

Abstract

Curse of dimensionality is the most prominent problem for higher dimensional data. Degree of data sparsity increases with the increasing number of dimensions in high performance scientific computing. Storing and applying operations on this highly sparse multidimensional data is still a challenge for data scientists. Multidimensional array is widely used as basic data structure. But the performance of multidimensional array is very poor when the number of dimension is very high. Data scientist suggests linearization of dimensions for higher dimensional data. But linearized data are highly compute intensive for retrieving into original data format as well as operations are very expensive on that data. It suffers from the higher index computational cost and lower data locality. Experts suggest special storage scheme over sparse array. In traditional sparse array storage scheme, $(n+1)$ one dimensional arrays are necessary to store n -dimensional array. But here we have used one of the most effective technique named G2A(Generalized 2 Dimensional Array) which can convert any dimensional array to 2 dimensional array. We used key value storage technique to generate keys against our stored values. Key value storage technique is flexible, highly scalable, high availability, high performance and provides operational simplicity. They can exist on distributed systems and don't need to worry about where to store indexes, how much data exists on each system or the speed of a network within a distributed system they just work. It stores a unique key in respect of each value. These keys are unique for each value and these keys are used to retrieve data from the real dataset. Also different types of query operations are performed by these keys rather than using the values. These keys are stored in a bin in secondary memory and their link is stored in a list in primary memory. At the end we also make some comparisons for storage and query operations(Exact, Single, Range key query) with MySQL query techniques.

1. Introduction

A key-value store or key-value database is a simple database that uses an associative array (as like map or dictionary) as the fundamental data model where each key is associated with one and only one value in a collection. This relationship is referred to as a key-value pair.

Here each pair has a key and an associated value for that key.

Key-value stores scale out by implementing partitioning (storing data on more than one node) replication and auto recovery. They can scale up by maintaining the database in RAM and minimize the effects of ACID (Atomicity, Consistency, Isolation, Durability) guarantees by avoiding locks, latches and low-overhead server calls. The simplicity and flexibility of a key-value store allows development staff to focus on tuning, scaling, monitoring and notification rather than architecture design and modeling. For most key-value stores, the secret to its speed lies in its simplicity. The path to retrieve data is a direct request to the object in memory or on disk. The relationship between data does not have to be calculated by a query language; there is no optimization performed. They can exist on distributed systems and don't need to worry about where to store indexes, how much data exists on each system or the speed of a network within a distributed system where they have to work.

2. Related Works

Many techniques have been proposed in literature for improving the cost of index computation for higher dimensional structured data [3]-[9]. Some techniques suggest linearization over higher dimensions[7]-[10]. Their argument is that the compiler allocates memory sequentially in all cases of memory allocation. EKMR scheme[3] is very effective but it is limited to four dimensions only. When the number of dimension is greater than four then it requires an abstract pointer array. Even there is no generalized indexing till four dimensional. Tamara G. Golda et.al [1][2] proposed an indexing scheme for two dimensional generalization. Though it is very effective for operation on generalized data but requires huge storage. Steve Carr [13] presents compiler optimizations to improve data locality but the scheme is based on the Traditional Multidimensional Array. It reorganizes the loops only for

compiler optimization. A new storage manager for complex, parallel array processing namely the Array Store is proposed in [12] in the area of multidimensional data storage. The main issue handled is the way to find for parallel processing. The SciDB[14][15] proposed an array based DB model. EJ Otoo[11] introduce new storage schemes for multidimensional sparse arrays that handle the sparsity of the array but the array is not n dimensional. D dynamic array structure namely EKA[8][10] becomes an hierarchical structure when n>4. Y. Zhao et. al.[9] introduced the chunking of arrays but linearization of n dimensional array is required. The simplified representation of higher dimensional array suggest two dimensional representation scheme in [5][6]. This representation highly affects for main memory computing saving huge cache locality. This two dimensional representation scheme is cost effective in terms of index computation specially when n is large.

3. Methodology

3.1 G2A:

The *Generalized Two-dimensional Array* (G2A) represents an algorithm to represent an n dimensional (nD) array by a 2 dimensional (2D) array. The nD array is converted to a 2D array. Hence the indexes of the nD array are also converted to 2D array. n/2 columns contribute to row direction and the rest n/2 columns to column direction. Hence a nD array can be drawn in a 2D plane to visualize the data.[2]

Let $A[l_1][l_2][l_3][l_4]$ be a TMA(4) of size $[l_1, l_2, l_3, l_4]$. A multidimensional array is linearized and programming languages map the array index into the linearized memory address. Therefore a location of the tuple $\langle x_1, x_2, x_3, x_4 \rangle$ can be linearized by

$$f(x_1, x_2, x_3, x_4) = x_1 l_2 l_3 l_4 + x_2 l_3 l_4 + x_3 l_4 + x_4$$

In G2A, the 4 dimensions are converted into 2 dimensions. Fig-2.1 shows the G2A $A'[l'_1][l'_2]$ for a TMA(4) $A[2][3][3][2]$ where $l'_1 = l_1 \times l_3$ and $l'_2 = l_2 \times l_4$; $x'_1 = x_1 l_3 + x_3$ and $x'_2 = x_2 l_4 + x_4$; $x_i = 0, 1, 2, \dots, (l_i - 1)$ ($1 \leq i \leq 4$).

Consider an element of $A[1][1][2][0]$. The corresponding element is therefore $A'[x'_1][x'_2]$ (see Fig. 2.1) where $x'_1 = 1 \times l_3 + 2 = 5$ and $x'_2 = 1 \times l_4 + 0 = 2$.

If an element in G2A is $A'[x'_1][x'_2]$ is known then it's equivalent $A[x_1][x_2][x_3][x_4]$ can be found as $x_3 = x'_1 \% l_3$ and $x_1 = x'_1 / l_3$; $x_4 = x'_2 \% l_2$ and $x_2 = x'_2 / l_2$.

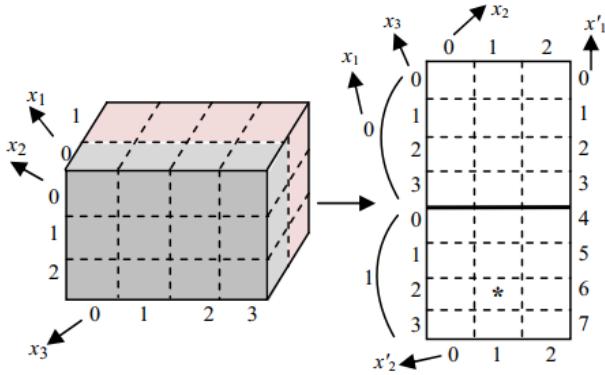


Figure 2.1: TMA(3) and it's equivalent G2A

Figure 2.2:G2A, equivalent to TMA(4) size(2*2*3*3)

3.1.1 G2A for TMA(n)

The generalization of the TMA(n) $A[l_1][l_2] \dots [l_n]$ to G2A $A'[l'_1][l'_2]$ is as follows :

$$l'_1 = \begin{cases} l_1 \times l_3 \times \dots \times l_{n-1}, & \text{if } n \text{ is even} \\ l_1 \times l_3 \times \dots \times l_n, & \text{if } n \text{ is odd} \end{cases} \text{ and } l'_2 = \begin{cases} l_2 \times l_4 \times \dots \times l_n, & \text{if } n \text{ is even} \\ l_2 \times l_4 \times \dots \times l_{n-1}, & \text{if } n \text{ is odd} \end{cases}$$

An element $A[x_1][x_2]\dots[x_n]$ in TMA(n) is equivalent to $A'[x'_1][x'_2]$ where

$$x'_1 = \begin{cases} x_1 l_3 l_5 \dots l_{n-3} l_{n-1} + x_3 l_5 l_7 \dots l_{n-3} l_{n-1} + \dots + x_{n-3} l_{n-1} + x_{n-1}, & \text{if } n \text{ is even} \\ x_1 l_3 l_5 \dots l_{n-2} l_n + x_3 l_5 l_7 \dots l_{n-2} l_n + \dots + x_{n-2} l_n + x_n, & \text{if } n \text{ is odd} \end{cases}$$

$$x'_2 = \begin{cases} x_2 l_4 l_6 \dots l_{n-3} l_{n-1} + x_3 l_5 l_7 \dots l_{n-3} l_{n-1} + \dots + x_{n-3} l_{n-1} + x_{n-1}, & \text{if } n \text{ is odd} \\ x_2 l_4 l_6 \dots l_{n-2} l_n + x_4 l_6 l_8 \dots l_{n-2} l_n + \dots + x_{n-2} l_n + x_n, & \text{if } n \text{ is even} \end{cases}$$

3.2 Key value Store Scheme:

A **key-value database**, or **key-value store**, is a data storage paradigm designed for storing, retrieving, and managing associative arrays, a data structure more commonly known today as a dictionary or hash. Dictionaries contain a collection of objects, or records, which in turn have many different fields within them, each containing data. These records are stored and retrieved using a key that uniquely identifies the record, and is used to quickly find the data within the database.

Key-value databases work in a very different fashion from the better known relational databases (RDB). RDBs pre-define the data structure in the database as a series of tables containing fields with well defined data types. Exposing the data types to the database program allows it to apply a number of optimizations. In contrast, key-value systems treat the data as a

single opaque collection, which may have different fields for every record. This offers considerable flexibility and more closely follows modern concepts like object-oriented programming. Because optional values are not represented by placeholders as in most RDBs, key-value databases often use far less memory to store the same database, which can lead to large performance gains in certain workloads. Performance, a lack of standardization and other issues limited key-value systems to niche uses for many years, but the rapid move to cloud computing after 2010 has led to a renaissance as part of the broader NoSQL movement. Some graph databases are also key-value databases internally, adding the concept of the relationships (pointers) between records as a first class data type.

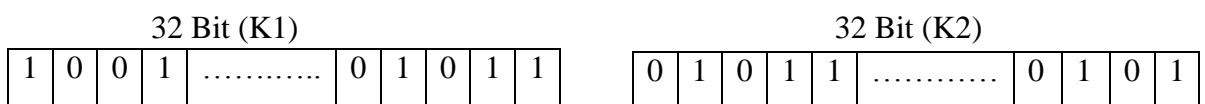
Here is an example of key value storage for stock market data where the list contains the stock ticker, whether it is a “buy” or “sell” order, the number of shares, and the price:

Key	Value
123456789	APPL, Buy, 100, 84.47
234567890	CERN, Sell, 50, 52.78
345678901	JAZZ, Buy, 235, 145.06
456789012	AVGO, Buy, 300, 124.50

3.3 Key value Storing Technique:

Key value generation process:

- (1) At first, take k1(x1) in 64 bit unsigned long long integer.
- (2) Then make 32 bit left shift operation.
- (3) Then take k2(x2) in another 64 bit unsigned long long integer.
- (4) Now make an OR operation between these two (k1 & k2) to make a key K.



K1 in 64 bit unsigned long long integer

0	0	0	0	0	0	0	1	0	1	1
---	---	---	---	---	---	-------	---	---	---	---	---

K1 in 64 bit unsigned long long integer (After 32 bit left Shifting)

1	0	0	1	0	0	0	0	0
---	---	---	---	-------	---	---	---	---	---

K2 in 64 bit unsigned long long integer

0	0	0	0	0	0	0	1	0	1
---	---	---	---	---	---	-------	---	---	---	---

Generation of 64 bit K(After OR operation between K1 & K2)

1	0	0	1	0	1	0	1
---	---	---	---	-------	---	---	---	---

Now we need to store the keys in fixed size bins. Because we want to save our keys in the permanent storage (Secondary memory) and the first index of each bin will be stored in a list. We stored a huge number of keys in the fixed size bins. It can be noted that the bin size will be a fixed number where each bin will be able to contain from thousands to millions of keys if needed.

These equal width bins are stored in secondary memory as .bin file and their links are maintained in a list which are stored in a primary memory.

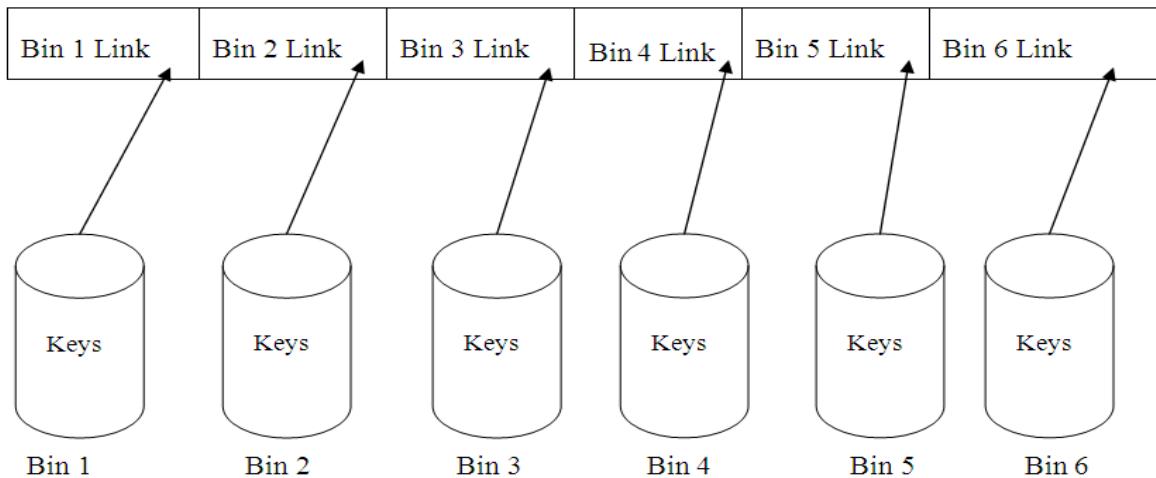


Figure 3.3.2.1: Keys stored in equal width bins in secondary memory linked with a list stored in primary memory.

4. Query Operations

4.1 Exact Key Query:

- (1) In exact match query we try to check the attributes of an objects whether it exists or not in the real datasets.
- (2) For each object we enter its attribute values one by one.
- (3) Then the program took those input and using the formula it creates a key.
- (4) After it started scanning the list of the indexes of the bins of key values.
- (5) When it got the related indexes of the bin then it searches for any match among the key values in the corresponding bin.
- (6) If it found any matches with the keys in that bin then the object is exist in the real datasets.
- (7) Otherwise the object does not exist in the real datasets.

4.2 Single Key Query:

- (1) In single key query we try to find out the relationships among objects based on a single attribute.
- (2) To find this relation we need to perform a permutation operation.
- (3) We need to determine all the permutations among the distinct values of each column of the dataset except the column for which value we are trying to find out relation.
- (4) We used recursive methods to do these huge number of permutations.
- (5) Then we store the index of all permutations in a set.
- (6) After we calculated the corresponding keys for each of the permuted objects using their index value.
- (7) Then for each key value the program scans for appropriate bin of key values.
- (8) When the corresponding bin found then the calculated key of each permuted object is searched there.
- (9) If the searching found any match then it denotes that there is relationship between two objects in respect of the single key.
- (10) If no match found then it denotes that there is no relationships between these two objects in respect of the single key.

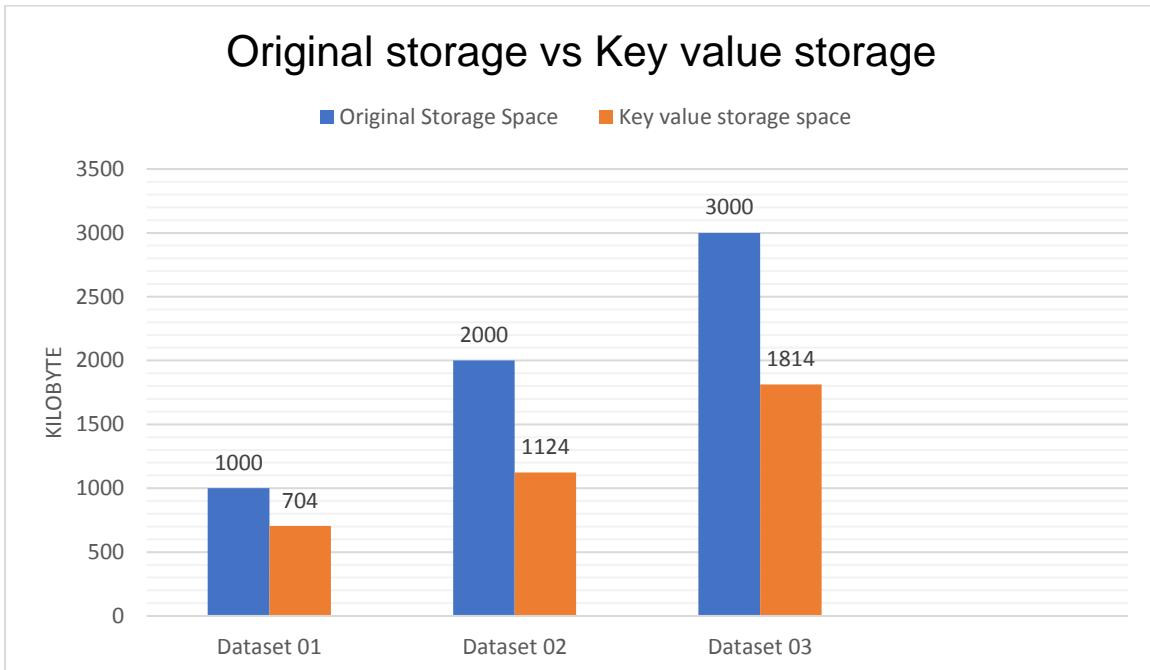
4.3 Range Key Query:

- (1) In range key query we try to find out the relationships among objects based on a range of values of a single attribute.
- (2) To find this relation we again need to perform a huge permutation operation.
- (3) We need to determine all the permutations among the distinct values of each column of the dataset except the column for which values in range we are trying to find out relation.
- (4) We used recursive methods to do these huge number of permutations.
- (5) Then we store the index of all permutations in a set.
- (6) After we calculated the corresponding keys for each of the permuted objects using their index value.
- (7) Then for each key value the program scans for appropriate bin of key values.
- (8) When the corresponding bin found then the calculated key of each permuted object is searched there.
- (9) If the searching found any match then it denotes that there is relationship between two objects in respect of the range keys.
- (10) If no match found then it denotes that there is no relationships between these two objects in respect of the range keys.

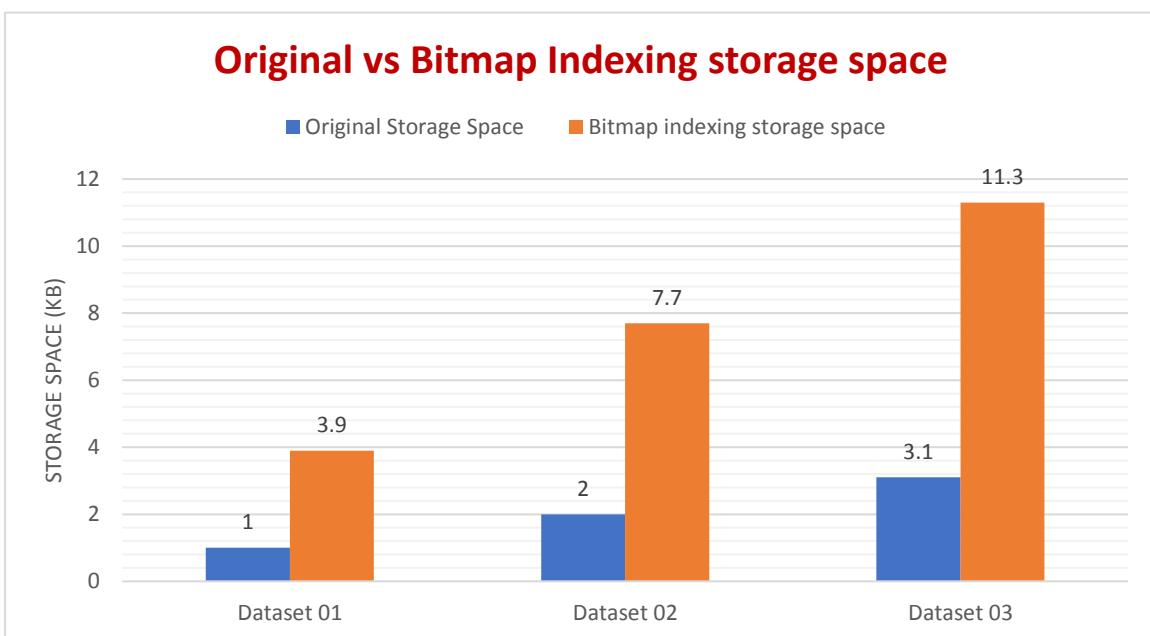
5. Experimental Performance

5.1 Storage Cost:

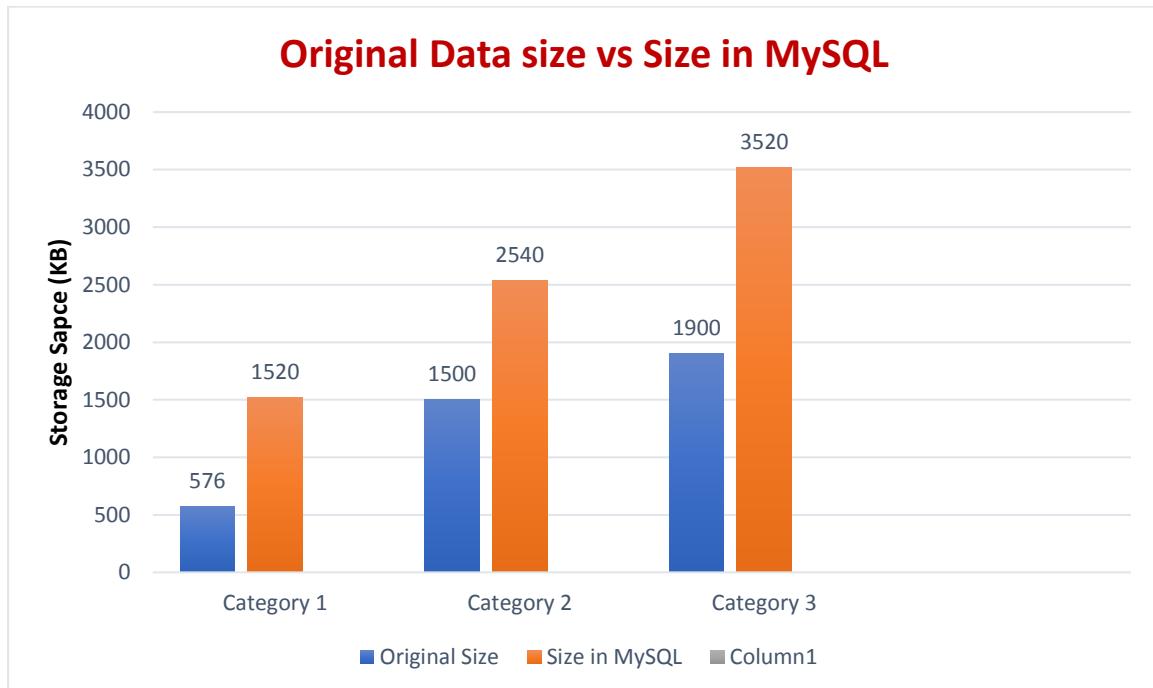
5.1.1 Original Storage Space vs Key Value Storage Space:



5.1.2 Original Storage Space vs Bit Map Indexing Storage Space:

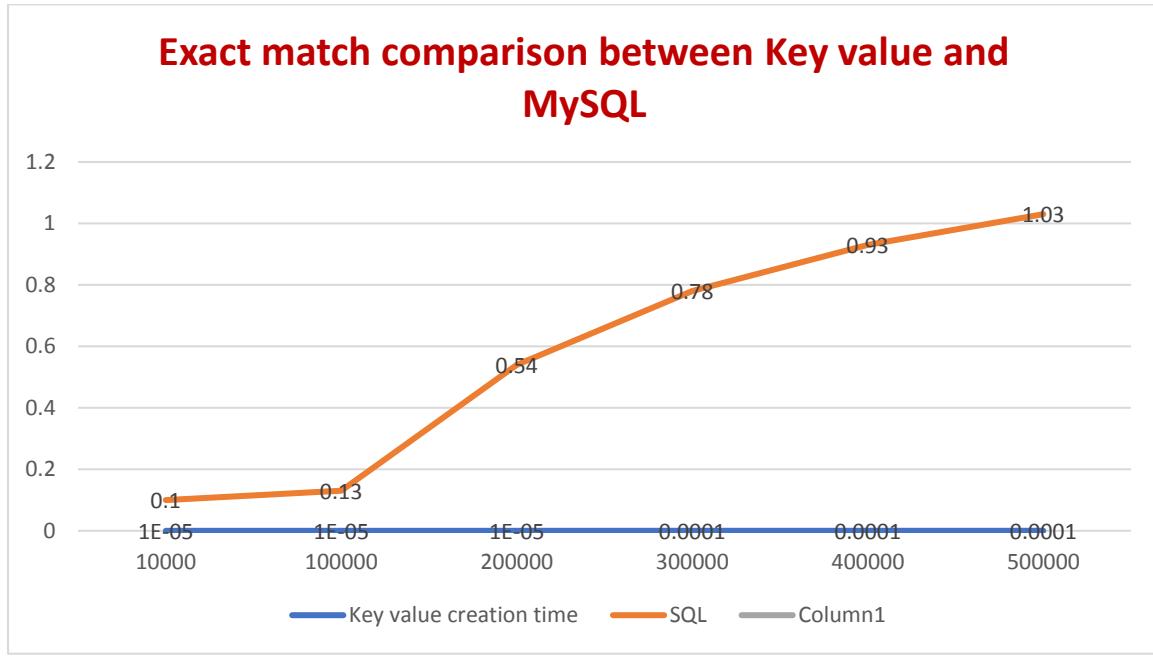


5.1.3 Original Storage Space vs MySQL Storage Space:

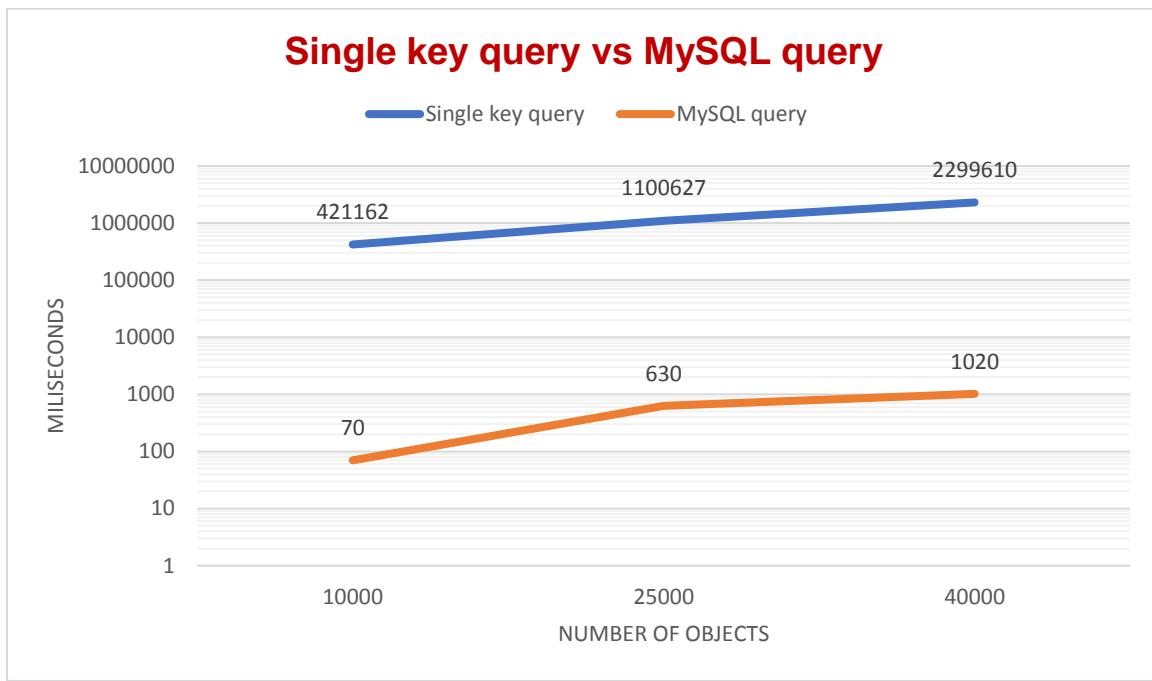


5.2 Query processing

5.2.1 Query processing time for exact key search :



5.2.2 Query processing time for single key search :



5.2.3 Query processing time for range key search :

Processing time for range key query is the multiplication of the number of single keys in the range.

6. Conclusions

We all know that the upcoming future is big data. As data is increasing enormously day by day. Different types of data such as online transaction data, social networking data, IOT(Internet Of Things) data, audio & video data, Online communication data, satellite dataetc and a lot of data from different sources. As the dimensionality of data increases data become more and more sparse. To perform operations on them such as storage, query, mining become too much complex. Key value storage is one of the best technique to handle higher dimensional data efficiently. Because we just need to store the corresponding keys here rather than a huge number of values. We can storing, retrieving, searching, query operation can be done more efficiently by this key value storage techniques. It also significantly reduces the storage space.

7. REFERENCES

1. K. M. Azharul Hasan, Md. Abu Hanif Shaikh, Computer Science and Engineering Department, Khulna University of Engineering & Technology, Khulna-9203, Bangladesh. "Representing Higher Dimensional Arrays into a Generalized Two-dimensional Array",
2. Tamara G. Kolda, Brett W. Bader, *Tensor Decompositions and Applications*, SIAM Review 51(3), Page:455-500, 2009.
3. Chun-Yuan Lin, Jen-Shiu Liu, and Yeh-Ching Chung, Efficient Representation Scheme for Multidimensional Array Operations, IEEE Transactions on Computers, 51(3), pp.327-345, 2002.
4. Kostas Zoumpatianos, Stratos Idreos, Themis Palpanas, Indexing for Interactive Exploration of Big Data Series, SIGMOD'14, June 22–27, 2014, Snowbird, UT, USA.
5. K. M. Azharul Hasan, Md Abu Hanif Shaikh, Representing Higher Dimensional Arrays into Generalized Two-dimensional Array: G2A, 16th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT 2015), Published by SpringerLNEE, Jeju, South Korea.
6. Md Abu Hanif Shaikh, K.M. Azharul Hasan, Efficient Storage Scheme for n-Dimensional Sparse Array: GCRS/GCCS, 13th International Conference on High Performance Computing & Simulation (HPCS2015), Page (137-142), Published by IEEE, Amsterdam, Netherlands.
7. B. Christian, M. Urs, "Multidimensional Index Structures in Relational Databases", Intelligent Information Systems, 15, pp. 51–70, 2000
8. Sk Md Masudul Ahsan, K M Azharul Hasan "An Implementation Scheme for Multidimensional Extendable Array Operations and Its Evaluation", ICIEIS, Part III, CCIS 253, pp. 136–150, 2011.
9. Y. Zhao, P. M. Deshpande, and J. F. Naughton, "An array based Algorithm for simultaneous multidimensional aggregates", ACM SIGMOD, pp. 159–170, 1997.
10. K M Azharul Hasan, Tatsuo Tsuji, Ken Higuchi "An Efficient MOLAP Basic Data structure and Its Evaluation", Proc. of DASFAA, pp. 288-299, LNCS 4443, 2011.
11. EJ Otoo, H Wang, G Nimako, "New Approaches to Storing and Manipulating Multi-Dimensional Sparse Arrays", Proc. of SSDBM'14, pp. 2014.
12. Emad Soroush and Magdalena Balazinska "ArrayStore: A Storage Manager for Complex Parallel Array Processing" In Proc. Of ACM SIGMOD International Conference on Management of data, pp.253-264, 2011.
13. Steve Carr , Kathryn S. McKinley , Chau-Wen Tseng "Compiler optimizations for improving data locality", In Proc. of the sixth international conference on Architectural support for programming languages and operating systems, p.252-262, 1994.
14. Michael Stonebraker, David Dewitt Requirements for Science DataBases and SciDB, CIDR Perspectives 2009.