International Journal of GEOMATE, Aug., 2016, Vol. 11, Issue 24, pp. 2267-2274
Geotec., Const. Mat. & Env., ISSN: 2186-2982(Print), 2186-2990(Online), Japan

# TUNING DIFFERENT TYPES OF COMPLEX QUERIES USING THE APPROPRIATE INDEXES IN PARALLEL/DISTRIBUTED DATABASE SYSTEMS

Mohamed Chakraoui[1], Abderrafiaa El Kalay[2] and Naoual Mouhni[3]

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

**ABSTRACT:** In this paper, we discuss the most powerful techniques of tuning parallel/distributed databases. As in engineering, database tuning becomes an inescapable part of big projects since the conception phase of research projects. The needs of companies including big data have increased to databases optimization. Systems that not take into account the optimization rules become heavy after five years of their production; these reasons were of a paramount of importance to prepare this paper. Indexing is the most suitable way to optimize database systems, further one of the top ways of optimizing index is the application of parallelization. In this paper, we will discuss parallelization and we will practice it with different complex queries and sub-queries using different types of indexes; then we will compare the results gotten from each index. To top it all, the most suitable interference between the major types of index: B*Tree index, Bitmap index, composite parallel index, local parallel index and global parallel index.

*Keywords: parallelization, sub-query, optimization, interference between indexes, b*tree, bitmap, local index, data partitioning, sub-query.*

## 1. INTRODUCTION

To satisfy the needs of data processing speed and decrease the response time of complex queries, optimizing a large database remains essentially to good query writing. A poorly written query can increase the input output gets, which leads to the increasing of the execution time. The most companies' needs have increased to store and analyze the ever-growing data transparently, such as search logs, crawled web content and click streams. Such analysis becomes crucial for businesses in different ways; such as to improve service quality and support novel features, to detect changes in patterns over time and to detect fraudulent activities[1] . As actual computers have powerful processors and very speed RAM, then applications require much higher data operation speed, the traditional RDBMS. Difference between memory and disk in terms of writing and reading speed is very large, so since 1980s, researchers try to move the whole database from disk into main memory to improve the execution time. From the one hand to optimize the latency time, because main memory is faster than disk and from the other hand, to minimize the interference between different processes when accessing to the data; this type of databases is called Main Memory Databases[2]. Our ultimate topic in this paper is not to discuss main memory databases, but to discuss and analyze different types of indexes and their interferences on a given query.

The most researches and practices focus on two types of indexes, BTree and Bitmap. BTree is three types, local index, global index and composite index. The same for Bitmap index, bitmap local index, bitmap global index and bitmap composite index. We will make a join query between two or more tables, each one is indexed by a distinct type of index; we analyze every method, and compare between them.

The list-based, view-based and disk-based methods are other optimization's methods. First, the list-based methods construct a set of lists by sorting all tuples based on their values in each attribute. It then finds the tuples by merging as many lists as needed. Second, the view-based methods pre-compute the results of multiple queries and store these results as a view. Third, disk-based methods build an index using disk[3].

Commercial optimizers classify query optimization techniques into two dimensions, optimization time and optimization granularity. Optimization time is when optimization decisions are made. However, optimization granularity defines if the optimization decisions are based upon dynamic sampling. In terms of optimization time, some database systems determine query plans in advance at compile time. While others forego pre-computed plans and "route" tuples on the fly at runtime[4].

Our contribution in this paper is to propose a set of index interference structures and algorithms, which allow us to decrease efficiently the costs of different complex queries in parallel/distributed database systems. These costs are real-time querying or real execution time and CPU cost or input output gets.

The following table describes a part of the

table CLIENTS. Is due to flexibility of the document, we cannot specify the full table.

Table. 1 The part of the running table CLIENTS

| NO | NAME | CITY | COUNTRY |
|---|---|---|---|
| 1 | Mohamed | Rak | Maroc |
| 3 | Hamid | Casa | Maroc |
| 25 | Khalid | Fes | Maroc |
| 32 | Salah | Casa | Maroc |
| 39 | Karim | Safi | Maroc |
| 43 | Houdi | Essaouia | Maroc |
| 46 | Jalal | Sfaqes | Tunisie |
| 50 | Charif | Casa | Maroc |
| 55 | Jamali | Agadir | Maroc |
| 66 | Gill | Doncast | United |
| 67 | Will | Arizona | USA |
| 70 | Bernar | Muniche | Germany |
| 76 | Mak | Curitiba | Brazil |
| 78 | Bridge | PointeCl | Canada |
| 80 | Fransis | Yamaga | Japan |
| 81 | Brolin | Rockfor | USA |
| 83 | Clark | Linz | Australia |
| 85 | Favreau | Zagreb | Croatia |
| 87 | Phillippe | Lyon | France |
| 88 | Nakai | New | India |

## 2. RELATED WORKS

The Structured Query Language (SQL) is the most used language in the existing database applications by database researchers as a standard language of querying[5]. SQL was designed for managing data in a Relational or Object Database Management Systems (RDBMS or ODBMS). SQL makes it possible to create, read, update and/or delete records. Actually SQL has many dialects that can well-establishes different query languages, it is widely applied in industrial context. SQL is specified around a set of operations on data stored in tables. Working with object concepts, requires traditional Object-Relational Mapping (ORM). Furthermore, the complexity of such queries would quickly increase beyond levels of feasibility. Even though the creation of nested, recursive queries using standard SQL is available through some implementations and extension modules. Even if there is, no standardized support that would be necessary to match the requirements formulated earlier[6] .

### 2.1 Btree Index:

BTree is a well-organized structure as a tree, so the information retrieval will be easier; a BTree index is based on either one column or more (composite BTree index).

The BTree contains many nodes, the highest node called root node, the descendent node called child node. Each node that have a child called internal node and the node that has no child called leaf node. Each node have k keys; we suppose that the root node has two keys n and m, the right child keys must be lower than n, the left child keys must be higher than m and the middle child node keys must be between n and m.

We distinguish between two major types of BTree, B+Tree discussed by David Taniar[7] and B*Tree discussed by Chakraoui Mohamed[8]. The Figure 1. Describes a part of tree structure.

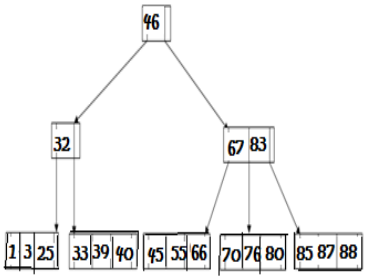


Fig. 1. A part of our B*Tree index structure

### 2.2 Bitmap Index

The bitmap index is also based on one or more columns; it is based on the bit masks for the separate values on the indexed column or columns.

This type of index is useful when the indexed column contains many distinct values and when the predicate in the query is an equal operation[9].

## 3. PARALLELISM

Resources such as memory space or CPU time used for buffering messages or temporal data can be released once a given query has consumed its quantum, being necessary to keep only the partial results calculated until that moment and the query state data used to enable its next quantum in the next super step. Thus processing a given query completely can take one or more super steps. In the case of asynchronous mode of parallel computing, the round-robin principle is emulated by performing proper thread scheduling at each processor to grant each active query its respective quantum of execution[10] .

With the arriving of big data, the classical index takes an important portion of the main memory and execution time; it is not match reduced on a given query. Then researchers migrate to parallelism as a good way to reduce this execution time, following the number of processors available on a given machine. Parallelism allows executing one query by more than one processor.

We take the following tables as a running example:

CLIENTS (noclient number not null, name varchar2(50) not null, city varchar2(50), country varchar2(25)).

CMD (nocmd number not null, noclient# number, datecmd date, etatcmd char(1)).

LINECMD (noline number not nul, nocmd# number not null, productId number not nul, quantity number not null, amount number not null).

### 3.1 Partitioning by Range

We divide the table CLIENTS by range into three partitions. We give to the first partition the range [0, 40], to the second partition the range] 40, 80] and to the third partition the rest] 80, +1000].

The Figure 2 describes the partitioning bounds of the underlying table by range.

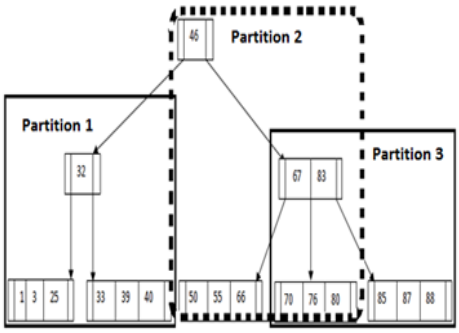


Fig. 2. Part of partitioning B*Tree index by range

### 3.2 Partitioning by List

In this case, the index is based on a varchar column and then the partitioning can perform as follows: if the second letter of name (partitioning column) is a consonant, we place the record in the first partition. If the second letter of name is a vowel of a letter a or e, the record go to partition two and finally if the second letter of the name is a vowel of letter i, o or u the record went to the third partition[7].

The following tables illustrate the partitioning of our running example of index B*Tree.

TAB. 2 – Part 1of index partitioning by list

| Partition 1 | |
|---|---|
| NOCLIENT | NAME |
| 25 | Khalid |
| 50 | Charif |
| 78 | Bridge |
| 80 | Fransis |
| 81 | Brolin |
| 83 | Clark |
| 87 | Phillipp |

TAB. 3 – Part 2 of index partitioning by list

| Partition 2 | |
|---|---|
| NOCLIENT | NAME |
| 3 | Hamid |
| 32 | Salah |
| 39 | Karim |
| 46 | Jalal |
| 55 | Jamali |
| 70 | Bernar |
| 76 | Mak |
| 85 | Favreau |
| 88 | Nakai |

TAB. 4 – Part 3 of index partitioning by list

| Partition 3 | |
|---|---|
| NOCLIENT | NAME |
| 1 | Mohamed |
| 43 | Houdi |
| 66 | Gill |
| 67 | Will |

### 3.3 Global Parallel Index

Global parallel index (GPI) is a BTree structure made on the underlying table globally, GPI do not like to partition the underlying table. However, it could do so; but partitioning methods and intervals on global index and partitioning methods and intervals on the underlying table could differ. One of the disadvantages of global index is when a data on the underlying table is moved or removed, all partitions of a global index are affected, and the index must be completely rebuilt[11].

### 3.4 Local Parallel Index

Local index is a BTree structure that can be partitioned. The partition methods, intervals and bounds must be respectively the same of the partition methods, intervals and bounds on the underlying table. LPI is one of the most useful indexes, among their advantages, the simplicity and the fact that the bounds of their partitions are the same of table partitions[9] and it is very dependent to the underlying table.

### 3.5 Composite Parallel Index (CPI)

Composite parallel index is a BTree index, it can be a global or local parallel index; it is based on two or more columns. Composite index can be useful when the predicate on the query is a logical and between two values on two columns. Parallelism can be exploited by partitioning one or more underlying attributes at the underlying table.

## 4. INTERFERENCE BETWEEN LOCAL AND GLOBAL PARALLEL INDEXES

For an efficient execution of the continuous location-dependent queries, incremental search algorithms are required, thus avoiding solving each search problem independently from scratch[12]. Incremental search implies reusing information from previous researches for each query, to obtain the current result without having to recomputing everything each time[13].

### 4.1 Simple Select Query (SSQ)

For search queries, the RDBMS cannot lock any partition, because the select query does not change any underlying data. We take the following query as a running example:

SELECT city FROM CLIENTS where noclient = 39 and name = 'jack'; The execution time is described in Figure 3.

### 4.2 Select Join Query (SJQ)

The same reasoning for the previous section, initially, we partition the table CLIENTS by list following the column 'name' as described before; then we create a local parallel index based on the same column 'name', and we partition it too following the underlying table partitioning. Secondly, we create a global index based on the table CMD, we take the following query as a testing example. SELECT datecommande FROM CMD, CLIENTS WHERE CLIENTS.noclient = CMD.noclient AND CLIENTS.name in 'mohamed'; The execution time for this type of interference is described in Figure 3.

### 4.3 Simple Update Query (SUQ)

Optimizing update query based on a single table is simple; we analyze the index interferences in this type of query by taking two predicates on the same running query each one is based on an index; then compare between obtained results; the following query is as a running example:

UPDATE CLIENTS SET name = 'ALI' WHERE noclient = 30;

Following the partitioning methods, if three processors try to send requests to the current table, three cases are possible. The first is when the range of the clause where belongs to one partition for example, the last will be locked by the first request; but others partitions remain unlocked, and accessible for other processors. Secondly, if the clause where of our update query is extended to two ranges (partitions), for example the partition [0, 40] and the partition] 80, 1000], these ranges will be locked, but the range] 40, 80] remains unlocked; then it behaves like two tables. Nevertheless, if the clause where is based on three ranges, it proceeds like three tables. The following query illustrates this case treatment:

UPDATE CLIENTS SET city = 'paris' WHERE noclient = 45 or noclient = 10 or noclient = 120; Taking x in milliseconds (ms) the execution time cost; the table CLIENTS behavior seems heavy, but when we partition it as described in this section, the execution time becomes: x/3 (ms) + interference (ms); with an interference $\approx$ x/9 and $x \in \mathbb{R}+$. Due to the parallelism, the RDBMS can handle more than one request in the same time, following the number of processors that we have.

### 4.4 Join Update Query on Local and Global Index

Join Update Query (JUQ) throws many problems, as concurrent access when the query is based on more than one partition. We take the following query as testing example:

UPDATE CLIENTS SET name = 'thomas' WHERE CLIENTS.noclient IN (select noclient from CMD where CLIENTS.noclient = CMD.noclient);

We create an index LPI on CLIENTS and an index GPI on CMD. Local parallel index allows the same partitioning rules as the CLIENTS table; then LPI partition the table CLIENTS. However, global parallel index did not partition the underlying table; then the table CMD remains non-partitioned. The execution time is described in Figure 3.

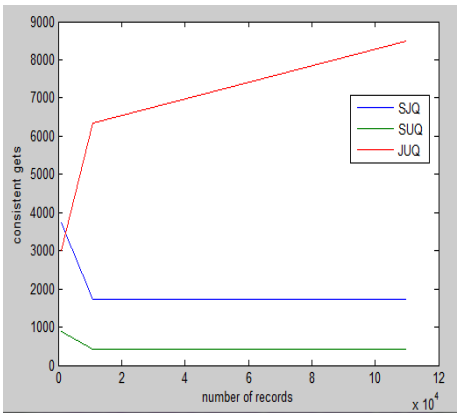


Fig. 3. Number of consistent gets for different query interferences between LPI and GPI

## 5. INTERFERENCES BETWEEN COMPOSITE PARALLEL INDEX AND OTHER PARALLEL INDEX ON COMPLEX QUERIES

### 5.1 Interference between composite parallel index and local parallel index

This type of query interference is rarely used, the execution time following the number of operations have to be performed by the RDBMS in the clause where:

UPDATE CLIENTS SET nom = 'simon' where ville = 'paris' AND pays= 'france';

The execution time is described in Figure 4.

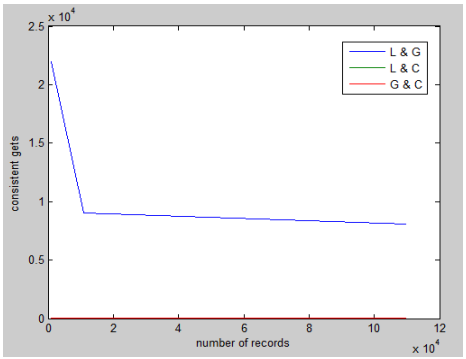


Fig. 4. Number of consistent gets of different index interferences

### 5.2 Interference between global parallel index and composite index:

This type of query is slightly close to the last cited interference on term of syntax, but different on term of execution time, following the number of operations have to be performed by the RDBMS in the clause where.

UPDATE CMD SET nocmd = 1234567890 WHERE datecmd = to_date('08-10-2013', 'dd-mm-yyyy') AND etatcmd = 'C';

The execution time is described in Figure 5.

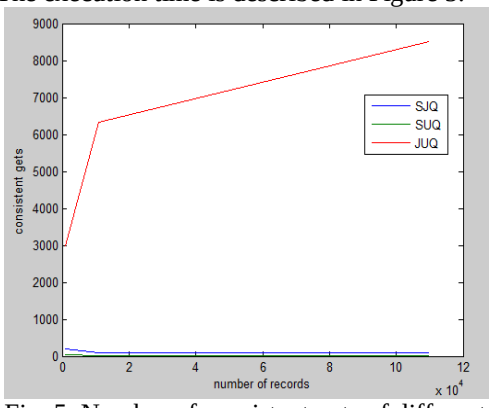


Fig. 5. Number of consistent gets of different types of queries between GPI and CPI

### 5.3 Interference between complex aggregate queries with LPI and with GPI

Complex aggregate query is a query with several query blocks (views or sub-queries) correlated together with a multiple joins[14].

To compare the performance of complex aggregate queries between LPI, GPI and CPI, we use the following two views attended by one query with our running example:

CREATE VIEW V1_AGG as SELECT noline, average = AVG (amount)
FROM LINECMD GROUP BY noline;

CREATE VIEW V2_AGG AS SELECT noline, maximum = MAX (average)
FROM V1_AGG GROUP BY noline;

SELECT V1_AGG.noline, V2_AGG.maximum
FROM V1_AGG, V1_AGG
WHERE V1_AGG.noline = V2_AGG.noline AND V2_1GG.maximum = V1_AGG.average;

The results obtained using this query with our running examples are presented in the Figure 8.

## 6. INTERFERENCE BETWEEN BITMAP LOCAL PARALLEL INDEX AND BITMAP GLOBAL PARALLEL INDEX

Bitmap index is the bit masks for distinct values of indexed columns: the binary AND and OR can make the equality tests.

For bitmap index, we do not prefer the global index because it cannot partition the table, but we use the bitmap local parallel index. We take the following as running query:

SELECT name FROM CLIENTS WHERE city = ‘paris’ AND country = ‘france’;

The column city is indexed by bitmap local parallel index; however, the column country is indexed by bitmap global parallel index. The local index has partition the underlying table, but global bitmap index did not partition it, then the local bitmap parallel index is most suitable in this case.

### 6.1 Sub-Query and Index Partitioning

Sub-query is a select query embedded in a clause of another SQL statement. We distinguish between two statements, the outer query or outer statement and the inner query or inner statement. Then we say that the sub-query is nested within the outer query; there are two strategies to executing sub-queries: serial and parallel execution scheduling strategies. When a sub-query is being processed, parallelization techniques must be applied[15]. One of the highlights of sub-queries is via query parallelization and data partitioning, a query can be segmented to multiple sub-queries; each of them, which contains joins on partitioned data sets and pre-aggregation. The final results are obtained by applying a final aggregation over the results of sub-queries [16].

[17] presents four other optimization methods for indexes that we use its interferences with sub-queries in this paper. Continuous index tuning, periodic index tuning, triggered index tuning and hybrid index tuning. The sub-query reconstruction mechanism consists of reconstructing the set of original queries before dispatching them to the data sources and computing the answers to the original queries based on answers to the

reconstructed queries [16].

### 6.2 Sub-query and global parallel index

In this section, we take the same global index used above with the following sub-query:

SELECT datecmd FROM CMD WHERE noclient IN (SELECT noclient FROM CLIENTS WHERE name = 'alain');

The results obtained are showed in Figure 6.

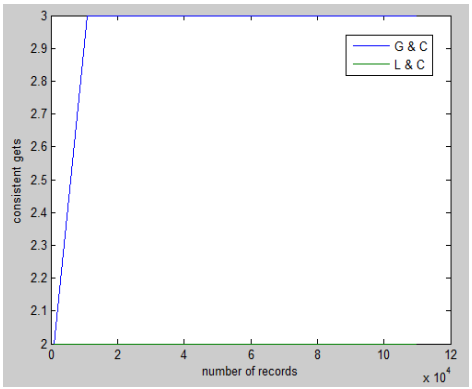


Fig.6. Number of consistent gets of different index interferences

### 6.3 Sub-query and local index

In this case, we use the same sub-query used in section A, with our local parallel index; then we get the results of Figure 7.

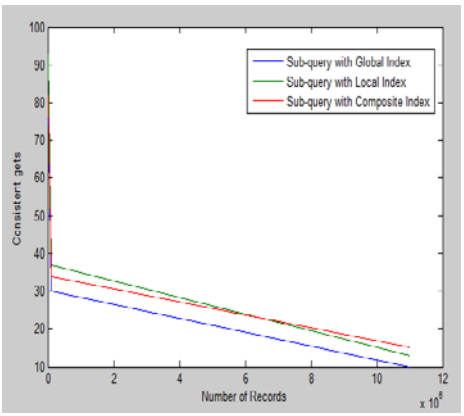


Fig.7. Number of consistent gets in sub-query with different indexes

### 6.4 Sub-query and composite index

With the last sub-query used in section A, indexed by composite parallel index, we get the results of Figure 8.

### 6.5 Complex sub-query with many types of index

Tables can reference the same table under a different correlation names. Adding two attributes to our table CLIENTS and considering a query to find all the clients who are younger than the oldest client of their gender[18]:

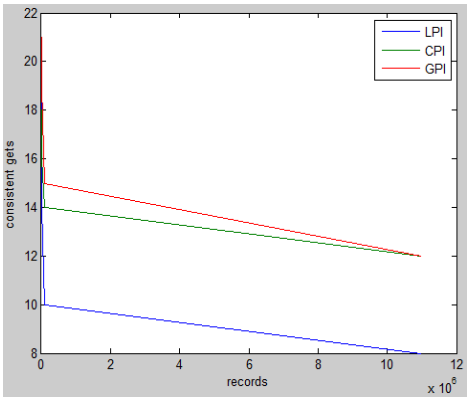


Fig.8. Number of consistent gets in complex aggregate queries with different indexes

SELECT S1.noclient, S1.name, S1.sex, S1. age
FROM CLIENTS AS S1 WHERE age < (SELECT MAX(age)
FROM CLIENTS AS S2 WHERE S1.sex = S2.sex);

We can also demonstrate the efficiency of inline views with the following sub-query:

SELECT C.noclient, M.name, CMD.datecmd FROM CLIENTS C, (SELECT LC.amount, CMD.noclient, MCD.nocmd FROM CMD, (SELECT LC.nocmd, amount, quantity FROM linecmd) LC WHERE CMD.nocmd = LC.nocmd) M WHERE M. noclient = G. noclient;

The results of two last queries are described in Figure 9 and Figure 10.

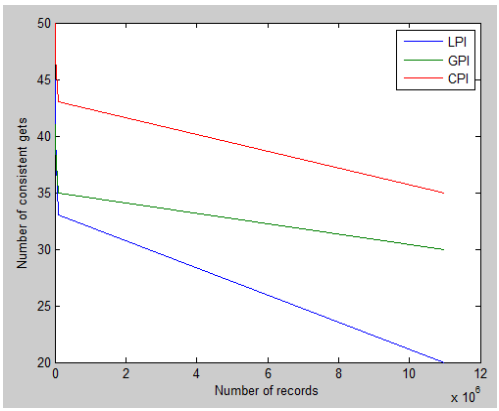


Fig.9. Number of consistent gets in complex sub-queries with different indexes

## 7. RESULTS AND ANALYSIS

Our experiments are performed on Intel (R) Pentium(R) Dual CPU T3200 @2.00 GHz machine with 3 GB of main memory running a Windows 7 Integral Edition operating system. All queries are performed on an oracle 11g release 2.

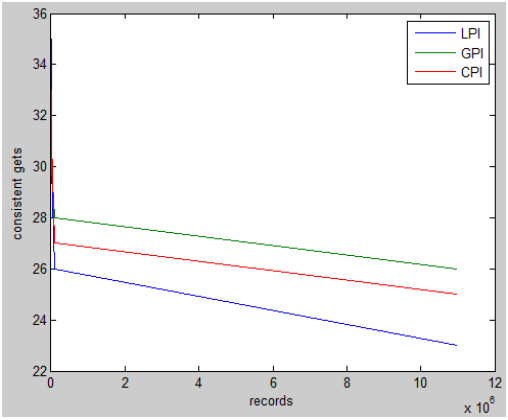


Fig.10. Number of consistent gets in complex sub-queries written on inline views with different indexes

The total cost of a parallel synchronous query program is the cumulative sum of the costs of its super steps and the cost of each superstep is the sum of the following three quantities, the maximum of the computations performed by each processor (CPU cost), the maximum of the messages sent/received by each processor (I/O CPU)[10]. Then we can minimize the cost of a query by minimizing the CPU cost and the number of I/O disk, these costs are the subject of this paper. Analyzing the results obtained, we can get the following explanations:

- SJQ: Following the Figure 3, the interference between local and global index, the number of input output decreases when the number of records increases, once the number of records on the underlying tables outgrew, the number of input output becomes stable and less than 2000.
- SUQ: Following the Figure 3, initially the number of consistent gets was around 2000; then it starts to decrease slightly and then stabilizes when the number of data in the underlying tables becomes too big. SUQ consumes less input output than SJQ.
- L & G: Following the Figure 4, this interference causes many inputs outputs bloc.
- L & C: Following the Figure 4, interference between local and composite allows many and stable input output gets.
- G & C: Following the Figure 6, interference between global and composite index is slightly close to the L & C.
- JUQ: Following Figure 3 and Figure 5, interference between local and global index, join update query increases the number of input-output gets with the increase of the data of the underlying tables.
- Sub-query and different indexes: following the Figure 7, for the small table, the global index and the composite index are the most suitable with sub-query; however when the underlying table is so big, the most suitable index is local parallel index followed by global index with sub-query.
- Complex query and complex sub-query, with different indexes: the results obtained in Figures 9, and Figure 10, by executing complex query, and complex sub-query successively, demonstrate that all time the LPI causes less consistent gets followed by CPI, itself followed by GPI.

### 7.1 Comparative analysis:

As there are different kinds of interferences between different indexes, it is important to analyze the efficiency of each case of interference scheme discussed above.

When it is a join query, is not recommended to use both of local parallel index and global parallel index, each one on a table; but in this case it is advisable to use either a global parallel index or a local parallel index and not both of them.

In the case of a single table, we can use either local parallel index, or global parallel index, or both of them following the type of query; but the number of consistent gets remains close for each case.

The use of global or local parallel index on join query with composite index is useful, according to the schemes 4 and 5. Generally, the use of different indexes is useful when the number of data in tables is big.

We can explain the efficiency of LPI by the consistency, between the partitioning of attribute, and the underlying table partitioning; the same reasoning of CPI, thing is not realized for GPI. In many past researches, GPI was the most whished index, but these researches partition just the index, but not both of index and underling table. Finally, we are in favor on our proposed partitioning method for the LPI as the most efficient and optimized index, followed by CPI and finally GPI.

Using sub-query also prefers the local parallel index as best optimization result.

## 8. CONCLUSION AND FUTURE WORK

On the first part of this paper, we introduce different parallel indexes, and a taxonomy of various parallel indexes, and their interferences.

Interferences between different parallel indexes mean that we can use more than one type of parallel index in the same query. Different queries are made in the current paper to conclude the most desirable combinations of indexes.Following different queries and analysis, we can conclude that the most useful interference is the local parallel index with the composite

parallel index, and the global parallel index with the composite parallel index. These interferences between different indexes and data structures, allows us to propose the optimized model for the use of both multiple indexes.

The second part of this paper studies the uses of sub-queries with different indexes used in the first part, the results obtained are in favor of the uses of the sub-queries with the LPI. For our future works, we plan to incorporate these methods in the backgrounds of a noncommercial RDBMS like PostgreSQL.

## 9. REFERENCES


1. Jingren Zhou, N.B., Ming-ChuanWu, Per-Ake Larson, Ronnie Chaiken, Darren Shakib, *SCOPE: parallel databases meet MapReduce.* Springer, 2012.
2. Xiaoqing Niu , X.J., Jing Han, Haihong E, and Xiaosu Zhan, *A Cache-Sensitive Hash Indexing Structure for Main Memory Database.* Springer, 2013.
3. Ihm, S.-Y., *A partitioned layer-based index for efficient processing top-k queries.* ELSEVIER, 2014.
4. Rimma V. Nehme , K.W., Chuan Lei, Elke A. Rundensteiner, Elisa Bertino, *Multi-route query processing and optimization.* Journal of Computer and System Sciences, 2013.
5. Codd, E.F., *A relational model of data for large shared data banks,*. 1970.
6. Wiet Mazairac, J.B., *BIMQL – An open query language for building information models.* Advanced Engineering Informatics, 2013.
7. Taniar, D. and J. Wenny Rahayu, *Global parallel index for multi-processors database systems.* Information Sciences, 2004. **165**(1-2): p. 103-127.
8. Mohamed CHAKRAOUI, A.E.K., Naoual MOUHNI, *Local Parallel Index in Databases System.* The 4th International Conference on Multimedia computing and systems (ICMCS'14), 2014.
9. Navarro, L., *Optimisation des Bases de Donnees.* Pearson, 2010.
10. Veronica Gil-Costa, M.M., Nora Reyes, *Parallel query processing on distributed clustering indexes.* Journal of Discrete Algorithms, 2009.
11. Lilian Hobbs, S.H., Shilpa Lawande, Pete Smith, *Oracle_10g_Data_Warehousing.* Elsevier Digital Press, 2005.
12. Sun X , Y.W., Koenig S, *Efficient incremental search for moving target search.* IJCAI International Joint Conference on Artificial Intelligence, 2009.
13. Imad Afyouni, C.R., Sergio Ilarri, Christophe Claramunt, *A PostgreSQL extension for continuous path and range queries in indoor mobile environments.* Pervasive and Mobile Computing, 2013.
14. Damianos Chatziantoniou, K.A.R., *Partitioned optimization of complex queries.* Information Systems, 2005.
15. David Taniar, C.H.C.L., *Query execution scheduling in parallel object-oriented databases.* Information and Software Technology, 1999.
16. Feng Chen, R.L., Xiaodong Zhang, *Essential Roles of Exploiting Internal Parallelism of Flash Memory based Solid State Drives in High-Speed Data Processing.* IEEE, 2011.
17. Karen Works, E.A.R., Emmanuel Agu, *Optimizing adaptive multi-route query processing via time-partitioned indices.* Journal of Computer and System Sciences, 2013.
18. Celko, J., *Advanced SQL Programming.* ScienceDirect, 2011.


*International Journal of GEOMATE, Aug., 2016, Vol. 11, Issue 24, pp. 2267-2274*
MS No. 5127j received on August 21, 2015 and reviewed under GEOMATE publication policies.

**Corresponding Author: Mohamed Chakraoui**