An Optimized Vertical Fragmentation Approach

Hichame Chaalel, Hafida Belbachir

Abstract- Vertical fragmentation in databases is considered as a difficult problem; it has attracted the interest of many researchers and has been the subject of several studies. In the literature, these studies suggest approaches to solving the problem of vertical fragmentation, these approaches always provide a solution, but we find no indication about the relevance of solutions, nor any clue about their qualities.

In this study we propose an algorithm that seems be best suited to the problem of vertical fragmentation and especially gives a best solution. To validate our approach we compared our solution to two existing algorithms based on two early studies (Genetic algorithm & Apriori algorithm).

Index Terms: Genetic Algorithm, Data mining, Physical Design, Vertical fragmentation.

I. INTRODUCTION

The vertical fragmentation is a technique that allows the users to change the physical design of database; it allows the user to fragment a table in a database into a set of fragments having for objective to improve the performances of database. The primary key of the table T is replicated in every fragment in order to reconstruct the original table using the joint operation on different fragments. Vertical fragments are obtained using the projection operation of relational algebra.

The vertical fragmentation is NP complete problem (Hammer & Niamir [6]); this is due to the huge number of possible alternatives to resolve the problem. For example with a table composed of m attributes, the number of alternatives obtained is calculated by the formula (1) (the Bell number):

$$B(\mathrm{m}+1)=\sum_{k=0}^{n} C_n^k B_k \quad (1)$$

For large values of m the number is calculated by formula (2):

$$(m >> 0) = m^m \quad (2)$$

The following values allow us to have an idea about the vastness of the Bell number where:

m = 5 ; B(m) ≈ 52.
m = 10 ; B(m) ≈ 115 975.
*m = 15 ; B(m) ≈ 1,38*109.*
*m = 22 ; B(m) ≈ 4,50*1015.*
*m = 50 ; B(m) ≈ 50*1048.*

Our major contribution is summarized in the proposal of a new developed approach for the purpose of providing not only a solution but an optimized solution for the problem of vertical fragmentation, to validate this work we tested our algorithm on TPCH benchmark comparing it to two other variants Genetic algorithm and Apriori algorithm inspired by previous work.

This paper is composed of three sections, in the first part, previous works in the domain are exposed with a more detailed explanation of the algorithms Genetic and Apriori, in the second section we will explain and develop the algorithm proposed in this study “Affiner”, and the last section is devoted to present different results.

II. PREVIOUS WORK

Vertical fragmentation on databases has been the subject of several studies; these studies converge towards a single goal: Optimizing the performance of the database. Approaches and orientations of researchers differ, and can be classified into three categories:

Algorithms based on the affinity: This class of algorithms calculates the affinity between attributes and regroups the most affine attributes in the same fragment; several authors have studied this approach Hoffer & al [7], Navathe et al [8], Navathe et Ra [9].

Not that the approach “Affine” proposed in this study can be classified in this category of algorithms.

Algorithms based on data mining: these are algorithms that generate attribute groups based on the methods of data mining; we can cite the work of Cheng & al [3], Gorla [4][4], and Song & Gorla [10].

Algorithms based on the cost model: these algorithms evaluate each fragmentation scheme with a cost model formula that estimates the response time of queries and applications running on the database. Genetic algorithms use the cost model as a fitness function. Among the works based on genetic algorithms: Hammer & Niamir [6], Song & Gorla [10], Angel & Zevala [2].

A. Genetic Algorithm:

Genetic algorithms are meta-heuristic algorithms that solve combinatorial problems. Many studies are based on GA to optimize the physical design using the technique of vertical fragmentation.

The basic principle of the algorithm is as follows:

- In the first step, algorithm begins randomly by generating initial solutions (population); each element (individual) in the population is subject to evaluation by the fitness function to measure the quality of the solution provided by the fitness function. In our case the fitness function is the time estimated of execution of queries, in other words, the cost of execution of the workload on database.
- After the operation of evaluation, comes the operation of selection of individuals in order to reproduce the new population,

Manuscript received on September, 2013.

Hichame Chaalel, Department of Computer Science, Laboratory Systems Signals Data, Faculty of Science. University of Science and Technology Mohamed Boudiaf-Oran, Algeria.

Pr.Hafida Belbachir, Department of Computer Science, Laboratory Systems Signals Data, Faculty of Science. University of Science and Technology Mohamed Boudiaf-Oran, Algeria.

according to Mendel's survival rule, the fittest element provides a strong offspring.

- Follows the reproduction operation on the basis of pre-selected individuals using the operations of mutation and crossover that allow us to obtain a new generation that will, in principle, get a fitness score greater than or equal to the fitness of the previous generation.

In general, at the end of the execution of the algorithm, we obtain an improved or better suited solution to the problem initially posed.

Adaptation of Genetic Algorithm to the Vertical Fragmentation:

In this section, we explain the adaptation of the genetic algorithm for vertical fragmentation.

The encoding: The encoding used is fairly simple:

- Individual represents the way that the table will be fragmented vertically, in other words, it is a digital representation of the fragmentation.
- The chromosome represents an individual and is a collection of integers. The length of the chromosome is equal to the number of attributes in the table, each integer expresses the partition to which the attribute will be assigned, and a population is the set of individuals in a generation.

Example: suppose that R is the table with 10 attributes where: R = {A1, A2, A3 …, A10}, and the encoding according to the table: R = {5, 0, 5, 5, 0, 1, 1, 2, 3, 4} schematized in figure 1, it can be interpreted as follows:

- *Partition_0* : {1, 4} ;
- *Partition_1* : {5, 6} ;
- *Partition_2* : {7} ;
- *Partition_3* : {8} ;
- *Partition_4* : {9} ;
- *Partition_5* : {0, 2, 3}.

Population: The initialization of the population occurs in a random way. Population size can be adjusted by the user, and the number of generations can also be specified by the user too.

Crossover: A point is randomly chosen in the first chromosome, with a certain probability, it will be active for crossing with a probability Pct (specified in advance by the user). Another crossover point is randomly chosen on the second chromosome using the same process.

Mutation: it performs a scan of each gene on chromosomes candidate in the population selected by the method of selection, and mutates it with a mutation probability (specified in advance by the user); each gene of the chromosome is a subject to a mutation probability Pm, if the gene is selected, it will randomly change the value.

Selection: in the literature there are several algorithms proposed for the selection of individuals which we cite: roulette-wheel selection, elitist selection, etc..; we have opted to use the standard selection mode that allows us to select some of the chromosomes of the population to enable them to continue surviving.

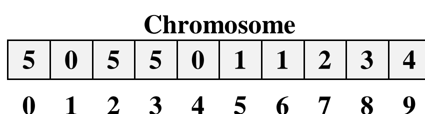

Fig. 1. : Individual Encoding.

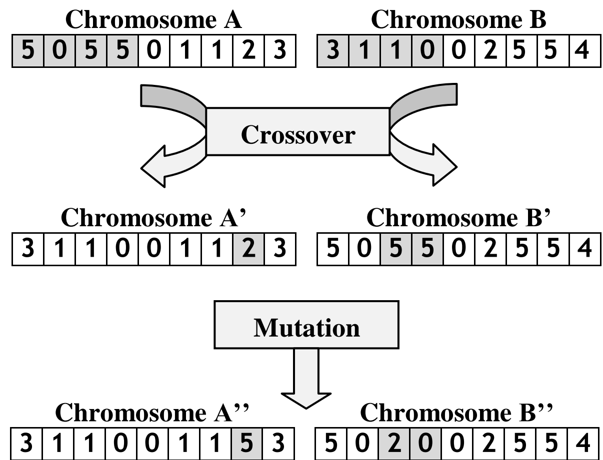

Fig. 2. : Crossover and mutation

This selection should be guided by the values of fitness, but the fitness function will be seen as a statistical probability of survival, not as the only determining factor for survival. In other words, the chromosomes with higher fitness values will be more likely selected than those with low fitness values, this rule is not always guaranteed, and individuals with low fitness can be selected but with a low probability.

Illustration: Figure 3 schematizes the process of crossover and mutation to both chromosomes A and B.

The fitness function: the fitness evaluation is based on the cost model; the cost model is used to estimate the time response of queries and applications that require interaction with the database.

The cost (fitness) query being estimated is based on two operations:

- The amount of data transferred between the processing kernel (CPU) and the data storage place.
- Time of data processing in CPU, but in our case processing time CPU is neglected compared to the time of data transfer because it is quite low.

Inspired from the work of Gorla & al [5], in this study the index access cost and the storage cost are not taken into account, the blocks estimate used is given in Yao (1977) [12]as the basis for cost of query processing. If there are n records divided into m blocks and if k records satisfying a query are distributed uniformly among the m blocks, then the number of blocks accessed by the query are equal to:

$m\,(1 - (1 - 1/m)^k\,)$ (Yao, 1977) [12].

In the partitioned database, we apply this formula for each Segment i accessed by the query q. Thus, the cost of processing a (retrieval) query q is estimated as shown in formula 1, where the first part computes the total number of blocks to be accessed from each of the Segment *q_j* partitions. The second part indicates the overhead to concatenate Segment *q_j* partitions in the memory. Where:

Freq_i = frequency of query q_i

Segment qj = Number of partition j required by the query q.

$$M_i = \left[T \times L_i / Bs \right]$$

K_q *: number of tuples satisfying a query q.*

T : Total number of tuples.

Li : size of partition.
Bs : block size .

$$\text{Cost_Query} = \text{freq}_q \times \left(\sum_{i=1}^{segment_{qj}} \left[M_i \left(1 - \left(1 - \frac{1}{M_i} \right)^{Kq} \right) \right] \right) \times \left(1 + \left(0.1 \times \left(segment_{qj} - 1 \right) \right) \right) \quad (3)$$

Apriori Algorithm:

Data mining is considered as a discipline related to the domain of decision making, it is a process that allows us to extract knowledge's and valid and exploitable information's from divert data sources. Such information can be used as the basis for decisions about marketing activities.

The following terminology will allow us to understand the process of knowledge extraction:

The items describe all articles or components of the studied subject, for example in the case of list shopping cart the items means all items available in the store (bread, milk, coffee, and bonbon).

The association rule is an association between several articles to discover from a set of transactions, a set of rules that expresses a possible association between different items, meaning regularities between products, for example in supermarket we can deduce that if a customer buys milk and coffee together, he or she is likely to also buy sugar.

An association rule is a rule of the form: If conditions then result, For example, a rule will be three items of the form:

If X and Y then Z: (X and Y) → Z; (4)

Rule expressing: If the Article X and Y appear simultaneously in a purchase then Article Z will appear.

To select an association rule, we need to define the numerical threshold that will be used to calculate the benefit of such a rule.

The *support* of a rule indicate the rate of records that satisfy the rule, it indicate the number of times, in rate terms, where the rule is applicable.

Consider the set of shopping carts T, R is a group of products and U is the set of shopping carts containing the subgroup R, we have:

$$Support(R) = \left(\frac{|U|}{|T|} \right) \times 100\% \quad (5)$$

Not that |U| and |T| denote respectively the number of elements of U and T.

Confidence is the ratio between the number of transactions where all the items contained in the rule appear, and the number of transactions in which items appear in the condition part.

Consider the association rule $R = S1 \rightarrow S2$, where S1 and S2 represent a list of product.

$$Confiance(S1) = \frac{Support(S1, S2)}{Support(S1)} \times 100\% \quad (6)$$

A rule is defined as solid if here support is greater than or equal to a fixed support, and here minimal confidence is greater than a predetermined confidence.

Apriori is a variant of the most interesting data mining algorithms and has been the subject of our study is inspired from the work of Gorla [4].

What interested us in this approach (Apriori) is the reduced complexity of the operations achieving in general good results.

Apriori is the result of work done by Agrawal & Srikant [1] is an algorithm for finding and extracting the association rules.

The Apriori algorithm is based on the principle that any set of non-frequent subset is infrequent implying a reduction of the search space.

The Apriori algorithm process:

Generate the itemsets,

1. Calculate the frequency of the itemsets,
2. Retain the frequent itemsets having a support greater than or equal to the minimum support predefined.
3. Generate from frequent itemsets robust association rules having sufficient confidence.

Adaptation of the Apriori Algorithm:

In summary, the work of Agrawal & Srikant [1] treat the adaptation of algorithm Apriori to generate vertical fragments follows this reasoning:

Let A and B be two attributes in a relation, we calculate the confidence of the rule A→B, if it is greater than the predefined minimum (min_conf), we have a strong argument to group A and B in the same partition.

We note that the order of tow attributes in the same partition is not significant, to calculate the confidence between A and B we calculate the values of confidences for A→B and for B→A and the lowest value is chosen to represent the association between attributes A and B, thus :

Confidence (A, B) = freq (A, B)/ max (freq (A), freq (B)). (7)

The proposed algorithm follows these steps:

• discover large itemsets: Large itemsets represent the combinations of attributes, in this step we generate all possible association rules and we calculate the corresponding confidence that satisfy the condition support greater the predefined minimum support.

• Filter large itemsets: we eliminate large itemsets having value of confidence less than predetermined minimum confidence (min_conf); this will significantly reduce the search space.

• Finally, produce the partitions, and choose the best fragmentation pattern, if we obtain several solution we calculate the cost of itch model, by using the cost model formula inspired from the last work (Genetic Algorithm), then the best one is selected.

For more details you can refer to the work of Gorla [4].

III. AFFINER ALGORITHM:

The main idea of Affiner is based on the analysis and evaluation of the affinity between each attribute and the combination of high affinity attributes, thereby algorithm Affiner can be classified in the algorithms based on affinity class.

One of the major advantages of this algorithm is that it performs a search on all attributes and deduces partitions without using a cost model, this will reduce processing time, and it is not easy to find the real estimating cost formula cause of the large difference between

architecture of DBMS's, from one version to another, and from one architecture to another, this affect the formula of the cost model.

In fact, after the analysis of each attribute separately, only the most interesting partitions will be retained for the composition of the final fragmentation pattern. Attributes will be compared one by one, relative to all the other attributes, taking into account the workload.

The affinity score calculation will be based relative to each request. Attributes with higher affinity will be grouped in the same set, each set will present a partition, this operation will be repeated until all the attributes are evaluated. The retained attributes will be removed from the set of potential solutions.

If an attribute has an acceptable affinity for all the attributes composing a given partition, it will be merged with this partition.

Affiner progress: The Affiner algorithm is stated as follows:

1. Generating the attribute usage matrix AUM,
2. Generating the attribute usage frequencies matrix,
3. calculation of the affinity matrix,
4. sorting the affinity matrix in descending order according to their affinity values,
5. Deduce the fragmentation pattern starting from the ordered set.

Generating the attribute usage matrix AUM:

For a given table, the attribute usage matrix AUM is a matrix that maps the relationship between queries and attributes in this matrix if the query requires the attribute, the corresponding box will be assigned to "1", otherwise "0".

$$AUM(q_j, A_i) = \begin{cases} 1 & \text{if } q_j \text{ uses } A_i \\ 0 & \text{otherwise.} \end{cases} \quad (8)$$

Example: Let T1 a table composed of attributes {*A1, A2, A3, A4, A5, A6*}, and this queries requests the following attributes, and the frequencies of all queries equals to 1.

q1 = {*A7*}
q2 = {*A1, A8*}
q3 = {*A2, A3, A4*}
q4 = {*A2*}
q5 = {*A1, A2, A3*}
q6 = {*A2, A3, A6, A8*}

The attribute usage matrix is presented in Table 1.

Generating the attribute usage frequency matrix:

It is a simple step where we calculate a new matrix AUFM based on the AUM, each element of the AUM is multiplied by the frequency of concerned request. $A'_{ij} = \text{freq}_i \times A_{ij}$

Where:

- *Aij*: element line i , column j, in AUM;
- *A'ij*: element line i, column j, in AUFM;
- *ferq_i*: frequencies of the query i.

	A1	*A2*	*A3*	*A4*	*A5*	*A6*	*A7*	*A8*
q1	0	0	0	0	0	0	1	0
q2	1	0	0	0	0	0	0	1
q3	0	1	1	1	0	0	0	0
q4	0	1	0	0	0	0	0	0
q5	1	1	0	1	0	0	0	0
q6	0	1	1	0	0	1	0	1

Table 1 : Attribute Usage Matrix

We suppose that for the previous example the frequencies of all queries are equal to 1, therefore the AUM will be similar to the AUFM.

Affinity Calculating: In our study we have observed that the affinity between attributes is mainly governed by the number of queries using the attributes. If two attributes are invoked by the same queries it would be beneficial to put them in the same partition, otherwise it is more interesting to put them in separate partitions.

Based on these assumptions, we propose the following formula to calculate the affinity between two attributes:

Ai: *column number i of the AUFM (attribute usage matrix), representing an attribute,*

Aik: *element of the AUFM, column number (i) and line number (k).*

t: *total number of queries.*

$$\text{Affinity}(A_i, A_j) = \frac{\text{nbr of commun invocation of } A_i \ \& \ A_j}{\text{nbr of total invocation of } A_i + \text{nbr of total invocation of } A_j}$$

$$= \frac{2 \times \sum_{k=0}^{t} (A_{ik} \times A_{jk})}{\sum_{k=0}^{t} A_{ik} + \sum_{k}^{t} A_{jk}} \quad (9)$$

The affinity formula reveals the following properties:

- The formula is symmetric :

  Affinity (A_i ; A_j) = Affinity(A_j ; A_i)
- $0 \le Affinity(A_i, A_j) \le 1$;
- *if* $Affinity(A_i, A_j) = 1$ then all queries that invoke Ai are the same with all the queries that invoke Aj; we deduce that each attribute is completely affine with himself, as follows: $Affinity(A_i, A_i) = 1$;
- *if* $Affinity(A_i, A_j) = 0.5$ then half of queries that invoke Ai also invokes half of queries that invoke Aj;
- *if* $Affinity(A_i, A_j) = 0$ All queries that invoke Ai do not make any appeal to Aj.

Table 2 shows an example of affinity matrix calculated on the basis of the AUFM in Table 1.

	A1	*A2*	*A3*	*A4*	*A5*	*A6*	*A7*	*A8*
A1	1	2/6	0	2/4	0	0	0	2/4
A2		1	4/6	4/6	0	2/5	0	2/6
A3			1	2/4	0	2/3	0	2/4
A4				1	0	0	0	0
A5					1	0	0	0
A6						1	0	2/3
A7							1	0
A8								1

Table 2 : Affinity Matrix

Example:

$$Affinity(A_3, A_4) = \frac{2 \times ((0\times0)+(0\times0)+(1\times1)+(0\times1)+(1\times0))}{((0+0+1+0+0+1)+(0+0+1+0+1+0))}$$

$$Affinity(A_3, A_4) = \frac{2\times1}{(2+2)} = \frac{1}{2}$$

$$Affinity(A_2, A_3) = \frac{2 \times ((0\times0)+(0\times0)+(1\times1)+(1\times0)+(1\times1))}{((0+0+1+1+1+1)+(0+0+1+0+0+1))}$$

$$Affinity(A_3, A_4) = \frac{2\times2}{(4+2)} = \frac{2}{3}$$

We also calculate the affinity matrix, given the symmetry character of the affinity formula affinity; we obtain a symmetric matrix of values between 1 and 0.

Sorting the affinity matrix: the step of sorting the matrix consists in sorting pairs of attributes affinities in descending order according to their affinity, we named this list Ordre_Affinity, sorting will help us for generating partitions based on the most affine attributes.

From the whole list Ordre_Affinity it will be retained attributes with affinities greater than or equal to 0.5 (Table 2, the gray box's), which means that the value attributes are referenced by at least half of the same applications, the reason we choose to wear this condition (affinity ≥ 0.5), is the need to find a compromise between too many partitions so many joins between partitions (for rebuilding the table) which is quite expensive, and the scan of all attributes that a majority can be useless for the query, which will unnecessarily occupy resources, and we note that a partition schema that satisfies few queries can degrade performance for other queries.

For the classification of attributes with a same affinity we calculate the number of calls in the attribute usage matrix AUFM($\sum_{k=0}^{t} A_{ik}$), so we obtain an ordered set of affinities which will allow us the classification of attributes and the generation of partitions.

Illustration: Continuing with the same example we obtain the following set Ordre_Affinity:

Ordre_Affinity = {*(A3; A2), (A2; A4), (A3; A6), (A6 ;A8), (A3; A4), (A3; A8), (A1; A4), (A1; A8)*}.

Partition Generation: the process of generating partitions (set of attributes) is formulated as follows, the algorithm performs an analysis of each element (Ai, Aj) across Ordre_Affinity starting with descending for each element Ordre_Affinity (Ai, Aj) algorithm consider one of the following three cases:

If both attributes do not belong to any partition (set of attributes), then the new partition will be generated combining the two attributes:

Px = {Ai, Aj};

a) If an attribute does not belong to any partition and the other belongs to a partition then let: $\{A_i \in \phi \,\&\, A_j \in P_y\}$ where *Py* denotes a partition; then we explore the possibility of assigning *Ai* at *Py*, in this case, *Ai* must satisfy an acceptable affinity (≥0.5) with all the elements of *Py*. The formula: $\{\forall A_i \in P_z \Rightarrow Affinity(A_i, A_x) \geq 0.5\}$, otherwise *Ai* will not be assigned to any partition;

b) Finally If both attributes belong to two distinct partitions: for example for the element *(Ai, Aj)* if $(A_i \in P_x)$ and $(A_j \in P_y)$ and *(Px ≠ Py)* we investigate the possibility of merging the two partition *Px* and *Py*, to do, affinity between the whole elements of the two partitions must be (≥0.5), else the two partitions *Px* and *Py* will stay disjointed.

The algorithm:

- *P_0 :initial Table without partition;*
- *P_k: Partition number k;*
- *Ai : Attribute number i ;*
- *Affinity (A_x, A_y)* : affinity value between Ax and Ay;

Begin:

- *Var k = 0 ;*
- *For each element(A_x,A_y)* $(A_x, A_y) \in \{Ordre_Affinity\}$ *do*

$$if\ ((A_x \in P_0)\ and\ (A_y \in P_0))\ then \begin{cases} P_k \leftarrow (A_x, A_y);\ \text{new partition} \\ P_0 \leftarrow P_0 - (A_x, A_y); \\ K \leftarrow K+1; \end{cases}$$

$$else\ if\ ((A_x \in P_0)\ and\ (A_y \in P_z))\ then \begin{cases} if\ (\forall A_i \in P_z \Rightarrow Affinity(A_i, A_x) \geq 0.5) \\ then \begin{cases} P_z \leftarrow P_z + (A_i); \\ P_0 \leftarrow P_0 - (A_i); \end{cases} \end{cases}$$

$$else\ if\ ((A_x \in P_z)\ and\ (A_y \in P_t)\ and\ (P_z \neq P_t))\ then \begin{cases} if \begin{pmatrix} (\forall A_i \in P_z \Rightarrow Affinity(A_i, A_x) \geq 0.5) \\ \&(\forall A_z \in P_t \Rightarrow Affinity(A_z, A_y) \geq 0.5) \end{pmatrix} \\ then \begin{cases} P_x \leftarrow (P_x \cup P_y); \\ P_y \leftarrow \phi; \text{delete } P_y. \end{cases} \end{cases}$$

End.

IV. IMPLMENTATION AND RESULTS:

The algorithms implemented in the developed tool that we named FRAGMAN are based on a theoretical estimation and provides the ability for fragmenting a virtual schema based on the real database scheme. To validate this study, we perform tests that we will present the different results, the results reflects the real vertical fragmentation of the database (benchmark) based on the recommendations of the algorithms.

Between several different existing benchmarks, we have opted to choose the TPCH benchmark [11], a benchmark with the ability to create database schemas and tables large enough that allow us to carry out our research. TPCH [11] is also very easy to implement, and to generate the database and the script does not require very much knowledge in the field of databases, in addition, we have noted that the TPCH benchmark [11] is a reference for testing different DBMS (Oracle, IBM,…).

For our application we generated the TPCH schema, and we chose to fragment the fact table called LineItem composed of 16 attributes containing 6,001,215 records with a database size around 1GB:

The attributes composing the LineItem table are: {L_ORDERKEY, l_partkey, l_suppkey, l_linenumber, l_quantity,
l_extendedprice,
l_discount, l_tax,

Database schema	Workload time time (Second's)	Reduction rate
Initial table (LineItem)	76,190	-
Fragmented table by Apriori Algorithm	65,870	13.54%
Fragmented table by Genetic Algorithm	52,581	30.99%
Fragmented table by Affiner Algorithm	29,702	70.298%

Table 3: Workload time execution

l_returnflag, l_linestatus, l_shipdate, l_commitdate, l_receiptdate, l_shipinstruct, l_shipmode, l_comment}.

The DBMS engine chosen to perform our tests is Oracle 11G Release2, a very stable version of DBMS, widely used in modeling OLAP business databases (On-Line Analytical Processing), the computer in which we have done tests was a Sony Vaio computer, dotted with Intel Core i5 processor and 4 GB of RAM, and hard disk of 500GB, the system is windows 7- 64 bits.

Knowing that the Oracle DBMS does not support vertical fragmentation, several choices were presented to us to simulate the vertical fragmentation: "materialized view", "composed index's", or "sub tables"; in our case we opted for "sub tables" which appears the most stable choice, and one that consumes fewer resources.

To maximize storage space, we were forced to create a new primary key based on the primary key of the initial table LineItem, and duplicate it on the generated fragments, so to allow us to reconstruct the LineItem table from different generated fragments.

Once we obtain the recommendations from the simulation using the tool developed, we proceed to the generation of sub tables representing recommendations, and then we have to rewrite all queries composing the workload so that they can be executed according to the generated fragments based on the new schema.

We have chosen 5 queries from the queries proposed in the TPCH benchmark: (Q3, Q5, Q6, Q12, Q14), with frequencies of 1 occurrence for each query (refer to the Table 5 in the appendix).

Table 3 presents the results of the query execution on a real database, based on the TPCH benchmark. The query execution time is estimated in seconds; see more details in table 5 in the appendix.

Ranked in descending order, in the Table 3, the values obtained represent respectively the execution cost which:

- The LineItem table is not partitioned,
- LineItem partitioned according to the Apriori recommendations,
- LineItem partitioned according to the Genetics recommendations,
- LineItem partitioned according to the recommendations of the Affiner Algorithm.

Based on actual results we note that the algorithm Affiner presents the best results compared with Genetic and Apriori algorithm's, the performance gain in execution time workload decreased significantly with a rate of 70 %, which is considerable.

We emphasize that the results are based on the work load aroused, which means that the test results can vary depending on the workload.

V. Conclusion

In this paper we have focused our study to test the vertical fragmentation on the centralized row-oriented DBMS, our choice was to test two approaches which are Genetic and Apriori algorithms and as contribution we proposed a third algorithm named Affiner.

This study has allowed us to simulate the application of the vertical fragmentation on the row-oriented databases and data warehouses.

The Affiner algorithm seems well respond to the resolution of the problem of Vertical Fragmentation.

As perspective this algorithm can be compared with other algorithms based on affinities and tested on distributed databases or even adapted on other models of physical design techniques, such as horizontal fragmentation.

Although the results appear to be convincing from an experimental point of view, but in practice it not the case, because the most common form known to implement vertical fragmentation on row-oriented databases is to create materialized views or sub tables representing the fragments. But this form does not meet the expectations of managers and users of databases; the reason is that the performance of a fragmented schema falls when queries are of a complex type or requires a lot of operations of joins and aggregations.

The vertical fragmentation will only have sense if we find the right and easy solution to apply on the row-oriented databases, and obtain good results on all different types of queries, as is the case for horizontal fragmentation. This has motivated us to continue in this way and contribute to provide a suitable architecture for the Vertical Fragmentation on row-oriented databases, a solution for both OLTP model (On-line Transaction Processing) and OLAP model (On-line Analytical Processing).

VI. Bibliographie :

[1] Agrawal, R and Srikant, R. "Fast algorithms for mining association rules in large databases". in 20th International VLDB, pages 487-499, Santiago, Chile, September 1994.

[2] Angel, F. & al. Taddei-Zavala "Simultaneous Vertical Fragmentation and Segment Assignment in Distributed Data Bases using Genetic Algorithms".

[3] Cheng. C.H; & Lee, W-K; Wong, K-F, "A Genetic Algorithm-Based Clustering Approach for Database Partitioning" IEEE Transactions on Systems, Man, and Cybernetics, 32(3), 2002, 215-230. 33.

[4] Gorla, N. & Pang Wing "vertical fragmentation in Databases Using Data-Mining technique", IGI Global Vol.4, Issue 3. 2008.

[5] Gorla, N. "A Methodology for Vertically Partitioning in a Multi-Relation Database Environment", JCS&T Vol.7 No. 3 October 2007.

[6] Hammer, M. & Niamir, B. "A heuristic approach to attribute partitioning. In Proceedings ACM SIGMOD Int. Conf. on Management of Data", (Boston, Mass., 1979), ACM, New York.

[7] Hoffer, J. & Severance,D. "The Uses of Cluster Analysis in Physical Database Design", Proc in 1st International Conference on VLDB, Framingham, MA, 1975.

[8] Navathe, S. & Ceri,S. & Weiderhold,G. and Dou,J. "Vertical Partitioning Algorithms for Database Design" ACM Transactions on Database Systems, Vol. 9, No. 4, 1984.

[9] Navathe, S. & Ra, M. "Vertical Partitioning for Database Design: A Graphical Algorithm". ACM SIGMOD, Portland, Juin 1989.

[10] Song, S.K. & Gorla, N., "A genetic Algorithm for Vertical Fragmentation and Access Path Selection," The Computer Journal, vol. 45, no. 1, 2000, pp 81-93.

[11] TPCH: ad-hoc, decision support benchmark. " Transaction Processing

Performance Council (TPC)” http://www.tpc.org/tpch.

[12] Yao, S. B. (1977). “Approximating block access in data-base organization. Communications of the ACM“, 20(4), 260-261.

APPENDIX

Table Lineitem	Q3	Q5	Q6	Q12	Q14	Workload time (Scd)	Reduction rate
Initial table	14,086	17,144	13,810	16,870	14,280	76,190	-
Fragmented by Apriori	2,565	17,725	6,970	15,845	29,735	65,870	13.54%
Fragmented by Genetic	5,741	8,807	14,605	17,595	14,640	52,581	30.99%
Fragmented by Affiner	0.593	7,191	2,184	12,558	7,176	29,702	70.298%

Table 4: Workload time execution details.

Queries	SQL core
Q3	**Select** l_orderkey, sum(l_extendedprice * (1 - l_discount)) as revenue, o_orderdate, o_shippriority **from** customer, orders, lineitem **where** c_mktsegment = ':1' and c_custkey = o_custkey **and** l_orderkey = o_orderkey **and** o_orderdate < date ':2' **and** l_shipdate > date ':2' **group by** l_orderkey, o_orderdate,o_shippriority **order by** revenue desc, o_orderdate;
Q5	**select** n_name, sum(l_extendedprice * (1 - l_discount)) as revenue **from** customer, orders, lineitem, supplier, nation, region **where** c_custkey = o_custkey **and** l_orderkey = o_orderkey **and** l_suppkey = s_suppkey **and** c_nationkey = s_nationkey **and** s_nationkey = n_nationkey **and** n_regionkey = r_regionkey **and** r_name = ':1' **and** o_orderdate >= date ':2' **and** o_orderdate < date ':2' + interval '1' year **group by** n_name **order by** revenue desc;
Q6	**select** sum(l_extendedprice * l_discount) as revenue **from** lineitem **where** l_shipdate >= date ':1' **and** l_shipdate < date ':1' + interval '1' year **and** l_discount between :2 - 0.01 **and** :2 + 0.01 **and** l_quantity < :3;
Q12	**Select** l_shipmode, sum(case when o_orderpriority = '1-URGENT' or o_orderpriority = '2-HIGH' then 1 else 0 end) as high_line_count, sum(case when o_orderpriority <> '1-URGENT' and o_orderpriority <> '2-HIGH' then 1 else 0 end) as low_line_count **from** orders, lineitem **where** o_orderkey = l_orderkey **and** l_shipmode in (':1', ':2') **and** l_commitdate < l_receiptdate **and** l_shipdate < l_commitdate **and** l_receiptdate >= date ':3' **and** l_receiptdate < date ':3' + interval '1' year **group by** l_shipmode **order by** l_shipmode;
Q14	**select** 100.00 * sum(case when p_type like 'PROMO%' then l_extendedprice * (1 - l_discount) else 0 end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue **from** lineitem, part **where** l_partkey = p_partkey **and** l_shipdate >= date ':1' **and** l_shipdate < date ':1' + interval '1' month;

Table 5: the workload detail.