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*—**

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. However, 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. However, new index-structures were introduced to efficiently handle multidimensional data. These index-based main memory structures process data sequentially. There is scope of further performance gain by parallelizing the sequential processes in these index-based main memory structures. One of the main memory index-based structure to store multidimensional data is Elf. 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.

- 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 item of each node in a dimension is processed by a single thread and the recursive call to process the remaining items in the node are pushed to the queue. The thread then proceeds to traverse the left child of the tree.

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 subtree level and node level. In Section V, we discuss the evaluation setup and results. Section VI provides a short insight into the conclusion and future work.

II. Parallelization Strategies in main memory index structure

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.

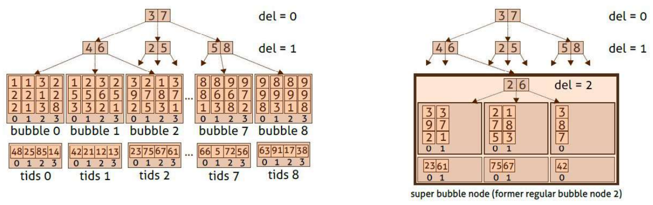


Fig. 1. BB Tree Structure [6]

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.

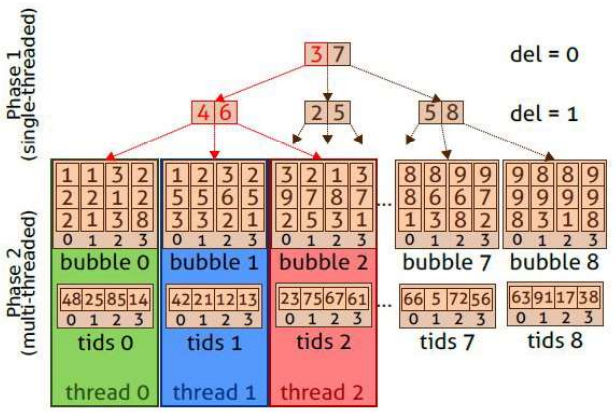


Fig. 2. Parallelization in BB Tree [6]

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 subtree and points on right of hyperplane forms right subtree.

1) KD-Tree Structure: At each level of K-D 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.

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

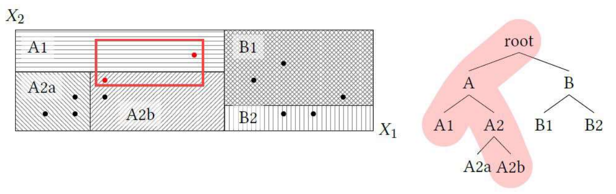


Fig. 3. KD Tree Structure [5]

next level the data is again divided into two parts by a hyperplane along dimension x2.

2) KD-Tree Parallelization: To process search queries parallelly, 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. Now 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.

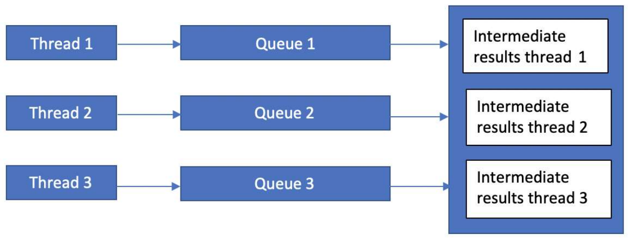

Thread 1
Queue 1
Intermediate results thread 1
Thread 2
Queue 2
Intermediate results thread 2
Thread 3
Queue 3
Intermediate results thread 3

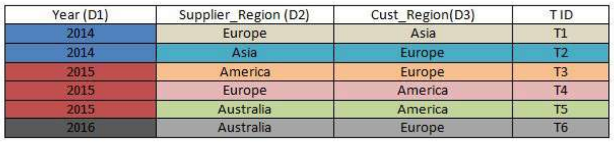
Fig. 4. Parallelization in K-D Trees[2]

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 parallelly. All the threads maintain their intermediate results which are finally merged to get the final query results.

III. Elf - 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”[3]. 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 be observed that the transactions



Year (D1)	Supplier_Region (D2)	Cust_Region(D3)	T ID
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

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[3]. 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 index structure maps the

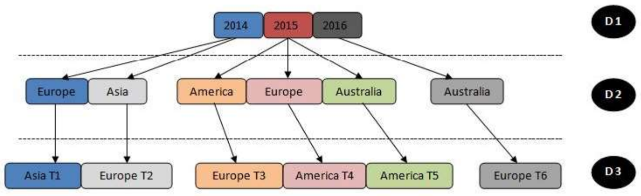


Fig. 6. Prefix redundancy elimination in Elf

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[3], 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. These backdrops have been covered in the optimized version of Elf.
(1) Since the first dimension in the example contains all the possible unique values which usually consists of large list of entries out of which we need to find the desired path.
(2) 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. For which it uses the unnecessary memory for saving pointers and also time taken for processing the traversal.
(1) HASH MAP:
The first dimension is already ordered which indeed allows us to directly create a hash map. Thus, we remove the overhead for processing a potentially high number of entries. In general we store the pointer to next dimension beside to the value. But, when it is hashed only the pointers to the next dimension are stored.

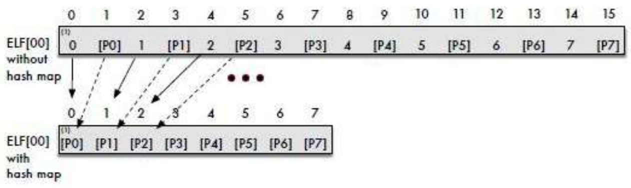


Fig. 7. Hash Map property within first dimension[3]

(2) MONOLISTS:
While we traverse towards the leaf node, the lists contain single object in it which leads to only one TID. This means that there is no prefix redundancy that can be found. Due to this the pointer and the value has to be stored which anyhow leads to single pointer. Therefore monolists are introduced where we directly append below dimensions to that list i.e we then shift from columnar layout to row wise layout. Here, for the year 2016 we have

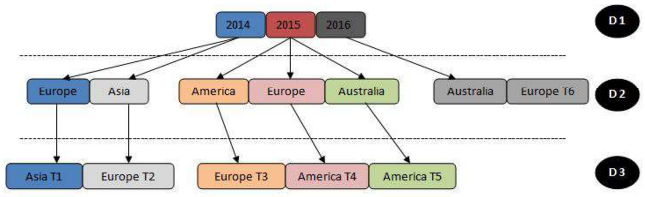
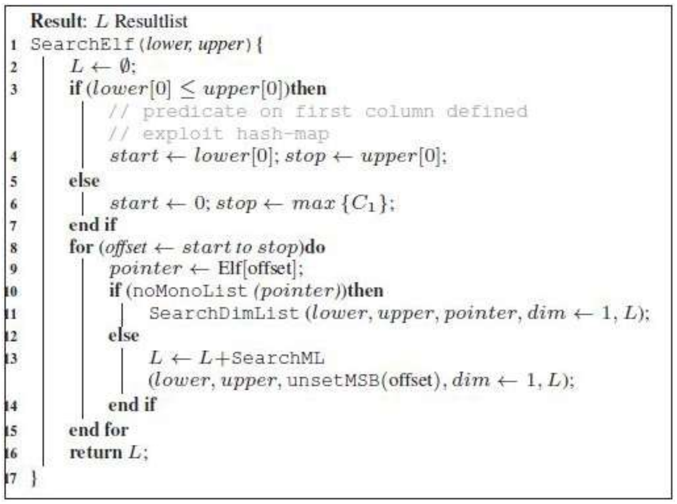


Fig. 8. Monolists

only one index pointing towards Australia and which further points towards Europe(Fig 8). But as discussed, with use of monolists we store the similar 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.

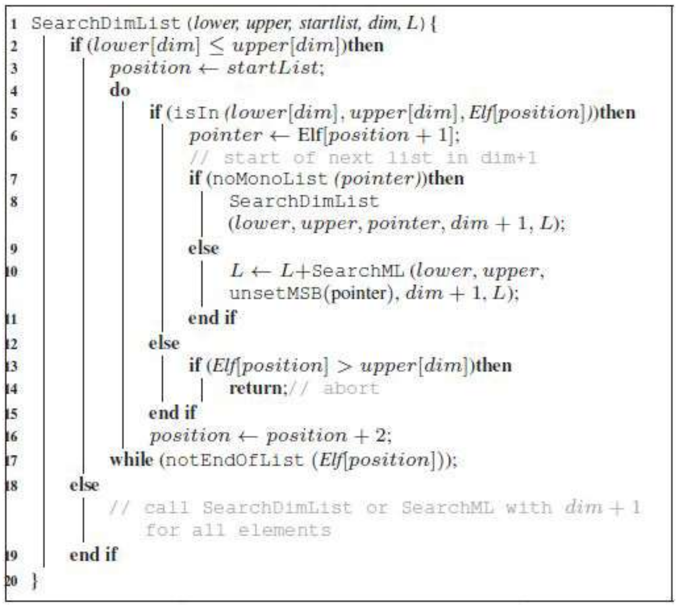
B. SEARCH ALGORITHM:

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



```
   Result: $L$ Resultlist
1  SearchElf (lower, upper) {
2      $L \leftarrow \emptyset$;
3      if ($lower[0] \leq upper[0]$)then
           // predicate on first column defined
           // exploit hash-map
4          $start \leftarrow lower[0]$; $stop \leftarrow upper[0]$;
5      else
6          $start \leftarrow 0$; $stop \leftarrow max\{C_1\}$;
7      end if
8      for (offset ← start to stop)do
9          $pointer \leftarrow$ Elf[offset];
10         if (noMonoList (pointer))then
11             SearchDimList ($lower, upper, pointer, dim \leftarrow 1, L$);
12         else
13             $L \leftarrow L+$SearchML
               ($lower, upper,$ unsetMSB(offset)$, dim \leftarrow 1, L$);
14         end if
15     end for
16     return $L$;
17 }
```

Fig. 9. Search Algorithm 1[3]



```
1  SearchDimList (lower, upper, startlist, dim, L) {
2      if ($lower[dim] \leq upper[dim]$)then
3          $position \leftarrow startList$;
4          do
5              if (isIn ($lower[dim], upper[dim], Elf[position]$))then
6                  $pointer \leftarrow$ 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                     $L \leftarrow L+$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             $position \leftarrow position + 2$;
17         while (notEndOfList ($Elf[position]$));
18     else
           // call SearchDimList or SearchML with $dim + 1$
              for all elements
19     end if
20 }
```

Fig. 10. search algorithm 2[3]

So far in search algorithm 1, we show how to deal with the data within the first dimension list which is hashed. The upper and lower boundaries of all the dimensions should be known as they are the required 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 point 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 whether the next dimension list is a monolist or not. All the elements are checked for the match and when it reaches a monolist, the results will be appended to the list (L).

We now see how to traverse further when the next dimension level is not a monolist(search algorithm 2). Apart from the lower and upper boundary ranges of the queries, the Elf algorithm needs the start offset of current dimension, respective dimension level and the result list (L) as input. The start offset points the first element of the list (line 3) When the predicate is defined on this dimension, 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 would be, when the next higher value is larger than the search attribute value (line 13) or when the list ends (line 17). As the elements in the dimension list are ordered already, so comparing the attribute value with the predicate value and stopping the search would be easy to implement. The Elf search algorithm is a depth-wise search[3], 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 of the data being multi dimensional, it results in few attribute value match which indeed results in visiting of deeper levels. 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 - subtree level 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 would just spawning ' 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*

For achieving parallelism at subtree level we check the maximum number of threads supported by the processor and spawn the same number of threads in the thread pool. After this, the first entry in 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. Meanwhile, 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 thread is available in threadpool. In Depth-first search branch is explored till a monolist is encountered. After this, the control is backtracked to the previous dimension to check if there are any unvisited entries left. If so then it is processed until we encounter a monolist. This process of exploring a branch until we encounter a monolist and then backtracking to explore the remaining unvisited entries from the previous dimension is continued until every entry present in every dimension is visited once. After this, the result in the form of tuple id is stored in the result vector if the query matches. 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. 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 subtree, the task present in the queue would have to wait till any of the thread finishes processing and comes back to the thread pool.

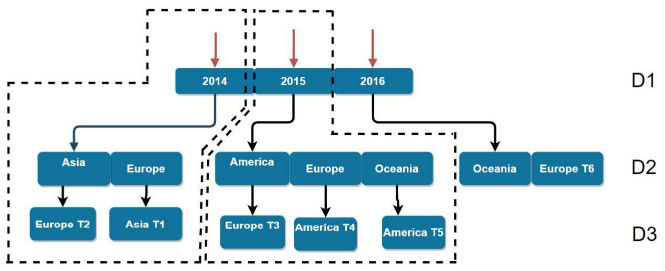


Fig. 11. Structure for Subtree

a) Example: Select * from table where cust_region='Europe' AND year BETWEEN 2014 AND 2015
As seen in the Figure 11 we now process the query using subtree level implementation of Elf. A thread starts processing and at start it encounters the first entry from dimension D1. It checks the year "2014" and it matches the query condition so now it keeps on processing children of "2014" using depth-first search. At the same time, push every entry towards the right of first entry in task queue if it satisfies the query condition. So it will check the second entry in first dimension "2015". As the condition is met second entry along with its children will be 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 the year will not be added to the task queue. Simultaneously as the node are added to the task queue, threadpool is notified about the same. So if there is any free thread in the threadpool it starts to fetch a new task. Coming back to 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 same depth first search.

b) Pseudo code: Each dimension represents a column. In first dimension we check if the dimension is relevant in our predicate that is if the dimension has some condition mentioned in the query. 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.

```
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(1, pointerNextDim, resultTIDs,
          lowerBoundQuery, upperBoundQuery, columnsForSelect))
    else
        pointerNextDim = RECOVER_MASK64;
        Queue(partialMatchMonoList(1, pointerNextDim, resultTIDs,
          lowerBoundQuery, upperBoundQuery, columnsForSelect))
    end
    offset ← offset + 2
end
return resultTIDs
```

Fig. 12. Pseudo code for Subtree-level

B. Node level

Parallelization can be achieved at subtree level however this approach has some disadvantages. For example, the thread utilization in subtree level parallelism is not balanced i.e. some of the threads in thread pool remain idle while other threads traverse entire subtree before returning to the thread pool. To overcome some of the drawbacks of subtree level parallelism, node level parallelism comes into picture. For node level parallelism again, the concept of multiple threads along with queues is used. While traversing the tree at each dimension the nodes are processed such that the first node is processed by a single thread and the recursive function call to other entries in same branch and same dimension are pushed to the task queue. These recursive function calls are then picked up from the task queue by threads from the thread pool whenever they are idle and processed by them. As mentioned earlier, a single thread traverses the first node in each branch till it reaches a monolist or a leaf node at the end or if the entry in the node do not satisfy the query.

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: SELECT * FROM table WHERE cust_region ='Europe' AND year BETWEEN 2014 AND 2015.
Now we will try to execute the query using node level

```
partialMatch1(DIMENSION, START_LIST, RESULTS,
  lowerBoundQuery, upperBoundQuery, columnsForSelect)
position ← START_LIST
if (columnsFor[DIMENSION]) then
    if (toCompare is in bounds) then
        pointer ← get64((ELF[position + 1])
        if (pointer < LAST_ENTRY_MASK64) then
            partialMatch1(DIMENSION + 1,..)
        else
            pointer = RECOVER_MASK64;
            partialMatchMonoList(DIMENSION + 1,..)
        end
    else
        if (upperBoundQuery[DIMENSION] < toCompare) then
            return;
        end
    end
    position ← position + 3
    pointer ← get64((ELF[position + 1])
    while ((toCompare = ELF[position]) < LAST_ENTRY_MASK) do
        if (toCompare is in bounds) then
            pointer ← get64((ELF[position + 1])
            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
                return;
            end
        end
        position ← position + 3;
    end
else
    Contd . .
end
```

Fig. 13. Psuedo code for node-level

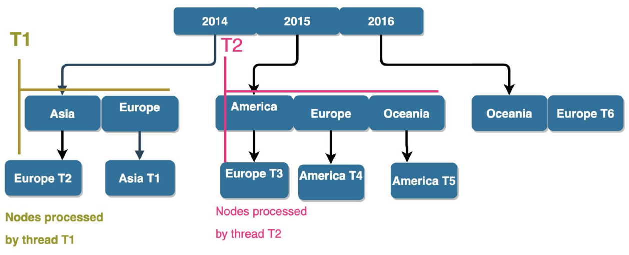


Fig. 14. node level

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)”.

b) Pseudo code: 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 else, 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- Experiment setup (TPC-H)

VI. Result

VII. Conclusion and Future work

APPENDIX

ACKNOWLEDGMENT

References

[1] David Broneske, Gunter Saake, and Martin Sch. Accelerating multi-column selection predicates in main-memory âĂŞ the Elf approach. pages 647–658, 2017.

[2] Tim Hering. Parallel Execution of kNN-Queries on in-memory K -D Trees. (Figure 1):257–266.

[3] Veit Köppen, David Broneske, Gunter Saake, Martin Schäler, Veit Köppen, David Broneske, Gunter Saake, and Martin Schäler. Elf : A Main-Memory Structure for Efficient Multi-Dimensional Range and Partial Match Queries Elf : A Main-Memory Structure for Efficient Multi-Dimensional Range and Partial Match Queries. 2015.

[4] Veit Köppen, David Broneske, Martin Schäler, and Gunter Saake. Elf : A Main-Memory Index for Efficient Multi- Dimensional Range and Partial Match Queries. 03(12):96–105, 2016.

[5] Jonas Schneider. Analytic Performance Model of a Main-Memory Index Structure. 2016.

[6] Stefan Sprenger and Patrick Schäfer. BB-Tree : A practical and efficient main-memory index structure for multidimensional workloads. pages 169–180, 2019.