation requirement of SSAI, $AR_{SSAI} = sgz = l^{\frac{n}{2}}$

Table 4.6. Analytical Result of maximum length of the compared Prototypes

For $AR_{CMA} = l^n$	For $AR_{SAI} = l^{n-1}$	For $AR_{SSAI} = l^{\frac{n}{2}}$
$l^{16} = 2^{64}$	$l^{15} = 2^{64}$	$l^8 = 2^{64}$
$\Rightarrow 16\log_2 l = \log_2(2^{64}) = 64$	$\Rightarrow 15\log_2 l = \log_2(2^{64}) = 64$	$\Rightarrow 8\log_2 l = \log_2(2^{64}) = 64$
$\Rightarrow \log_2 l = 4$	$\Rightarrow \log_2 l = 4.3$	$\Rightarrow \log_2 l = 8$
$\Rightarrow l = 2^4 = 16,$	$\Rightarrow l = 2^{4.3} = 20,$	$\Rightarrow l = 2^8 = 256,$
$V = l^n = 16^{16}$	$V = l^n = 20^{16}$	$V = l^n = 256^{16}$
$= 1.8 \times 10^{19}$	$= 6.6 \times 10^{20}$	$= 3.4 \times 10^{38}$



(a) (b)

Figure 4.8: Analytical Result of Storage Utilization

Let $n = 16$. Theoretically, for a 64 bit address space the maximum length of each dimension can be calculated as mentioned in Table 4.6 which shows that the SAI scheme requires lowest allocation space and highest usable length and hence offers maximum volume of data or maximum storage utilaization. Fig. 4.8(a) shows the allocation requirements and Fig. 4.8(b) shows the maximum usable length of CMA, EA (or SAI) and SSAI.

b) Experimental Analysis.

In storage utilization we have discussed two types of overflow situations. First one is resource overflow where the structure has enough address space to allocate but the system has no space to store. Another one is allocation overflow where we have enough space to store but the address space overflows. The CMA has very less memory utilization because an increase in n and l causes the total address space to increase as l^n. Consequently, it overflows quickly. Again, the CMA requires l^n consecutive memory locations. In case of storage utilization, the SAI acts like an EA. The index array models do not require l^n consecutive memory locations. Instead, the index array models dynamically allocate consecutive SA of size l^{n-1}. Hence storage utilization of index array models is higher than CMA. But the increase in n and l triggers the address space to overflow in index array models too. In SSAI the allocation grows in the form of $l^{\frac{n}{2}}$. This is because the dimensions are divided into 2 and the allocation depends on the segment size which corresponds to the length of either dimension from the 2 dimensions. Fig. 4.9(a) shows how the allocation for CMA, EA (or SAI) and SSAI increases with the increase in dimension value. For this reason, the maximum usable length of dimension decreases even though enough resource

available which is shown in Fig. 4.9(b). As SSAI demands least allocation and largest usable length, hence SSAI manages highest storage utilization than others as shown in Fig. 4.9(c). From the mentioned figure it can be seen that the CMA structure always shows address space overflow. On the other hand, the EA (or SAI) shows resource overflow when n = 4~8 and shows address space overflow when $n \geq 10$. Using our available resources, we have observed that the proposed SSAI scheme always faces resource overflow rather than address space overflow.

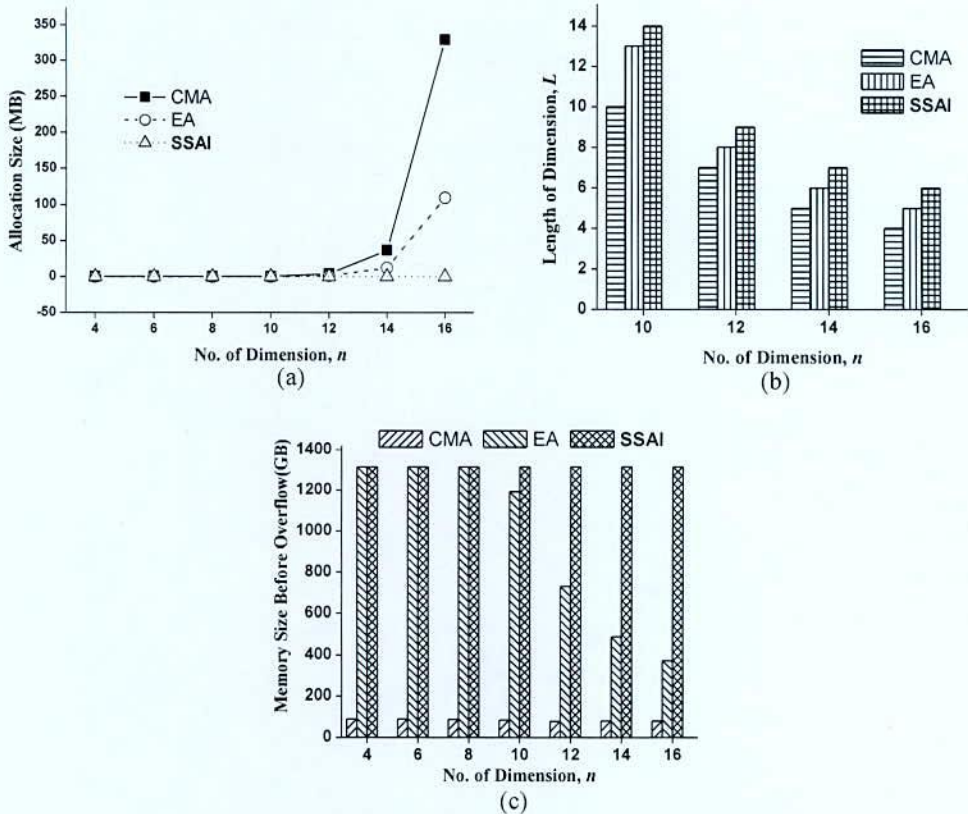


Figure 4.9: Experimental Result of Storage Utilization

4.3 Performance Analysis of the Encoding

In the previous section we have seen that the SSAI has better performance than SAI. Hence, we have applied our encoding technique on SSAI scheme named as 2DKVE (mentioned in chapter 3). We have compared the performance of our encoding scheme with the history-offset scheme [16] which is based on EA1. In rest of the section we will denote history-offset scheme [16] as HOE.

4.3.1. Index Overhead (ϒ)

The structure replaces the 2D Address Table (AT) entry from the supplementary table of SSAI with an individual 1D First Address Table (FAT) that contains the first non-empty address of a SA. The HOE is an encoding scheme based on EA1. Hence, the encoding cost in HOE is same as EA1.

a) Theoretical Analysis.

i. No. of index in 2DKVE scheme, $noi_{2DKVE} = noi_{SSAI} = 5$

   Total size of index in 2DKVE scheme, $tsi_{2DKVE} = (4 \times 2 + 1)\gamma = 9\gamma$

   Total index cost in 2DKVE scheme:$\Upsilon_{2DKVE} = \big(tsi(hc + 1)\big) = 9\gamma(hc + 1)$

ii. Total index cost in HOE: $\Upsilon_{HOE} = \Upsilon_{EA1} = 3n\gamma(hc + 1)$

Table 4.7. Analytical Index Overhead for Encoding Schemes

γ	CASE	hc	n	Υ_{SSAI} $10\gamma \times (hc + 1)$	Υ_{2DKVE} $10\gamma \times (hc + 1)$	Υ_{HOE} $3n\gamma \times (hc + 1)$
8	1	0	4	80	72	96
			5	80	72	120
			6	80	72	144
	2	0	4	80	72	96
		1		160	144	192
		2		240	216	288

Consider index size $\gamma = 8$. Let $l = 4$. Now consider two cases CASE 1 and CASE 2. In first case or CASE 1, vary $n = 4{\sim}6$ and in second case or CASE 2, vary $hc = 0{\sim}2$. Then, the index overheads of the structures can be calculated as mentioned in Table. 4.7. The first case is mentioned in Fig. 4.10(a) for $n = 4{\sim}9$. Here, it can be seen that the 2DKVE scheme requires smallest and constant index overhead which does not depend on the value of n. The second case is depicted in Fig. 4.10(b) for $hc = 0{\sim}5$. Here, in accordance with the dynamic extensions, the index overhead of 2DKVE scheme increases but still consumes lowest overhead than the other schemes.

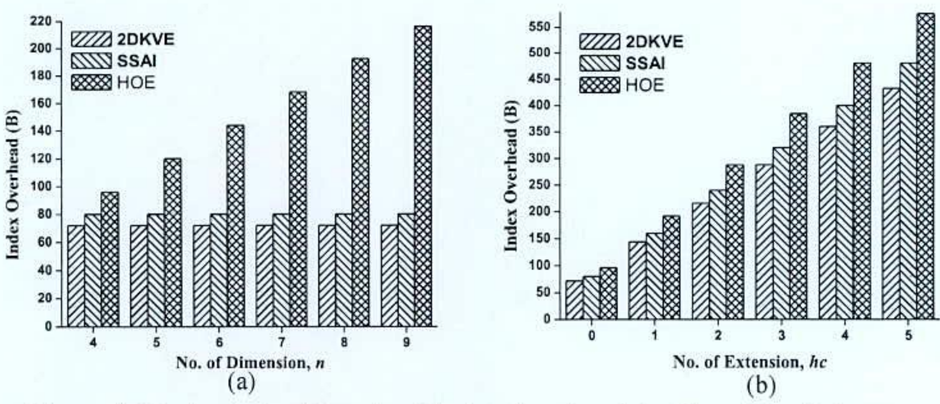


Figure 4.10: Analytical Result of Index Overhead for Encoding Schemes

b) Experimental Analysis.

The indexing in 2DKVE involves 2 dimensional 4 tupple <*history value*, *initial index*, *coefficient vector*, *extended dimension*> and one dimensional <*first address*> and has small overhead compared to SSAI (sec 4.2.1). Fig. 4.11(a) shows the storage overhead for 2DKVE, SSAI (or SAI) and HOE for $l = 4$, $hc = 0$ and varying n. Fig. 4.11(b) shows the storage overhead for 2DKVE, SSAI (or SAI) and HOE for $n = 4$ and varying hc. The SAI or SSAI shows better perfoormance compared to HOE and the 2DKVE scheme shows better performance compared to SAI or SSAI.

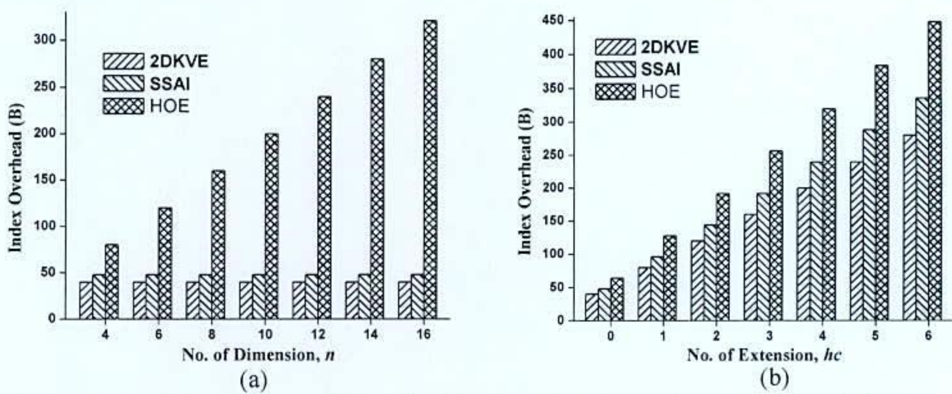


Figure 4.11: Experimental Result of Index Overhead for Encoding Schemes

4.3.2. Range of Usability(μ)

The Compression Ratio (η) of an encoding scheme is defined as the ratio between the compressed array and uncompressed array. The value of η is preferred to be $0 < \eta < 1$. The Range of Usability(μ) of an encoding scheme is defined as the maximum value of ρ up to which the compression ratio η is less than 1.

a) Theoretical Analysis.

The array cell size is α, key size of 2DKVE is β, offset and history of HOE is δ and λ respectively. The array volume of α cells is $V = \alpha \times l^n$. Then,

The total data size of compressed array is: $\sigma = N \times \alpha$

The total key size of compressed array in HOE is: $\tau_{HOE} = N \times \delta + N \times \lambda = N(\delta + \lambda)$

The volume of compressed array is: $v_{HOE} = \sigma + \tau_{HOE} = N \times (\alpha + \delta + \lambda)$

So, the compression ratio of the HOE scheme is as follows:

$$\eta_{HOE} = \frac{v_{HOE}}{V} = \frac{N\,(\alpha+\delta+\lambda)}{\alpha \times l^n} = \frac{N}{l^n} \times \frac{(\alpha+\delta+\lambda)}{\alpha} = \rho \times \frac{(\alpha+\delta+\lambda)}{\alpha} = \rho \times \left(1 + \frac{\delta}{\alpha} + \frac{\lambda}{\alpha}\right) \quad \text{.......(4.1)}$$

The total key size of compressed array in 2DKVE is: $\tau_{2DKVE} = N \times \beta$

The volume of compressed array is: $v_{2DKVE} = \sigma + \tau_{2DKVE} = N \times (\alpha + \beta)$

So, the compression ratio of the proposed scheme is as follows:

$$\eta_{2DKVE} = \frac{v_{2DKVE}}{V} = \frac{N\,(\alpha+\beta)}{\alpha \times l^n} = \frac{N}{l^n} \times \frac{(\alpha+\beta)}{\alpha} = \rho \times \frac{(\alpha+\beta)}{\alpha} = \rho \times \left(1 + \frac{\beta}{\alpha}\right) \quad \text{.....................(4.2)}$$

Table 4.8. Analytical Compression Ratio of Encoding Schemes

ρ	α	HOE $\rho \times \left(1 + \frac{\delta}{\alpha} + \frac{\lambda}{\alpha}\right)$		2DKVE $\rho \times \left(1 + \frac{\beta}{\alpha}\right)$	
		CASE 1: $\alpha = \delta = \lambda$	CASE 2: $\alpha = 2\lambda$	CASE A: $\alpha = \beta$	CASE B: $\alpha = 2\beta$
		$\eta_{HOE} = 3\rho$	$\eta_{HOE} = 2.5\rho$	$\eta_{2DKVE} = 2\rho$	$\eta_{2DKVE} = 1.5\rho$
0.39	8	1.17	0.975	0.78	0.56

Table 4.9. Analytical Usable Length of Encoding Schemes

η	α	HOE $\rho \times \left(1 + \frac{\delta}{\alpha} + \frac{\lambda}{\alpha}\right)$		2DKVE $\rho \times \left(1 + \frac{\beta}{\alpha}\right)$	
		CASE 1: $\alpha = \delta = \lambda$	CASE 2: $\alpha = 2\lambda$	CASE A: $\alpha = \beta$	CASE B: $\alpha = 2\beta$
		$\rho = \frac{\eta}{3}$	$\rho = \frac{\eta}{2.5}$	$\rho = \frac{\eta}{2}$	$\rho = \frac{\eta}{1.5}$
0.9	8	$\mu_{HOE} < 0.34$	$or\ \mu_{HOE} < 0.4$	$\mu_{2DKVE} < 0.5$	$\mu_{2DKVE} < 0.67$

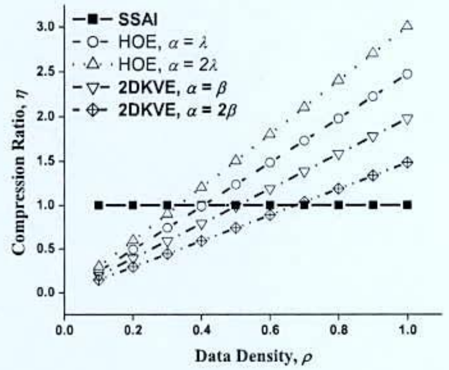


Figure 4.12: Analytical Result of Range of Usabilities of Encoding Schemes

Now, let the value of data density or ρ is fixed and $\rho = 0.39$. Hence, depending on the value of α, β, δ and λ we have two cases for HOE denoted as CASE 1 and CASE 2. The first one is for $\alpha = \delta = \lambda$. As for a HOE scheme $\alpha \neq 2\delta$ hence, the second case CASE 2 is for $\alpha = 2\lambda$. Similary we can have two cases for 2DKVE scheme denoted as CASE A when $\alpha = \beta$ and CASE B when $\alpha = 2\beta$. Then from Eq. 4.1 and Eq. 4.2 we can get some compression ratios such as mentioned in Table. 4.8. From the above table it can be seen that in every cases the 2DKVE scheme has better compression ratio than HOE. For determining the range of usability (μ) of an encoding scheme for the cases such as CASE 1, CASE 2, CASE A and CASE B, let usable $\eta_{HOE} = \eta_{2DKVE} = \eta = 0.9$. Then from Eq. 4.1 and Eq. 4.2 we can get some data density as mentioned in Table. 4.9. In every case the 2DKVE scheme has higher usability than HOE. Hence, the 2DKVE scheme outperforms the HOE scheme as depicted in Fig. 4.12.

b) Experimental Analysis.

As the SSAI structure is an ucompessed data representation, hence value of μ is always 1. From the theoritical analysis we have seen that, the performances of the encoding schemes depend on the value of total key size τ. The HOE scheme involves n dimensional history and $(n-1)$ dimensional offset information. The history information is a small integer to track the dynamic extensions. Hence, it can be assumed the the size of history is less than the size of array cell. For this reason, we can have two cases: $\alpha = \delta = \lambda$ and $\alpha = 2\lambda$. In every cases, the encoding depends on the value of n for an n dimensional history and n-1 dimensional offset. Depending on the value of α for 2DKVE, we have two possible values for β, $\alpha = \beta$ and $\alpha = 2\beta$. But in every cases of 2DKVE scheme, the encoding cost depends only on two parameters $< x_1', x_2' >$ despite of the value of n. So, the 2DKVE

scheme always outperforms the HOE scheme. For every cases , the usability range of HOE is lower compared to 2DKVE scheme. The comparison of the range usabilities of the compared schemes has been depicted in Fig. 4.13(a) which is similar to the theoretical analysis. Fig. 4.13(b) shows the logical volume of the two schemes. From the above figures it can be said that the maximum usability range of HOE is 0.39 whereas the maximum usability range of 2DKVE is 0.66.

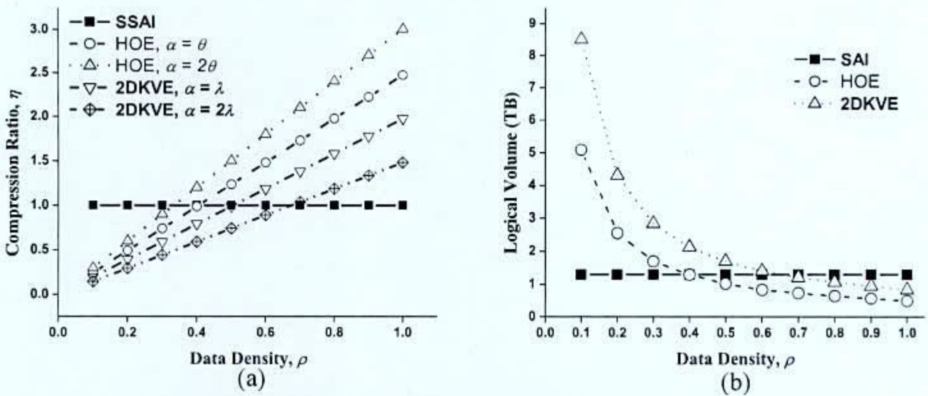


Figure 4.13: Experimental Result of Range of Usabilities

4.3.3. Storage Cost (ζ)

The storage cost is the cost needed to encode a given volume of data from a sparsed structure. This cost includes the volume of non-empty data, the cost of encoding of this volume and also the cost of indexing which manages the scalability of the schemes.

a) Theoretical Analysis.

The storage cost of an encoding scheme ζ is as follows:

$$\zeta = \text{encoding cost} + indexing\ cost$$

$$\zeta = \mathrm{N} \times (\alpha + \tau) + \gamma = \rho \times V \times (\alpha + \tau) + \gamma = \rho \times l^n \times (\alpha + \tau) + \gamma$$

Now, for 2DKVE scheme, the storage cost ζ_{2DKVE} is as follows:

$$\zeta_{2DKVE} = \rho \times l^n(\alpha + \tau_{2DKVE}) + \gamma_{2DKVE} = \rho \times l^n(\alpha + \beta) + 9\gamma(hc + 1)$$

If $\alpha = \beta$, then

$$\zeta_{2DKVE} = 2\rho\alpha \times l^n + \gamma_{2DKVE}$$

And if $\alpha = 2\beta$, then

$$\zeta_{2DKVE} = 3\rho\beta \times l^n + \gamma_{2DKVE} = 1.5\rho\alpha \times l^n + \gamma_{2DKVE}$$

Now, for HOE scheme, the storage cost ζ_{HOE} is as follows:

$$\zeta_{HOE} = \rho \times l^n \times (\alpha + \tau_{HOE}) + \gamma_{HOE} = \rho \times l^n\{\alpha + \beta + \lambda\} + 3n\gamma(hc + 1)$$

If $\alpha = \beta = \lambda$, then

$$\zeta_{HOE} = 3\rho\alpha \times l^n + \gamma_{HOE}$$

and if $\alpha = 2\lambda$, then

$$\zeta_{HOE} = 5\rho\lambda \times l^n + \gamma_{HOE} = 2.5\rho\alpha \times l^n + \gamma_{HOE}$$

Now, consider two cases. In first case denoted as CASE I vary $n = 4, 5$ and set $\rho = 0.2$. In second case denoted as CASE II vary $\rho = 0.4, 0.5$ and set $n = 4$. Let $l = 12$. Then, for the cases mentioned in sec. 4.3.2, the storage cost of the schemes can be calculated as mentioned in Table. 4.10. For CASE I, we can see that the encoding cost in 2DKVE scheme is smaller than HOE as depicted in Fig. 4.14(a) for $n = 4 \sim 9$. Again for CASE II, the encoding cost in 2DKVE scheme is smaller than HOE as depicted in Fig. 4.14(b) for $\rho = 0.1 \sim 0.6$.

Table 4.10. Analytical Storage Cost of Encoding Scemes

α	l	CASE	n	ρ	HOE $\zeta_{HOE} = \rho l^n\{\alpha + \delta + \lambda\} + \gamma_{HOE}$		2DKVE $\zeta_{2DKVE} = \rho l^n(\alpha + \beta) + \gamma_{2DKVE}$	
					CASE 1 $\alpha = \delta = \lambda$	CASE 2 $\alpha = 2\lambda$	CASE A $\alpha = \beta$	CASE B $\alpha = 2\beta$
					$3\rho\alpha l^n + \gamma_{HOE}$	$2.5\rho\alpha l^n + \gamma_{HOE}$	$2\rho\alpha l^n + \gamma_{2DKVE}$	$1.5\rho\alpha l^n + \gamma_{2DKVE}$
8	12	I	4	0.2	99,629	83,040	66,427	49,838
			5		1,194,514	995,448	796,334	597,269
		II	4	0.4	199,162	165,984	132,782	99,605
				0.5	248,928	207,456	165,960	124,488

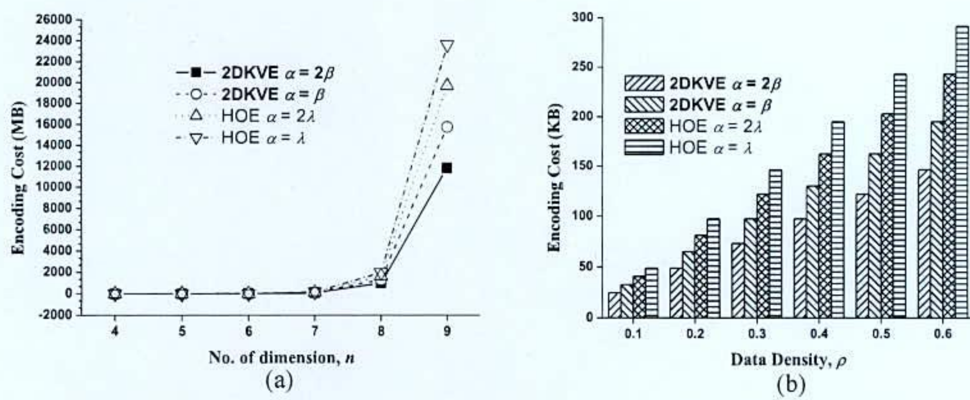


Figure 4.14: Analytical Result of Storage Cost of Encoding Schemes

So, it can be concluded that in every cases, the 2DKVE scheme outperforms the HOE scheme.

b) Experimental Analysis.

In 2DKVE scheme, the indexing cost depends only on the number of extensions (*hc*) of the SA which is constant for initial construction and lowest for varying extensions compared to others. The encoding of 2DKVE involves only data of size α and key of size β which comprises only 2 indices. For HOE scheme the indexing cost not only depends on the the number of extensions (*hc*) of the SA, but also on the value of *n*. Again,the encoding of HOE involves data of size α, $(n-1)$ dimensional offset of size δ and *n* dimensional history of size λ. Thus, the storage cost of HOE is always larger than the storage cost of 2DKVE. Again, dependending on the value of α, β, δ and λ, the performance of ζ_{HOE} and ζ_{2DKVE} varies. For example, the value of ζ_{2DKVE} and ζ_{HOE} is larger when $\alpha = 2\beta$ and $\alpha = 2\lambda$ respectively. The performances of the storage cost of the underlying schemes for fixed *hc*, ρ and for varying *n*, l^n has been depicted in Fig. 4.15(a). Fig. 4.15(b) shows the storage costs for fixed *n* (*n* = 12), $l^n(4^{12})$, and varying ρ.

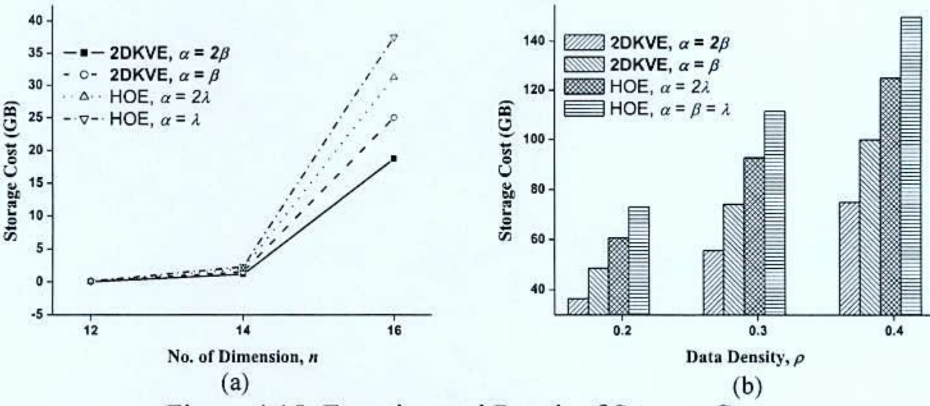


Figure 4.15: Experimental Result of Storage Cost

4.4 Discussion

In this chapter we present the experimental outcomes and also the theoretical analyses of the proposed schemes. We have compared our schemes with the static model like CMA and also with the dynamic models like EA1 and EA2. We have also made comparison between compressed and uncompressed version of the proposed model. In each case we found relevancy with the theoretical analysis and hence we validate the theory. We have showed that the SAI scheme outperforms the EA1 and EA2 schemes and the SSAI outperforms the SAI scheme. Furthermore, our encoding scheme 2DKVE outperforms the conventional HOE scheme.

CHAPTER V

Conclusion

5.1 Summary

Now-a-days large volume of current and future data maintenance has been a key concern in different aspects of data computing like Big Data. But the margin of large volume is changing day bay day as the required size of data is expanding gradually. On the other hand, in real world application the amount of effective data among the large volume is very small as the structure is extremely sparsed. So, it is very important to handle large volume application efficiently with meaningful data only. The conventional multidimensional array systems may comprise many advantages but they cause address space overflow with the increase in length or number of dimension (or both) as they demand reallocation. This consequence degrades their performance drastically even the system has available resources. The Extendible Array strategy can improve the performance of conventional systems by avoiding reallocation but they also suffer from address space overflow as the subarray size grows exponentially. In this research work, we have managed four practical problems of higher order multidimensional data namely (i) n dimensional data representation (ii) extending the length or size of the array dynamically, (iii) decreasing index cost, (iv) dealing address space overflow, and (v) handling sparsity of array.

We describe a new scalable array structure that represents an n dimensional array by a 2 dimensional extendible array named as Scalable Array Indexing (SAI). But like an Extendible Array, the structure also shows address space overflow. For this reason, we modify the SAI structure and renamed the new scalable structure as Segment based Scalable Array Indexing (SSAI) where SAs are represented by a set of segments. The memory is allocated for individual small segments instead of exponential sized SAs. Therefore, the allocation requires less size compared to the other schemes even for large values of length of dimension and number of dimension. Hence, the proposed SSAI structure is able to delay address space overflow with smallest storage overhead. We also propose an encoding scheme based on our proposed SSAI

structure that can encode the sparse data that reduces the indexing cost and encoding cost effectively and named as 2 Dimensional Key Value Encoding (2DKVE).

We have evaluated the proposed SAI and its variant i.e. the SSAI structure and the encoding of SSAI or 2DKVE scheme by theories and experiments. The experimental results confirm the theory for various array operations. Again we have compared our proposed schemes with the static CMA and also with the dynamic models EA1 and EA2 and have found better results for the proposed model.

5.2 Recommendation for Future Work

Since the proposed model is a multidimensional array representation scheme, any application or system that uses multidimensional array to represent data can use the scheme. More specifically –

- This scheme can be successfully applied to database applications especially for multidimensional database or multidimensional data warehousing system [2, 3].
- One important future direction of the work is that; the scheme can be easily implemented in parallel platform [34].
- Because most of the operations described here is independent to each other. Hence it will be very efficient to apply this scheme in distributed array storage, parallel and distributed array storage [8].
- This scheme can be successfully applied to key value storage for big data storage.

REFERENCES

1. Florin Rusu and Yu Cheng, "A Survey on Array Storage, Query Languages, and Systems." arXiv preprint arXiv: 1302.0103, 2013.

2. Chun, Y. L., Jen, S.L. and Yeh, C.C., "Efficient Representation Scheme for Multidimensional Array Operations," IEEE Transactions on Computers, vol. 51(3), pp. 327-354, 2002.

3. P. Baumann, "On the management of multi-dimensional discrete data", The VLDB Journal, vol. 4(3), pp. 401-444, 1994.

4. S. Idreos, F. Groffen, N. Nes, S. Manegold, S. K. Mullender and M. L. Kersten, "MonetDB: Two decades of research in column-oriented database architectures," IEEE Data Engineering Bulletin, vol. 35(1), pp. 40-45, 2012.

5. Y. Zhang, M. L. Kersten, M. Ivanova and N. Nes, "SciQL, Bridging gap between science and relational DBMS", In Proceedings of the 15th Symposium on International Database Engineering & Applications, pp. 124-133, 2011.

6. Katarina Grolinger, Wilson A Higashino, Abhinav Tiwari and Miriam AM Capretz, "Data management in cloud environments: NoSQL and NewSQL data stores", Journal of Cloud Computing: Advances, Systems and Applications, vol.2, 2013.

7. Seyong Lee and Jeffrey S. Vetter, "Early evaluation of directive-based gpu programming models for productive Exascale computing", Proceedings of the 12th International Conference on High Performance Computing, Networking, Storage and Analysis, Article No. 23, 2012.

8. Mingxing Zhang, Yongwei Wu, Kang Chen, Teng Ma and Weimin Zheng, "Measuring and optimizing distributed array programs", In Proceedings of the VLDB Endowment, vol. 9(12), pp. 912-923, 2016.

9. D. Rotem and J. L. Zhao, "Extendible arrays for statistical databases and olap applications", In 8th International Conference on Scientific and Statistical Database Systems, pp. 108-117, 1996.

10. K. M. A. Hasan, M. Kuroda, N. Azuma, T. Tsuji and K. Higuchi, "An extendible array based implementation of relational tables for multi-dimensional databases", Proceedings of the 7th International Conference on Data Warehousing and Knowledge Discovery, pp: 233-242, 2005.

11. E. Otoo, G. Nimako and D. Ohene-Kwoee, "Chunked extendible dense arrays for scientific data storage", Parallel Computing, vol. 39(12), pp. 802-818, 2013.

12. S. M. M. Ahsan and K. M. A. Hasan, "An implementation scheme for multidimensional extendable array operations and its evaluation", International

Conference on Informatics Engineering and Information Science, Part III, CCIS(253), pp: 136-150, 2011.

13. E. Otoo and T. Merrett, "A storage scheme for extendible arrays", Computing, vol. 31(1), pp: 1-9, 1983.
14. Daniel Ohene-Kwofie, E.J. Otoo and Gideon Nimako, "O2-Tree: A Fast Memory Resident Index for In-Memory Databases", International Conference on Information and Knowledge Management, vol. 45, pp: 78-87, 2012.
15. Steve Carr, Kathryn S. McKinley and Chau-Wen Tseng, "Compiler Optimizations for Improving Data Locality", In Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 252-262, 1994.
16. K M Azharul Hasan, Tatsuo Tsuji and Ken Higuchi, "An Efficient MOLAP Basic Data structure and Its Evaluation", In Proceedings of 12th International Conference on Database Systems for Advanced Applications, LNCS, vol. 4443, pp. 288-299, 2007.
17. Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra and Andrew Fikes, and Robert E. Gruber, "Bigtable: A distributed storage system for structured data", ACM Transactions on Computer Systems, vol. 26(2), Article No. 4, pp. 1-26, 2008.
18. Alex Mircea Dumitru, Vlad Merticariu and Peter Baumann, "Array database scalability: intercontinental queries on petabyte datasets", In Proceedings of the 28th International Conference on Scientific and Statistical Database Management, pp. 1-5, 2016.
19. Alex Dumitru, Vlad Merticariu and Peter Baumann, "Exploring cloud opportunities from an array database perspective", In Proceedings of Workshop on Data analytics in the Cloud, pp. 1-4, 2014.
20. Ben Lippmeier, Manuel M. T. Chakravarty, Gabriele Keller and Simon Peyton Jones, "Guiding parallel array fusion with indexed types". In Proceedings of the 2012 Haskell Symposium, pp. 25-36, 2012.
21. Min Chen, Shiwen Mao and Yunhao Liu, "Big Data: A survey", Mobile Networks and Applications, vol. 19(2), pp. 171–209, 2014.
22. Kostas Zoumpatianos, Stratos Idreos and Themis Palpanas, "Indexing for Interactive Exploration of Big Data Series", In Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data, pp. 1555-1566, 2014.
23. Michael Stonebraker and David Dewitt, "Requirements for Science Data Bases and SciDB", In 4th Biennial Conference on Innovative Data System Research Perspectives, 2009.

24. Rosenberg, A.L., "Allocating Storage for Extendible Arrays", Journal of the ACM (JACM), vol. 21, pp. 652-670, 1974.

25. Mano, M.M., "Digital Logic and Computer Design", Prentice Hall, 2005.

26. K. M. Azharul Hasan and Md Abu Hanif Shaikh, "Efficient representation of higher-dimensional arrays by dimension transformation", Journal of Supercomputing, vol. 73(6), pp. 2801-2822, 2017.

27. P. Baumann, A. Dehmel, P. Furtado, R. Ritsch and N. Widmann, "The multidimensional database system RasDaMan", In Proceedings of the 1998 ACM SIGMOD International Conference on Management of data, vol. 25(2), pp. 575-577, 1998.

28. Maarten Vermeij, Wilko Quak, Martin Kersten and Niels Nes, "MonetDB, a novel spatial column-store DBMS", In Academic Proceedings of the 2008 Free and Open Source for Geospatial Conference, OSGeo, pp. 193–199, 2008.

29. Weixiong Rao, "MonetDB And The Application For IR Searches", University Of Helsinki. Seminar Paper. Column-Oriented Systems, 2012.

30. Peter Baumann, Alex Mircea Dumitru and Vlad Merticariu, "The array database that is not a data-base: file based array query answering in Rasdaman", In Proceedings of the 13th International Conference on Advances in Spatial and Temporal Databases, pp. 478-483, 2013.

31. Sándor Héman, Marcin Zukowski, Arjen De Vries and Peter Boncz, "Efficient and flexible infor-mation retrieval using MonetDB/X100", In Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research, pp. 96-101, 2007.

32. Haozhou Wang, Kai Zheng, Xiaofang Zhou and Shazia Sadiq, "SharkDB: An in-memory storage system for massive trajectory data", In Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management, pp.1409-1418, 2015.

33. Yaoliang Chen, Xiaomin Xu, Pohan Li, Siyuan Lu, Sheng Huang, Wei Lu and Kevin Brown, "Geo-Mix: Scalable geoscientific array data management", In Proceedings of the Industrial Track of the 13th ACM/IFIP/USENIX International Middleware Conference, Article No. 1, pp. 1-6, 2013.

34. Viet-Trung Tran, Bogdan Nicolae and Gabriel Antoniu, "Towards scalable array-oriented active storage: the pyramid approach", ACM SIGOPS Operating Systems Review, vol. 46(1), pp. 19-25, 2012.

35. Yihong Zhao, Prasad M. Deshpande and Jeffrey F. Naughton, "An Array Based Algorithm for Simultaneous Multidimensional Aggregate", In Proceedings of the

1997 ACM SIGMOD International Conference on Management of data, pp. 159-170, 1997.

36. Tsuji, T., Hara, A. and Higuchi, K., "An Extendible Multidimensional Array System for MOLAP", In Proceedings of the ACM symposium on Applied computing, pp. 23-27, 2006.

37. Sk. Md. Masudul Ahsan and K. M. Azharul Hasan, "Segment Oriented Compression Scheme for MOLAP Based on Extendible Multidimensional Arrays", Journal of Computing and Information Technology, vol. 23(2), pp: 111-121, 2015.

38. Masafumi Makino, Tatsuo Tsuji and Ken Higuchi, "History-Pattern Implementation for Large-Scale Dynamic Multidimensional Datasets and Its Evaluations", In Proceedings of the 20th International Conference on Database Systems for Advanced Applications, Part II, LNCS, vol. 9050, pp. 275–291, 2015.

39. A. Sudoh, T. Tsuji and K. Higuchii, "A Partitioning Scheme for Big Dynamic Trees", In Proceedings of the 22th International Conference on Database Systems for Advanced Applications, LNCS, vol. 10179, pp.18-34, 2017.

40. D. Lemire and C. Rupp, "Upscaledb: Efficient integer-key compression in a key-value store using simd instructions", Information Systems, vol. 66, pp.13-23, 2017.

41. R. Barrett, M. W. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine and H. van der Vorst., "Templates for the solution of linear systems: Building blocks for iterative methods", SIAM Press, 1994.

42. Chun-Yuan Lin, Yeh-Ching Chung and Jen-Shiuh Liu, "Efficient data compression methods for multidimensional sparse array operations based on the EKMR scheme", IEEE Transactions on Computers, vol. 52(12), pp. 1640-1646, 2003.

43. Bei Li, Katsuya Kawaguchi, Tatsuo Tsuji and Ken Higuchi, "A Labeling Scheme for Dynamic XML Trees Based on History-offset Encoding", 2nd International Conference on Future Computer and Communication, pp: 71-87, 2010.

44. Tsuchida, T., Tsuji, T. and Higuchi, K., "Implementing Vertical Splitting for Large Scale Multidimensional Datasets and Its Evaluations". In Proceedings of the 13th International Conference on Data warehousing and knowledge discovery, LNCS vol. 6862, pp. 208-223, 2011.

45. Tatsuo Tsuji, Keita Amaki, Hiroomi Nishino and Ken Higuchi, "History-Offset Implementation Scheme of XML Documents and Its Evaluations", In Proceedings of the 18th International Conference on Database Systems for Advanced Applications, Part I, LNCS, vol. 7825, pp. 315-330, 2013.