Parallel Selections in Elf

Karthik Kadajji[1], Madhu[2], Neel Mishra[3], Rabi Kumar K C[4], Rathan kumar Chikkam[5], and Ravi Gunti[6]

[1,2,3,4,6]Data and Knowledge Engineering, Otto von Guericke University Magdeburg
[5]Digital Engineering, Otto von Guericke University Magdeburg

***Abstract*— Database design nowadays is transformed to avoid disk-based access because of an increase in availability of low-cost main memory. There are various index-based structures for handling multidimensional data which take advantage of this increased main memory. However, these well-known index-based main memory structures process data sequentially. In this paper, we tried to parallelize the search operation in one such index-based main memory structure, Elf. We implemented parallelization at two levels, node level and sub-tree level. We performed an experimental evaluation of effectiveness and efficiency using TPC-H benchmark and concluded that out of the parallelism at two level, sub-tree level performed better than node level. Our parallelization approach could not compete with the sequential access approach because of the locking overhead involved.**

I. INTRODUCTION

The availability of low cost main memory has led to a significant transformation in the design of database systems. This partially solved disk-based access latency and shifted the bottleneck between RAM and processor. Even with much faster RAM and the shift from the disk-based storage to RAM storage, index-based data structures which were designed earlier to work with disk-based systems are now one of the main factors causing memory bottleneck. Elf is an index-structure introduced to efficiently handle multidimensional data and overcome the memory bottleneck. These index structures are traversed sequentially. There is scope of further performance gain by parallelizing the traversal. Current implementation of Elf uses depth-first search to navigate through the tree, starting from the first dimension and traverses the tree down till the last dimension. In this paper, we investigate how to parallelize the search operations on Elf. We introduce two types of parallelization: Sub-tree level and Node level parallelization.

- Sub-Tree level parallelization: First dimension of the Elf only has pointers to the next dimension. In sub-tree level parallelization, each thread considers the pointer as root node and starts the depth first sequential traversal.
- Node level parallelization: In Node level parallelization, parallel processing starts from the second dimension to the last dimension. The first node in a dimension is processed by a single thread and the recursive call to other entries in same branch and same dimension are pushed to the queue.

The remainder of the paper is organized as follows. Section II deals with analyzing data structure and parallelization achieved in structures similar to Elf. In Section III, we explain the details of the current implementation and optimization of Elf. In Section IV, we provide parallelization approaches for Elf namely sub-tree level and node level. In Section V, we discuss the evaluation setup and followed by results in Section VI. Conclusion and future work are given in Section VII.

II. PARALLELIZATION STRATEGIES IN MAIN MEMORY INDEX STRUCTURES

We start with an overview of BB-Tree, which is designed for parallel query execution, and then discuss the present parallelization techniques for the KD-Tree.

A. BB-Tree

BB-tree is a main memory, space efficient and a fast index structure for handling multidimensional data that can be used for both reading and write operations. BB-tree has two main components: K-ary search tree and bubble bucket. BB-tree relies heavily on a novel structure known as a bubble bucket which is used to store the data.

1) BB-Tree Structure: The novel concept of bubble bucket is able to handle large amount of insertions and deletions. The bubble bucket in BB-tree ensures that there is minimum reorganization of tree structure. Apart from this, the structure of bucket helps in accommodating multiple insertion or deletion requests at the same time because the data is spread over multiple buckets and multiple partition in that bucket giving easier accessibility. Each of these buckets can only accommodate a maximum amount of data which is referred to as bmax.

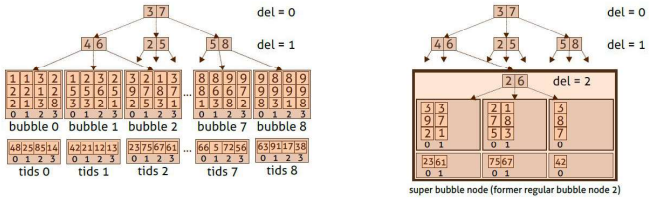

Fig. 1. BB-Tree Structure [5]

Value of bmax plays a vital role in determining the structure of the tree. Lower bmax value will result in

deeper tree useful for selective predicate and not so useful for query scans. However, a higher bmax value will result in a tree with a sparse structure capable of handling simple workloads. In the figure above the bmax value is 4. According to the unique values present in the first dimension, it is split into 3 partitions. Values less than 3 are stored in the left part. Values more than 3 but less than 7 is stored in the middle and values more than 7 are stored in the right part. In the bubble bucket, tuple data is stored horizontally. Also, tuple ids can be stored corresponding to id and bucket it is present.

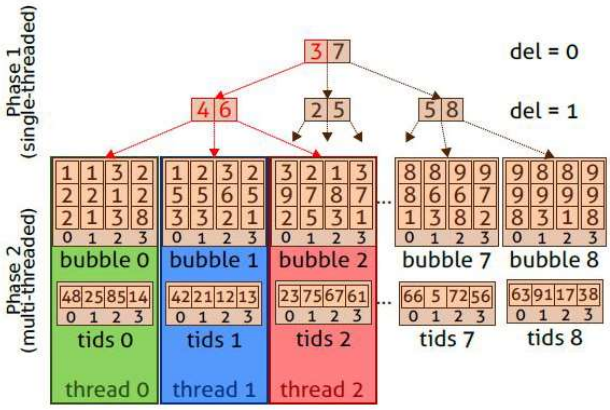

Fig. 2. Parallelization in BB-Tree [5]

2) BB-Tree Parallelization: There are two main phases of the parallelization: phase 1 is single threaded and phase 2 is multi-threaded. In the first phase, the single thread is used to navigate nodes using depth-first search as little is to gain by applying multithreading at this level because the tree structure is mostly flat and introducing the multithreading at this level will introduce more process overhead than doing some good. So in the first phase, the candidate BB is determined and in second phase parallel processing is started. Each of the buckets would be navigated with a single thread parallelly but usually, this is not the case and partition based on the number of candidate BB(bmatch) and the number of threads (t) is to be computed. If bmatch = t then we have perfect parallelism where each thread processes each of the candidate BB. If bmatch >= t each thread processes a single candidate BB and once any thread finishes its part they go on to execute the remaining candidate BB. If bmatch<t we can go with two types of implementation: going with one thread per bucket and keeping the rest of the threads idle or partitioning the candidate BB and assigning multiple threads to a bucket.

B. KD-Tree

KD-Tree is a binary tree index structure. It is used to store multi-dimensional data where every leaf node is a point in k-dimensional space. Every non-leaf node divides the space into two parts by generating a hyperplane where points on left of the hyperplane forms left sub-tree and points on right of hyperplane forms right sub-tree.

1) KD-Tree Structure: At each level of KD-tree the data is split into two parts along a hyperplane. The KD-tree cycles through the dimensions as the tree descends and divides the data into two parts at each level. The dimension at level l which divides the data into two parts is given by dl = d mod dmax , where dmax is maximum dimensionality of data.

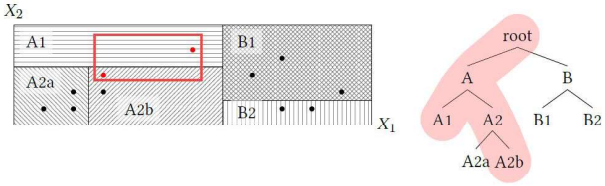

Fig. 3. KD-Tree Structure [4]

In above figure, the data at root node is divided into two parts by a hyperplane along dimension x1, then at next level the data is again divided into two parts by a hyperplane along dimension x2.

2) KD-Tree Parallelization: To process search queries in parallel, multiple threads are used. A queue containing all the nodes which are yet to be visited is introduced. If one or both child nodes of the current node being examined are part of the query results then they are inserted at the end of the queue. To avoid the overhead because of interthread communication each thread stores its own intermediate results which are finally merged to form global query results. To avoid the queue as a bottleneck for multiple threads, it is divided into multiple queues. All the unvisited nodes are divided into multiple queues. To access a queue to process its nodes, a thread first polls a random queue to check whether it is locked or it is not empty. If the queue being polled is empty or is locked by another thread then the thread which polled checks all queues sequentially and whenever it finds a queue to be processed it starts processing it. This also ensures that all the queues are empty before the final results are merged.

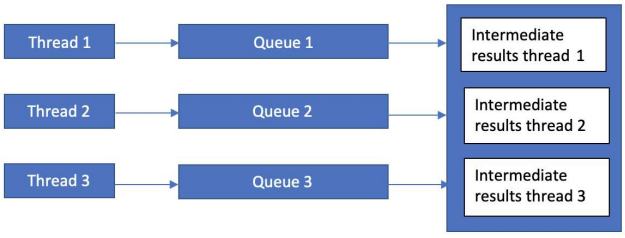

Fig. 4. Parallelization in KD-Tree [2]

In the above figure all the unvisited nodes are there in three queues (queue 1, queue 2 and queue 3), there

are three threads which are processing these queues in parallel. All the threads maintain their intermediate results which are finally merged to get the final query results.

III. ELF - INDEX STRUCTURE

In this section we analyze Elf, discuss the current working and provide an example to understand the process of creating an Elf structure from a given table of data.

In main-memory databases, the query while processing has to sequentially scan over several columns. Such collections of predicates on several columns are termed as "Multi Column Selection Predicates" [1]. Since the database has been stored in the main memory, in order to process the query there would be the need of an index structure which would efficiently find the required tuple from the hot data. Elf is one such index structure that would process multi column selection predicates and access the memory efficiently. For a better understanding, we present an example of Elf's efficient memory utilization in the following example table (Fig 5). It can

Year (D1)	Supplier_Region (D2)	Cust_Region(D3)	TID
2014	Europe	Asia	T1
2014	Asia	Europe	T2
2015	America	Europe	T3
2015	Europe	America	T4
2015	Australia	America	T5
2016	Australia	Europe	T6

Fig. 5. An example table

be observed that the transactions between the suppliers and customers based on their regions for the mentioned years(Fig 5). In this, we have three dimensions namely Year, Supplier_Region, and Cust_Region. Now this data has to be converted into Elf structure using prefix redundancy elimination [1]. Each column has attribute values that repeat multiple times in the table and share the same prefix value and Elf eliminates the redundancy and enters an instance only once in a Dimension list. The

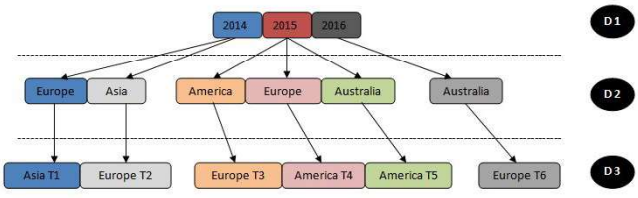

Fig. 6. Prefix redundancy elimination in Elf

index structure maps the distinct values of a column to one single Dimension list in the first column, we have three distinct values 2014, 2015, and 2016 (Fig 6). Therefore, the first dimension consists of three entries in the dimension list and their pointers to next dimensions respectively.Thereby achieving prefix redundancy elimination. In the next column, we cannot see the prefix redundancy elimination as all attribute columns in this dimension are unique.The third dimension consists of tuple identifier.This structure has fixed depth[1], as the columns in the table are fixed number. Hence the dimensions are also fixed.

A. Optimization:

So far this structure has performed well in reducing prefix redundancy elimination and efficient memory access but this had two drawbacks.

- The first dimension in the example contains all the possible unique values. This usually consists of large list of entries out of which we need to find the desired path.
- As we traverse from top to bottom, the dimension lists in the deeper levels of the tree contains only one value. This means, the traversal path becomes unique. Unnecessary memory for saving pointers is used and also time taken for processing the traversal increases.

These drawbacks have been addressed in the optimized version of Elf using the techniques mentioned below.

1) Hash Map: Since the first dimension is already ordered, this allows us to directly create a hash map. Thus, we remove the overhead for processing a potentially high number of entries. In general pointer to next dimension is stored next to the value. But, when it is hashed only the pointers to the next dimension are stored.

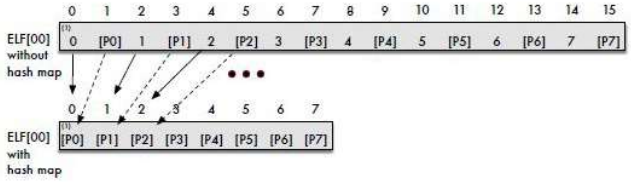

Fig. 7. Hash Map property within first dimension [1]

2) Monolists: In a sparse index structure, as we traverse towards the leaf node, the lists probably contain single object which leads to only one TID. This is due to the absence of prefix redundancy. Due to this, the pointer and the value are being stored which lead 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. Here, for the year 2016 we have only one index pointing towards Australia and which further points towards Europe(Fig 8). But as discussed, using monolists we store the path in one list by switching from columnar layout to row wise layout. Hence, the overall lists and dimensions that have to be processed will be reduced by means of hashing and monolists.

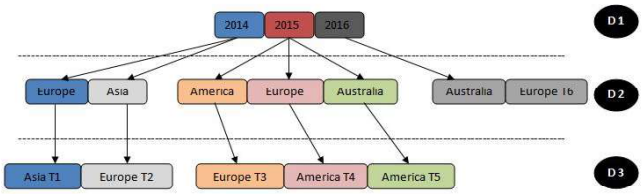

Fig. 8. Monolists

B. Search Algorithm:

As directed above in this section, we provide an overview of the search functionality of Elf.

```
   Result: L Resultlist
1  SearchElf (lower, upper) {
2      L ← ∅;
3      if (lower[0] ≤ upper[0])then
           // predicate on first column defined
           // exploit hash-map
4          start ← lower[0]; stop ← upper[0];
5      else
6          start ← 0; stop ← max {C1};
7      end if
8      for (offset ← start to stop)do
9          pointer ← Elf[offset];
10         if (noMonoList (pointer))then
11             SearchDimList (lower, upper, pointer, dim ← 1, L);
12         else
13             L ← L+SearchML
               (lower, upper, unsetMSB(offset), dim ← 1, L);
14         end if
15     end for
16     return L;
17 }
```

Fig. 9. SearchElf Algorithm [1]

In SearchElf algorithm, we show how to deal with the data within the first dimension list which is hashed. The algorithm takes the upper and lower boundaries of the first dimension as parameters. If the query is a partial match query, that is, if there is no definition for the single node boundary, then all the elements in that node have to be retrieved. The result of this algorithm is a list (L) with all the pointers to next dimension which fulfill the search criteria. We exploit the hash map property for a scenario that contains a predicate on the first dimension (line 3). If not, we continue the search for each value in the first list to next level. The subsequent level can be either a monolist that refers to only one point which has its value next to it in the dimension list. But prior to this, we have to check if the next dimension list is a monolist. All the elements are checked for the match and when a monolist is reached, the results are appended to the list (L).

We now see how to traverse further when the next dimension level is not a monolist(SearchDimList algorithm). Apart from the lower and upper boundary ranges of the queries, the Elf algorithm needs the start offset of current dimension, respective dimension level

```
1  SearchDimList (lower, upper, startlist, dim, L) {
2      if (lower[dim] ≤ upper[dim])then
3          position ← startList;
4          do
5              if (isIn (lower[dim], upper[dim], Elf[position]))then
6                  pointer ← Elf[position + 1];
                   // start of next list in dim+1
7                  if (noMonoList (pointer))then
8                      SearchDimList
                       (lower, upper, pointer, dim + 1, L);
9                  else
10                     L ← L+SearchML (lower, upper,
                       unsetMSB(pointer), dim + 1, L);
11                 end if
12             else
13                 if (Elf[position] > upper[dim])then
14                     return;// abort
15             end if
16             position ← position + 2;
17         while (notEndOfList (Elf[position]));
18     else
           // call SearchDimList or SearchML with dim + 1
           for all elements
19     end if
20 }
```

Fig. 10. SearchDimList Algorithm [1]

and the result list (L) as input. The start offset points the first element of the list (line 3). When the dimension is part of the predicate, we compare every element in this dimension with the search predicate value for the match. When there is a match, we traverse further to the next subsequent level or a monolist. The stopping criteria is the next higher value being larger than the search attribute value (line 13) or when the list ends (line 17). As elements in the dimension list are ordered already, comparing the attribute value with the predicate value and stopping the search is easy to implement. The Elf search algorithm is a depth-wise search [1], that means when the match is found we directly go to the next child level to identify the results (line 5) rather than continuing the search in the same dimension list. Because the data being multidimensional, it results in few attribute value match which results in visiting the lower dimensions. The ideology behind this approach is to reach to a level quickly that has low selectivity, so that we do not need to jump to next level.

IV. PARALLELIZATION: SUB-TREE AND NODE LEVEL

In this section we are going to discuss the parallelization of Elf implemented in two very distinct manner.
Most of the modern processors allow parallel execution of various processes and provides developers interfaces to exploit this concurrency easily and efficiently. A naive way of achieving the parallelism in Elf is to spawn 'n' number of threads supported by the processor and manage them. However, this increases process overhead as we need to manage the creation, deletion and synchronization of many threads involved in the process. Instead of just spawning threads and processing the structure

parallely we go with implementation of threadpool to effectively utilize the concurrency feature supported in modern systems. The concept of a thread pool allows various threads to work concurrently without having a need to bother about thread creation and destruction every time everytime a new node is traversed or control moves to the next dimension. Furthermore, in our case considerable overhead is avoided by using threadpool because threads are created only once. A task queue stores the tasks that needs to be executed. A Queue is the data structure that is suitable for storing parallel tasks because it follows the principle of "First in First out", hence tasks will be executed in the order in which they are submitted. A drawback of the task queue is that concurrent accesses have to be synchronized and this issue is resolved by mutex locks which ensures that there is no concurrent access of the tasks by threads. The usage of mutex locks also makes the queue thread safe.

A. Sub-Tree Level Parallelization

A simple way to imagine sub-tree parallelization would be to consider multiple depth-first searches running simultaneously by taking the entries present in the first dimension as the root. In Depth-first search branch is explored until a monolist is encountered. Achieving parallelism at sub-tree level we check the maximum number of threads supported by the processor and spawn the same number of threads in the thread pool. At any given point of time threads from threadpool can only process a task if a thread is present in the thread pool. Suppose all the threads from the thread pool are busy processing a particular sub-tree, the task present in the queue would have to wait till any of the thread finishes processing and comes back to the thread pool. At the start, the first entry in the first dimension is processed and the rest of the entries present in the first dimension along with its children if any is pushed into the task queue. First thread proceeds to the second dimension and starts to process the remaining entries and nodes using depth-first search. Simlarly, the entries from the first dimension added as tasks in the task queue are also processed by threads from thread pool using depth-first search when a thread is available in threadpool.

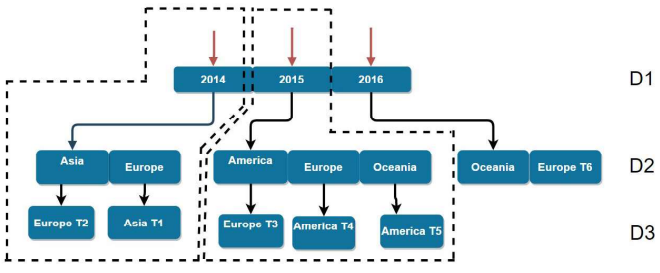

Fig. 11. Structure for Sub-tree parallelization

a) Example: Consider the query below

```
select * from table where cust_region='Europe'
AND year BETWEEN 2014 AND 2015
```

As seen in the Figure 11 we process the query using sub-tree level implementation of Elf. As the main thread starts processing, it encounters the first entry from dimension D1. It checks if the year "2014" matches the query condition. Since it is a match, the thread pushes the call to function partialmatch1 (with pointer to next dimension as one of the arguments) to the task queue and keeps processing the elements next to "2014". So the thread checks the second entry in first dimension "2015". As the condition is met the call to function partialmatch1 of the second entry is pushed to the task queue. Then move to the next entry which in our case does not satisfy the condition as the year "2016" is not "2014" or "2015". Therefore this element will not be added to the task queue. Once D1 is traversed, threadpool is notified by popping the items from task queue. If there is any free thread in the threadpool it starts to fetch a new task. The first thread which moved to second dimension will now process the entry "Asia" in second dimension using depth-first search. Since second dimension represents supplier_region which is not part of predicate any node value is acceptable. Then the thread will move to third dimension which represents cust_region and to satisfy the condition it has to be "Europe" and here the first entry value is "Europe (T2)" which does satisfy the condition. As it encounters a match, its corresponding tuple identifier is stored in the result vector. After encountering a monolist the control will be backtracked to visit the unvisited entries from the previous dimension. So thread would go to the second dimension's second entry "Europe" and then come to the next dimension. Value of entry in the next dimension (D3) is "Asia (T1)" which does not satisfies the query condition in our case. Simultaneously, another thread from the threadpool will process the YEAR= "2015" using the depth first search.

b) Pseudocode: Each dimension represents a column. In first dimension, we check if the dimension is part of the predicate. If the dimension has some condition we check for the particular value of the condition. These particular values are represented by lower bound and upper bound. After this, the pointer points to the lower bound entry and we restrict our processing till the upper bound value of that dimension. However, if the dimension does not have any condition associated with it, we have to select all the entries present in the dimension starting from the first entry. Next we loop through each entry present in the first dimension and push the pointer to the next dimension of that entry as argument to the function. We check if each entry is monolist or not and in case of a regular DimensionList, we pass that entry as argument to *partialmatch1*. Entry is passed as argument to *partialmatchmonolist* if it is a monolist. Function *partialmatch1* performs recursive depth-first search on

the entries.

```
Algorithm 1: Subtree Level Parallelism
  Result: Subtree Level Parallelism
  partialMatch(DIMENSION, START_LIST, RESULTS, lowerBoundQuery, upperBoundQuery,
    columnsForSelect)
  position ← START_LIST
  if (columnsForSelect[FIRST_DIM]) then
      LOWER ← lowerBoundQuery[FIRST_DIM] ∗ 2;
      UPPER ← upperBoundQuery[FIRST_DIM] ∗ 2;
  else
      LOWER ← 0;
      UPPER ← MAX_FIRST_DIM ∗ 2;
  end
  for offset ≤ UPPER do
      pointerNextDim ← get64((ELF[offset]));
      if (pointerNextDim <LAST_ENTRY_MASK64) then
          Queue(partialMatch1(..,pointerNextDim,..))
      else
          pointerNextDim = RECOVER_MASK64;
          Queue(partialMatchMonoList(..,pointerNextDim,..))
      end
      offset ← offset + 2
  end
  return resultTIDs
```

Fig. 12. Pseudocode for Sub-tree level parallelization

B. Node Level Parallelization

One challenge sub-tree level parallelization has is the imbalance in thread utilization i.e. some of the threads in thread pool remain idle while other threads traverse entire sub-tree before returning to the thread pool. To overcome this, we propose Node level parallelism. Node level parallelism also uses the concept of multiple threads along with queues. While traversing the tree, at each dimension the nodes are processed such that the first element in each node is buffered by a single thread and the recursive function call to other elements in same node and same dimension are pushed to the task queue. Once all the elements of the node are pushed to the task queue, the thread uses the buffer to traverse to the left child of the node. The recursive function calls are picked up from the task queue by threads from the thread pool whenever there is an idle thread. As mentioned earlier, a single thread traverses the first node in each branch till it reaches a monolist or a leaf node.

After processing of nodes or tasks by threads their intermediate results are stored. When all the threads are done with the processing of tasks from the queue and there are no more tasks left in the queue to process, these intermediate results are finally merged to get final query results.

a) Example: Consider the query below

```
select * from table where cust_region='Europe'
AND year BETWEEN 2014 AND 2015
```

Now we try to execute the query using node level parallel implementation of Elf. Here in the first dimension the predicate is evaluated and one thread call it t1, starts processing the "2014" branch of the tree and another thread call it t2 starts processing the "2015" branch of the tree. We do not need to process the '2016' branch because it is not in the range of "2014" and "2015". The thread t1 processing "2014" branch descends to next dimension and processes the entries of first node having value "Asia" and "Europe" at dimension 2 and "Europe (T2)" (the monolist) at dimension 3. The thread t1 also pushes the child of "Europe" at dimension 2 i.e. "Asia (T1)" (the monolist) to the queue. Likewise, thread t2 processes entries of the second node "America", "Europe", "Oceania" at dimension 2 and "Europe (T3)" (the monolist) at level 3 and pushes the children of "Europe" and "Oceania" to the queue, i.e. the pointers of "America (T4)" and "America (T5)".

```
Algorithm 1: If dimension IS part of predicate
  Result: Node Level Parallelism
  partialMatch1(DIMENSION, START_LIST, RESULTS, lowerBoundQuery, upperBoundQuery,
    columnsForSelect)
  position ← START_LIST
  buffer ← 0
  if (columnsForSelect[DIMENSION]) then
      while ((toCompare = ELF[position]) <LAST_ENTRY_MASK) do
          if (toCompare is in bounds) then
              pointer ← get64((ELF[position + 1])
              if (buffer == 0) then
                  buffer ← pointer
                  position ← position + 3;
                  continue;
              end
              if (pointer <LAST_ENTRY_MASK64) then
                  Queue(partialMatch1(DIMENSION + 1,..))
              else
                  pointer = RECOVER_MASK64;
                  Queue(partialMatchMonoList(DIMENSION + 1,..))
              end
          else
              if (upperBoundQuery[DIMENSION] <toCompare) then
                  break;
              end
          end
          position ← position + 3;
      end
      toCompare = RECOVER_MASK64 and Process the last value
      if (buffer != 0) then
          if (pointer <LAST_ENTRY_MASK64) then
              partialMatch1(DIMENSION + 1,..)
          else
              pointer = RECOVER_MASK64;
              partialMatchMonoList(DIMENSION + 1,..)
          end
      end
  else
      Contd . .
  end
```

Fig. 13. Pseudocode for node level parallelization when dimension IS part of the predicate

```
Algorithm 2: If dimension IS NOT part of predicate
  Result: Node Level Parallelism
  partialMatch1(DIMENSION, START_LIST, RESULTS, lowerBoundQuery, upperBoundQuery,
    columnsForSelect)
  position ← START_LIST
  buffer ← 0
  if (columnsForSelect[DIMENSION]) then
      Contd . .
  else
      while (toCompare = ELF[position]) <LAST_ENTRY_MASK do
          pointer = get64((ELF[position + 1])
          if (buffer −− 0) then
              buffer ← pointer
              position ← position + 3;
              continue;
          end
          if pointer <LAST_ENTRY_MASK64 then
              Queue(partialMatch1(DIMENSION + 1,..))
          else
              pointer & = RECOVER_MASK64;
              Queue(partialMatchMonoList(DIMENSION + 1,..))
          end
          position ← position + 3;
      end
      toCompare = RECOVER_MASK64 and Process the last value
      if (buffer != 0) then
          if (pointer <LAST_ENTRY_MASK64) then
              partialMatch1(DIMENSION + 1,..)
          else
              pointer = RECOVER_MASK64;
              partialMatchMonoList(DIMENSION + 1,..)
          end
      end
  end
```

Fig. 14. Pseudocode for node level parallelization when dimension IS NOT part of the predicate

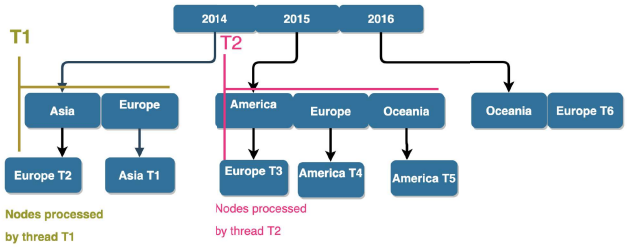

Fig. 15. Structure for Node level parallelization

b) Pseudocode: In node level parallelism, processing actually starts at second dimension. First a thread checks if the value under processing is within bounds or not. If it is within bounds then the thread processes it using *partialmatch1* if it is not a monolist, and if it is a monolist then it is processed using *partialmatchmonolist*. Then we check the pointers of next entries in same dimension. If the next entry pointer points to a monolist we push *partialmatchmonolist* to the queue else we push *partialmatch1* to the queue. Furthermore, function pushed in the task queue are processed by threads from threadpool if any idle thread is available.

V. EVALUATION

In this section, we introduce the evaluation setup for the parallel Elf. We are concerned in complex queries where the output would affect business and the organizational decision that has to be made. Therefore, we use the TPC-H benchmark data and queries as it contains the data which is not artificial. From the TPC-H benchmark generator, we generated tables of scale factor 1 for Lineitem and 10 for Part(i.e., the biggest table LineItem has 6 million tuples). As discussed in the above section, we now have sub-tree level parallelization and node level parallelization for evaluation and its response time is compared to the sequential Elf.All approaches have been implemented in C++ and are optimized to allow for a fair comparison. We performed the evaluation using single and multi-threaded execution on an intel (R) Core(TM) i7-6500U CPU @ 2.50 GHz clocked at 100MHz with Ubuntu 18.04 LTS and 16 GB RAM on an instance hosted on Google cloud platform. This setup consists of 4 cores to have 4 threads running in parallel. The performance measure for our evaluation are the variation of number of threads(1 to 4), dimensions(9 for part and 15 for lineitem), data points(2Million for Part and 6Million for lineitem). Initially, we process the queries using sequential Elf and compare the processing time with the variants of parallel Elf. We also compare the performance of sequential Elf with varying number of threads for sub-tree and Node level.

A. TPC-H Evaluation:

For evaluation, we use the selection predicates from the TPC-H benchmark. Consequently, the results of this experiment are better revised to conclude real-world workloads. In mono-column selection predicates are chosen to explore the normal applicability for a real-world scenario. For this we have chosen queries Q1, Q6 and Q14. For multi-column selection predicate we have query Q19 which is defined on two tables (lineItem and part). Query Q17 was defined on the Part table. Query Q10 is defined on the lineItem table.

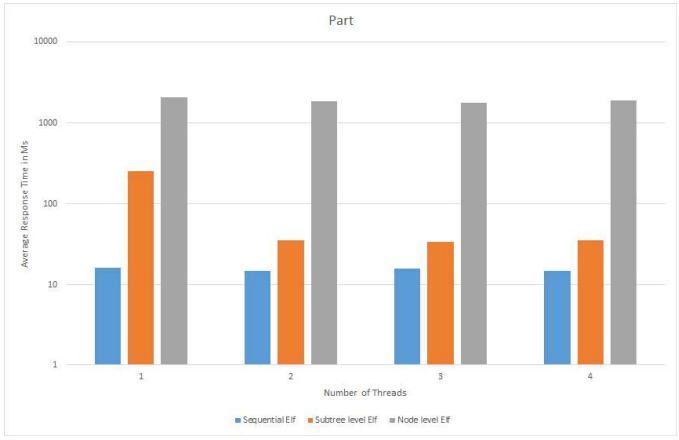

Fig. 16. Response time vs number of threads for Part

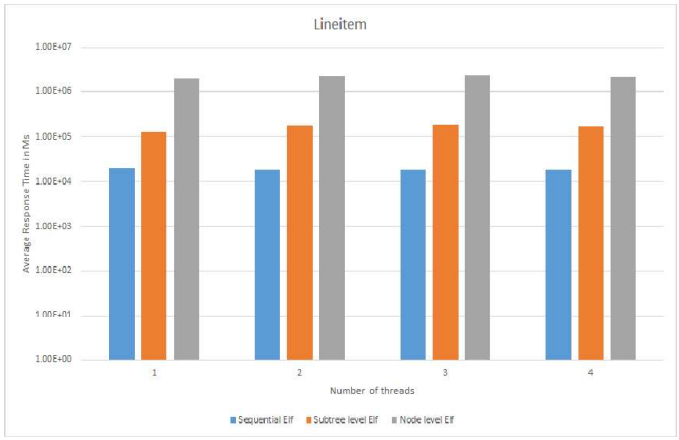

Fig. 17. Response time vs number of threads for Lineitem

In figure 16, we depict the average response time for the fore-mentioned queries with respect to the number of threads spawned for Part. Figure 17 depicts the average response time for Lineitem. We compared both sub-tree level and node level with sequential Elf. For a better visibility, we used logarithmic representation of our response time. We see that there is a significant difference between the parallel variants and sequential Elf. Moreover, we can see that traversing with multiple threads do not improve the average response time as there is an overhead due to the locks on the shared resources. As we have seen in the implementation section, we used a task queue to hold the function calls. But, to avoid the conflict of having concurrent accesses to the

task queue we used mutex locks. There is also a result vector, where the threads after processing append their results into. Here also both the variants have suffered the conflict of concurrent access. This resulted in significant difference between the sequential and parallel variants. Sub-tree level performed better when compared to node level implementation as it requires fewer locks on the task queue and result vector when compared to the node level. In node level, the overhead increases at a large scale as the queue and result vector are under high pressure due to locking and unlocking.

VI. DISCUSSION

Based on the results of all the experiments, we understood that the parallelization could not improve the response times when compared to sequential. Among the proposed variants, we noticed that the sub-tree level gives better performance than the node level. However, the overhead due to locking of task queue on both the variants is still a challenging factor on the implementation of parallelization.

When we increase the data points for evaluation, sub-tree Elf could perform as good as sequential Elf. Better load distribution in terms of the number of threads and elimination of lock overhead might give us better performance when compared to sequential Elf.

VII. CONCLUSION AND FUTURE WORK

Predicate evaluation is an important task when it comes to main-memory databases. Elf was introduced as an index structure that helps to evaluate the predicate easily and use the memory layout efficiently. We tried to implement parallelization to traverse the TIDs in parallel. Thereby, assuming to attain a better response time.

We proposed two granularities: sub-tree level and node level. The implementations are described in the above sections. Furthermore, when we evaluated our hypothesis we understood that our variants of parallel Elf cannot compete with sequential Elf. As our approaches had lock overhead due to which we had a huge response time while processing the queries. Among the variants, the sub-tree level performed better as it had few bottleneck situations when compared to the node level. In node level parallelization, the concurrent access to task queue increased rapidly due to which it cannot perform well when correlated to sequential and sub-tree level.

For future work, we examine the opportunity of implementing a distributed task queue to compensate the lock overhead when we spawn more threads. To improve the response time we can also try optimizing the locks on the shared resources like task queue and result vector. Furthermore, an intermediate result vector can be used for each thread and can be merged at the end using prefix sum.

References

[1] David Broneske, Gunter Saake, and Martin Sch. Accelerating multi-column selection predicates in main-memory :the Elf approach. pages 647–658. 2017 IEEE 33rd International Conference on Data Engineering (ICDE), 2017.

[2] Tim Hering. Parallel Execution of kNN-Queries on in-memory K -D Trees. Number LNI:216, pages 257–266. GI, 2013.

[3] Veit Köppen, David Broneske, Martin Schäler, and Gunter Saake. Elf : A Main-Memory Index for Efficient Multi- Dimensional Range and Partial Match Queries. *International Journal Of Advancement In Engineering Technology, Management and Applied Science (IJAETMAS)*, 03(12):96–105, 2016.

[4] Jonas Schneider. *Analytic Performance Model of a Main-Memory Index Structure*. PhD thesis, Karlsruher Institut für Technologie, 2016.

[5] Stefan Sprenger and Patrick Schäfer. BB-Tree : A practical and efficient main-memory index structure for multidimensional workloads. pages 169–180. Proceedings of the 22nd International Conference on Extending Database Technology (EDBT), 2019.