International Journal of GEOMATE, Nov., 2016, Vol. 11, Issue 27, pp. 2755-2762
Geotec., Const. Mat. & Env., ISSN: 2186-2982(Print), 2186-2990(Online), Japan

OPTIMIZATION OF LOCAL PARALLEL INDEX (LPI) IN PARALLEL/DISTRIBUTED DATABASE SYSTEMS

Mohamed Chakraoui and Abderrafiaa El Kalay

Faculty of Sciences and Techniques Marrakech, Cadi Ayyad University, Marrakech, Morocco

ABSTRACT: The widespread growth of data has created many problems for businesses, such as delay requests; in this paper, we propose several methods of partitioning an index B*Tree in multi-processor machines in parallel/distributed database systems and collaboration between processors when executing multi-queries. When optimizing, indexing automatically comes to mind; we distinguish two types of indexing: B*Tree and Bitmap. Since the advent of multicore computers (multi processors) parallelism becomes an indispensable part of optimization. Our work will focus on partitioning each table on three parts following indexing key partitioning; each processor will host a partition of the index, and the first processor that will finish will immediately take another partition of the index pending according to the priority. The parallelism will reduce the CPU cost then reduces execution time; collaboration between processors will further reduce these costs.

*Keywords: Tuning indexes, Collaboration between processors, Optimization, B*Tree, Partitioning*

1. INTRODUCTION

Tuning databases is an essential task since the design phase to the maintenance phase of parallel/distributed database systems. When a request is sent to the RDBMS, it will be parsed and translated into RDBMS language, then the RDBMS establishes several execution plans possible, then the RDBMS optimizer chooses the most suitable one; finally, it runs the execution plan chosen. All RDBMS provide two types of optimizers: Rules Based Optimizer (RBO) and Cost Based Optimizer (CBO), all of actual RDBMSs use CBO[1]. The CBO is an optimizer that is based on the estimated costs of performing the operations execution plans. For a given query, the RDBMS creates several possible execution plans and the RDBMS optimizer estimates for each one the cost performance and chooses the lowest.

Many solicited issues in this research field are about the efficiency, speed and reliability of database systems. Many papers have discussed optimization of databases; however, they still remain insufficient and could not get a top requested by the researcher community. Our paper comes in this context to a progressive thread, then provides a complete theme in parallel databases indexing and supports it by yielded experimental results never been established. Asking specialists in this interesting discussion and provides a solid idea to RDBMS designers, then asking CPU designers to take into consideration the collaboration between processors when accessing the parallel databases with this method since the results are there.

To estimate the cost of an execution plan the RDBMS evaluates the cost of resources used to implement the plan following the priority:

CPU time.

The number of input / output (I/O) disk storage.

The amount of memory (Random Access Memory) required.

This cost depends not only on the query itself, but also on the data that it bears. For example, given a table of 100 records, a single query can be rapid, but for a request of 1.000.000 records, a simple query can be very slow and expensive (input/output disk and CPU utilization). Therefore, the cost depends on the data of the query and not only on the query itself; it is the reason why we resort to indexing.

Data retrieval does cause serious problems when schemas and indexes are not created properly. However, inserting data can often causes serious performance issues as well[2].

Our contribution in this manuscript can be listed as:

Parallelism is an excellent technique to reduce the execution time and optimize databases.

Multiple query execution time, partitioning every query to three parts, and make a waiting line to every processor to accessing to each part.

Collaboration between processors is a new technique based on the parallelism and benefits from sleeping time of each processor.

Table 1 A part of the running table CLIENTS

NCLIEN	NAME	CITY	COUNTRY
1	Mohame	Marrakech	Maroc
3	Hamid	Casa	Maroc
25	Khalid	Fes	Maroc
32	Salah	Casa	Maroc
39	Karim	Safi	Maroc
43	Houdi	Essaouira	Maroc

46	Jalal	Sfaqes	Tunisie
50	Charif	Casa	Maroc
55	Jamali	Agadir	Maroc
66	Gill	Doncaster	United
67	Will	Arizona	USA
70	Bernar	Munichen	Germany
76	Mak	Curitiba	Brazil
78	Bridge	PointeClai	Canada
80	Fransis	Yamagata	Japan
81	Brolin	Rockford	USA
83	Clark	Linz	Australia
85	Favreau	Zagreb	Croatia
87	Phillippe	Lyon	France
88	Nakai	New Delhi	India

Index B*Tree:

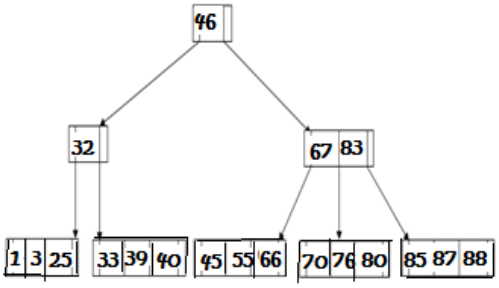

Fig.1 Part of index B*Tree for the Table CLIENTS

2. RELATED WORK

The indexes in databases are like indexes in books; it addresses directly the desired information, without going through the whole book. Indexes are divided into two major types, Bitmap index and B*Tree index.

Several researches on different indexes and optimization of relational distributed databases are discussed before; DAVID Taniar[4] discussed B+Tree, SERGEY Bereg and all[5] discussed K-Tree, FM-Index[6], AR-Tree[7], R-Tree[8] and others. In addition, many studies have been optimized complex queries in distributed databases and cloud computing using partitioned databases[9]. Following our investigations there is no approach like our, making the collaboration between processors in partitioned databases on multi-cores and multiprocessors machine[10]. New database architecture based on batching queries and shared computation across many concurrent queries in a shared disk[10].

In B*Tree, internal nodes (non-leaf) can have a variable number of child nodes within some pre-defined ranges. When data are inserted or removed from a node, its number of child nodes changes. In order to maintain the pre-defined range, internal nodes may joined or split. Because a range of child nodes is permitted, B*Trees do not need re-balancing as frequently as other self-balancing search trees, but may waste memory space, since nodes are not entirely full[11].

Each internal node of a B*Tree will contain a number of keys. Keys act as separation values, which divide its node. For example, if an internal node has three child nodes, then it must have two keys: a and b. All values in the leftmost node will be less than a; all values in the middle node will be between a and b; then all values in the rightmost node will be greater than b. usually, the number of pages is the fixed size capable of holding up to 2*k keys, but pages need only be partially replete.

These trees grow and contract; the nodes can split into brothers, or two brothers can merge or "concatenate" into a single node. The splitting and concatenation processes are initiated at the leaves only and propagates them to the root. When the root node splits, a new root must be introduced, and this is the way in which the height of the tree can increase[12].

The opposite process occurs if the tree contracts.

Definition: We suppose h >= 0 an integer, k is a natural number. A directed tree T is in the class Z(k,h) of B*Trees if T is either empty (h=0) or has the following properties:

i) Each path from the root to any leaf has the same length h, also called the height of T, i.e., h = number of nodes in path.

ii) Each node except the root and the leaves have at least k + 1 son. The root is a leaf or has at least two sons.

iii) Each node has at most 2*k + 1 daughters[13].

3. OPTIMIZING INDEX

Our contribution in this paper is within the scope of parallel/distributed databases indexing, as this subject is not discussed since 2004[4]. Following an investigation into the business and technology services, we saw a dire need in terms of tuning the search and update data. Several specialists in the design of database systems, fails index usages, strong reasons that push us to propose two strong methods about database optimization, and report to researchers that there are still things to do in the optimization of parallel/distributed database systems.

With the advent of multi-core computers, it is essential to take advantage of these cores, so we appeal to the parallelism. This paper presents different database optimization techniques that can be employed for parallel/distributed processing. Our motivation to optimize parallel/distributed database systems is its importance on rapidity and reliability of information retrieval. Rapidity has become an important thing everywhere, then onerous database

applications become known as inefficient and inacceptable.

In the first time, we partition our table by range (attribute NoClient) into three parts, then we create and partition a local parallel index (attribute NoClient) into three parts too, then we attribute each part to one processor, and so on for each table that participate to related queries, then we collaborate between processors.

In the second time, we partition our tables by list into three parts, then, we create and partition our local parallel index into three parts by list too and subsequently we always attribute each part to one processor, then we collaborate processors.

The table 1 illustrates a simple table with four attributes client (noclient, name, city, country) as a testing example.

Finally, we compare between the results obtained by the two methods.

We would like to note that David Taniar discussed the Global Parallel index GPI[4]. However, our proposed method is a combination of new and old methods and technologies, our contribution improves the results already obtained, by changing these principles by ours, and GPI by local parallel index (LPI) and then, we add a great optimizing method that consists on the collaboration between processors.

We use java 1.7 to programming our test application,

We use MySQL 5 and Oracle database 11g release 2 to execute our methods.

We use the MPJ (Message Passing Interface for java) to communicate between processors.

4. GLOBAL PARALLEL INDEX

Global index is a tree structure that can be built from an attribute or more, of number or varchar type and not lob or bfile. We can partition it by range, by hash or by list, and it can be based on a partitioned or non-partitioned table.

Global Parallel Index (GPI)[14] can be partitioned indifferently with the underlying table; but the problem is harder to maintain when the based table is partitioned.

5. LOCAL PARALLEL INDEX

To discuss the parallel/distributed databases automatically we discuss table partitioning. Local Parallel Index (LPI) has the advantage that the index and the underlying table partition identically. In this paper, we propose two types of table partitioning. The first time is to partition our running tables as CLIENTS (NOCLIENT, NAME, CITY, COUNTRY) by RANGE into three parts and we suggest that we have a multiprocessors computer (3 processors or more). Then we create and partition our local index by range into 3 parts too, then we assign each partition to one processor to benefit from the parallelization, and finally our processors have to work together I.e. The processor that finish its work gives help to the next and so on.

When partitioning the table by Hash, the hash index uses the same hash function to arrange the RowIDs on different segments in ascendant order. The optimizer sends the value of each data to the hash function to build segments of data elements[15]. The following section presents our proposed methods briefly.

The figure 2 shows distributed partitioning queries and how allowing every part to one processor, following the algorithm of allowing distributed queries to processors.

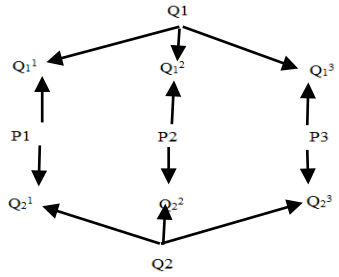

Fig. 2 Distributed partitioned queries

We partition Q1 to Q_1^1 Q_1^2 Q_1^3. P1, P2 and P3 are successively processor number one, processor number two and processor number three. The number n of distributed queries is Qn.

Algorithm1: allowing distributed queries to processors:

Sorting (Q_1^n); // n =1, 2, 3
For i=1 until i=k; // k is the number of queries
    If (Q_i^n is given); // n=1, 2, 3
    Then free Pn;
    Allow (Pn) to Qm; //m!=n and m=1, 2, 3
    When Q_i^m is given;
Allow (Pn and Pm to Pl); // l !=m and l !=n and // l =1, 2, 3
When Q_i^l is given
Return (Q1)
Free P1, P2 and P3
END

For multiple distributed join queries, we use the one-to-many algorithm, to assign each processor to one table (generally, we have at most a join of three tables).

Algorithm2: One-to-many

1 n: denotes the number of tables
2 Ti: denotes the table number i
3 Pi: denotes the processor number i
4 If (n < 3) then

5 We assign each processor to one table
6 Else
7 Assign T1 to P1, T2 to P2, T3 to P3,
8 then T4 to P1, T5 to P2 and T6 to P6 and so on.
9 END

5.1. First Partitioned Method

In this proposed method we will use all the last algorithms, partition our table into 3 parts by range and the index into 3 parts by range too; the attribute of index partitioning is the same of table partitioning attribute like GPI 1 [4]. In this case, the attribute of index is NoClient. Then we assign each part to one processor, following the availability; the range of NoClient(attribute partitioning) the sets from 1 to 40 is assigned to the processor number one following the availability, from 41 to 80 are assigned to processor two, more than 80 are assigned to processor three. The figure 3 illustrates processors allocation following the first method. The processor has finished its part giving a helping hand to the next who has not yet finished.

To implement an LPI we must be careful. However, it is not difficult as the global parallel index (GPI); this is one of the strength points of the LPI. We explain that the root node is replicated to the processor 2 and not to all processors; the child node 32 and their children are not replicated to processor 2 but to the processor 1.

The child node 67, 70, 76 and 50, 55, 66 and 80, 81, 83 are replicated to the processor 2; the child node 80, 81, 83 is replicated to processor 3 too, because 81, 83 are replicated to processor 3 and 80 is replicated to both processor 2 and 3.
The node 85, 87, 88 is replicated to the processor number 3.

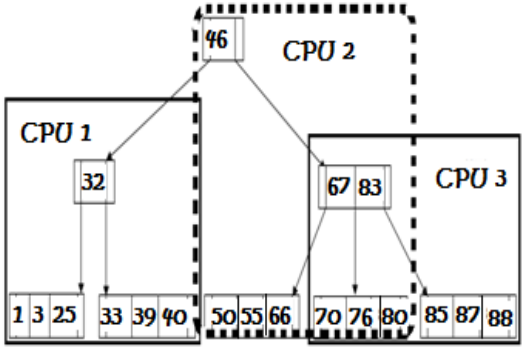

Fig. 3 LPI first method

5.2. Second Method

In this method we will use the same running table example called CLIENTS for simplicity and we will partition it into 3 parts by list (attribute country) like GPI 2[14]. The first partition takes Morocco and Tunisia following the table 2. The second takes United Kingdom, Germany, France, and Croatia as described on table 3. The third partition takes the rest, the table 4 shows the n-uplets components of this part; we index and partition our table following the same attribute of table partitioning, then we assign each partition to one processor following the availability, and finally the processor that finish its work gives help to the next.

The LPI 2 is based on a Varchar2 attribute (NAME Varchar2 (30)), so this is different from the first method; the three lists partitioning (Morocco, Tunisia) and (United Kingdom, Germany, France, Croatia) and (USA, Brazil, Canada, Japan, Australia, India and others) gives the following results:

- The root node 46 is replicated to processor 1
- The Childs node (32) and (1,3,25) and (32, 39, 40) are replicated to processor 1
- The child node (50, 55, 66) is replicated to both of processor 1 and 2, because 50 and 60 are replicated to processor 1 and 66 is replicates to processor 2
- The child node (67, 70, 76) is replicated to both of processor 2 and 3, because 67 and 76 are replicated to 3 and 70 is replicated to processor 2
- The child node (85, 87, 88) is replicated to both of processor 2 and 3, because 85 and 87 are replicated to processor 2 and 88 is replicated to processor 3
- The child node (80, 81, 83) is replicated to processor 3

Table 2 Lines attributed to CPU1 on the second method

CPU1			
1	Mohamed	Marrakech	Morocco
3	Ali	Casa	Morocco
25	Khaled	Fas	Morocco
32	Salah	B. Mellal	Morocco
39	Karim	Safi	Morocco
43	Houdi	Essaouira	Morocco
46	Omar	Sfaqes	Tunisia
50	Charif	Tetouan	Morocco
55	Adam	Agadir	Morocco

Table 3 Lines attributed to CPU2 on the second method

CPU2			
66	Gill	Doncaster	U. Kingdom
70	Bernar	Munichen	Germany
85	Favreau	Zagreb	Croatia
87	Phillippe	Lyon	France

Table 4 Lines attributed to CPU3 on the second method

CPU3			
67	Will	Arizona	USA
76	Mak	Curitiba	Brazil

78	Bridge	Pointe	Canada
80	Fransis	Yamagata	Japan
81	Brolin	Rockford	USA
83	Clark	Linz	Australia
88	Nakai	New Delhi	India

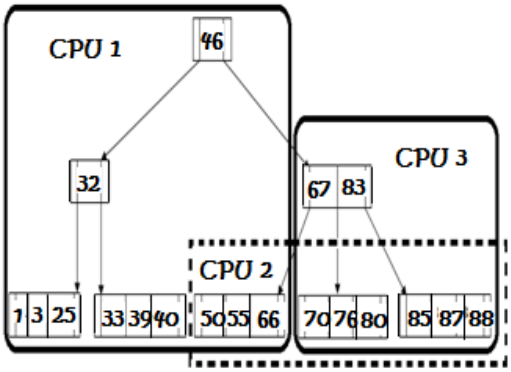

Fig. 4 Local parallel Index (LPI) schema Second method

5.3. competitor access

Concurrent access is among the real problems in the parallelization index, so we think of introducing this algorithm to arrange access to nodes replicated to two processors

Algorithm1: Node-Concurrent-Access

```
1  if (node is replicated to two processors: p1and
2  p2)
3    prohibit (p2)
4    allow (p1)
5   if the operation is update
6      lock (node)
7      if (p1 has finished)
8        unlock (node)
9      end if
10   end if
11 end if
```

6. MAINTENANCE ALGORITHM OF PARALLEL B*TREE

Many methods of concurrent operations on B*Tree and B+Tree have been discussed by Bayer and Schkolnick, David Taniar and others. The solution given in the current paper has the advantage that we use B*Tree and we benefit of parallelism and **collaboration between processors**. In addition, no search through the tree is ever prevented from reading any node (locks only prevent multiple update access). These characteristics do not apply to the previous solution.

6.1 Node Insertion

Node insertion is one of the frequent operations applied to the B*Tree. Inserting an element can merge the node if it is full down, or collapsing it if it is full up. The figure 5, figure 6 and figure 7 bellow illustrate the steps for one case of node insertion:

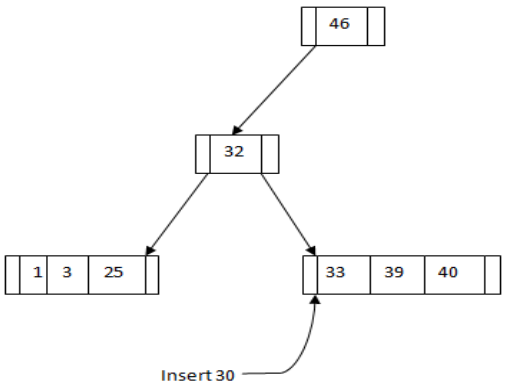

Fig. 5 Node insertion step 1

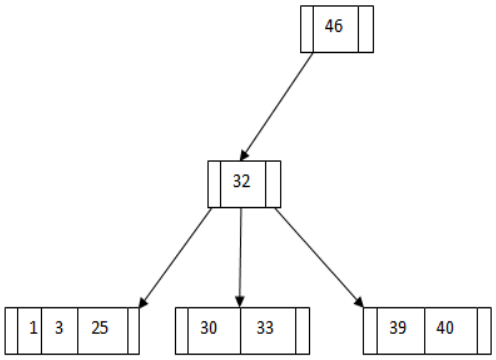

Fig. 6 Node insertion step 2

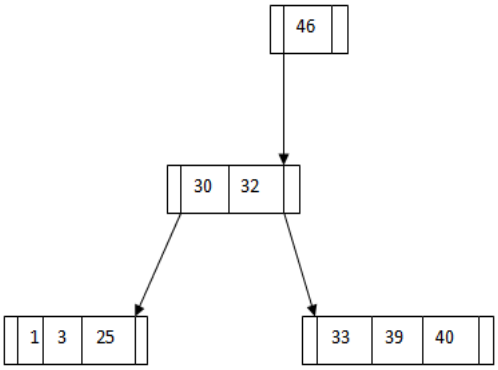

Fig. 7 Node insertion step 3

6.2 Node deletion

Node deletion also usually called.

The following schemas: figure 8, figure 9 and figure 10 describe the steps for one case of node deletion:

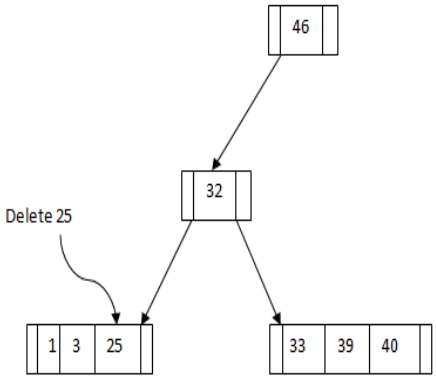

Fig. 8 Node deletion step 1

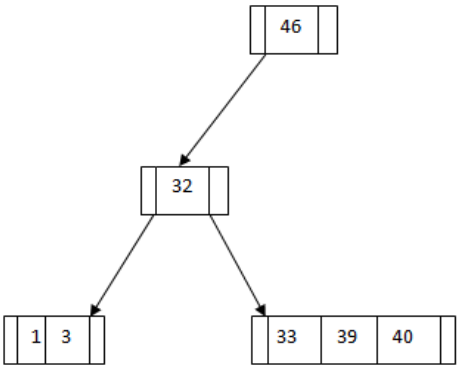

Fig. 9 Node deletion step 2

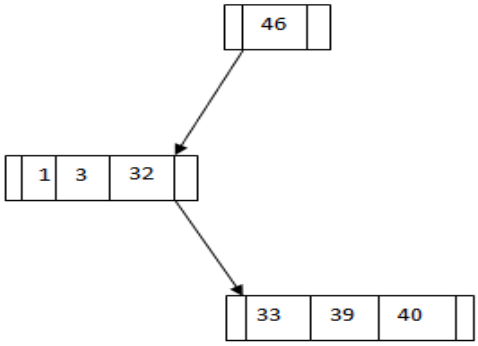

Fig. 10 Node deletion step 3

The following algorithm describes how processors work together:

Algorithm 2: collaboration between processors

1 (range varray)
2 Find the available processor or processors
3 Establish an array of number of size 3: the
4 numbers of the processors, and order it
5 following the availability of each one
6 Assign each range of index to one processor
7 following the order of array making in last step
8 If the processor that key i is finishing its work,
9 gives help to processor i+1 and so 10 on.

7. EXPERIMENTAL RESULTS AND ANALYSIS

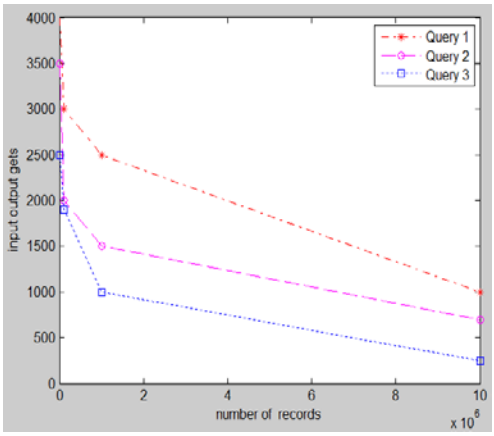

Fig. 11 Costs of different distributed queries for the first method

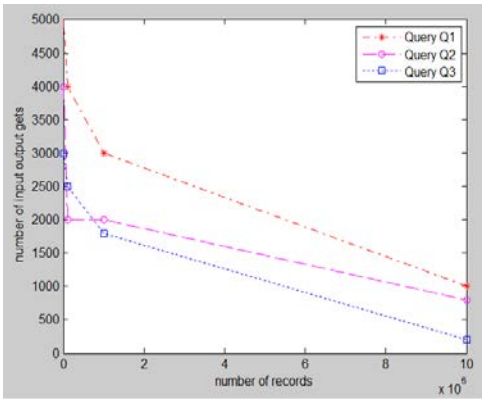

Fig. 12 Costs of different distributed queries for the second method

There are various methods of partitioning an index in parallel database systems, but in this paper, we discuss and improve two most powerful methods for the reason of avoiding redundancy in this current.

7.1 Existing analysis

In shared-memory and shared-disk systems, the major problem for multi-processors machines is the interference between processors in both memory and disk. To reduce network traffic and to minimize latency, each processor is given a large private cache[16]. Parallelism increases performance, but shared resources increase interferences and limit performances. Multi-processor computers often use many processes to reduce interferences. Partitioning a shared-memory system creates many interferences and problems; we find that the performance of shared memory machines is not cost-effective with some processors when running database systems. The shared-disk architecture is not very effective for

database systems. The processor that want to update the data must declare its intention to update the data, once this declaration has been honored and acknowledged by all other processors, the declared processor can read the shared data from disk and updates it. This creates interferences between processors, and then creates delays.

7.2 Multiple query analysis

When we launch a multiple query in parallel search processing, generally it proceeds three steps: processors involvement, index scan and record (data) loading[4] for everyone. In the first step, the RDBMS finds the processor or processors selected by the algorithm of collaboration between processors. In tree traversal, we can localize the record in each processor following the range of the tree or list of name and of course the method used.

The three major methods of table's access are as following. The first is Full Table Scan, when the table is parsed entirely following the order of blocs in the tablespaces. Secondly, the partitioning methods made when the query is performed on a partition of table and not on the table entirely, in this case the table must be partitioned. Moreover, if the optimizer does not accept the partitioning method, we can force it through using HINTs. Finally, the third method is the Table Access by RowID, this method allows the access directly of the RowID in this case the query is based on an index.

Then let us analyze the three major index access methods, UNIQUE SCAN, RANGE SCAN and PARTITION SCAN. Regarding UNIQUE SCAN, the optimizer chooses to parse the tree to find a unique record, generally used for the type of query whose the clause where is an equal like NoClient= 234. For RANGE SCAN, the optimizer parses a part of the tree that host the range searched often used for the type of query of the clause where is an interval like NoClient between 2.000 and 3.000;. And thirdly, the PARTITION SCAN is used by index accessing if the index is partitioned, this method allows the optimizer to parse just the partition of index that host the key or the range of keys on the clause where of the query.

Finally, we cite the three major join operations performing methods. The first is NESTED LOOP; we suggest that we have two tables. CLIENTS table and COMMANDES table. CLIENTS is 10 times bigger than COMMANDES. The NESTED LOOP parses COMMANDES entirely for each data of the table CIENTS, generally used for the sub-query. MERGE JOIN, in this case we use the same tables but we suggest that the sizes of them are approximately close, then we sort both of them following the same criteria for simplify the data search. The third method is the HASH JOIN that we construct a hash table following the index key and then we parse the second table for each value of join column in the hash table.

7.3 Algorithms analysis

Based on Figure 11 and Figure 12, the first method (Fig. 11) presents less consistent gets then less input outputs blocs than the second method (Fig. 12). We can confirm that first method is more accepted as an optimized method than the second method. Following the Figure 4 and Figure 12 they illustrate the second method, we find more correlation between processors, since only selected processors are used and tree traversal and record loading are locally done. In parallel searching, we search single values (for exact match) or several values (for range search). In this type of query, both of the local parallel index first method and second method are efficient but the most optimized is local parallel index first method, because there are no correlation between processors. Which means only selected processors by the algorithm are used (implicated), and data loading are locally done.

When we launch a query in parallel one-index join processing, we search on the indexed table by the attribute of index and the record loading is pointed by RowID. The problem in this processing (one-index join) is that we search each record on the non-indexed attribute (on non-indexed table) this takes a lot of input/output on blocs, which takes a lot of memory. These constraints increase the execution time. In parallel one-index join, we search single values (for exact match) or several values (for range search) from the indexed table and we search for all values of join attribute from the non-indexed table. This processing is not efficient for big tables (table that contains more than 100.000 tuples, but not indexed). In this processing, both parts of the local parallel index are not efficient, but the most wished one is the local parallel index used in the first method, because it bears on the same attribute that uses the join operation in the indexed table.

About parallel two-index join processing, we search single values (for exact match) or several values (for range search) from the first table, then the same processing from the second table and finally we compare the results done according to join operation predicate. If the tables involved in the join operation contain more than 100.000 n-uplets, this processing is preferred; else, if one of them is small, this processing is not efficient. In this case the local parallel index first method is the most suitable because it is based on the same join attribute index [4].

8. CONCLUSION AND FUTURE WORK

In this paper, we have presented in first time two algorithms of tuning parallel databases. The

first is based on partitioning our table, create and partitioning a local parallel index by range. Moreover, the second method is based on partitioning both of them by list. In both of these methods, we assign each part of one processor. The first part is assigned for the processor number one. The second part is assigned for the processor number two and the third part is assigned for the processor number three. Finally, the processor that finished its work giving a helping hand to the processor that not yet finished (collaboration between processors). Following the figures 11 and 12, we are in favor of the first method (partitioning by range) and their algorithms as the most optimized algorithm.

In a second time, we have discussed (presented) three of major methods of query optimization. No-replicated-index, partially-replicated-index and fully-replicated-index [14]; all of them are used with the parallelization and collaboration between processors. We used each of these three methods separately with our proposed methods, for eventually find the most optimal result, according to the results obtained is the third method (fully replicated index). Throughout this paper, we discuss the local parallel index, thanks to these advantages like the absence of correlations between the index and table partitioning, contrariwise the global parallel index.

For our future work, we plan to implement the collaboration between processors in the background of a RDBMS like postgreSQL.

9. REFERENCES

1. Navarro, L., Optimisation des Bases de Donnees. Pearson, 2010.
2. Flora S. Tsai, A.T.K., Database optimization for novelty mining of business blogs. Expert Systems with Applications, 2011.
3. Xianhui Li, C.R., Menglong Yue, A Distributed Real-time Database Index Algorithm Based on B+ Tree and Consistent Hashing. ELSEVIER, 2011.
4. David Taniar, J.W.R., Global parallel index for multi-processors database systems. elsevier, 2004.
5. Sergey Berega, H., Wiener indices of balanced binary trees. Discrete Applied Mathematics, 2007.
6. Alejandro Chacon, J.C.M., Antonio Espinosa, Porfidio Hernandez, n-step FM-Index for faster pattern matching. Procedia Computer Science, 2013.
7. HaRim Jung, Y.S.K., Yon Dohn Chung, QR-tree: An efficient and scalable method for evaluation of continuous range queries. Information Sciences, 2014.
8. Yunjun Gao, Q.L., Baihua Zheng, Gang Chen, On efficient reverse skyline query processing. Expert Systems with Applications, 2014.
9. Tansel Dokeroglu, M.A.B., Ahmet Cosar, Robust heuristic algorithms for exploiting the common tasks ofrelational cloud database queries. Applied Soft Computing, 2015.
10. G. Giannikis, G.A., D.Kossmann, SharedDB, SharedDB: killing one thousand queries with one stone,. Proc. VLDB, 2012. Vol. 5, no. 6.
11. R. Bayer, M.S., Concurrency of Operations on B-Trees. Acta Informatica, 1977.
12. Lilian Hobbs, S.H., Shilpa Lawande, Pete Smith, Oracle_10g_Data_Warehousing. Elsevier Digital Press, 2005.
13. R. Bayer, E.M., Organization and Maintenance of Large Ordered Indexes. Acta Informatica, 1972.
14. DAVID TANIAR, J.W.R., A Taxonomy of Indexing Schemes for Parallel Database Systems. Distributed and Parallel Databases, 2002.
15. Xiaoqing Niu , X.J., Jing Han, Haihong E, and Xiaosu Zhan, A Cache-Sensitive Hash Indexing Structure for Main Memory Database. Springer, 2013.
16. David J. DeWitt, J.G., Parallel Database Systems: The Future of Database Processing or a Passing Fad? 1991.

International Journal of GEOMATE, Nov., 2016, Vol. 11, Issue 27, pp. 2755-2762.
MS No. 1322 received on July 7, 2015 and reviewed under GEOMATE publication policies.

Corresponding Author: Mohamed Chakraoui