Order-By-Queries in Elf

Supervised By - Dr-Ing. David Broneske

Aravind Lade
M.Sc. Digital Engineering
University of Magdeburg
aravind.lade@st.ovgu.de

Azima Islam Fariha
M.Sc. Data and Knowledge Engineering
University of Magdeburg
azima.fariha@st.ovgu.de

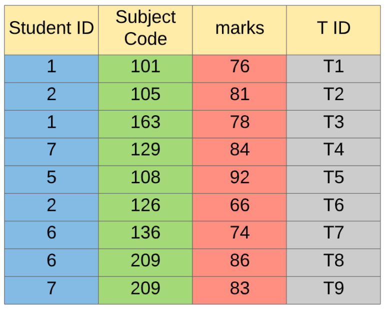
Jalaj Vora
M.Sc. Data and Knowledge Engineering
University of Magdeburg
jalaj.vora@st.ovgu.de

Lakshmikanth Chippa
M.Sc. Digital Engineering
University of Magdeburg
lakshmikanth.chippa@st.ovgu.de

Shilpa Babu
M.Sc. Digital Engineering
University of Magdeburg
shilpa.babu@st.ovgu.de

Vedamsh Reddy Vanja
M.Sc. Digital Engineering
University of Magdeburg
vedamsh.vanja@st.ovgu.de

***Abstract*—The increase in processing time and memory consumption while processing data in rows has been a significant problem that we are facing. With the explosion of Big-Data the ordering performance of our analytical queries is notably slower in Column Sequential. As a solution to this problem, we introduced an index structure, Elf, which has the property of pre-sorted nodes in the first dimension, that we intend to exploit in our implementation of Order-By queries. However, we also introduced an extension to the Elf structure namely LOD (Locally Ordered at every dimension) Elf to further demonstrate which one outperforms the Order-By query. In this paper we investigated the performance criteria between Column Sequential, Elf and LOD Elf. Our results depict that, for a smaller dataset the Column-Sequential outperforms both Elf and LOD Elf whereas for larger datasets both the Elf versions, especially LOD Elf is competitively performing better at the initial dimensions.**



Student ID	Subject Code	marks	T ID
1	101	76	T1
2	105	81	T2
1	163	78	T3
7	129	84	T4
5	108	92	T5
2	126	66	T6
6	136	74	T7
6	209	86	T8
7	209	83	T9

Fig. 1. An Example Table

I. Introduction

Order-By query is an important operator in analytical systems and a very resource intensive operation that is used to sort the fetched data in either ascending or descending order according to the user query. In order to run through a set of data it needs a fair amount of CPU time, but the main problem is that the database must temporarily buffer the results. So, the Order-By operation must read the complete input before it can produce the first output[4]. With the current data explosion in science and technology poses difficulties for data management and data analytics [6]. Though the runtime in Order-By is logarithmic, still it goes up really high for Big-Data. In order to have an acceptable runtime, we have to come up with a concrete solution.

The basic design of Elf contains pre-sorted dimensions which is designed for efficient multi-dimensional querying that reduces the effort of sorting. On the other hand, we have the column store where we have to sort the full plain data. The fastest would be to use sorted index structures. For a better understanding, we present an example of Elf's efficient memory utilization in the following example table (Fig 1).

We observe in fig 1 that, the subject code and marks are mentioned with respect to student IDs. We have three dimensions as student ID, subject code and marks. This data will be converted into Elf structure using Prefix Redundancy Elimination. Each column has attribute values that repeats multiple times in the table and shares the same prefix value. Here, Elf helps us eliminating the redundancy by entering an instance only once in a `Dimension List`. For example, we refer to the Fig 1, where the index structure maps the distinct values of a column to one single `Dimension List`. From this first column we get the distinct values for the student IDs. The first dimension consists redundant student IDs in the `Dimension List`. Here with the help of Prefix Redundancy Elimination property of Elf, we have unique `Dimension Elements` in the First dimension. In the next dimension, we cannot see the prefix redundancy elimination as all attribute columns in this dimension are unique. The third dimension consists of Tuple Identier. This structure has fixed depth, as the number of columns in the table are fixed. Hence, the dimensions are also fixed.

Ordered node elements:

Each `Dimension List` is an ordered list of entries. This property is beneficial for equality or range predicates as we can stop the search in a list if the current value is bigger than the searched constant/range.

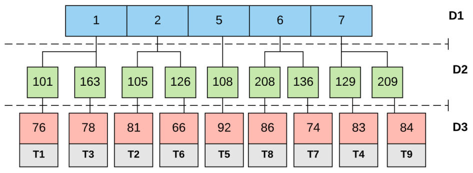


Fig. 2. Prefix Redundancy Elimination in Elf

Fixed depth:

Here a column of a table corresponds to a level in the Elf, for a table with n columns, we have to descend at most n nodes to find the corresponding `Tuple IDs`(TIDs). This sets an upper bound on the search cost that does not depend on the amount of stored tuples, but mostly on the amount of used columns.

Hash Map:

The first dimension is already ordered, this allows us to directly create a Hashmap. Thereby removing the overhead for processing a potentially higher number of entries. In general, Pointer to the next dimension is stored right next to the value. But when it is hashed only the Pointers to the next dimension are stored.

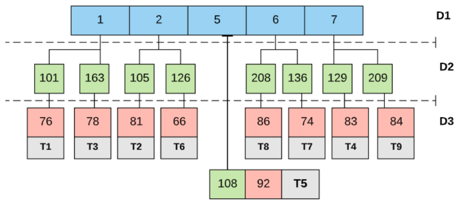


Fig. 3. Monolist in Elf

MonoLists:

In a sparse index structure, we traverse towards the leaf node, the lists probably contain single object which leads to only one TID. This is because of the absence of Prefix redundancy. Due to this, both the Pointer and the value are being stored which leads to a single TID. `MonoLists` are introduced to append next dimensions to the same list i.e. we shift from columnar layout to row wise layout as it can be seen from fig 3, it is observed that the `MonoList` for node 5 in the Elf structure. We would like to get a promising structure as current analytical queries-Elf. The data stored is pre-sorted and combines Order-By with Elf.

In this paper we investigate efficient ways to implement Order-By queries in Elf. In order to do that we compare it to the Column Sequential, Elf and the LOD (Locally Ordered at every Dimension) Elf. Our results show that, with smaller datasets Column Sequential is performing better. However, in a bigger prospect, Elf and LOD Elf is outperforming Column Sequential for larger datasets. We have divided our paper into the following section as below:

- Section II deals with analyzing the traditional data structure of Order-By operators and an introduction to the functionality of Elf.
- Section III deals with the Implementation of Order-By queries used in Column Sequential, Elf and LOD Elf.
- Section IV follows the Evaluation of our results.
- Section V summarizes about our entire contribution and results.

II. Related Work

A. B Trees

B-Trees are used in relational database mainly for storing and retrieving data. Based on partial key information we can retrieve data, avoiding large scans of huge tables. B-Trees are not well suited for sequential processing environment. Many users often prefer sequential view on files where the next operation is used widely for processing the records in a key sequential order. For example, the smallest key is located in the leftmost of the leaf, finding this key requires all the nodes of the path from the root of the leaf itself. MDAM (Multidimensional Access Method) is a method where multiple columns or dimensions are designed for efficient access of data. MDAM uses the existing B Tree indexing for a broad range of queries for efficient usefulness by mainly improving the response time and reducing additional secondary indexes.

B. MonetDB

The availability of main memory for performing computer applications has become a bottleneck. As the speed of CPU's has outpaced in RAM latency causing a phenomenon called memory wall. The database technology was affected by the growing imbalance between CPU clock-speed and RAM latency. MonetDB is a database system led to redesign the database architecture. It focuses on redefining major database architecture components (data storage, query execution, query processing algorithms, query optimization) to make applications analyze large database volumes. MonetDB uses columnar storage rather than row-wise storage. The columnar database has fast query speed and great for highly analytical query models whereas row-wise storage is great for transaction processing and can write data very quickly but not applicable for the heavy-query environment[3]. "The storage model deployed in MonetDB is a significant deviation of traditional database systems. It uses the decomposed storage model (DSM), as we can see from Fig 4, it represents relational tables using vertical fragmentation, by storing each column in a separate (surrogate, value table), called binary association table (BAT)"[3].

From Fig 4 the design of MonetDB shows BAT algebra virtual machine as a back-end process, with many popular data models and query languages as front end modules. BAT storage forms two simple memory arrays, the surrogate or object-identifier (OID) on left column are called head and right column as a tail. BAT is a virtual id and has a low-level relational algebra called the BAT algebra. The BAT is a collection of an SQL query. Our main focus is on virtual BAT

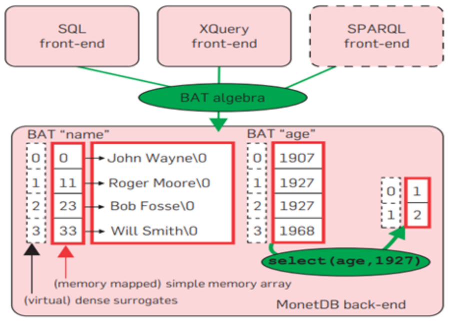


Fig. 4. MonetDB: A BAT Algebra Machine

IDs to be used as `Tuple IDs`(TIDs) which are stored at the end of the `MonoList` leaf nodes.

C. MPSM

For the main-memory database systems, the database software has to be designed in accordance with the futuristic hardware developments. In order to utilize the multi-core hardware effectively, it is required to investigate intra-operator parallelism. If large main memory databases are to be explored, this can be achieved by the query engines that are relayed on intra-operator parallelism. The smaller partitioning of the arguments of cache locality has been focused more on the radix join algorithm of MonetDB.

Modern hardware extends the main memory via non-uniform memory access(NUMA). Hence depending on straight forward partitioning techniques to maintain cache locality and to keep all cores busy is not possible. At the same time, RAM and cache hierarchies and multi-core parallelism have to be considered. The remote and local memory resources are accessed by the number of nodes that are divided logically by the NUMA system. NUMA is a friendly data processing system which mainly focuses on core working of the local data. By proper data placement which is the primary for high performance and scalability, a processor can access its local memory faster than its neighboring memory[1]. The three rules that are introduced for scalable multi-core parallelization for NUMA are as follows:

- ”Chunk the data, redistribute and then sort on the data locally.”
- ”Let the prefetcher hide the remote access latency.”
- ”Don’t use fine-grained latching or locking and avoid synchronization points of parallel threads”[1]

The massively parallel sort-merge join algorithms (MPSM) adhere to the three above mentioned rules where there will be no stability issues.It is designed to take NUMA architectures into consideration for parallel join processing for main memory systems. MPSM is distinct to specific NUMA architectures as it assumes the locality of the partitioning of RAM for a single core. MPSM follows the three rules as it first processes each data chunk which is sorted locally. Data accesses across NUMA partitions are sequential while subsequent join phase, this tends to hide the access overhead by the prefetcher. No upscale synchronization is enforced as there are no shared data structures occupied.[1].

D. K-d Tree

K-d tree is a binary tree in which each record contains k keys, right and left pointers to its sub-trees and an index integer between 1 and k that indicates which key in the record is used for splitting [5]. It is a space partitioning data structure that is useful to search multidimensional search key. The k-d tree is a generalization of the binary search tree used for sorting and searching[2]. The search cost depends upon the nature of the query.

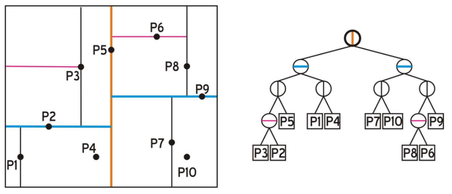


Fig. 5. K-d Tree Structure

It is a useful data structures for problems in diverse fields, especially those having to do with range queries, nearest neighbor for search or partial, exact, or approximate match queries[5]. For example, a tree is storing information about income and age of the employees of a particular organization and it is looking for employees between the age 30 to 45 years having the income between 60,000 to 80,000. In such a scenario, at each level K-d trees divide the range in half which is very useful for doing range searches.The search cost depends upon the nature of the query.

III. Implementation

In the this section, we explain the implementation of the Order-By in Column Sequential, Elf and LOD Elf. In all the three index structures, Order-By approach was used to observe the performance changes. Firstly, in Column Sequential we used the mapper function which initially selects all the values, orders and then these values are mapped as key value pairs to their respective TIDs. In Elf and in LOD Elf we have used the same functionality to select and map the values from each dimension to order and then to assign respective TIDs at the end.

A. Order-By in Column Sequential

Column Sequential is a relational database system where the entry look-up is done in a columnar manner. Column Sequential has faster selection and follows traditional Order-By approach. We have implemented the Order-By in Column-Sequential thereby comparing the performances of index structures with respect to Benchmark on various number of dimensions.

1.Column Sequential Scan

Algorithm 1: Order-By in Col-Seq

```
1  ColSeq(lower, upper,columnsForOrder,
     columnsForSelection);
2  BitVector.setAll(1);
3  MultiMap order = ArrayList.create();
4  for (column = start to stop) do
5      if (columnsForSelect[col])) then
6          for (Tid = start to stop) do
7              value = column[tid];
8              if (notisIn(value, lower, upper)) then
9                  BitVector.unset(tid);
10             end
11         end
12     end
13 end
14 for (column = start to stop) do
15     if (columnsForOrder [col]) then
16         for (Tid = start and stop) do
17             values = column[tid];
18             order.put(key, tid);
19         end
20     end
21 end
22 return Result
```

The Selection function `columnsForSelect` (Line 5-12) selects the values in a columnar fashion by checking the lower and upper boundary. If the selected column are in the range boundaries,it selects the values column-by-column and at the end it returns the TID to the end of the column. Finally, it terminates by returning the result. We use sorted map approach by using MultiMap-Function (Line 1) for ordering which returns the sorted values to keys (Line 14-22). This returns the sorted values as result along with their TIDs' and then terminates.

B. *Order-By in Elf*

In the previous sections, we have elaborated the properties of Elf. Prefix redundancy elimination is one such property which helps in memory optimization. By reducing the memory overhead which is advantageous, considering Big-Data with respect to varying dimensions and large sizes. Thereby helping to order the first dimension in the Elf. However, in the following dimensions it consists of the `Dimension Elements` of that respective `Dimension Lists` which are sorted and not the entire elements of the concurrent dimension.

From Fig 6, we can see that the `Dimension List` L1 is a ordered dimension which uses the Prefix Redundancy. However, the `List` L2 is not ordered entirely, so as to order further dimension elements, we used the Mapper Function (Line 2) called MultiMap. The advantage of using MultiMap is that it allows the redundant values i.e. by assigning multiple values to the same key. As compared to Standard Map Libraries this

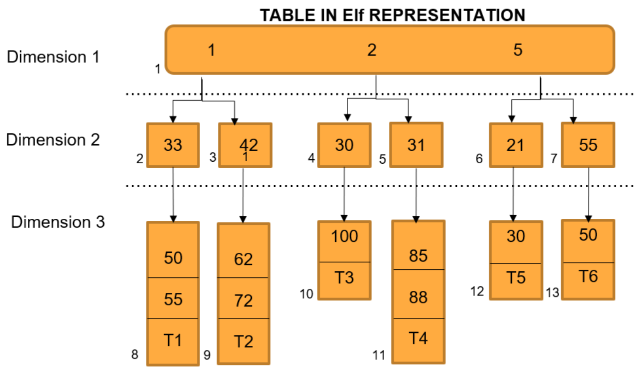


Fig. 6. Table represented in Elf Structure

is not possible. Our goal is to add all the ordered values in the form of an array list.

Algorithm 2: Order-By in Elf

```
1  OrderByElf(lower, upper, columnsForOrder,
     columnsForSelection)
2  MultiMap order = ArrayList.create();
3  if (lower[0] is less than and equal to Upper[0]) then
4      start lower[0];
5      stop upper[0];
6      else
7          start 0;
8          stop Real size-1;
9      end
10 end
11 for (offset start to stop) do
12     pointer = Elf[offset];
13     if (columnsForOrder[0]) then
14         key = offset;
15     end
16     if noMonoList(pointer) then
17         call the collectDimElement function;
18         else
19             call the scan MonoList function;
20         end
21     end
22 end
23 return Result;
```

MultiMap is a collection of Key and Value pairs that gives the results in the ordered form. In our implementation, we consider the `Dimension Elements` as Keys and the TIDs which is at the end of the `MonoList` as Value. From fig 6, we can see that the Pointer from the `Dimension List` L1 points to the next dimension to the concurrent `Dimension List` L2, thereby checking whether the Pointer points to a `Dimension List` or a `MonoList` (Line 16-21). When the Pointer points to a `Dimension List`, we select the `Dimension Elements` and make it available for ordering. As we observe from fig 6 that the Pointer from the `Dimension List` L2 is connected to a `MonoList` in `Dimension List` L3.

C. Locally Ordered at every Dimension(LOD) Elf

Algorithm 3: Collect DimElement

```
1  collectDimElement(DIMENSION,START_LIST,
     RESULTS,lower, upper, colForOrder, key, colForSel)
2  if (columnsForOrder[DIMENSION]) then
3  |  MultiMap orderDim =ArrayList.create();
4  end
5  if (columnsForSelection[DIMENSION]) then
6  |  while (notFoundEndOfList(Elf[position])) do
7  |  |  if (isIn(lower[DIMENSION],
         upper[DIMENSION], Elf[position])) then
8  |  |  |  pointer = ELF2[position + 1];
9  |  |  |  if (columnsForOrder[DIMENSION]) then
10 |  |  |  |  key = toCompare;
11 |  |  |  end
12 |  |  |  if (MonoListNotFound(pointer)) then
13 |  |  |  |  //CollectDimElemnt Function is called
   |  |  |  |    for Dim+1
14 |  |  |  end
15 |  |  |  else
16 |  |  |  |  //scanMonoList Function is called for
   |  |  |  |    Dim+1
17 |  |  |  end
18 |  |  end
19 |  |  else if (upper[DIMENSION] greater than
         toCompare) then
20 |  |  |  if (columnsForOrder[DIMENSION]) then
21 |  |  |  |  order.putAll(order-dim);
22 |  |  |  end
23 |  |  |  return;
24 |  |  end
25 |  |  position += DIM-ELEMENT-SIZE;
26 |  end
27 end
```

We search the values in the similar way for Dimension List as we reach the last value, it is known that the last value of the MonoList; these TID values get paired with the Dimension Elements. Likewise, in the Prefix tree based Elf structure, all the values get traversed from the left sub trees to the right sub trees.The paired values will be entered into the MultiMap. By doing this, we can obtain all the ordered Dimension Elements. The above discussed method defines the MultiMap globally to all the dimensions.The algorithm of the Elf is similar to the algorithm of LOD Elf, the only addition is the highlighted Lines which are shown in the Algorithms 3 and 4.

We have observed that the possibility of optimization and collection of results of the dimensions individually followed by merging them at the end. The methods are explained as follows:

- In this method, the functionality of searching and ordering of the Dimension Elements remains the same in the LOD Elf. Since, we want to make Elf more robust, when the data is huge in amount (Big-Data) it is unnecessary to search all dimensions when we want only the result of a single dimension.
- We use a Mapper Function for collecting the Keys and Values.To store them in an arraylist (Line 3) by checking whether the elements are in our required boundary range (Line 7). The function calls itself multiple times, it scans all the values from all dimensions (Line 12) and with a check condition to recovery mask. By this we mean, if all the values are selected completely then it calls scanMonoList function (Line 15) where at last the following values of the column as Keys and TIDs as Values are attached and are finally returned, thereby terminating the process. We find that if the current value is greater then than the upperboundary value, then we abort the scan (Line 18) and by adding the elements into the ArrayList.

Algorithm 4: Scan MonoList

```
1  ScanMonoList (DIM+ 1, pointer, RESULTS, lower,
     upper,colForOrder, key, order, colForSel) if
     (columnsForSelection[dim])) then
2  |  value = ELF[START-MONO-LIST + i];
3  |  if (notisIn(lower[dim], upper[dim], value)) then
4  |  |  return;
5  |  end
6  end
7  for ListElements Start to Stop do
8  |  if (columnsForOrder[dim]) then
9  |  |  value = [START-MONO-LIST + i];
10 |  |  key = value;
11 |  |  foundOrder = true;
12 |  end
13 end
14 if (foundOrder = false) then
15 |  order-dim.put(key, Last MonoList Element);
16 end
17 else
18 |  order.put(key, Last MonoList Element);
19 end
```

scanMonoList function is called when there is no DimnensionList. We scan all the elements of the MonoList until we find the last element which is our TIDs (Line 7-10). Then we merge the Dimension Lists results with the MonoList, thereby checking whether there are any Dimension Lists results available to merge (Line 14). After merging all the key and value pairs, that are ordered and entered into the Arraylist.

IV. Evaluation

From the experiments of the above section, an empirical evaluation discussion is delineated in this section. The first subsection defines the Evaluation Methodology followed by critical analysis of the factors affecting the performance of all the three index structures and in the end the results are discussed in detail.

A. The Evaluation methodology

We have evaluated and compared performances of Column Sequential, Elf and LOD Elf based on diverse factors. The index structures are evaluated on a computing system with Intel Core-i5 8300H @ 2.30GHz 8 MB of cache size and 12 GB of RAM. We conduct several experiments to gain insights into the benefits and drawbacks of Column-Sequential, Elf and LOD Elf. We evaluate with a benchmark that systematically evaluates the influence of parameters such as DimMax, columnForOrder and size on the response time. In this evaluation, we are interested in the break-even points regarding selectivity and ordering that indicate when a Column-Sequential scan becomes faster than our approach. Furthermore, we are interested in the results between size, columns for order versus run-time. We have considered our Benchmark with different sizes (i.e 1000, 10000, 100000, 1 Million) dimensions limited to 10 and with with different dimension sizes (i.e. 32, 64, 128, 256, 512). We have repeated each experiment 20times. In the later paper, we have addressed dimension size as DIM MAX.

B. Performance Factors

Performance Factors shows the behaviour of our data set. The factors affecting performance of the Index structures with respect to Benchmark are discussed below:

1) DimMax - It refers to the maximum number of `Dimension Elements` that can occupy a single Dimension. With the increasing time the DimMax also increases.
2) Size - It refers to the total number of `Dimension Elements` that are present in all the dimensions. Size of Elements doesn't have an impact in the Elf whereas in Column Sequential it has relatively more impact.
3) Column for Order - It is the query, which is used to set which column to order at a particular run. In this case, it is very effective for Elf since Dimension 1 is always sorted while in Column-Sequential scan it doesn't change for Dimension 1 and Dimension 2.
4) Dimensionality - As the number of dimensions increases the number of queries also increases resulting increase in time.

C. Evaluation results

To get the final result by implementing the Order-By query in Elf, LOD Elf and Column Sequential we used the outputs to plot our graphs. As mentioned before, we conducted several experiments to do evaluation between Column-Sequential, Elf and LOD Elf. Taking those results into account we plotted several graphs and selected only those that would help us demonstrate our findings.

1) Experiment 1 - Run-Time vs Columns: Our first experiment is to depict by comparing the results between the run-times of the Elf versions and the Column Sequencial when the query is generated to Order each dimension. As shown in fig. 7, we have set DIM-Max=32, and the size=1000 which are distributed among Dim=10. We interpret from fig.7 that the run-times are better in ColSeq when compared to Elf versions. We have observed that when the size of the data is small,

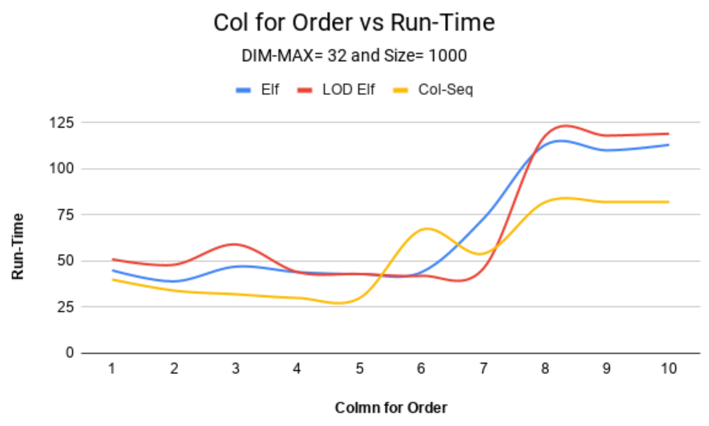


Fig. 7. DIM-MAX= 32 Col for Order vs Run-time

ColSeq is performing more efficiently than Elf Versions. We have also noted the same pattern even when the DimMax is varying.

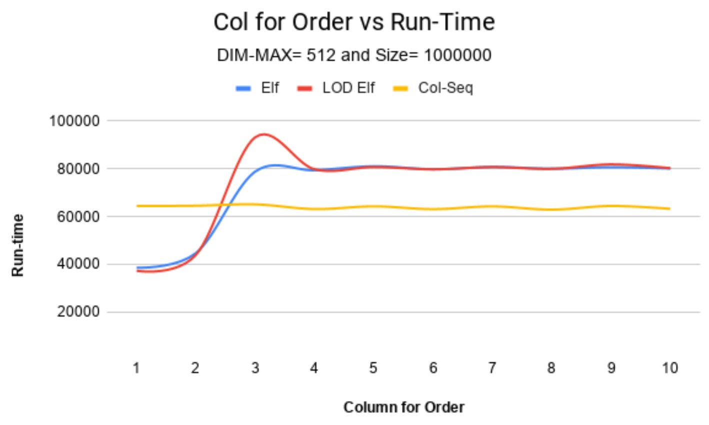


Fig. 8. DIM-MAX= 512 Col for Order vs Run-time

As we refer to fig. 8, considering the run-time as performance metric, we again compared the Elf versions with the Column-Sequential. While Ordering for the first 3 dimensions we can clearly see that, when there is an increase in size to 1 Million, the Elf versions are performing better than Column-Sequential.

In the fig.9 shown above, we compare the results of LOD Elf and Column-Sequential of size = 1 Million taking into account all the DimMax ranges (i.e. 32, 64, 128, 256, 512). We observe that, the dimensions have no impact on Column-Sequential. We also noted that the run-times have a very small deflection of spline across all the dimensions, whereas as in LOD Elf the run-times vary from dimension to dimension. Moreover, for the Lower DimMax ranges we can see that the performance of LOD Elf is better compared to the higher DimMax ranges. This could be justified as the Column selectivity increases, the cardinality for each dimension varies and which results in

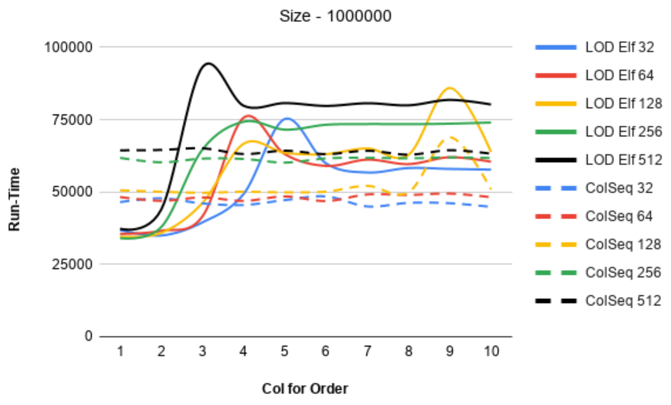


Fig. 9. LOD Elf vs ColSeq

increase in effort of selection and then ordering.

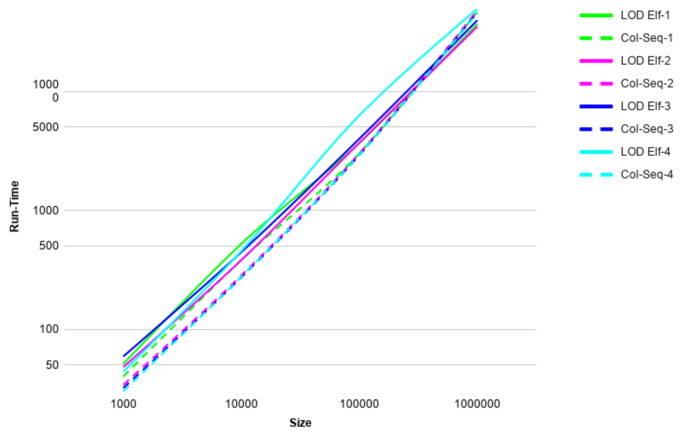


Fig. 10. DIM-MAX 32 from Col for Order 1-4

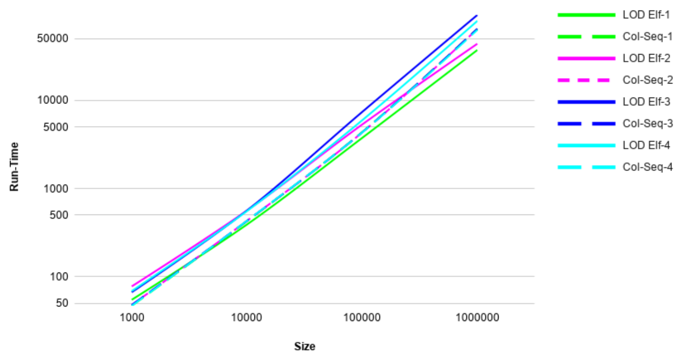


Fig. 11. DIM-MAX 512 from Col for Order 1-4

2) Experiment 2 - Run-Time vs Size: In this experiment we visualized the performance of the first four dimensions of LOD Elf and Column Sequential for different sizes. From both of the fig.10 and fig.11, we have interpreted that for the first three dimensions the performance is better but from the fourth dimension there is a decrease in the performance. We have also noticed in the fig.10 that, LOD Elf at dimension 4 has almost similar results as Column Sequencial at Size 1 Million, by that we can say, there is an increase in the performance with the increase in size. Although, the sorting (ordering) complexity O(log n) remains same through every dimension but as the size is increasing this effects the complexity, leading to variation in depth for selection in every dimension. Hence, the graphs imitate these changes.

V. Conclusion

Our work highlights the advantages of the Elf versions over Column Sequential considering properties such as Prefix Redundancy Elimination and Mono-List Optimization. The graphs clearly exhibit that both the Elf versions are performing better with larger datasets,with varying DIM-MAX and diverse Columns for Order. The LOD Elf and Elf with Order-By proved to be competitive at the initial dimensions and showed a possible scope of performing better with the increase in size. From this we can infer that it might also perform better with Big-Data; which is one of our concerns. The results discussed in the previous section states that, the extension of the Elf version known as the LOD Elf performs competitively better than Elf and at Column Sequential in initial dimensions.

References

[1] Martina Cezara Albutiu, Alfons Kemper, and Thomas Neumann. "Massively parallel sort-merge joins in main memory multicore database systems". In: *Proceedings of the VLDB Endowment* 5.10 (2012), pp. 1064–1075. ISSN: 21508097. DOI: 10 . 14778 / 2336664 . 2336678. arXiv: 1207.0145.

[2] Jon Louis Bentley and Jerome Friedman. "a Survey of Algorithms and Data Structures for Range Searching". In: *ACM Trans.Database Syst.* August (1978).

[3] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. "Breaking the memory wall in MonetDB". In: *Communications of the ACM* 51.12 (2008), pp. 77–85. ISSN: 00010782. DOI: 10.1145/1409360.1409380.

[4] David Broneske et al. "Accelerating multi-column selection predicates in main-memory -The Elf approach". In: *Proceedings - International Conference on Data Engineering* 0 (2017), pp. 647–658. ISSN: 10844627. DOI: 10.1109/ICDE.2017.118.

[5] Philippe Chanzy, Luc Devroye, and Carlos Zamora-Cura. "Analysis of range search for random k-d trees". In: *Acta Informatica* 37.4-5 (2001), pp. 355–383. ISSN: 00015903. DOI: 10.1007/s002360000044.

[6] Linnea Passing et al. "SQL- and operator-centric data analytics in relational main-Memory databases". In: *Advances in Database Technology - EDBT* 2017-March (2017), pp. 84–95. ISSN: 23672005. DOI: 10.5441/002/edbt.2017.09.