RESEARCH ARTICLE OPEN ACCESS

DOMI: A Dynamic Ordering Multi-field Index for Data Processing

SuraI.Mohammed[1], Hussien M. Sharaf[2], Fatma A. Omara[3]
1(Computer science department, Cairo University/Faculty of computer science, Egypt, Cairo)
2 (IT Experts from Masr (I.T.E.M.), Software Corp., Egypt, Cairo)
3 (Computer science department, Cairo University/Faculty of computer science, Egypt, Cairo)

Abstract:

The information retrieval from Big Data requires more efficient techniques for data indexing. According to the work in this paper, a Dynamic Order Multi-field Index (DOMI) structure has been introduced and implemented. The proposed DOMI indexing structure allows dynamic rather than sequential ordering of fields, in addition, compacting values with common prefixes. Hence, the DOMI allows efficiently indexing of huge data sets and answering queries that involve multi-fields, as well as, queries that involve a single field. A comparative study by building a composite-field index among the proposed DOMI and other popular indexing data structures such as B and B+ trees has been implemented. The comparison results show that DOMI composite indexing soutperform other composite indexing structures in case of answering a single field query that addresses a non-leading field. The performance of DOMI is slightly better than that of B+ in case of answering a composite-field query. Therefore, DOMI covers different cases of data retrieval more efficiently. Generally, DOMI could be the most suitable indexing structure for intensive-data such as Big Data because it is based on radix trees which compacts common prefixes into a single occurrence. Our experiments show that DOMI is effective, scalable, and efficient for supporting structural data queries.

Keywords:- **Dynamic Order Multi-field Index (DOMI), Composite field query, Single field query, Indexing structures, Data processing, Big Data.**

I. INTRODUCTION :

The growing of datasets indexing becomes more important. Many researchers have directed their efforts to facilitate faster execution of queries. Index and query technologies are critical for facilitating interactive exploration of large datasets which leads to the new era of Big data [2]. Big data refers to interesting high-velocity, high value, and/or high-variety data with volumes beyond the ability of commonly-used software to capture, manage and process within a tolerable elapsed time[1]. Also, Big data refers to voluminous data which surpass the abilities of the current database technology. So, a delay in the retrieval of data can be expected. For this reason, a query over vast amounts of unsorted data might be considered time consuming. Otherwise, the size of the data itself becomes part of the problem.

However, there are two major phases for supporting queries on large data sets. The first phase concerns about preparing data sets. The second phase concerns about providing an answer to a query after using indexing of data that results from the first phase. Regarding fields' indexing, it is necessary to perform the process of organizing data into tuples before query execution. The chosen index should need minimal preprocessing of the input data stream.

Now adays, an attention is concerned about the processes of sorting, indexing and searching of data through the Data Stream Model, data arrives at high speed, and the algorithms which process this data under space and time constraints [3]. But, datastream introduces various challenges for designing data indexing approach where the limited purse "Time" should be considered. This purse must be dealt with efficiently and at a low cost. So, indexing mechanisms are needed to help fast information retrieval especially within large data sets where the main objective of indexing is to optimize the speed of queries processing [4]. According to the work in this paper, a Dynamic Multi-Field Index (DOMI) based on the Radix Tree as a basic structure has been introduced. DOMI covers different cases of data retrieval more efficiently. Also, DOMI presents a single index structure that can be used for single-field and composite-field queries. In case of single-field queries, DOMI avoids sequential search among the stored index tuples because it can dynamically sort data with respect to any indexed field even if it is a non-leading field. Moreover, the process of building DOMI does not need much preprocessing of the input data whether it is a static or a streaming data

.

An implementation of a dynamic index using layers of radix trees together with a single hash table has been proposed. The proposed DOMI structure improves the performance of handleling different types queries. Each layer of the radix trees can be traversed without sequentially visiting the unrelated layers. Hence, the look-up performance for queries addressing a single field is handled efficiently, as well as, queries addressing multiple fields.

This paper is organized as follows; Section II discusses some of the related work. Section III describes DOMI structure. Section IV presents the implementation of DOMI architecture including the construction of the index and the search performance. Section V lists the experiments of using DOMI architecture and entails with the results of each experiment. Finally, section VI presents the conclusions of the paper.

II. BACKGROUND AND RELATED WORK:

Predilection of processing queries over large data set has been contributed in the improvement of index structure, such as projections strategy indexes (RB+), Value-List Index (B+ tree) and more complex index structures to speed-up the process of queries evaluation [7]. Although, efficient query processing specifications have been achieved from these indexes, querying huge data sets, especially streaming data, have suffered from the time overheads during answering the queries that involve multiple fields in addition to queries that involve single field in their filtering criteria.

According to the survey in [6], the problem of designing efficient indexes to support spatial objects has been addressed. On the other hands, the reason of creating an index for a data set is to speed up the access of a subset of the data [8]. Index structures are different in terms of structure, query support, data type support and applications [9]. Query Processing in QPPT (Query Processing Prfix Trees) keeps the index materialization cost low, and uses optimal prefix trees to satisfy balanced read/write operations (i.e., memory optimization) [11]. Based on the used data structure to store tuples of fields, B-tree is considered a simple data strucutre that permits storing vertical partitions in traditional B-tree indexes with practically zero overheads for storing the tuples [10]. Yet, B-trees and B+ trees store composite fields into one tuple with a static order. As a result, B-trees and B+ trees respond to queries of the non-leading fields with almost a sequential search which affect the search efficiency.

Compared to the existing indexing strucutres, using radix trees relatively compact the data by storing common prefixes once. Reducing the needed space is very helpful especially with huge static data-sets or streaming data. An efficient search is considered the most important criterion for selecting data structures because searching of data is normally carried out on-line which needs fast response and will be carried out many times [12].

Bitmap indexes have become popularin in this context. In a bitmap index, leaf pages of an index structure don't contain lists of record ids but bit vectors with one bit for each data record. Several types of bitmap index structures suitable for different query types are discussed in [13]. Bitmap indexes, however, are static because the insertion of a data record needs all index entries to be updated.

Therfore, , The proposed DOMI architecture focuses on scenarios of searching where searching might involve multiple fields, as well as, a single field where one index structure should be used to handle all scenarios. DOMI is structured in form of layers of radix trees where each layer is dedicated for a field. Layers can be accessed independently or collaboratively resulting in a dynamical mechanism (see Fig.1). A comarative study has been done to exam the offered speedup by our DOMI technique and and other techniques. An implementation has been done using three index strucutres; DOMI, B-trees and B+ trees. The structure of DOMI and query processing strategies have been discussed in details in [5].

III. DOMI STRUCTURE:

In this section, the proposed Dynamic Order Multi-field Index (DOMI) structure is presented. DOMI is based on radix trees, and combines a set of fields into layers of radix trees. A hashtable is used to refer only to the roots of each level to allow direct and fast access to lower layers of the DOMI. Candidate fields are decided according to the choice of the index designer before runtime. The designer of such index should take into account the relationship between the fields. Fields with less distinct values should be at the higher levels in order to decrease the branching.

Each level consists of ***n*** number of the radix trees. The leaf nodes of each radix tree at level ***i*** may refer to a single radix tree root in level ***i+1***. Hence, the values of each field are grouped into a layer that may contain one or more Sub-Radix Trees (SRT). The DOMI can be partitioned with respect to semantic categorization where each semantic partition can be mapped to a separate node in a distributed system.

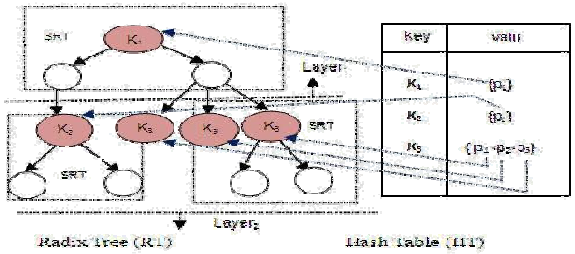


Fig. 1 **DOMI Structure**

In Fig.1, the radix tree has two types of nodes; dark nodes and white nodes. The white nodes contain the data value. Conversely, the dark nodes contain keyes. Each pointer (p) refers to a root node of the relevant field. All pointers of a field are stored in a single entry at the Hash Table (HT) to eliminate the need of traversing the upper layers of the DOMI. As a result, when a query is received, the DOMI immediately moves towards the hash table to determine the suitable root at the right level, and then process the query accordingly

IV. THE IMPLEMENTATION DETAILS:

The main goal of DOMI implementation is to enable building and querying of sets of data concurrently. Hence, the implementation is designed in form of concurrent processes. The main components of the system areas are as follows (see Fig.2):

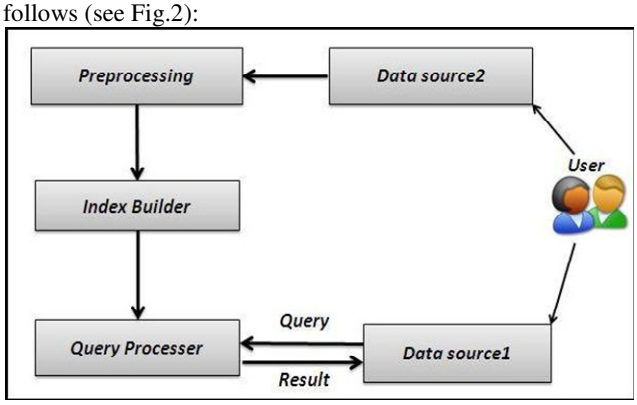


Fig. 2 **Architecture of the Implementation**

Preprocessing; It is intended to partition data into fields for indexing and querying. It reads a chunk of the input data set in a buffer. Each buffer is processed independently without losing the sequencing of the dataset.

Index Builder; It builds the indexes for a subset of a dataset and stores the indexes inthe memory.

The Query Processor; It accepts dataset queries from the user, then performs compare data selection using the stored indexes and returns the common records as a finalreport to the user.

These processes are implemented in form of multi producers and consumers. Each process is instantiated as soon as it receives an input from its predecessor. All processes work concurrently and collaboratively. The architecture allows processing large amounts of streaming data. So, the memory and CPU are used efficiently. Data is transferred from one component to the next component through a shared FIFO-queue. Processes are grouped in form of a pipeline. The first process in the pipeline splits a stream of data into a set of equal sized buffers and passes them as in-memory buffers to the next process. The Scenario of the implementation consists of eight processes as shown in (Fig.3).

User

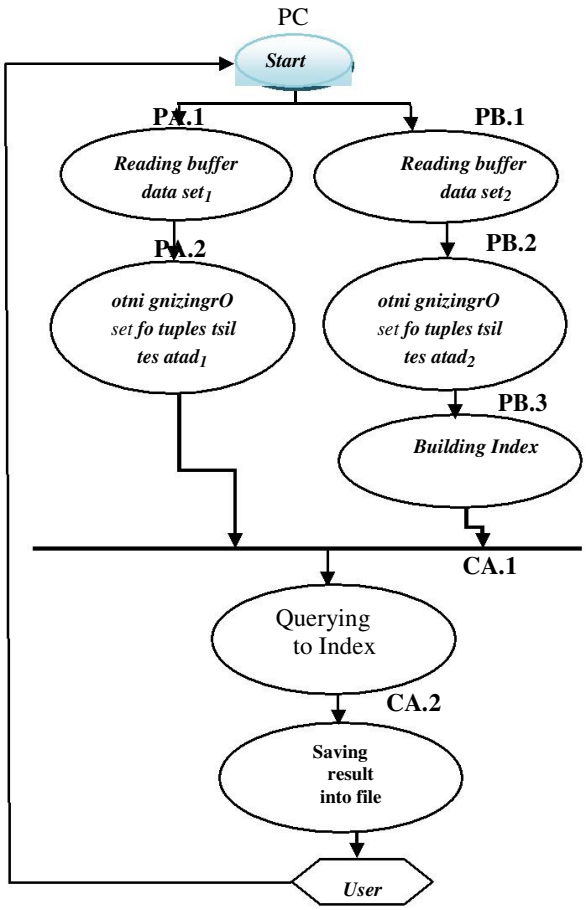


Fig. 3 **Implementation processes query**

The program starts with process PC that recieves two datasets, initiates one shared queue and two threads for processing each dataset. The start process is briefly described in the the following algorthim.

PC: Algorithm Start (DataSet1, Dataset2)

1: StartTime ← Now()
2: SharedQueue← Initiate a shared queue
3: Initiate a thread for PA.1 (DataSet1,SharedQueue)
4: Initiate a thread for PB.1 (DataSet2,SharedQueue)
5: Initiate a thread for CA.1 (SharedQueue)

Algorthim 1. Start PC

The partitioning of a stream is presented in Fig.4. The partitioning of a stream can be controlled using three parameters; Stream, Nbytes, and Start Position. 'Stream' indicates a data source stream, 'Nbytes' controls the buffer size, and 'Start Position' is an optional parameter that starts reading from a chosen location within the input stream.

Next, both PB.2 and PA.processes organize each buffer by processes PB.1 PA.1 into a list of tuples where each tuple consists of well defined fields.

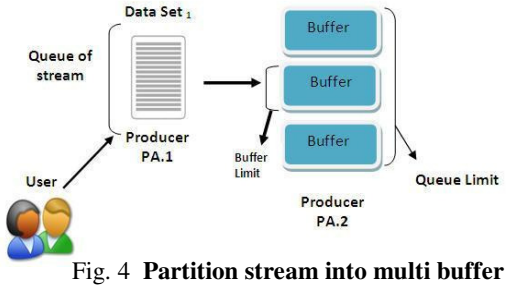


Fig. 4 **Partition stream into multi buffer**

The convertion of two data set into a list of tuples is stated in the the following algorthim.

PA.2,PB.2:Algorithm list of tuples (DataSet1, Dataset2)

1: for each buffer
2: for Line buffer
3: Tuple← LineToTuple (line, ListofFieldNames) 4: Append "Tuple" into list of tuples

Algorthim 2. List of tuples PA.2,PB.2

The algorithm consists of three steps. First, it scans the buffer that is received from process PB.1 and process PA.1, and plits buffer into line (Line 1-2). Second, it converts each line into tuples (Line 3). Third, it appends each Tuple into the list that is shared with the next process (Line 4).

Process PB.3 constructs the index by selecting a set of values that map to the chosen set of indexed fields. The details of the building of the index process (PB.3) is stated in the the following algorthim.

PB.3: Algorithm Build Index from buffer ()

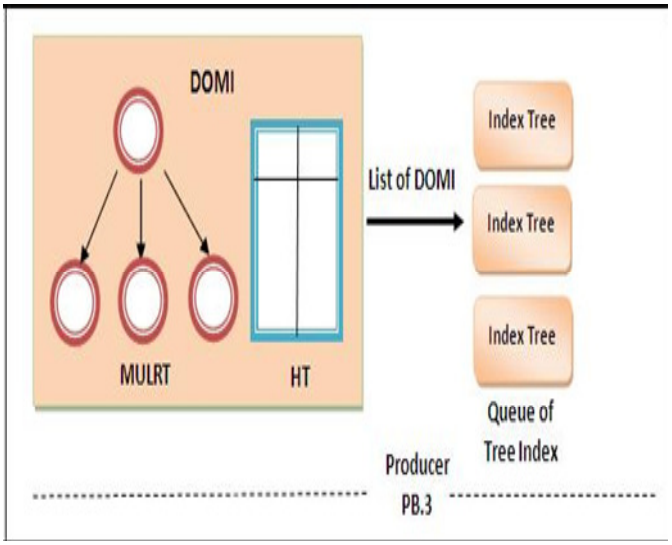
1: for each buffer
2: for Line buffer
3: Indexed Values←select fields from buffer()
4: IndexTree.Insert(Indexed Values,completetuple)
5: Append "index Tree" into a shared queue

Algorthim 3. BuildingIndex PB.3

The algorithm consists of three steps. First, it scans the buffer and adds qualifying tuples into the result (Line 1-2). Second, it selects the fields that should be indexed and inserts a complete tuple into the chosen index structure (DOMI, B-tree or B+ tree) (Line 3-4). Third, it appends each index into the queue that is shared with the next process (Line 5). Each buffer is loaded into a separate index (See Fig.5). The size of a buffer controls the depth of the tree of the index.

Algorithm CA.1 search performance ()

1: indexTree_Lstset ← of indexTrees "DOMI"
2: for indexTree indexTree_Lst
3: for $Ddict_2$ Tupleslist of query
4: if "the query is a composite-field query"
5: Result=indexTree.findTuple(Ddict1,List of compare fields1)
6:else
7: Result=indexTree.findvalue(Ddict1, List of compare fields1)
8: if Result is not NULL
9: Append Result to MatchingList



DOMI internal strucutre

Then, PB.3 reads each index tree "DOMI" and adds it into a list of indexes. Both processes PA.2 and PB.3 are synchronized before CA.1 starts to execute. CA.1 is the process that does the actual data comparison by joining the indexed data tuples extracted from data source2 and given stream tuples from the data source1. A join operation starts by pulling one list of tuples produced by PA.2, where each list maps to a buffer to compare it with indexed data (see Fig.6).

The CA.1 algorthim consists of three essential steps. First, it extracts the set of values that are intended for comparing the two datasets (Line 1-2). Next, it sends this dictionary of values to the "DOMI" to check for its existing (Line 3). The DOMI answers with a complete tuple if the data is found or answers with NULL if the tuple is not found (Line 4-8).

Algorthim 4. Algorthim of search peformance

It returns the result of the comparing data-source1 and data-source2 after joining the common compound values to the user in form of a list of all matching lines (Line 9). Finally, each list is appended as a single entry into a shared queue. The shared queueis are received and processed by process CA.1. Process CA.2 retrieves each list from the shared queue and writes it into a file saved on disk as a final result for the end user (see Fig. 6).

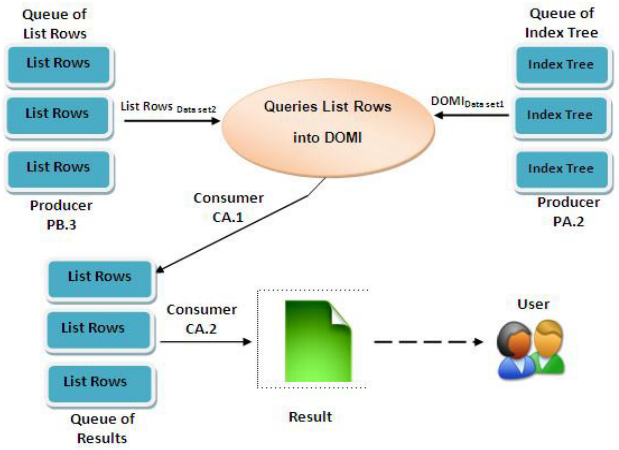


Fig. 6 **Comparative to Retrieval Information of user**

VI. The Performance Evaluation:

A. Experimental Enviroment

Experiments are executed on a hardware environment of an AMD FX(tm)-8320-Eight Core processor at 3.50 GHz with 32 GB of main memory. The operating system is Windows 7 64 bit edition and the coding language is Python 2.7. Each run is performed in form of a pool of background threads. The objective of these experiments is to compare the execution time of building an index and execution time of querying this index several times to get common records between two homogeneous datasets.

B. Experimental Data

The star schema benchmark (SSB) is used [14]. The datasets are generated using a scale factor (SF) parameter of size six million records for "Order" table. The query involved in the experiment is based on Order table. For all experiments, the "**lo_orderkey", "lo_linenumber"** and **"lo_custkey"** fields compound together into a single composite index. A separate subset of the "Order" table of size five hundard thousand records is created. The subset is used to query the index five hundard thousand times to get the common records. The performance of a join is measured by calculating the total time which is taken to process all records of the subset table against the index built for the original "Order" table. The experiments measure the performance of building index with a variety of sizes and using it. For each buffer size, a comparsion is made against indexes built based on B, B+ trees.

C. Building Composite-Field Index Results

The index is created in memory based on the chosen data strucutre. The construction time of building index is evaluated, as well as, the performance of querying the index five hundard thousand times to get a join of two given datasets. All measurements use the same cost model. Three parameters are identified to describe the behavior of the different indexing structures. The three parameters are the available buffer size, the number of records for building index, and the number of tuples for querying. In all cases, the index is built as a single composite index on the three fields mentioned above.

Experiment C.1: The first experiment illustrates the building a composite-field index based on DOMI, B-tree and B+ tree with an input buffer of 5 MB of data. As a result, a series of 109 index trees is constructed for holding 6,000,000 records of the "Order" table where each index holds relatively few amount of data. The depth of each tree whether it is a DOMI, B-tree, B+ tree, is relatively small. The insertion time for the DOMI, B+ tree and B-tree is presented in Fig. 7. The insertion time is significantly high for the index based on B-tree. DOMI and B+ tree have the best performance. DOMI is faster by 10.8% and 49% than B+ tree and B-tree respectively.

Experiment C.2: The second experiment illustrates building a composite-field index based on DOMI, B-tree and B+ tree with an input buffer of 250 MB of data. Accordingly, each index holds much number of records and the depth of each tree whether it is a B-tree, or B+ tree is relatively higher than those constructed in experiment C.1. Yet, the number of branches of the DOMI are bigger than the DOMI constructed in experiment C.1 leading to longer insertion time.

A series of three index trees is constructed for holding 6,000,000 records of the "Order" table where each index holds relatively higher amount of data. The building of DOMI index is faster by 22.17% and by 52% than B+ tree index, and B-tree index respectively (see Fig. 8).

Experiment C.3: The third experiment illustrates the building a composite-field index based on DOMI, B-tree and B+ tree with an input buffer of 1000 MB of data. Accordingly, each index holds much number of records and the depth of each tree whether it is a B-tree, B+ tree is relatively higher than those constructed in experiment C.2.

Yet, the depth and the width of the DOMI do not grow much bigger than the DOMI constructed at experiment C.2 which leads to a nearly equal insertion time. Since DOMI is based on radix trees that stores common prefixes once, therefore the insertion time becomes constant.

A single index tree is constructed for holding 6,000,000 records of the "Order" table where each index holds relatively higher amount of data than each index which is constructed in experiments C.1 and C.2. The building DOMI index is faster by 4.7% and 51% than B+ tree index B-tree index respectively (see Fig. 9).

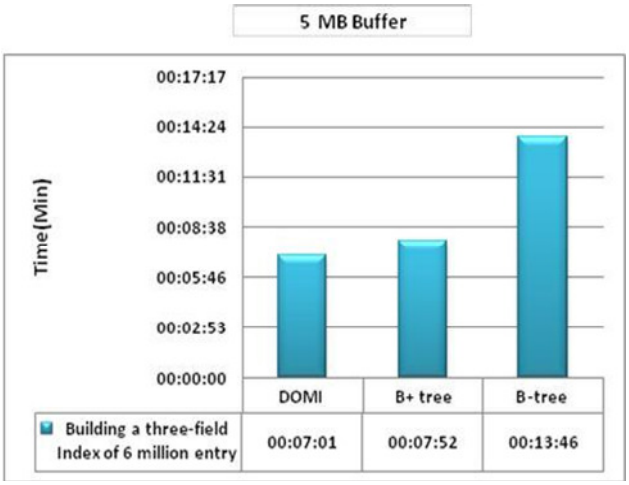


Fig. 7 **Insertion Time of three index with 5 MB buffer**

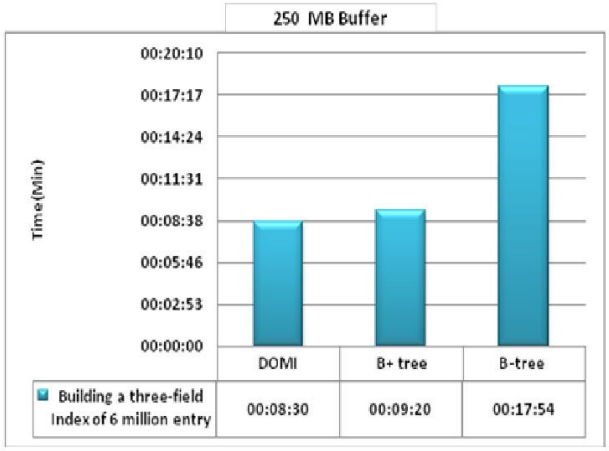


Fig. 8 **Insertion Time of three index with 250 MB buffer**

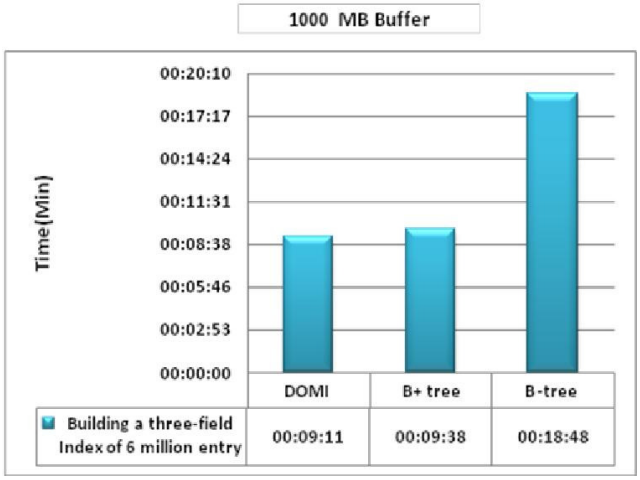


Fig. 9 **Insertion Time of three index with 1000 MB buffer**

The relative performances of the data structures are different for the three indexes with different buffers sizes. This indicates that small buffer (5 MB) is faster than larger buffers (250 MB and 1000 MB) because each index has least depth. Regarding experiments C.2 and C.3, the depth of B and B+ trees tuples much bigger which leads to a higher insertion time. Regarding experiments C.2 and C.3, DOMI does not grow in depth but it grows in width which leads to a higher insertion time than the constructed DOMI in experiment C.1 (see Fig. 9).

D. Executing a composite-field Query Results

The goal of this experiment is to evaluate the performance of the composite-field index for each of the three index structures; DOMI, B-tree and B+-tree. In each experiment, the workload consists of 500,000 queries. Each query is executed several times with different input data sizes. During a join operation, the system stores the matched stream tuples based on a values of composed of multi-fields.

Experiment D.1: A workload of 500,000 queries is run against a series of 109 indexes that hold a volume of 6,000,000 records. For the comparison, the execution time of 500,000 queries against the constructed composite-field index is presented in Fig. 10. In case of the index constructed by 5 MB of data, the lookup time is significantly higher for B-tree based index. Experiment D.1 shows that DOMI and B+ trees have better searching performance than B-trees. According to Fig.10, it is found that DOMI is faster by 25.9% and 69% than B+ tree and B-tree respectively.

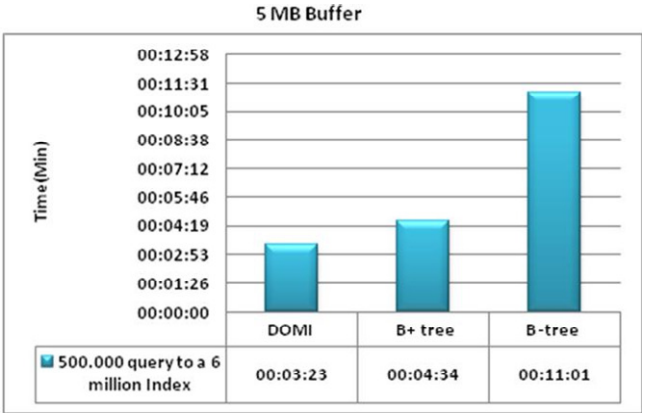


Fig. 10 **Query Time of multi value with 5 MB buffer**

Experiment D.2: A workload of 500,000 queries is run against a series of three indexes that hold a volume of 6,000,000 records. The lookup time of DOMI is faster by 13% and 47.5% than B+ tree and B-tree respectively (see Fig.11).

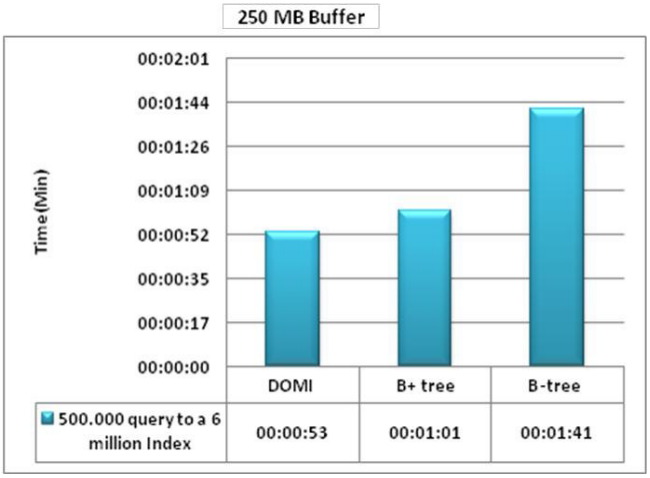


Fig. 11 **Query Timeof multi field with 250 MB buffer**

Experiment D.3: A workload of 500,000 queries is run against a single index that holds a volume of 6,000,000 records. Fig.12 shows that in case of the constructed index using 1000 MB of data, the lookup time of DOMI is faster by 7% and 25.9% than B+tree and B-tree respectively. Fig.12 shows that the performance of searching enhances as the data are grouped into a big index rather than a list of small indexes. This indicates that large index (with 1000 MB of data) is the best for all index types. So, the performance of querying a single big index is much better that that of querying a list of small-sized index. In case of a single big index, the elimination of the loop against a list of small sized-indexes provids better performance.

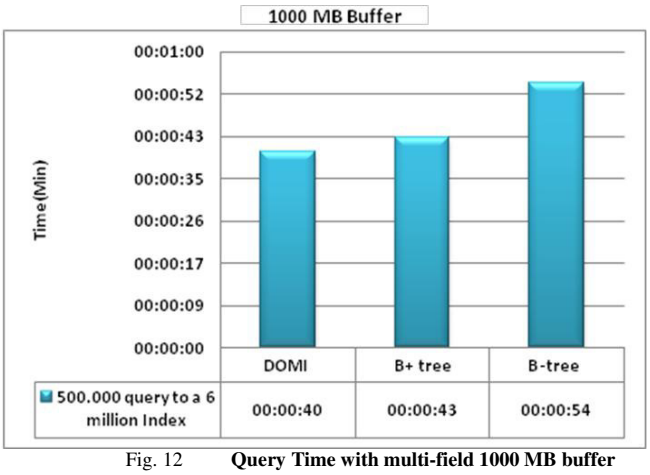


Fig. 12 **Query Time with multi-field 1000 MB buffer**

E. Executing a single-field queryResults

The goal of the single-field query experiments is to search for a value of a single field among a composite-field index. In each experiment, the workload consists of executing 1000 query operations against the same composite indexes that are constructed and explained in section C. Each query searches for values of the field named

"lo_custkey". "lo_custkey" is the third field of the composite fields that constitute the indexes mentioned in section C

Experiment E.1: A workload of 1,000 queries is run against a list of 109 composite-field indexes that holds a volume of 6,000,000 records. Each index holds 5MB of data. Fig.13 represents the execution time of 1000 queries. The lookup time of DOMI is faster by 8% and 48% than B+ tree and B-tree based index respectively (see Fig. 13).

Experiment E.2: A workload of 1,000 queries is run against a list of three composite-field indexes that holds a volume of 6,000,000 records. Each index holds 250MB of data. The lookup time of DOMI is faster by 9% and 73% than B+ tree and B-tree respectively (see Fig.14).

Experiment E.3: A workload of 1,000 queries is run against a single composite-field index that holds a volume of 6,000,000 records. Each index holds less than 1000 MB of data. According to the experiment E.3, it is found that the lookup time of DOMI is faster by 36% and 81% than B+ tree B-tree respectively (see Fig. 15).

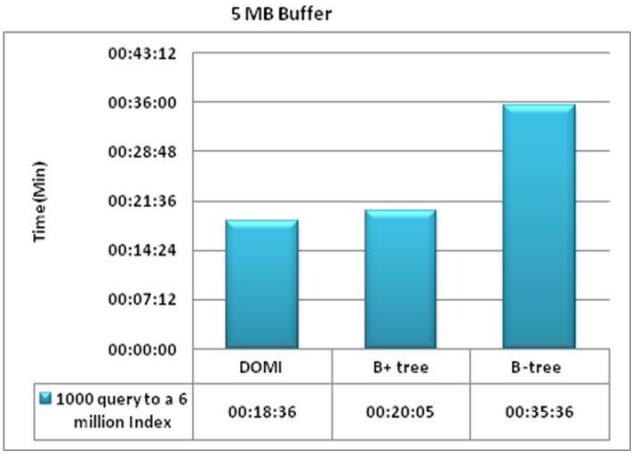


Fig. 13 **Query Time with single field 5 MB buffer**

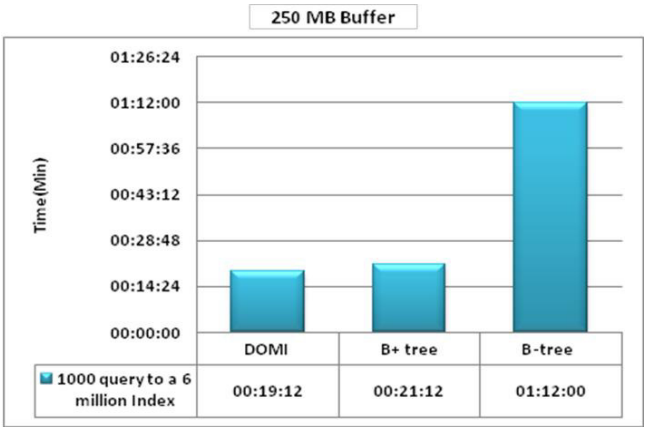


Fig. 14 **Query Time with single field 250 MB buffer**

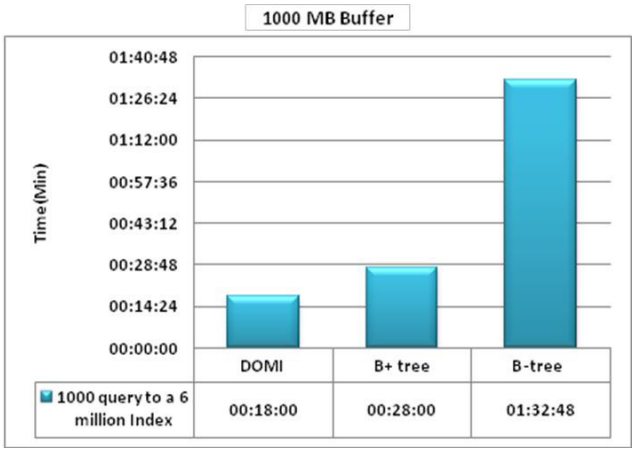


Fig. 15 **Query Time with single field 1000 MB buffer**

VII. Conclusion:

The proposed DOMI architecture presents a dynamic data structure that consists of layers of radix trees. It allows a query processing to be proceed directly to the relevant layer. So, the query of a single value can be answered without traversing unnecessary levels of nodes. This structure allows answering multiple queries based on composed-fields and to build compact index based on radix trees.

A comparative study has been done among our proposed DOMI, B-tree and B+-tree by implementing and executing different experiments to measure the speed-up in case of a composite-field query, and a single-field query using differnet buffer sizes to measure the building time and query time of different index sizes.

One of the main and common inability in the plurality of the B and B+ indexing structures is the space and the required time to store and search composite-fields. Experiments are conducted to show the efficiency of the proposed DOMI indexing structure comapering to B-tree and B+-tree indexing structures. Generally, the proposed DOMI indexing structure is considered a promosing indexing structure for the execution of queries over big data. Future enhancements with respect to the compression can produce a more successful Big Data-indexing structutre.

REFERENCES:

[1] Jiang, Fan,"Reducing the Search Space for Big Data Mining for Interesting Patterns from Uncertain Data", Big Data (BigData Congress), 2014 IEEE International Congress, Anchorage, AK , pp 315 - 322 , 2014.

[2] S. Madden, "From databases to big data," IEEE Internet Computing, Vol. 16, No. 3, pp. 4–6, May–June 2012

[3] Albert Bifet, "Mining Big Data in Real Time ", The International Journal of Computing and Informatics, Vol. 37, No.1, pp. 15–20, March 2013.

[4] Ajit Singh, Deepak Garg "Implementation and Performance Analysis of Exponential Tree Sorting," the IJCA International Journal of Computer Applications, Vol. 24, No.3 , pp. 34-38, June 2011.

[5] Sura I. Mohammed, Hussien M. Sharaf, and Fatma A. Omara, "Information Retrieval using Dynamic Indexing," Proceedings of the 9th International Conferenceof the Informatics and Systems (INFOS 2014), Cairo, Egypt, pp. PDC-93 - PDC-101, Dec. 2014.

[6] V. Gaede and O. Gu¨ nther, "Multidimensional Access Methods," ACM Computing Surveys, Vol. 30, No. 6, pp. 170-231, June 1998.

[7] P. O'Neil, D. Quass, "Improved Query Performance with Variant Indexes", International Conference on Management of Data (MOD) , ACM, New York, USA, pp. 38-49, 1997.

[8] Lisa A. Horwitz, "Techniques for Managing Large Data Sets:Compression, Indexing and Summarization," Proceedings of the 1997 North East SAS Users Group, pp. 30-37, 1997.

[9] P. Patel, D Garg," Comparison of Advance Tree Data Structures," the IJCA International Journal of Computer Applications,Vol. 41, No. 2, 2012.

[10] Goetz Graefe, "Efficient columnar storage in B-trees," ACM SIGMOD Record 36 (1), New York, USA pp. 3-6 , 2007.

[11] K.Ramamohanarao, John W. Lloyd, "Dynamic Hashing Schemes," ACM Computing Surveys, Vol.20, No.2, pp. 850-113, June1998.

[12] Guojun Lu, "Techniques and Data Structures for Efficient Multimedia Retrieval Based on Similarity," IEEE Circuits & Systems Society, Vol.4, No.3, pp. 372 - 384, Sep 2002.

[13] O'Neil P., Graefe G.: "Multi-Table Joins through Bitmapped Join Indexes," SIGMOD Record, Vol. 24, No. 3, pp. 8-11, 1995.

[14] Electronic Publication: Pat O'Neil, Patrick E., Elizabeth J. O'Neil, and Xuedong Chen, "The star schema benchmark (SSB)", Jan 2007.