Local Parallel Index(LPI) in Databases System

Mohamed CHAKRAOUI and Abderrafiaa EL KALAY

Laboratory of applied Mathematics and Computer Science, Faculty of Sciences and Techniques Gueliz Cadi Ayyad University, B.P 549, Av.Abdelkarim Elkhattabi, Guéliz Marrakech, Morocco.

Received: Nov. 2014 & Published: Jan 2015

Abstract: In this paper we propose a model of partitioning a B*tree index in multi-processor machines for parallel database systems and collaboration between the processors. Indexing a large database is an essential task; 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 in the optimization. Our work will focus on partitioning a table on 3 parts following indexing key partitioning; each processor will host a partition of the index, and the first processor that will finish its work will immediately take the first partition of the index pending according to the priority.

The parallelism will reduce the CPU cost and execution time, and then collaboration between processors will further reduce these costs.

Keywords: tuning, index, collaboration between processors, optimization, b*tree, partitioning, parallel database systems, setting, parallelization,

1. Introduction

Tuning databases is an essential task since the design phase to the maintenance phase. When a request is sent to an RDBMS, the latter will be parsed and translated into RDBMS language, and then the RDBMS establishes several execution plans possible, then the RDBMS optimizer choose the most suitable one, and finally it runs the execution plan chosen. We have two types of optimizers: Rules Based Optimizer (RBO) and Cost Based Optimizer (CBO), all of actual RDBMS use the CBO[1]. The CBO is an optimizer which 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.

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) hard disk;
- The amount of memory (Random Access Memory) required.

This cost depends not only on the query itself, but also on the data which it bears; for example, given a table of 100 records a single query can be powerful, but given a request for 10000 records a single query can be so expensive (input/output disk and CPU utilization), so the cost depends on the data of the query and not only in the query itself; it's the reason why we resort to index.

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)
.

With the arriving of big data and the development of automation technology the data access also increase exponentially; the constraint is the time, response time increase too. To understand and decrease the real response time of these applications we have to sample in real time such equipment[2].

2. Notion of Index

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; our issue is not to discuss the bitmap index and their maintenance, but the index B*tree and their maintenance in parallel database systems.

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 be 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 some space, since nodes are not entirely full[3].

Each internal node of a B*tree will contain a number of keys. The keys act as separation values which divide its node. For example, if an internal node has 3 child nodes then it must have 2 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; and 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 are merged or "catenated" into a single node. The splitting and catenation processes are initiated at the leaves only and propagate 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[4].

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[5].

3. Indexing

An index improves a lot of performance of query, but it is associated with two types of costs: it takes up disk space, and it takes time to maintain when the underlying data changes[1]. Space requirement for index is larger than the space needed to store the data.

With the advent of multi-core computers, it is essential to take advantage of these cores, so we appeal to the parallelism.

In the first time we partition our table by range (attribute NoClient) into 3 parts, then we create and partition a local index (attribute NoClient) into 3 parts too, then we attribute each part to one processor, then we collaborate between processors.

In the second time we partition our table by list to 3 parts and, then we create and partition our local parallel index into 3 parts by list too, and subsequently we always attribute each part to one processor, then we collaborate between processors.

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

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

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

I note that the Global Parallel index GPI was discussed by David Taniar, but we'll come in 2013 with new methods and new technologies to improve the results obtained by him, and change his method to use the local parallel index (LPI) and 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.

Table 1. A part of the running table client

NOCLIENT	NAME	CITY	COUNTRY
1	Mohamed	Marrakech	Maroc
3	Hamid	Casa	Maroc
25	Khalid	Fes	Maroc
32	Salah	Casa	Maroc
39	Karim	Safi	Maroc
43	Houdi	Essaouira	Maroc
46	Jalal	Sfages	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	PointeClair	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:

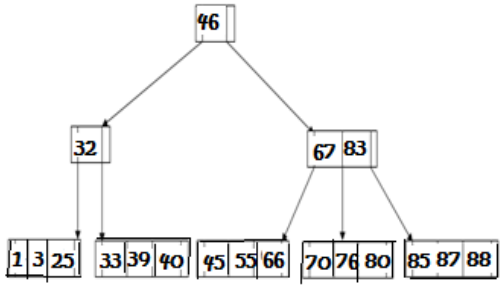

Fig. 1. An Index B*Tree for the Table Client beneath

4. Global Parallel Index

Global Index is a tree structure that can be made in an attribute or more of number or varchar type and not lob and bfile.

Global index can be partitioned by the range, the hash and by list and it can be based on a partitioned or non-partitioned table.

Global Parallel Index (GPI)[6] can be partitioned indifferently with the underlying table; but the problem with Global Parallel Index is harder to maintain when the based table is partitioned.

Our work focuses principally on the Local Parallel Index (LPI), and we take GPI as reference and we demonstrate improved results.

5. Local Parallel Index

To discuss the parallel databases automatically we discuss table partitioning, in this paper we propose two types of table partitioning; the first time is to partition our running table Client (NOCLIENT, NAME, CITY, COUNTRY) by RANGE into 3 parts and we suggest that we have a multiprocessor 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 by group I.e. the processor that finish its work gives help to the next and so on.

Global indexes may not be efficient for this type of query processing, because there is no correlation between the index and table partitioning, so accessing a specific value in a table may involve access to several or all of the index partitions; the same for the index involvement. But sometimes, you need a unique index, which does not include the partitioning key of the table; you will have no choice, then you use a global index. Another reason why global index can be expensive is that when the data in an underlying table partition is moved or removed using a partition maintenance operation, all partitions of a global index are affected, and the

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

index must be completely rebuilt. The shortcomings of a global index are addressed by local indexes.

Local indexes are the preferred indexes to use when a table is partitioned. A local index inherits its partitioning criteria from the underlying table. It has the same number of partitions, sub partitions, partitions and sub partitions bounds as the table partitions, because the index is partitioned identically to the table partitioning. When operation partition maintenance is done on the index. Thus, when partitions are added, dropped, split, or merged in the underlying table, the corresponding index partitions are automatically modified by the RDBMS as part of the same statement. This makes maintenance of the local index extremely efficient, since the entire index does not need to be rebuilt, unlike a global index.

When partitioning the table by Hash, the hash table 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[7].

A. First Method

In this method we will 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 [8]. 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 2 illustrates the processors allocation following the first method. The processor has finished its part giving a helping hand to the next who has not yet finished.

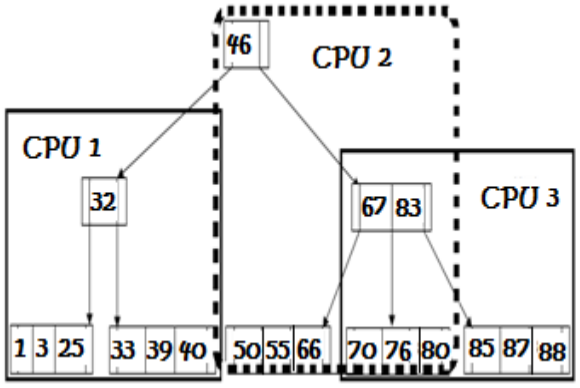

Fig. 2. LPI method 1

To implement an LPI we must be careful. But it is not a big problematic like 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 3.

B. Second Method

In this method we will use the same running table example called Client for simplicity and we will partition it into 3 partitions by list (attribute country) like GPI 2[6]. The first partition takes Maroc and Tunisie following the table 2; the second (United Kingdom, Germany, France, Croatia) as described on table 3; and the third partition takes the rest, the table 4 shows the n-uplets components of this part; we will index and partition our table in the same attribute of table partitioning, then we will assign each partition to one processor following the availability, and finally the processor that finished its work gives help to the next.

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)
.

Table 2. Lines attributed to CPU1 on the second method

CPU1			
1	Mohame	Marrakech	Maro
3	Ali	Casa	Maro
25	Khaled	Fas	Maro
32	Salah	Beni	Maro
39	Karim	Safi	Maro
43	Houdi	Essaouira	Maro
46	Omar	Sfaqes	Tunis
50	Charif	Tetouan	Maro
55	Jamali	Agadir	Maro

Table 3. Lines attributed to CPU1 on the second method

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

Table 4. Lines attributed to CPU1 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	India

The LPI 2 is based on a Varchar2 attribute (NAME Varchar2 (
30)), so this is different from the first method; the list partitioning (Maroc, Tunisie) 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

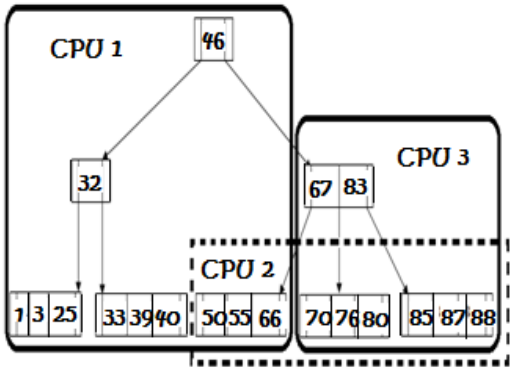

Fig. 3. Local parallel Index schema (Second method)

C. 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

Algorithm: node-concurrent-access

If (node is replicated to two processors: p1and p2)

Prohibit (p2)
Allow (p1)
If the operation is update
Lock (node)
If (p1 has finished)
Unlock (node)

End if

End

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

6. Maintenance Algorithm of Parallel B*tree

A lot of methods of concurrent operations on B*trees have been discussed by Bayer and Schkolnick, David Taniar and others. The solution given in the current paper has the advantage that we benefit from parallelism and collaboration between processors. Also, 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.

D. 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 figure4, figure5 and figure6 bellow illustrate the steps for one case of node insertion:

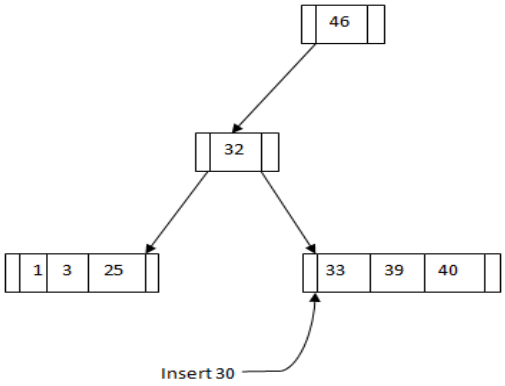

Fig. 4. Node insertion step 1

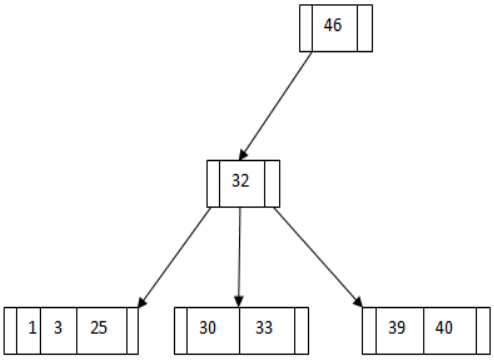

Fig. 5. Node insertion step 2

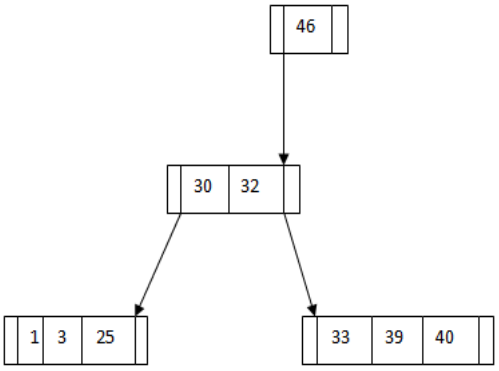

Fig. 6. Node insertion step 3

E. Node deletion

Node deletion is also usually called.

The following schemas: figuire7, figuire8 and figure9 describe the steps for one case of node deletion:

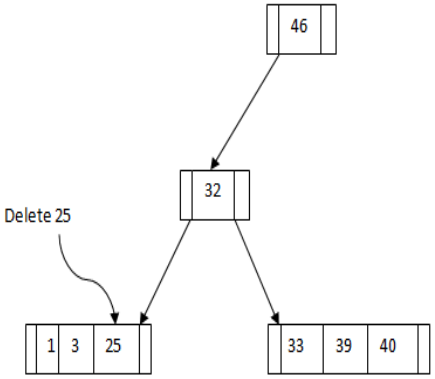

Fig. 7. Node deletion step 1

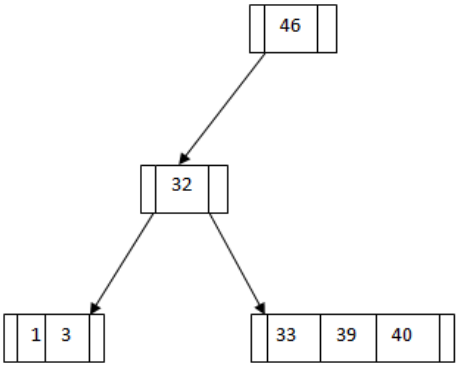

Fig. 8. Node deletion step 2

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

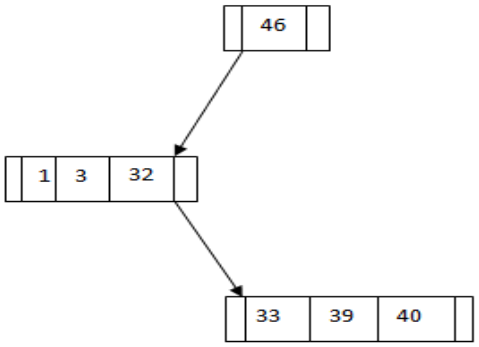

Fig. 9. Node deletion step 3

The following algorithm describes how processors work together:

Algorithm: collaboration between processors (range varray)

- Find the available processor or processors
- Establish an array of number of size 3: the numbers of the processors, and order it following the availability of each one
- Assign each range of index to one processor following the order of array making in last step
- If the processor that key i is finishing its work, gives help to processor i+1 and so on.

7. Analysis

There are various methods of partitioning an index in parallel database systems, but in this paper we focus on two major methods for the reason of avoiding redundancy in this current.

Existing analysis:

In shared-memory and shared-disk systems the major problem for shared-memory for multi-processors 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[9]. Parallelism increases, interference but shared resources limits performance. Multi-processor computers often use many processes to reduce this interference. 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 applications database. The shared-disk architecture is not very effective for database applications that write a shared database. 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 the other processors, the declared processor can read the shared data from disk and update it. This creates processor interference and delays. It creates heavy traffic on the shared interconnection network[9]

General query analysis:

When we launch a query in parallel search processing, generally it proceeds three steps: processors involvement, index scan, and record (data) loading[8]. In the first step, the RDBMS find the processor or processors selected by the algorithm: 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 accessing of tables are: the first is Full Table Scan when the table is parsed entirely following the order of blocs in the tablespaces; the PARTITION methods made when the query is performed on the partition of table and not on the table entirely in this case the table must be partitioned and if the optimizer doesn't accept the PARTITION method we can force it through the HINT; then third method is the Table Access By RowID, this method allow the access directly of the RowID, in this case the request is based on an index.

The three major methods of accessing of index are: UNIQUE SCAN, RANGE SCAN and PARTITION SCAN. Regarding UNIQUE SCAN, the optimizer choose to parse the tree to find an unique record, generally used for the type of query of the clause where is an equal like NoClient= 234; RANGE SCAN the optimizer

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

parse 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 2000 and 3000; and the third of major methods PARTITION SCAN is used for index accessing is the partition that use the partitioning index, this method allows the optimizer to parse just the partition of index that host the key or the range of key on the clause where of the query.

And finally we cite the three major methods of performing the join operation, the first is NESTED LOOP we suggest that we have two tables: Client table and Commande table, Client is 10 time more big than Commande table, the NESTED LOOP parse Commande table entirely for each data of the table Client: generally used for the sub query; MERGE JOIN, on 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; and the third method is the HASH JOIN that we construct an hash table following the index key, then we parse the second table for each value of column join in the hash table.

Algorithms analysis:

Based on Table 1, Figure 2 and Figure 3, the first method is more accepted as an optimized method than the second method because in the Figure 3 that 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 suitable is the 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 the index, and the record loading is pointed by the 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, then it takes 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 the big table (table that contain more than 10000 tuples and not indexed). In this processing both of parts of the local parallel index are not efficient, but the most suitable one is the local parallel index used at 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 the predicate of join operation, if the tables involved in the joint operation contain more than 10000 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 the parallel index is based on the same attribute of the index join attribute[8].

8. Conclusion and future work

In this paper we have presented in first time two algorithms of tuning databases; the first is based on partitioning these tables and create and partitioning a local parallel index by range, and the second is based on partitioning all by list, in both of the methods we assign each part of one processor: the first part is assigned for the processor number 1, the second part is assigned for the processor number 2 and the third part is assigned for the processor number three and finally the processor that finished its work giving

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

a helping hand to the processor that not yet finished (collaboration between processors). We choose the first method (partitioning by range) 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 [6]; all of them use the parallelization and collaboration between processors; we test each of these three methods separately, to conclude finally that the third method (fully replicated index) is the most optimal According to the results obtained. And on all of time we discuss the local parallel index, because in this type of index no correlation between the index and table partitioning, contrariwise the global parallel index can use a lot of correlations between index and underlying table partitioning which increases the execution time, and input output gets.

For our future work, we will implement these methods in the background of an RDBMS like postgresql. And we plan to execute these methods on three separated machines and the server in another machine.

References

1. Navarro, L., *Optimisation des Bases de Donnees.* Pearson, 2010.
2. Xianhui Li, C.R., Menglong Yue, *A Distributed Real-time Database Index Algorithm Based on B+ Tree and Consistent Hashing.* ELSEVIER, 2011.
3. R. Bayer, M.S., *Concurrency of Operations on B-Trees.* Acta Informatica, 1977.
4. Lilian Hobbs, S.H., Shilpa Lawande, Pete Smith, *Oracle_10g_Data_Warehousing.* Elsevier Digital Press, 2005.
5. R. Bayer, E.M., *Organization and Maintenance of Large Ordered Indexes.* Acta Informatica, 1972.
6. DAVID TANIAR, J.W.R., *A Taxonomy of Indexing Schemes for Parallel Database Systems.* Distributed and Parallel Databases, 2002.
7. Xiaoqing Niu , X.J., Jing Han, Haihong E, and Xiaosu Zhan, *A Cache-Sensitive Hash Indexing Structure for Main Memory Database.* Springer, 2013.
8. David Taniar, J.W.R., *Global parallel index for multi-processors database systems.* elsevier, 2004.
9. David J. DeWitt, J.G., *Parallel Database Systems: The Future of Database Processing or a Passing Fad?* 1991.

(DOI: dx.doi.org/14.9831/1444-8939.2015/3-1/MAGNT.3)

.