A mobile relational algebra

Franck Morvan and Abdelkader Hameurlain*
Institut de Recherche en Informatique de Toulouse IRIT, Université Paul Sabatier, Toulouse, France

Abstract. Most of the dynamic query optimization methods proposed in the literature are centralized. This centralization, in a large-scale environment, generates a bottleneck due to relatively important message exchange on a network with a weak bandwidth and strong latency. It becomes thus convenient to render autonomous and self-adaptable the query execution on a large-scale network. In this perspective, we propose a mobile relational algebra to decentralize the control of dynamic query optimization processes. Experiments, in local and large-scale distributed environments, allow: (i) to validate the proposed proactive migration policy, and (ii) to identify the efficiency intervals of proposed mobile relational algebra.

Keywords: Large scale data distribution, dynamic query optimization, decentralization, mobile agents, performance evaluation

1. Introduction

More and more applications try to integrate heterogeneous data sources scattered on a large-scale network. An example is the biomedical research domain where high numbers of data sources are accessible. These data sources have different query processing capacities. For example, a phone book service which requires the name of a person to return a phone number is a data source where the access is restricted.

In a large-scale environment, networks have a relatively low bandwidth, the accessed data volume can be huge and the number of users and data sources is important. In this environment, one of the major challenges for the viability of applications is the performance optimization of the query evaluation. It is thus necessary to propose query optimization methods.

The static optimization methods relying on estimations to generate an optimal execution plan or close to the optimal cannot be used. Indeed, the operation costs executed by sources are very difficult to estimate because of the autonomy and of the heterogeneity of data sources. The statistics describing the data stemming from sources and the formulae associated to the operations processed by these sources can be not revealed [11]. Furthermore, when the statistics exist, they are not necessarily up to date. So, the use of a dynamic optimization method is necessary to correct sub-optimality of the execution plans and to adapt itself to events which can occur during query execution.

The dynamic query optimization methods can be classified in two categories according to the nature of the decision-making [27]: the centralised and decentralized methods. The centralised methods [4–6,13, 16,23] are supervised by a master process which controls all processes participating to the optimization or re-optimization. These methods cannot be adapted to a large-scale environment because of: (i) the

*Corresponding author: Abdelkader Hameurlain, Institut de Recherche en Informatique de Toulouse IRIT, Université Paul Sabatier, 118 Route de Narbonne, 31062 Toulouse Codex 9, France. Tel.: +33 5 61 55 82 48; Fax: +33 5 61 55 62 58; E-mail: {morvan, hameur}@irit.fr.
important number of messages on networks with relatively low bandwidth and strong latency and (ii) the bottleneck that forms the optimizer, since all the messages converge to a single point.

In the decentralized optimization methods [10,19,37,38] many authors improve the local processing cost by adapting the use of the CPU, I/O and memory resources with the changes of the execution state (e.g. estimation errors, delays in data arrival rates). However, the proposed methods are focused mainly on the resources (CPU, I/O and memory) and do not take into account the network resource. In particular, these methods do not minimize the volume of data transferred on networks. The data transfers determined during compilation remain identical whatever the estimation errors detected at runtime. The network resource is important to consider because it can represent up to 85% of the query response time [32]. To take into account this resource [22], leans on the mobile agents to decentralize the control and the modifications of the execution plans. In this method, the interactions between the agents participating in the same query are based on message broadcasting. This broadcast allows making a decision with a global view on statistics of the query. However, this broadcast generates a network congestion when the number of agents becomes significant. In a large scale distributed environment, the congestion of the network is quickly affected because of the strong latency and the low bandwidth. As a consequence, an optimization method must be decentralized and must minimize the number of exchanged messages.

In this perspective, we propose a mobile relational algebra. For that purpose, we use the mobile agents to exploit their properties of autonomy, reactivity and mobility. Autonomy and reactivity, allow decentralizing the control. As for the mobility, it can replace distant interactions by local interactions. The originality of our proposal is to make a complete algebra where some parts are published in [3,17,32,36]. In the database field, two kinds of mobility exist: unit [26,39,40] and software [14] mobility. In this paper, we are only interested in software mobility.

The remainder of the paper is organized as follow: in Section 2 we describe the mobile relational operators without restricted access. In Section 3 we explain the mobile relational operator with restricted access. In Section 4 the cost model is presented. Section 5 shows the performance evaluation of the mobile hash join operator and of a mobile multi-join query. Section 5 is a survey of the related works on dynamic optimization. In Section 6 we conclude.

2. Relational operators without restricted access

In the proposed mobile relational algebra, each relational operation is processed by a mobile agent [14]. Every agent is autonomous by observing its execution environment and by choosing a site to migrate or not to continue its operation. So, the control of the dynamic query optimization is decentralized and the operations are processed as close as possible to the data. In this section we describe all the mobile relational operators which have no restricted access to the base relations. In the rest of the paper, to facilitate the comprehension of the mobile relational operators, we use an algorithmic notation supposing that the base or temporary relations are completely sent. In the implementation of our prototype, these communication operations are made in pipeline mode for performance reasons.

2.1. Scan, select and project

A scan operator of a base relation R is initially placed on the site of the relation R. This operator is the only fixed operator. As for the select and project operators, they are mobile in order to be processed on the production site of their operand relation. The objective is to replace distant interactions by local interactions in order to reduce the transferred data volume. The behavior of these operators is illustrated with a project operation in Fig. 1.
2.2. Mobile join

In a distributed environment, an important aspect is the way to process join of two relations mapped on two different sites. In the literature, there are two approaches to process inter-site joins [35]: (i) the direct join by moving one relation or both relations, and (ii) the semi-join based join. This alternative consists in replacing a join, whatever is the class of algorithm implanting this join, by the combination of a projection, a semi-join ended by a join [9].

In this paper, we describe only the direct hash join algorithm, and semi-join based hash join algorithm which we call mobile join algorithm [3]. In order to illustrate the behaviour of these two algorithms, we consider a join \( J \) between two relations \( R_1 \) and \( R_2 \) placed respectively on the sites \( S_1 \) and \( S_2 \). We also suppose that \( |R_1| < |R_2| \) (Where \( |R_1| \) the size of the relation \( R_1 \)) and the result \( T \) of the join must be materialized on an indefinite site.

2.2.1. Direct mobile hash join

The hash join [30] is processed in two steps: (i) build the hash table from the smallest relation \( R_1 \), and (ii) the hash table is probed with \( R_2 \) to produce the result. In the hash join, during the building of the hash table, the \( R_1 \) profile can be computed precisely at runtime. Hence, from runtime profile on \( R_1 \), the compile-time profile on \( R_2 \), the profile on \( T \) can be revised. Thus, it is possible to make a decision about the location of the probe step and maybe to move it on another site. For example, let us suppose that \( |R_2| < |R_1| + |T| \). However, if after building the hash table, \( J \) estimates that \( |R_2| > |R_1| + |T| \), then \( J \) migrates with the hash table on \( S_2 \) and resumes its execution on this site. For that purpose, we extend the hash join algorithm by allowing the agent to choose its execution site between the build and probe phases.

In our example, the query is processed with three agents: two fix agents which are in charge of scanning both relations \( R_1 \) and \( R_2 \), respectively on sites \( S_1 \) and \( S_2 \), and a mobile agent which is in charge of processing the join between both relations. This last one begins its execution on a site determined by the optimizer where the query was emitted [33]. The behaviour of a mobile hash join operator is described in the Fig. 2. First, the agent begins by building the hash table from \( R_1 \) (\( build (R_1, HT) \)) and simultaneously, it computes the \( R_1 \) profile. As in the works of [5,6], to limit the statistics collect cost, this collect is integrated within the operation. Then, the agent makes a decision about the probe step location. This decision is made in an autonomous and decentralized way. For that, the agent leans on a decision function. The parameters of this function are:

- \( StatInfo \): includes \( R_1 \) profile estimated at runtime, \( R_2 \) profile estimated at compile-time and the revised profile of \( T \).
- \( DataAvail \): gives information about relation availability. For example, the waiting time to produce the first tuple or the waiting time to produce each tuple [17].
Syntax: contains information describing the execution state (e.g. memory capacity, CPU load, bandwidth).

When a mobile agent decides to move onto another execution site, to process the probe step (probe (HT, R₂, T)), it migrates with its hash table. Finally, the result tuples are sent to the client (send (T)) or are written on the local disk (materialize (T)).

2.2.2. Semi-join based mobile hash join

The semi-join based join operator was introduced to reduce the quantity of transferred data between distant sites. The classic execution of a semi-join based join is composed of three steps:

1. A project on S₁ of the first relation R₁ on the join attributes (P). All tuples of P are transferred from S₁ to S₂,
2. a semi-join on S₂ between the second relation R₂ and the result of the project. The result of this semi-join is quoted RSJ. The RSJ tuples are transferred from S₂ to S₁,
3. a join on S₁ between RSJ and R₁.

As for the mobile hash join, we suggest to process the semi-join based join by means of one mobile agent. The behaviour of this operator is described in Fig. 3. Having made the project on R₁, the mobile agent has better knowledge concerning the R₁ profile, the P profile and it can revise the RSJ and T₁ profiles. Hence, by using these profiles, it verifies if the semi-join based join is always more beneficial than a direct join. The primitive ContinueSJ (P, RSJ, R₂, T) allows the agent to determine if it continues its execution or if it replaces the semi-join based join by a direct mobile join. For example, let us suppose that the optimizer considered that |R₁| + |T| > |R₂| and |P| + |RSJ| < |R₂|. So, having made the project on R₁, if the agent considers that |P| + |RSJ| < |R₂|, it continues the execution of the join based on semi-join. However, if the agent considers that |P| + |RSJ| > |R₂|, it replaces the semi-join based mobile join by a direct mobile join. If the agent decides to continue the execution of the semi-join based join, the agent moves with P onto S₂. After, it computes the RSJ tuples by processing locally (on S₂) a classical mobile hash join between P and R₂. The temporary relation RSJ is again used with R₁ by the agent to realize a mobile hash join. Between the build and probe steps of the join (RSJ, R₁), the mobile agent can make a decision on the choice of its execution site.

![Fig. 2. Mobile hash join algorithm.](image)

| if (not local (R_i)) then receive (R_i); |
| build (R_i, HT); // compute statistics on R_i |
| Site ← decision (StatInfo, DataAvail, SyState); |
| if (not local (Site)) then migrate on Site; |
| if (not local (R_j)) then receive (R_j); |
| probe (HT, R_j, T); |
| if (not local (T)) then send (T) |
| else materialize (T); |

\(^{1}\)T is always materialized on S₁ site.
When the semi-join based join is replaced by a direct join after the project, the agent compares the size of both relations to choose its initial execution site. If it considers that \( |R_1| < |R_2| \), it begins its execution on \( S_1 \). Elsewise, it moves on \( S_2 \) to begin its execution.

2.3. Mobile minus and union

As for the join, the minus operator can be processed according to two approaches: (i) the direct minus and (ii) the minus based on semi-join. Because of the similarity with the join operation, we shall describe, hereafter, only the direct minus operator. The minus \( T = \text{minus} (R_1, R_2) \) can be obtained by building a hash table with the \( R_1 \) relation. Then, this hash table is probed with the relation \( R_2 \). When two tuples match, the \( R_1 \) tuple is discarded in the hash table. This implementation forbids any pipeline execution. Indeed, it is necessary to wait that all the relation \( R_2 \) is scanned to know if a tuple is a part or not of the result \( T \). To allow this pipeline execution, the hash table is built with \( R_2 \). The algorithm, Fig. 4, illustrates this behaviour.

The union operator can be expressed from the minus operator. Indeed, \( R_1 \cup R_2 = (R_1 – R_2) \oplus R_2 \) where \( \oplus \) denote the concatenation operator. The union operator algorithm being very close to the minus operator, it will not be illustrated.

3. Relational operators with restricted access

Some data sources impose restricted access on relations. Restricted access can be formalized by using binding patterns. A binding pattern for a table \( R(X_1, X_2, \ldots, X_n) \) is a partial mapping from \{\( X_1, X_2, \ldots, X_n \)\} to the alphabet \{\( b, f \)\} [25]. For those attributes mapped to ‘b’, the values should be supplied in order to get information from \( R \) while the attributes mapping to ‘f’ do not require any input in order to
return tuples from R. If all the attributes of R are mapped to ‘f’ then it is possible to get all the tuples of R without any restriction (e.g. with a relational scan operator).

In our example, in the introduction, the mapping of the relation phone (name, phoneNumber) is Phone (name, phoneNumber). It means that the relation Phone gives the phone number only if the value of the name attribute is known.

Because of the restricted access to a data source, some operations of the algebra are not possible as the union and the others must be adapted as the join and the minus. In this section, we describe the specific mobile relational operators for data sources with restricted access.

3.1. Dependent access operator

The restricted access to a data source prevents the use of the scan operator. Indeed, to reach tuples, it is necessary to supply to the data source all the attributes which are mapped to ‘b’. So, we need a new fixed operator named Daccess as D indicates its dependency on the value of the input attribute(s). Formal semantics of Daccess is as follows [25]: Consider a table R(X, Y) and \( \chi \) be a set of values for X. Then, DAccess(R(X, Y), \( \chi \)) = \( \sigma (X \in \chi, R(X, Y)) \).

3.2. Mobile dependent join operator

In this section we will describe a join algorithm designed for restricted sources and based on the hash join algorithm. Consider an execution plan that needs a relational join between Employee (empId, name) and Phone (name, phoneNumber) tables on the name attribute. In relational join both of the following fragments: Join (Employee, Phone) and Join(Phone, Employee) are valid since join is commutative. But with restricted sources the second fragment Join (Phone, Employee) on name attribute is not valid since the binding pattern of the table Phone requires the value of the name attribute in order to return the value of the phoneNumber. So, we need a new join operator which is asymmetric in nature, also known as dependent join Djoin [18]. In Djoin (R1, R2) a hash table HT1 is built with the first relation R1 which is a relation where all attributes are mapped to ‘f’. At the same time we retrieve the distinct values of the join attribute(s) (e.g. name attribute in our example) into a relation P. Then, Daccess operator is called to compute R2’ = Daccess(R2, P). Finally, the hash table HT1 is probed with R2’ to produce the result.
The algorithm of the mobile dependent join operator, MDJoin, is presented in Fig. 5. We build a hash table from the first relation (HT1). By relational project operation we retrieve the distinct values of the attribute(s) V and store them in the table P just like before. After building the hash table we are now able to calculate run-time profile of R\textsubscript{1} and P and we have a better idea about the execution environment. MDJoin can now make a decision about where to continue its execution and migrates to the Site according to its decision function (e.g. \(|R_2| > |R_1| + |T| + |P|\)). After the migration step the rest of the algorithm is the same as the fixed version.

This algorithm resembles the semi-join based mobile join algorithm introduced previously, in a way having an initial projection step on R\textsubscript{1}. But the difference between the two operators is two-fold: (i) it is not possible to switch to mobile join in some part of MDJoin, and (ii) it is not possible to probe hash table build from P with R\textsubscript{2} like in mobile semi-join.

### 3.3. Mobile dependent minus

In the presence of data sources with restricted access, the operation minus (R\textsubscript{1}, R\textsubscript{2}) is possible only if all the attributes of R\textsubscript{1} are mapped to ‘f’. Hence, in dependent minus, it is not possible to build the hash table with the relation R\textsubscript{2} since some attributes are mapped to ‘b’. As a consequence, the hash table is built with the relation R\textsubscript{1} forbidding any production in pipeline mode of the result. The mobile dependent minus algorithm is presented in Fig. 6.

As for union operator, it is not possible with restricted access because it needs tuples of R\textsubscript{1} not belonging to R\textsubscript{2} and vice versa.

### 4. Cost model

In the execution model presented previously, we supposed that each mobile agent had a cost model allowing choosing its execution site. In a large-scale distributed environment, it is not realistic to
Fig. 6. Mobile dependent minus algorithm.

```plaintext
if (not local (R1)) then receive (R1);
build (R1, HT); // compute statistics on R1
P ← project (R1, V);
Site ← decision (StatInfo, DataAvail, SyState);
if (not local (Site)) then migrate on Site;
if (not local (R2)) then
    {send P to S2; // DAcess runs on the remote site
     receive R2' from S2;}
else R2' ← DAcess(R2, P);
minus (HT, R2', T);
if (not local (T)) then send (T)
else materialize (T);
```

replicate, on all sites, a catalog: including all data profiles, the architecture descriptions of all sites and all the cost formulas. Hence, the catalog is distributed and partially replicated. This approach consists in maintaining, on each site S, a catalog which contains: the data profiles referenced by the submitted queries from S, the architecture characteristics of sites interacting with S and a part of the costs formulas.

The migration site is chosen by a mobile agent according to several metrics, called agent metrics. These metrics are: local production costs of the operand relations, the local execution cost of an agent, the migration cost of an agent, the transfer costs of the operands from the operand localization site to the execution site of an agent, and the transfer or writing cost of the result. These metrics are computed by the site where the agent is localized. The problem here is that the site hosting an agent does not contain, necessarily, all the necessary information to calculate its metrics (e.g. data profiles). For example, let us suppose that initially the agent executing the join \( T = \text{join} (R_1, R_2) \) was placed by the optimizer on the site \( S_1 \). The site \( S_1 \) hosting the agent does not know necessarily the \( R_2 \) profile located on \( S_2 \). So, to ensure the mobile agent autonomy and to avoid distant interactions with the site where the query was submitted, the agent has to supply to \( S_1 \) the data profiles of its operands, its migration space constituted by all the sites on which it can move, and the parameters involved in the formula estimating the production costs of the second operand (\( R_2 \)). These components are necessary for \( S_1 \) in order to compute the agent metrics. These components constitute the cost model embedded in an agent [17]. In [17] is described in detail the cost model of each site, the cost model embedded in an agent, as well as all the interactions between an agent and a site. These interactions, with information stamping, allow updating the catalog of site hosting agents.

### 4.1. Local DBMS cost model

The cost model [1,12,28,29] of a site estimates execution plan costs. It allows the optimizer to choose the optimal execution plan or closest to the optimal among various plans. This model is constituted
by three units: the relation profiles, the architecture characteristics, and the cost formulas. The DBMS catalog records this information, with the exception of the cost formulas, with the stamp of the last update. The stamping allows comparing the information embedded by an agent and the DBMS information, to use the most recent.

4.2. Embedded cost model

The cost model of a site and the cost model embedded into an agent allow an agent to choose its execution site. The embedded cost model is provided to the agent during the optimization phase. Each agent participating in the execution of a query moves with its embedded cost model. Hence, the embedded cost model has to be as small as possible to minimize the additional cost due to the agent migration. The embedded cost model consists of three units: (i) the migration space, (ii) the operand data profile, and (iii) the parameters of formulas estimating the operand production costs. An agent does not need to embed the cost formulas estimating its metrics because they are present on all the sites.

4.3. Interactions between an agent and a site

A site hosting a mobile agent computes the metrics which allow the agent to choose its execution site. So, an agent interacts with the site which hosts it. These interactions occur at two moments: during the installation phase and the decision phase.

4.3.1. Installation Phase

As soon as an agent ends its installation, it asks, to the hosting site, to estimate the following metrics: (i) the site workloads of the migration space, (ii) the availability of its operands, and (iii) the network bandwidth between the agent site and the sites of the space of migration. This estimation is made with a monitoring tool presented in [36]. In this step, an agent supplies to the site its migration space, the architecture characteristics of every site of its migration space and the site addresses producing its operands. After this step, an agent starts its execution.

4.3.2. Decision phase

During this phase, an agent interacts with the hosting site in two steps. In a first step, an agent asks the site to compute the profile of the result relation. For example, if the operator executed by an agent is a mobile hash join, then the profile of T is computed according to the R1 profile and of the stamped profile of R2 embedded in the agent. If the site also knows R2, thanks to the stamp, the site chooses the most recent profile between that embedded by the agent and that stored in its catalog. In a second step, the agent asks to the site to compute: (i) its execution cost on every site of its migration space, and (ii) the communication costs to receive operands and to send the result. The execution cost is computed with the formulas stored by the site and the architecture characteristics supplied by the agent. Naturally, if the site knows some sites of the migration space, it uses the most recent characteristics.

5. Performance evaluation

The objective of this section is to evaluate the performances of the direct mobile hash join operator and mobile multi-join queries. We concentrate on the join because it is the most widely used operator and because the behaviour of all the mobile relational operators is similar with the exception of the select and project operators.
To evaluate the performances of the mobile hash join, we compare its response times with those of the standard hash join executed on each site belonging to the migration space of the mobile hash join. As for the multi-join queries, we compare the execution of a mobile query with a fixed execution.

Our experiments are conducted in multi-user environments. So, the workload of an execution site can vary at any time like the network load. In the rest of this section, we describe the environment, the prototype and the results of our experiments.

5.1. Environment and prototype

To evaluate mobile hash join operator and mobile multi-join query performances, we consider that the relations $R_1$, $R_2$, $R_3$ and $R_4$ are located respectively on the sites $S_1$, $S_2$, $S_3$ and $S_4$. The number of tuples estimated by the optimizer are respectively 10 000, 20 000, 20 000 and 25 000. The join selectivity factor of each join between $R_i$ and $R_j$ is estimated to $\frac{1.5}{\max(|R_i|, |R_j|)}$ [31]. The average number of tuples in a 4KB disk page is 32 for every relation.

The performances of the mobile hash join operator are evaluated by the query $T = R_1 \bowtie R_2$ submitted from the site $S_1$ [21]. The component migration space of the cost model embedded inside the mobile hash join, noted $MJ$, is constituted by sites $S_1$ and $S_2$. The execution plan, generated by the optimizer from the estimations, consists in building the hash table from $R_1$ on $S_1$, then receiving $R_2$ on $S_1$ to probe the hash table. For this query, we compare:

1. the response time of $MJ$ execution which we will quote proactive execution,
2. the join response time of the standard execution on site $S_1$ and $S_2$ which will quote respectively execution on $S_1$, and execution on $S_2$.

We have conducted the same experiments in two different experimentation environments:

1. local environment: it is constituted of two workstations $S_1$ (HP) and $S_2$ (SUN) interconnected by a local area network,
2. large scale environment: it is constituted of two workstations $S_1$ (HP) and $S_2$ (SUN) interconnected by Internet. The first is located in France at Paul Sabatier University (Toulouse), it will be called Toulouse. The second is located in Turkey at EGE (Izmir), it will be called Izmir.

The performances of a mobile multi-join query are evaluated with the query $T = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$ submitted from site $S_5$. For the multi-join query, we compare only the performances in a local environment. Indeed, we do not possess enough sites connected on a large-scale network to realize this evaluation. Our experiment environment is constituted by 4 workstations SUN ($S_1$, $S_3$, $S_4$ and $S_5$) and of a workstation HP ($S_2$). We compare the performances of execution plans obtained from a bushy tree and from a right-deep tree (Cf. Fig. 7). We have chosen these two structures because they allow a better use of the parallelism with regard to a left-deep tree.

To conduct our experiments, we installed on every workstation a platform of mobile agents. In this one, every mobile agent runs on a Virtual Machine Java (JDK 1.4.2).

The costs of the network parameters and the migration cost of an agent are given in Table 1. The migration cost of an agent includes its serialization cost, its transfer cost and its de-serialization cost. Of course, when an agent migrates, it is added to its migration cost, the serialization cost, the transfer cost and the de-serialization cost of its hash table which are proportional in the size of this table.
5.2. Evaluation of mobile hash join operator

5.2.1. Experiments on a local environment

In this section, we describe the results of the experiments evaluated in a local environment. Here, we measure the impact on response time of an estimation error concerning (i) the number of tuples of $R_1$ noted $||R_{1es}||$ and (ii) the selectivity factor of a join noted $FS_{es}$. One estimation error on the selectivity factor influences the cardinality of $T$ ($||T|| = ||R_1|| * ||R_2|| * 1.5/\max(||R_1||, ||R_2||)$ and consequently its size because the average number of tuples by page is fixed. On the other hand, an error of estimation on the cardinality of $R_1$ acts on the size of $R_1$, the cardinality of $T$ and the size of $T$.

The curves of the Figs 8 and 9 show the measures of response time of the join by decreasing the cardinality of $R_1$ constated by MJ, $||R_{1cons}||$, with regard to $||R_{1es}||$ (Fig. 9) and the selectivity factor calculated by MJ, $FS_{cons}$, with regard to $FS_{es}$ (Fig. 10). From these curves, we observe that:

1. Whatever the variation found, the mobile hash join has approximately the same response time as the minimum of the response time between the standard hash join executed on $S_1$ and of this executed on $S_2$.
2. The behaviour of the mobile hash join in the Figs 8 and 9 is similar.

In Figs 8 and 9 respectively for a variation of $||R_{1cons}||$ with regard to $||R_{1es}||$ included between 0% and $-20\%$ and for a variation of $FS_{cons}$ with regard to $FS_{es}$ included between 0% and $-30\%$, the agent...
MJ executes the execution plan generated at compile time. On the other hand, when the variation of $|R_{\text{cons}}|$ with regard to $|R_{\text{res}}|$ is lower than $-20\%$ (lower than $-30\%$ for a variation of $FS_{\text{cons}}$ with regard to $FS_{\text{es}}$), the agent decides to modify this plan by moving on $S_2$ having built the hash table on $S_1$ because the volume of the hash table plus the revised volume of $T$ is lower than the volume of $R_2$.

For a variation of $|R_{\text{cons}}|$ with regard to $|R_{\text{res}}|$ (respectively of $FS_{\text{cons}}$ with regard to $FS_{\text{es}}$) included between $-40\%$ and $-90\%$, the Fig. 8 (respectively Fig. 9) shows some differences not proportional between response time of a proactive execution and response time of an execution on $S_2$. These two executions are identical but carried out at two different moments’ $t_1$ and $t_2$. This difference is explained by the workload of the execution site (memory available, number of active processes, number of connected users, etc.) which is different between the moments $t_1$ and $t_2$. For example, for a variation of $-80\%$ of $|R_{\text{cons}}|$ compared with $|R_{\text{res}}|$ (Fig. 8), the response time of the proactive execution is two seconds higher than the response time of the execution on $S_2$ just because of the workload of the site $S_2$.

5.2.2. Experiments on a large-scale environment

In this section, we make the same experiments as in Section 5.2.1, but we carry out these experiments in a large scale environment. In the experiments of the Figs 10 and 11, the join response times (execution on Toulouse, execution on Izmir, proactive execution) are measured by decreasing $|R_{\text{cons}}|$ with regard to $|R_{\text{res}}|$ (Fig. 10) and $FS_{\text{cons}}$ with regard to $FS_{\text{es}}$ (Fig. 11). In a large-scale environment, we make the same observations as those in a local environment, namely:
Fig. 10. Response time of a mobile hash join in a large-scale environment by decreasing $|R_{1cons}|$.

Fig. 11. Response time of a mobile hash join in a large-scale environment by decreasing $FS_{cons}$.

1. Whatever the observed variation, the mobile hash join has approximately the same response time as the minimum of response time between the standard hash join executed on Toulouse and the standard hash join executed on Izmir,

2. The behaviour of the mobile hash join in Fig. 10 and in Fig. 11 is similar.

Although, in our experiment environments (local or large-scale), the variation between $|R_{1cons}|$ and $|R_{1es}|$ and between $FS_{cons}$ and $FS_e$ has similar influences on the behaviour of the agents, the workload of an execution site does not have the same influence on response time in both experiment environments. Indeed, for a variation of $|R_{1cons}|$ with regard to $|R_{1es}|$ included between $-40\%$ and $-90\%$ (respectively $FS$), the Fig. 10 (respectively Fig. 11) shows few differences between the response times of a proactive execution and those of an execution on Izmir.

5.3. Mobile query evaluation

After the performance evaluation of mobile hash join operator, we evaluate those of a mobile query. This evaluation consists in comparing the response time of a mobile query with that of a fix query called standard query. We study the behaviour of a mobile query according to the tree structure used to generate the execution plan.
5.3.1. Bushy tree

In a bushy tree, the joins $J_1: T_1 = \text{Join}(R_1, R_2), J_2: T_2 = \text{Join}(R_3, R_4)$, and $J_3: T_3 = \text{Join}(T_1, T_2)$, are initially placed on the site $S_1$, $S_2$ and $S_5$ respectively. The curves of Figs 12, 13, 14 and 15 measure respectively the response times of a mobile query and a standard query by decreasing $||R_{cons}||$ with regard to $||R_{res}||$ in the cases 0\%, −30\%, −60\% and −90\% of error on $||R_{cons}||$ with regard to $||R_{res}||$. In Fig. 12, we observe that when the variation of $||R_{cons}||$ with regard to $||R_{res}||$ is included between 0\% and −30\%, the response time of the mobile query is superior by about 6\% to the response time of a standard query. However, when this variation is included between −30\% and −90\% the mobile query improves the response time with regard to the standard query from 5\% to 15\%. In Figs 13, 14 and 15 we observe that whatever the error on the $R_1$ size, the mobile query always improves the response time with regard to standard. This improvement is included between 10\% and 55\% in the best case.

5.3.2. Right deep tree

To evaluate the right-deep tree performances, the joins $J_1: T_1 = \text{Join}(R_1, R_2), J_2: T_2 = \text{Join}(R_3, T_1), J_3: T_3 = \text{Join}(R_4, T_2)$, are initially placed on the sites $S_1$, $S_3$ and $S_4$ respectively. As for the previous experiments, the curves of Figs 16, 17, 18 and 19 measure the response time of a mobile query
and a standard query by decreasing $|R_{1cons}|$ with regard to $|R_{1es}|$ in the cases 0%, −30%, −60% and −90% of error on $|R_{3cons}|$ with regard to $|R_{3es}|$. We notice in Fig. 16 that when the estimation error on $R_1$ is included between 0% and −30% the response time of the mobile query is superior at most of 6% to the response time of a standard query. We have same observation in the Fig. 17 when there is a −30% error on $R_3$ and when there is no error on $R_1$. In all other cases the mobile query improves the response time with regard to a standard query. In these experiments, the maximum improvement is 28%. This improvement is relatively weaker than that noticed with the bushy tree. This explains by the last join which builds a hash table with the most important of the base relations. Indeed, the building of hash table with $R_1$, $R_3$ and $R_4$ is made in parallel. So, the response time is conditioned by the building of the most important hash table.

5.4. Discussion

From the results of the mobile hash join on local and large-scale environments, we can make the following observations

1. The response time obtained with the mobile hash join operator is approximately equivalent at least response time of the standard hash join executed on each sites belonging to the migration space whatever is the experiment environment (local or large-scale),
2. the experiments, on a local environment, allowed to observe the sensibility of the response time of the mobile hash join operator to the execution site workload,

3. from the experiments, on a large-scale environment, we notice that the data transfer cost between both sites is the parameter the most important in the response time. So, the workload influence of the execution site on the response time is lesser with regard to the local environment.

Concerning the performance evaluation of the mobile queries, we noticed that for a relatively weak estimation error the mobility overhead compared to a standard execution is at most 6%. We noticed that the gains of the mobility can be at most 55%. These performance results on a local environment are encouraging and rest to confirm on a large-scale environment.

To confirm these observations, more exhaustive experiments must be tested. From our first results, we think that a migration policy allowing an agent to minimize, as much as possible, the transferred data volume is dominating on a large-scale environment. While, on a local environment, this migration policy has to take into account the execution site workload and the transferred data volume.
Fig. 18. Response time for a right-deep tree by decreasing $|R_{1cons}|$ with a $-60\%$ error on $|R_{3cons}|$.

Fig. 19. Response time for a right-deep tree by decreasing $|R_{1cons}|$ with a $-90\%$ error on $|R_{3cons}|$.

6. Related work

Many research works proposed dynamic query optimization methods. These methods can be classified in two categories: the centralized dynamic optimization methods [2,4–8,13,15,16,19,20,24,34,41] and the decentralized dynamic optimization methods [10,22,37,38]. In the centralized dynamic optimization methods, the control and the modifications of the execution plan are made by a centralized optimizer. This optimiser collects, at run time, information concerning the statistics on the data, the available resources and the communication costs. Then, from the collected information, it decides if an execution must be or not updated. All these methods have the capacity to detect the sub-optimality of the execution plans and to modify these execution plans to improve their performances. They allow the query evaluation process to be more robust to the optimizer errors and to the changes of the execution conditions during the query execution. However, this centralization does not allow adapting these methods to a large scale distributed environment because: (i) the number of messages is relatively important on a network with low bandwidth and strong latency, and (ii) the bottleneck formed at the optimizer, because all the messages converge on a single point.

The decentralized dynamic optimization methods proposed by [10,19,37,38] improve the local processing cost by adapting the use of the CPU, I/O and memory resources with the changes in the execution state (e.g. estimation errors, delays in data arrival rates). However, the methods proposed are focused mainly on the resources (CPU, I/O and memory) and do not take into account the network resource. In particular, these methods do not minimize the volume of data transferred on networks. The data transfers determined during compilation remain identical whatever the estimation errors detected at runtime.

The network resource is taken into account in the method proposed by Jones and Brown [22]. This method relies on mobile agents to decentralize the control and the modifications of execution plans.
Each local sub-query is executed by a mobile agent. An agent supervises the execution of its sub-query and can move to another execution site to minimize the transferred data volume. In this method, the interactions between the agents participating in the same query are based on message broadcasting. This broadcast allows making a decision with a global view on statistics of the query. However, it generates a congestion of the network when the number of agents becomes large. In a large scale distributed environment, the congestion of the network is quickly affected because of the strong latency and of the low bandwidth. In the proposed mobile relational algebra, we limit the control messages to avoid that the overhead inferred by these messages are bigger than that saved by the reduction of the data volume transferred.

7. Conclusion

The analysis of the dynamic optimization methods allowed us to notice that in a large-scale environment, it is necessary to decentralize the control and the execution plan modification. Furthermore, we have observed that it is important to limit the data and control messages to avoid the risk of congestion of the network due to the strong latency and of the low bandwidth in a large-scale distributed environment. In this objective, we proposed, a mobile algebra. We defined the operators without restricted access and with restricted access where the control of the execution plan is integrated within every operator. Every operator is autonomous by observing its execution environment and by choosing an execution site to migrate or not.

A site hosting a mobile operator does not contain, necessarily, all the necessary information to compute the values of the parameters used in its decision function (e.g. data profiles). So, we embedded inside the mobile operator, a cost model which allows to insure its autonomy and to avoid distant interactions with the site on which was submitted the query.

The experiment platform which we designed and developed, allowed us to evaluate the performances of the direct mobile hash join operator and mobile multi-join query. From this performance evaluation, we were able to make the following observations:

1. in our experiment environment, the direct mobile hash join chooses the site belonging to its migration space for which its response time will be minimum, independently of the estimation error on R₁ or on the selectivity factor,
2. the execution site workload influences significantly the response time of a direct mobile hash join in a local environment, while its influence is very weak in a large-scale environment,
3. the mobility overhead is at most 6% for a mobile query,
4. the gains of the mobility can be at most 55% for a mobile query in a local environment.

The performance evaluation results are encouraging and have to confirm by a more important evaluation campaign. This work can be completed by two following issues: (i) conducting an experience in a large-scale environment to analyze the performance of mobile queries and (ii) studying the behaviour of mobile relational algebra when the environment is unstable (i.e. a node can join or leave the system at any time).

Acknowledgements

This work is supported in part by the Ministry of Foreign and European Affairs and the Ministry of Higher Education and Research. Amadeus program 2010 (French-Austrian Hubert Curien Partnership – PHC). Grant Number 23066XC.
References

Abdelkader Hameurlain is full professor in Computer Science at Paul Sabatier University, Toulouse, France. He is a member of the Institute of Research in Computer Science of Toulouse (IRIT). His current research interests are in query optimization in parallel and large scale distributed environments, mobile databases, and database performance. Prof. Hameurlain has been the general chair of the International Conference on Database and Expert Systems Applications (DEXA’02). He is co-editor in chief of the International Journal “Transactions on Large-Scale Data- and Knowledge-Centered Systems” (LNCS, Springer). He was guest editor of two special issues of “International Journal of Computer Systems Science and Engineering” on “Mobile Databases” and “Data Management in Grid and P2P Systems”.

Franck Morvan received a PhD degree in Computer Science from Paul Sabatier University in 1994. He worked at Dassault Data Services society for 3 years before he joined Paul Sabatier University. He is currently full professor and member of the Institute of Research in Computer Science of Toulouse (IRIT). His main research interest are optimization in distributed and parallel databases, mobile agents and mobile computing.