International Journal of Applied Information Systems (IJAIS) – ISSN : 2249-0868
Foundation of Computer Science FCS, New York, USA
Volume 4– No.1, September 2012 – www.ijais.org

Mobile Join Algorithms based on Mobiles Agents for Large Scale Distributed Query Optimization

Mohammad HUSSEIN
Dept. of business information system
Lebanese university, faculty of business
Tripoli, Lebanon

ABSTRACT
In the large scale distributed environment, the query optimization presents new problems because of the data unavailability, the estimations inaccuracies and environment instability. In this paper, we address the sub-optimality of executions plans caused by these problems. We propose to extend the join algorithms based on mobile agents in order to correct the sub-optimality. This extension allows the join to change their execution site. Indeed, the mobile agent executing a join adapts to changes in characteristics of the execution environment (e.g. network bandwidth, available memory) and responds to the estimations inaccuracies (e.g. size of intermediate relations). The performance evaluation shows that the proposed algorithms improve the response time whatever the variation of estimations errors.

Keywords
Distributed data bases systems, Query optimization, mobile agents, Data integration.

1. INTRODUCTION
In database management systems, the query processing gathers all the necessaries steps to calculate the results of queries. It consists of several successive phases [32]: rewriting, optimization and execution. Optimization is the producing process that will minimize the cost function of an execution plan of a query. This function has an objective either to minimize the total time [35], or to minimize the response time [14]. The query optimizer based on a cost model [17, 19, 20, 29, 39], generates an optimal execution plan or close to optimal. The methods of classical optimization produce firstly an optimal execution plan. Then, the generated execution plans are sent to the execution engine in order to be executed. This methods suppose that the values of the parameters utilized during the generation of an optimal execution plan (e.g. size of relations, selectivity of operators, available resources), are always valid at runtime. But this assumption is often unjustified. Indeed, values of the parameters used during optimization can become invalid during the execution because of several factors:

- Absence of statistics: the statistics describing the data (e.g. size, cardinality, maximal and minimal values) necessary to estimates the costs of the execution plans can be non-existent.
- Estimation errors: the estimations of the sizes of the temporary relations and the operator costs of an execution plan can be erroneous because of the absence or the imprecision of the statistics describing the data.
- Instability of the execution environment: in a large scale distributed environment, thousands of users can submit their queries. The complexity of these queries is different and gives data arrival rates of these queries to the system is irregular. For that, the characteristics of the execution sites (e.g. memory available, load of CPU) and networks (band-width and latency) vary from one moment to another.

Because of the factors mentioned previously, the execution plans generated, during compilation, can be sub-optimal. To correct this sub-optimality, several solutions were proposed. The first solution consists to improve the quality of the statistics describing the data relied on the previous executions. This solution was used by [9, 11] to improve the estimations quality of the selectivity factors, and by [36] to estimate the correlation between the predicates. The second solution proposed by Antoshenkov et al. [3] executes, in parallel, several execution plans for the same query in order to choose the best plan among them. After at certain time, the executions of all plans are suspended except for only one which continues its execution. A third solution proposed by [27] concentrates on the multi-join distributed queries. The optimizer generates an optimal or closes to optimal execution plan after having deduced the costs of inter-sites data transfer and the cardinalities of temporary relations. These parameters are deduced by calibration. Hence, the operators of the query are executed on a subset of tuples of the operands in order to estimate the costs of inter-site data transfer and the cardinalities of temporary relations. The fourth solution consists in modifying the sub-optimal execution at runtime, this solution is called dynamic optimization (named also adaptive) [4, 5, 7, 8, 18, 23, 24, 26].

In this paper, we propose autonomous and mobile execution of every relational operator of an execution plan by mobile agents. The agent executing an operator is conscious of its environment, able to react in an autonomous and decentralized way with the evolution of the system state (e.g. workload of sites, bandwidth) and with the estimation errors. Furthermore, it can move from site to another to continue its execution.

The remainder of paper is organized in the following way: in the section 2 we describe state of the art of the main optimization dynamic method. Section 3 proposes an extension of join algorithms. This extension based on mobile agents in order to allow the join to change their execution site. The decision and change control of the execution site are made in a decentralized and autonomous manner. Section 4 presents the experimentation environment and the results of the experiments. Finally, we conclude and present the perspectives.

2. RELATED WORKS

Several dynamic optimization methods [1, 2, 6, 7, 21, 22, 34] are proposed in the literature for correcting the sub-optimal execution plans. These methods are classified in four types (i) Replacement,(ii) Scheduling[15], (iii): Re-optimization[1,2,12], and (iv) Uses of dynamic operators[23,24]. Each dynamic optimization method is able in adapting the execution plans in order to react to one or several events: (i) Estimation errors [15],(ii) Memory available[7,8,28,30], (iii): Delays in data arrival rates[1,2], and (iv) Users Preferences. The modification level of the executions plans can also be differ from a method to another. Methods propose to modify the execution plans either between executions of two operators, or after materialization of the temporary relations. Others propose to modify the execution plans during the execution of a physical operator. In the remainder of the paper, we focus only on methods that t modify the execution plans during the execution of a physical operator.

Eddy [4] is a mechanism of query processing which changes continuously the execution schedule of operators in order to adapt to the changes of the execution environment. Eddy can be considered as a router of tuples positioned between a number of data sources and a set of operators. Each operator participate in an Eddy must have one or two input queues to receive the tuples sent by Eddy and an output queue to return the results tuples to Eddy. The tuples received by an Eddy are redirected towards the operators in different orders. Thus, the scheduling of the operators is encapsulated by the dynamic routing of tuples. The routing of tuples is carried out in the following way: (i) select a tuple for the next processing. This one can be a tuple coming from a basic relation or a temporary relation, and (ii) choose an operator for the selected tuple from the valid operators, redirect this tuple to the chosen operator and to store the result in Eddy buffer. The operators valid for a tuple are determined starting from the semantic properties of the operators. The key point in Eddy is the routing of tuples. Thus, the policy of the routing of tuples must be effective and intelligent to minimize the response time of queries.

Eddy was proposed initially for a single site environment, where all operators and Eddy are carried out on the same site while the operands can be distributed. At same time, Eddy can be exploited in the processing of distributed queries. In this objective, Zhou et al. [38] are based on the Eddy in order to define architecture, named SWAP, for the processing of distributed execution plans. In SWAP, an Eddy is placed on each execution site. Thus, when a query is submitted to an execution site, this site becomes the coordinator of the query. It transforms this query into an execution plan and determines for each operator the execution site. The localizations of execution sites of the operators are unchangeable at runtime. On the other hand, the execution of the operators is dynamic. Indeed, instead of fixing a scheduling between the operators, the tuples are routed dynamically (locally or between the different execution sites) according to the selectivity of operators, of the workload of execution sites and the bandwidth.

In [24] the execution plan of a query is supervised, at runtime, and it can be replaced by a new plan in the case where we consider that the current plan is sub-optimal. The tuples processed by each used plan represent a data partitioning which is dynamically given. When an execution plan is replaced, a new data partitioning is produced. Thus, the number of partitioning of the operands is equal to the number of used execution plans. Each used execution plan, during the query execution, produces a part of the total result from the associated data partitioning. The union of the tuples produced by the various used execution plans provides only part of the total result. Thus, to calculate the final result of the query, it must also calculate the results of all the combinations of various data partitioning. This method is similar to that of Eddy [4]. But contrary to Eddy which uses a local decision routing, this method is based on more total information to generate the new plans.

In the conventional hash join a hash table is created from the tuples of the operand having the smallest size, then this hash table is probed with the tuples of the second operand in order to produce the join results. This algorithm requires the reception of all tuples of the first operand before beginning the probe step. Thus, the time to produce the first tuple can be high if the size of the operands is large, or when the data arrival rate is irregular. Contrary to the conventional hash join (with only one hash table), the double hash join (DHJ) introduced by Ives [23] built a hash table for each operand. When a tuple arrives, it is inserted firstly in the associated hash table. Then, it is used to probe the other hash table. If the probe stage allowing to produce tuples of results, then these tuples is immediately delivered. DHJ was proposed in TUKWILA project [23] to deal with the problems of conventional hash join in the context of data integration: (i) the time of production of the first tuple is minimized, (ii) the optimizer does not need to know the sizes of the operands in order to choose the operand used in the construction of the only hash table, and (iii) it masks the slow arrival rate of tuples from an operand by processing the tuples of the other operand. However, DHJ requires maintaining the two hash tables in memory. This can limit the use of DHJ with operands having large sizes or with queries constituted of several joins. To solve this problem, parts of the hash tables residing in the memory are moved towards a secondary storage space. When the memory becomes saturated, a partition of the one of the two tables is chosen to move the tuples of this partition towards the secondary storage space in order to reduce the memory allocation. DHJ be executed in two successive phases: Regular and Cleanup. In the regular phase, the tuples which arrive are inserted in the associated hash table and those are used to probe the portions of the other hash table residing in memory. Moreover, this phase is responsible for the moving of the parts of the hash tables towards the secondary storage space. The cleanup phase starts after the reception of the totality of the tuples of the two operands. It ensures the total production of all results. This stage is necessary because the regular phase produces only part of the result. To avoid the duplicated production of the results, algorithms of marking with stamp can be used.

The operator of double hash join improve the local processing by adapting the use of resources CPU, I/O and memory with the changes of the execution environment (e.g. estimation errors, delays in data arrivals rates) and does not take in account the resource network. However, in distributed environment, the change of execution site can reduce the quantity of data transferred on the network and consequently can minimize the response time. In this objective, we propose, based on the mobile agents [16], to extend the algorithms of direct join and semi-join in order to allow them to change their execution sites proactively. A mobile agent is an autonomous software entity which can move (code, data, and

execution state) from a site to another in order to carrying out a task.

3. MOBILE JOIN ALGORITHMS

In a distributed environment [6, 31, 33], an interesting aspect in the query optimization is the selection of execution sites of the operators. The unary operators (e.g. selection, projection) are placed on the sites of their operands. However, for the binary operators (e.g. join, union), the optimizer chooses a site in order to execute these operators. Another interesting aspect is the execution of joining two operands residing on different sites. In literature, there are two approaches to execute the joining of two operands residing on different sites: (i) the direct join, and (ii) the join based semi-join [10].

In order to deal with the unexpected changes in large scale distributed environment, we propose to use the mobile agent [16, 25] for extension of join algorithms. This extension allows the join to change their execution site. The decision and change control of the execution site are made in a decentralized and autonomous manner. It no longer the optimizer chooses the join execution site, but the join itself that chooses its execution site. Indeed, the mobile agent executing a join adapts to changes in characteristics of the execution environment (e.g. network bandwidth, available memory) and responds to the estimations inaccuracies (e.g. size of intermediate relations). In the following subsections, we describe extensions of join algorithms which is called mobile join for two reasons: (i) they are executed by mobile agents, and (ii) they can change their execution site locations. For a clear illustration of these algorithms behavior, we consider a join between two relations R1 and R2 respectively located at sites S1 and S2. We also assume that | R1 | <| R2 | (where | R 1 | is the size of the relation R1) T and the result of the join between R1 and R2 should be materialized to any site

3.1 Mobile sort-merge join algorithm

In this section, we describe the mobile sort-merge join algorithm. Firstly, we present briefly the classical sort-merge join algorithm. This algorithm is composed of two steps. The first step sorts locally and in parallel the two relations on their join attribute. The second step merges the two sorted relations on a site chosen by the optimizer. After sorting of the two relations, the profiles of these relations (e.g. number of tuples, minimum and maximum values of attributes) can be known precisely and therefore the result size and the selectivity factor of the join can be revised. Thus, based on the revised profiles of the relations after sorting, it is possible to make a decision on the execution site of the join. For example, suppose that the relation T must be materialized on S3 (Figure 1) and the optimizer decided to place on S3 the mobile agent ASM in order to merge the relations because the |R1| + |R2| <|R1| + |T| and |R1| + |R2| <|R2| + |T| (i.e. |R1| <|T| and |R2| <|T|). However, if after sorting of two relations, ASM estimate that|R1| + |T| <|R2| + |T| and |R1| + |T| <|R1| + |R2| (i.e. |R1| <|T| <|R2|), then ASM migrates to S2 to merge the two relations. For this, we propose to extend the classical sort-merge join algorithm by adding a decision phase after sorting step. The extended algorithm will be called mobile sort-merge join algorithm.

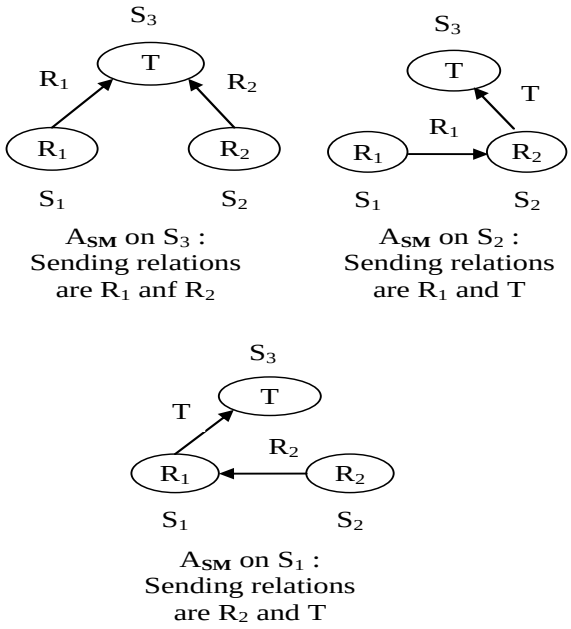


Fig 1 : relations sending by function of join execution site

A mobile sort-merge join algorithm is evaluated by three agents (Figure 2): (i) two agents called Asort1 and Asort2 are used for sorting the two relations, and (ii) the third agent called ASM, is used to calculate the result of the join from the two sorted relations. The behavior of a sorting agent is illustrated in Figure 3. This agent starts by sorting relations (Sort (R)) and simultaneously calculates the profile of R. Then it sends the calculated profile of R to the agent ASM. Afterwards, it waits until the agent ASM sends his location site (the execution site of merge) in order to start the sending the tuples of R (Send (R)).

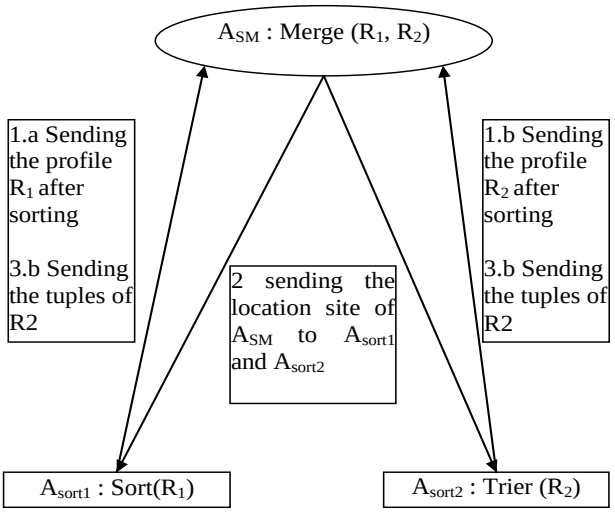


Fig 2 : communications between agents of mobile sort-merge join algorithm

The Figure 4 describes the behavior of an agent (ASM) executing merge between two sorted relations. This agent receives the revised profiles of the two sorted relations produced by the sorting agents (Asort1 and Asort2). Then the agent decides to continue its execution on the site chosen by the optimizer at compilation step or migrates to another site. This decision of ASM is autonomous and decentralized. For

this, it based on a decision function that determines the site where it migrates to continue its execution. The parameters of this decision function are:

- RelProf: it contains profiles recalculated of the relations (R_1 and R_2) and the revised profile of the result T.
- RelUnav: it describes the unavailability relations. For example, the time required to produce the first tuple or the production time of each tuple.
- (iii) EnvState: it contains information describing the status of the execution environment (e.g. memory, CPU load, bandwidth, etc.).

After migrating, A_{SM} sends its decision to sorting agents. These agents A_{SM} send to the tuples of the two sorted relations in order to produce the result tuples (Merge (Sorted_R_1, Sorted_R_2, T)). Finally, the results tuple are sent to the client (Send (T)) or are materialized to local disk (materialize (T)).

```
Sort (R) ;
SendEstimation (R.Profil, A_SM) ;
ReceiveSite (A_SM)
Send(R, A_SM);
```

Fig 3 : behavior of sorting agent

```
Site ← Decision (RelProf, RelUnav, EnvSate) ;
If (not local (Site) then migrate on Site ;
SendSite (Site, A_sort1) ;
SendSite (Site, A_sort2) ;
If (not local (Sorted_R1)) then Receive (Sorted_R1) ;
If (not local (Sorted_R2)) then Receive (Sorted_R2) ;
Merge (Sorted_R1, Sorted_R2, T));
If (not local (T)) then Send (T) else Materialize (T) ;
```

Fig 4 : behavior of merging agent

3.2 Mobile hash join algorithm

In the simple hash join algorithm, during the building of the hash table, the characteristics of R_1 (e.g. size, values distribution of every attribute) can be calculated precisely. So, from the precise statistics of R_1, the statistics of R_2 estimated at the optimization, the statistics revised of T from R_1 and R_2, the unavailability of R_2 and the state of the system, it is possible to make a decision on the localization of the probe step and eventually move this step to another site. The behavior of a mobile agent executing a simple hash join is described in Figure 5.

```
If (not local (R1) ) then Receive (R1) ;
Build  (R1, HT) ;
Site ← décision RelProf, RelUnav, EnvSate) ;
If  (not local (site))    then migrate on site ;
If (not local ((R2)) ) then Receive (R2) ;
Probe ( HT, R2, T) ;
If (not local (T))  then send (T)    Else materialize(T) ;
```

Fig 5 : behavior of mobile hash join agent

3.3 Mobile hash join based on semi-join algorithm

The join operator based on semi-join is introduced to reduce the volume of data transferred between sites. The join based on the semi-join enhances response time by reducing the size of relations exchanged between sites because the communication cost is the dominant factor in the response time. This enhancement is proportional to the selectivity join factor. The classical join based semi-join is composed of three steps:

- A projection on S1 of R1 on the attributes join (ΠR1).
- A semi-join on S2 between the projection result and R2. Thus, tuples of ΠR1 are transferred from S1 to S2. The result of this semi-join is noted RSJ.
- A join on S1 between R1 and RSJ. Here, the tuples of RSJ are transferred from S2 to S1.

We propose to execute the join based semi-join using a mobile agent, called A_{SHJ}. The behavior of A_{SHJ} is described in Figure 6. After projection of R1, the agent A_{SHJ} has better knowledge about the profile of R_1 and the size of RSJ can be re-estimated. Thus, using these new parameters, the agent A_{SHJ} checks if the semi-join is always better than direct join. The primitive SJ (ΠR_1, RSJ, R_2, T) allows the agent A_{SHJ} to determine if the semi-join must be replaced by a direct join. For example, suppose that T must be materialized on S_1 and the optimizer estimates that $|R_1| + |T| > |R_2|$ and $|\Pi R_1| + |RSJ| < |R_2|$. Thus, after the projection of R_1, if the agent A_{SHJ} determines that $|\Pi R_1| + |RSJ| < |R_2|$, it continues the execution of semi-join. However, if the agent A_{SHJ} determines that $|\Pi R_1| + |RSJ| > |R_2|$, it replaces the semi-join by a direct join direct. In case the agent A_{SHJ} decides to continue the execution semi-join, it migrates with ΠR_1 on S_2, and then calculates locally (on S_2) the tuples of RSJ by executing a classical hash join between ΠR_1 and R_2. The temporal relation RSJ is again used by A_{SHJ} with R_1 to execute a mobile hash join. In the case the agent A_{SHJ} decides to replace the semi-join by a direct join, it compares the size of two relations to choose its initial execution site. If it estimates that $|R_1| < |R_2|$, executes the mobile hash join on S_1, Else, it migrates on S_2 to start its execution.

```
If (notlocal (R1)) then receive (R1) ;
ΠR1=Projection (R1) ;
If (ContinueSJ (ΠR1, RSJ, R2, T) then
    {
    Migrate on S2 ;
    HashJoin(ΠR1, R2, RSJ) ; // semi-join
    MobileHashJoin (RSJ, R1, T) ;
    }
Else      If (|R1|<|R2|)  then
    MobileHashJoin (R1, R2, T) ; // direct join
    else
    {
    Migrate on S2 ;
    MobileHashJoin (R2, R1, T) ; // direct join
    }
```

Fig 6 : behavior of mobile hash join agent based on semi-join

4. PERFORMANCE EVALUATION

The objective of our performance evaluation is to validate if an agent chooses the site which allows him to minimize its response time. For that, we compare: (i) the response time of the agent execution of mobile join algorithm by allowing him to choose its execution site; and (ii) the response times of the agent execution of classical join algorithm on each site.

We realized our experiments in distributed environment. It is constituted of two workstations S1 (HP) and S2 (SUN) interconnected by Internet. The first is located in Lebanon at Tripoli and it will be called Tripoli. The second is located in Lebanon at Beirut and it will be called Beirut.

To handle our experiments, we installed on every workstation a platform of mobile agents [16, 37] including the mobile sort-merge join algorithm, the mobile hash join algorithm, and the mobile hash join algorithm based on semi-join. In this one, every mobile agent runs on a Virtual Machine Java (JDK 1.6.2).

The response times are measured by real executions. These are carried out between sites interconnected via a network. Our experiments are handled in multi-user environments, where several users can start up applications. In these environments, it is difficult to reproduce an experiment in identical conditions. Indeed, the workload of an execution site varies from moments to another (available memory, number of running processes, etc.) and the amount of the data transferred through the network also varies.

The base costs associated with the execution of these algorithms are deduced by calibration. The migration cost of an agent given in the table Tab.1 includes the serialization cost, the transfer cost and the de-serialization cost. Of course, when the agent migrates with its data, it must be added the serialization cost, the transfer and the de-serialization of the data which is proportional to its size.

Tab. 1: Environment parameters

Networks parameters

	Banwidth (KB/s)	Time to send a page(ms)
Tripoli → Beirut	106.3	749.52
Beirut → Tripoli	112.64	723.18

Mobile agent parameters

	Agent migration(ms)
Tripoli → Beirut	29623
Beirut → Tripoli	28967

Workstations parameters

	Time to write a page(ms)	Time to read a page (ms)
Tripoli	0.79	0.65
Beirut	1.12	1.03

To compare quantitatively the various algorithms, we consider two relations R_1 and R_2 residents respectively on Tripoli and Beirut, with number of tuples estimated at 20 000 and 40 000 respectively. Also let us consider a simple hash join J: RES1 = $R_1 \infty R_2$. The join J is emitted on Tripoli, this one will be executed by a mobile agent noted AJ. The optimal execution plan of J is to receive R_2 on Tripoli in order to join with R_2. The selectivity factor of J is 1.5/max ($||R_1||$, $||R_2||$) where $||R_i||$ indicates the R_i number of tuples. In the rest of this section, we describe the results of our experiments.

4.1 Expirementations results

In this section, we describe the results of the experiments handled for the mobile sort-merge join algorithm and the mobile hash join algorithm. Here, we verify the choice of an agent according to the variation between the parameters estimated at compile-time (the estimated number of tuples of R1 noted ||R1es||, and the estimated selectivity factor of J noted SFes) and that computed by agent at runtime (the computed number of tuples of R1 noted ||R1comp||, and the computed selectivity factor of J noted SFcomp).

In the next sub-sections, we evaluate the impact of an estimation error of the R_1 number of tuples and of the selectivity factor on the mobile sort-merge join algorithm and the mobile hash join algorithm.

4.1.1 Expirements of mobile sort-merge join algorithm according to estimation errors of $||R_1||$ and SF

The curves of the Figure 7 and of the Figure 8 show the measures of the response times of the join J (sort-merge join algorithm execution on Tripoli noted "execution SMJ on Tripoli", sort-merge join algorithm execution on Beirut noted "execution on SMJ Beirut ", mobile sort-merge join algorithm execution noted "execution MSMJ") by decreasing the ||R1comp|| compared with ||R1es|| (Figure 3) and SFcomp compared with SFes (Figure 4). We observe that the behavior of MSMJ in the Figure 7 and in the Figure 8 is similar.

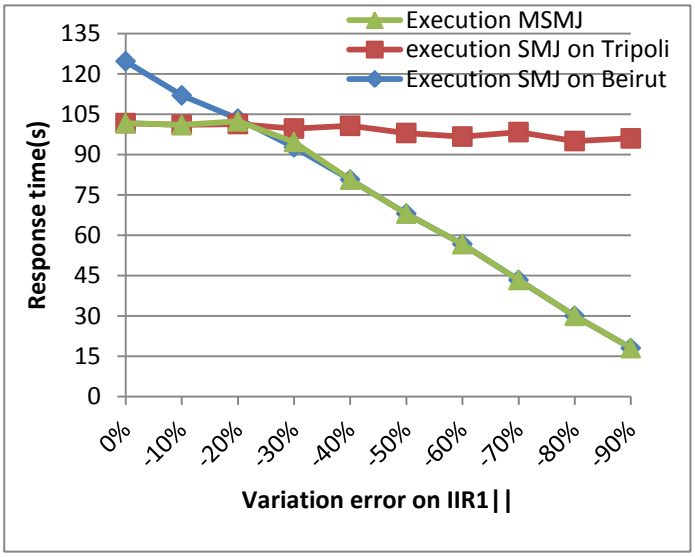


Fig 7: Sort-merge join performance by decreasing $||R_1||$

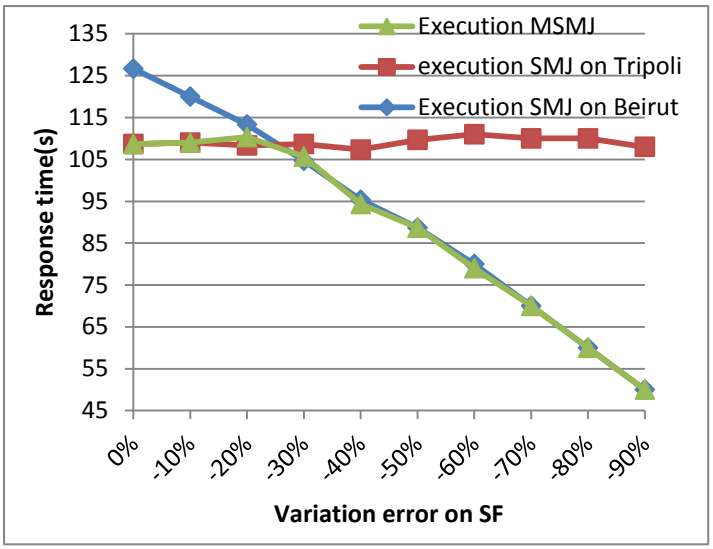


Fig 8: Sort-merge join performance by decreasing SF

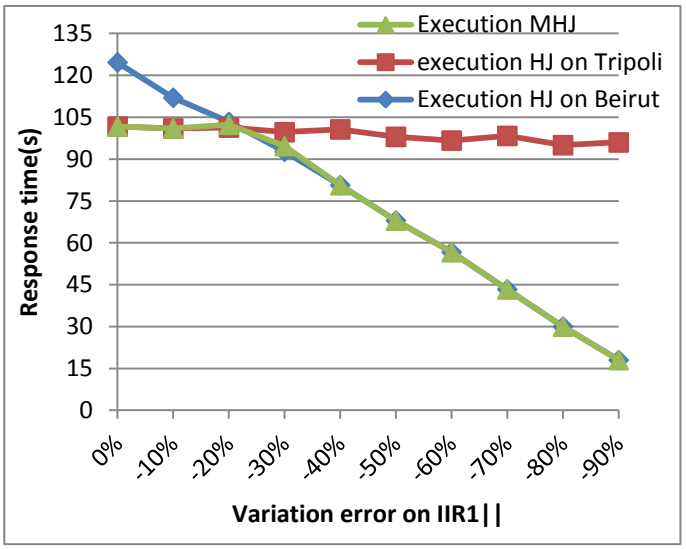


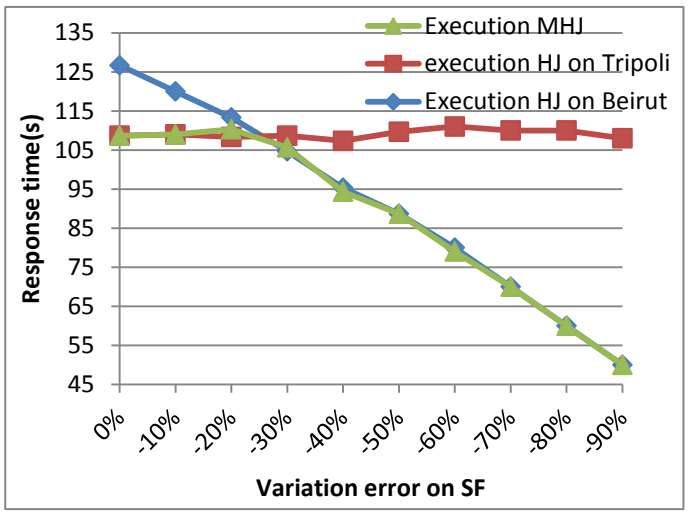
Fig 9:Hash join performance by decreasing $||R_1||$

The Figure 7 and the Figure 8 show respectively that for a variation of ||R1comp|| compared with ||R1es|| between 0% and -20% and for a variation of SFcomp compared with SFes between 0% and -30%, the agent executes the execution plan generated during the compilation of J. On the other hand, when the variation of ||R1comp|| compared with ||R1es|| is higher than -20% (higher than -30% for a variation of SFcomp compared with SFes), the agent decides to move on Beirut in order to calculate the result of join. Thus, it modifies the execution plan generated during the compilation.

Although, the variation between $||R1_{comp}||$ and $||R1_{es}||$ and that between SF_{comp} and SF_{es} have similar influences on the behavior of the MSMJ whatever the estimations error. After -30%, MSMJ improve the response time compared to classical execution of SMJ on Tripoli or on Beirut.

4.1.2 Expirements of Mobile hash join algorithm according to estimation errors of $||R_1||$ and SF

The curves of the Figure 9 and of the Figure 10 show the measures of the response times of the join J (hash join algorithm execution on Tripoli noted "execution HJ on Tripoli", hash join algorithm execution on Beirut noted "execution on HJ Beirut ", mobile hash join algorithm execution noted "execution MHJ") by decreasing the ||R1comp|| compared with ||R1es|| (Figure 3) and SFcomp compared with SFes (Figure 4). We observe that the behavior of MHJ in the Figure 9 and in the Figure 10 is similar.



.Fig 10: Hash join performance by decreasing SF

The Figure 9 and the Figure 10 show respectively that for a variation of ||R1comp|| compared with ||R1es|| between 0% and -20% and for a variation of SFcomp compared with SFes between 0% and -30%, the agent executes the execution plan generated during the compilation of J. On the other hand, when the variation of ||R1comp|| compared with ||R1es|| is higher than -20% (higher than -30% for a variation of SFcomp compared with SFes), the agent decides to move on Beirut in order to calculate the result of join. Thus, it modifies the execution plan generated during the compilation.

Although, the variation between $||R1_{comp}||$ and $||R1_{es}||$ and that between SF_{comp} and SF_{es} have similar influences on the behavior of the MHJ whatever the estimations error. After -30%, MHJ improve the response time compared to classical execution of HJ on Tripoli or on Beirut.

4.1.3 Discussion

In this performance evaluation, we realized our experiments according to estimation error. From the results of these experiments, we can note the following observations: (i) the mobile join algorithms improve the responses times whatever the estimation error; (ii) the experiments handled in a distributed environment, we notice that the workload of the execution site does not influence on the response time. Here, the costs of local processing (CPU, I/O) account for 5% of the response time of the join. On the other hand, the transfer cost of data between the two sites accounts for 95% of the response time of the join. Hence, in this environment we do not to see the variation of the response time of the join (execution SMJ on Tripoli and execution SMJ on Tripoli), because the influence of the scale of the axis (response time) and of the weak weighting of costs (CPU, I/O) in the response time, and (iii) We observe that the behavior of sort-merge join algorithm and the hash join algorithm is similar because the costs of the local processing are almost negligible in front of the transfer cost of data.

5. CONCLUSIONS AND PERSPECTIVES

In this paper, we proposed an extension of join algorithms based on mobile agents to correct the sub-optimality of the execution plans while decentralizing the control and the modifications of the execution plans. They improve the cost of local processing and the communication cost by minimizing the volume of data transferred on the network. The performance evaluation shows that the proposed algorithms improve the response time whatever the variation of ||R1comp|| compared with ||R1es|| or of SFcomp compared with SFes.

The next objectives of research are focused mainly on : (i) the definition of the methodes which determine the migration space of the agents participating in execution of a query. In this paper, the migration space of an agent is calculated, according to operand localizations of the operator executed by the agent, independently of the migration spaces of the other agents participating in the execution of the same query. For that, it is important to define methods whose take into account the migration spaces of the other agents participating in the execution of a query and the tree structure of execution plan of the query; (ii) the extension of our performance evaluation. Here, they plan to increase the number of the site of our experimentation environments and to make more exhaustive experiments in order to study the behaviors of the agents on the level of the complex queries.

6. REFERENCES

[1] L. AMSALEG et al.; Scrambling query plans to cope with unexpected delays, Proc. of the Fourth International Conference on Parallel and Distributed Information Systems, IEEE Computer Society, Miami, Florida, USA, December 1996, pp. 208-219.

[2] L. AMSALEG, M. FRANKLIN, A. TOMASIC; Dynamic query operator scheduling for wide-area remote access, Distributed and Parallel Databases, vol. 6, no3, Kluwer Academic Publishers, 1998, pp. 217-246.

[3] G. ANTOSHENKOV, M. ZIAUDDIN; Query processing and optimization in Oracle Rdb, Journal of VLDB, Springer Verlag Publishers, vol. 5, no4, December1996, pp. 229 237.

[4] R. AVNUR, J.-M HELLERSTEIN; Eddies: continuously adaptive query processing, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Dallas, Texas, USA, May 2000, pp. 261-272.

[5] S. Babu, P. Bizarro, D. -J. DeWitt; Proactive Re-optimization. Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Baltimore, Maryland, USA, June 2005, pp.107-118.

[6] Bose, S.K., Krishnamoorthy, S., Ranade, N.: Allocating Resources to Parallel Query Plans in Data Grids. In: Proc. of the 6th Intl. Conf. on Grid and Cooperative Computing, pp. 210–220. IEEE CS, Los Alamitos (2007)

[7] L. BOUGANIM et al.; A dynamic query process-ing architecture for data integration systems. Journal of IEEE Data Engineering Bulletin, IEEE Computer Society, vol. 23, no2, June 2000, pp. 42-48.

[8] L. BOUGANIM et al.; Dynamic query scheduling in data integration systems, Proc. of the 16th International Conference on Data Engineering, IEEE Computer Society, San Diego, California, USA, March 2000, pp. 425-434.

[9] N. BRUNO, S. CHAUDHURI; Efficient Creation of Statistics over Query Expressions, Proc. of the 19th International Conference on Data Engineering, IEEE Computer Society, Bangalore, India, March 2003, pp.201-212.

[10] D.-M. Chiu, Y.-C. Ho ; A Methodology for Interpreting Tree Queries Into Optimal Semi-Join Expressions, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Santa Monica, California, USA, Mai 1980, pp. 169-178.

[11] C.-M. CHEN, N. ROUSSOPOULOS; Adaptive Selectivity Estimation Using Query Feedback, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Minneapolis, Minnesota, USA, May 1994, pp. 161-172.

[12] C. COLLET, T.-T. VU ; QBF: A Query Broker Framework for Adaptable Query Evaluation, Proc. of 6th International Conference on Flexible Query Answering Systems, Springer Verlag Publishers, Lyon, France, June 2004, pp. 362-375.

[13] A. DESHPANDE, J.-M. HELLERSTEIN; Lifting the Burden of History from Adaptive Query Processing, Proc. of the Thirtieth International Conference on Very Large Data Bases, Morgan Kaufmann, Toronto, Canada, August 2004, pp. 948-959.

[14] R.-S. EPSTEIN, M. STONEBRAKER, E. WONG ; Distributed Query Processing in a Relational Data Base System, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Austin, Texas, June 1978, pp. 169-180.

[15] C. EVRENDILEK et al.; Multidatabase Query Optimization, Journal of Distributed and Parallel Databases, Kluwer Academic Publishers, vol 5, no1, January 1997, pp. 77-113.

[16] A. FUGGETTA, G.-P. PICCO, G. VIGNA; Understanding Code Mobility, IEEE Transactions on Software Engineering, IEEE Computer Society, vol. 24, no5, May 1998, pp. 342-361.

[17] G. GARDARIN, F. SHA, Z.-H. TANG; Calibrating the Query Optimizer Cost Model of IRO-DB, an Object-Oriented Federated Database System, Proc. of 22th International Conference on Very Large Data Bases, Morgan Kaufmann, Mumbai (Bombay), India, September 1996, pp. 378-389.

[18] A. GOUNARIS et al.; Adaptive Query Processing: A Survey, Proc. of the 19th British National Conference on Databases, Sheffield, UK, July 2002, pp. 11-25.

[19] A. HAMEURLAIN, P. BAZEX, F. MORVAN; Traitement parallèle dans les bases de données relationnelles, EDITIONS CÉPADUÈS, 1996.

[20] A. HAMEURLAIN, F. MORVAN; Parallel Query Optimization Methods and Approaches: a Survey, Journal of Computers Systems Science & Engineering, CRL Publishing Ltd9 De Montfort Mews, vol. 19, no5, September 2004, pp. 95-114.

[21] J.-M. HELLERSTEIN et al.; Adaptive query processing: Technology in evolution, IEEE Data Engineering Bulletin, IEEE Computer Society, vol. 23, no2, June 2000, pp. 7-18.

[22] Hu, N., Wang, Y., Zhao, L.: Dynamic Optimization of Sub query Processing in Grid Da-tabase, Natural omputation. In: Proc of the 3rd Intl Conf. on Natural Computation, vol. 5, pp. 8–13. IEEE Computer Society Press, Los Alamitos (2007).

[23] Z.-G. IVES et al.; An Adaptive Query Execution System for Data Integration, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Philadelphia, Pennsylvania, USA, June 1999, pp. 299-310.

[24] Z.-G. IVES, A.-Y. HALEVY, D.-S. WELD; Adapting to Source Properties in Processing Data Integration Queries, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Paris, France, June 2004, pp. 395-406.

[25] R. JONES, J. BROWN; Distributed Query Processing Via Mobile Agents, find the 14 november 2002, accessible via: http://www.cs.umd.edu/~rjones/paper.html, 1997.

[26] N. KABRA, D.-J. DEWITT; Efficient Mid-Query Re-Optimization of sub-optimal query execution plans, Proc. of the ACM SIGMOD International Conference on Management of Data, ACM Press, Seattle, Washington, USA, June 1998, pp. 106-117.

[27] L. KHAN, D. MCLEOD, C. SHAHABI; An Adaptive Probe-Based Technique to Optimize Join Queries in Distributed Internet Databases, Journal of Database Management Idea Group, vol. 12, no4, Octobre 2001, pp. 3-14.

[28] F. MORVAN, A. HAMEURLAIN; Dynamic Memory Allocation Strategies For Parallel Query Execution, Proc. of the ACM Symposium on Applied Computing, ACM Press, Madrid, Spain, March 2002, pp. 897-901.

[29] H. NAACKE, G. GARDARIN, A. TOMASIC ; Leveraging Mediator Cost Models with Heterogeneous Data Sources, Proc. of the Fourteenth International Conference on Data Engineering, IEEE Computer Society, Orlando, Florida, USA, February 1998, pp. 351-360.

[30] B. NAG, D.-J. DEWITT; Memory Allocation Strategies for Complex Decision Support Queries, Proc. of the ACM CIKM International Conference on Information and Knowledge Management, ACM Press, Bethesda, Maryland, USA, November 1998, pp. 116-123.

[31] M. OUZZANI, A. BOUGUETTAYA; Query Processing and Optimization on the Web, Distributed and Parallel Databases, Kluwer Academic Publishers, vol. 15, no3, May 2004, pp. 187-218..

[32] M.-T. ÖZSU, PATRICK VALDURIEZ; Principles of Distributed Database Systems, Second Edition, Prentice-Hall, 1999.

[33] Paton, N.W., Chávez, J.B., Chen, M., Raman, V., Swart, G., Narang, I., Yellin, D.M., Fernandes, A.A.A.: Autonomic query parallelization using non-dedicated computers: an evaluation of adaptivity options. VLDB Journal 18(1), 119–140 (2009)

[34] V. RAMAN, A. DESHPANDE, J.-M. HELLERSTEIN; Using State Modules for Adaptive Query Processing, Proc. of the 19th International Conference on Data Engineering, IEEE Computer Society, Bangalore, India, March 2003, pp. 353-362.

[35] G.-M. SACCO, S.-B. YAO; Query Optimization in Distributed Data Base Systems, Advances in Computers, vol. 21, 1982, pp. 225-273.

[36] M. STILLGER et al.; LEO - DB2's LEarning Optimizer. Proc.of 27th International Conference on Very Large Data Bases, Morgan Kaufmann, Roma, Italy , September 2001, pp. 19-28.

[37] P. WOJCIECHOWSKI; Algorithms for location-independent communication between mobile agents, Technical Report DSC-2001/13, Ecole Polytechnique Fédérale de Lausanne, Département Systèmes de Communication, 2001.

[38] Y. ZHOU et al. ; An adaptable distributed query processing architecture, Data & Knowledge Engineering, vol. 53, no3, June 2005, pp. 283-309.

[39] Q. ZHU, S. MOTHERAMGARI, Y. SUN; Cost Estimation for Queries Experiencing Multiple Contention States in Dynamic Multidatabase Environments, Journal of Knowledge and Information Systems, Springer Verlag Publishers, vol. 5, no1, Februray 2003, pp. 26-49.