Parachute Queries in the Presence of Unavailable Data Sources*

Philippe Bonnet
GIE Dyade
655 Avenue de l'Europe
38330 Montbonnot, France
Philippe.Bonnet@dyade.fr

Anthony Tomasic
INRIA Rocquencourt
78153 Le Chesnay, France
Anthony.Tomasic@inria.fr

Abstract

Mediator systems are used today in a wide variety of unreliable environments. When processing a query, a mediator may try to access a data source which is unavailable. In this situation, existing systems either silently ignore unavailable data sources or generate an error. This behavior is inefficient in environments with a non-negligible probability that a data source is unavailable (e.g., the Internet). In the case that some data sources are unavailable, the complete answer to a query cannot be obtained; however useful work can be done with the available data sources. In this paper, we describe a novel approach to mediator query processing where, in the presence of unavailable data sources, the answer to a query is computed incrementally. It is possible to access data obtained at intermediate steps of the computation. We define two new evaluation models and analytically model for these evaluation models the probability of obtaining the answer to a query in the presence of unavailable data sources. The analysis shows that complete answers are more likely in our two evaluation models than in a classical system. We measure the performance of our evaluation models via simulations and show that, in the case that all data sources are available, the performance penalty for our approach is negligible.

Keywords

Heterogeneous databases, Query Processing, Partial Evaluation, Unavailable Data

Résumé

Lorsqu'un médiateur traite une requête, dans un système de bases de données hétérogène, il est probable qu'il tente d'accéder à une source de données qui est temporairement indisponible. Dans cette situation, les systémes existants génèrent une erreur ou ignorent ces sources. Ce comportement est inefficace dans des environnements où il existe une probabilité non négligeable qu'un site soit indisponible (e.g., Internet). Dans le cas où des sources de données sont indisponibles, la réponse complète à une requête ne peut pas être produite ; cependant du travail utile peut être effectué avec les sources de données disponibles. Dans cet article, nous décrivons une nouvelle approche du traitement de requête dans un médiateur en présence de sources de données temporairement indisponibles. Nous définissons deux nouveaux modèles d'évaluation et fournissons pour chacun d'eux un modèle analytique qui représente la probabilité d'obtenir la réponse à une requête en présence de sources de données indisponibles. L'analyse montre que la probabilite d'obtenir une réponse est plus forte dans les deux modèles d'évaluation que nous proposons que dans un système classique. Nous mesurons les performances de nos modèles d'évaluation en utilisant une simulation ; nous montrons que, dans le cas où toutes les sources de données sont disponibles, les pertes de performances imputables à notre approche sont négligeables.

Mot Clés

Bases de Donnes hétérogènes distribuées, traitement de requêtes, évaluation partielle, données indisponibles

*This work has been done in the context of Dyade, a joint R&D venture between Bull and INRIA.

1 Introduction

Many current application environments use mediators (e.g., [19, 7, 1, 12, 4, 22]) to provide query access to a wide variety of heterogeneous data sources. Providing timely answers to queries in this environment is difficult due to the unpredictable response-time nature of data sources and of the interconnection network. Data sources become overloaded and networks become congested. Both can cease to function due to power loss, administrative operations, etc.

In cases where a data source or network does not respond sufficiently quickly, it can be considered unavailable. In such situations, when processing a query q, existing systems either silently ignore missing data or generate an error notification n (replicated data sources are considered in Section 7). In either case, to obtain the complete answer, the query must be resubmitted to the system and reprocessed from scratch. If some sources are unavailable, the system will again generate an error and again the query must be resubmitted. The complete answer a to a query will be generated only when all data sources are available. Thus, we can model the sequential interaction between the application program and the mediator as the following sequence of steps: $q, n, q, n, \ldots\ q, n, q, a$. We call this sequential model of interaction a *classical evaluation model*.

However, even when some data sources are unavailable, useful work can be done with the available data sources; a mediator can access, process and materialize their data. We call a representation of the mediator state at the point of notification a *partial answer* (the notification n contains the partial answer). The mediator uses its state to construct an *incremental query* i which is equivalent to the original query but cheaper to evaluate. The application program obtains the incremental query through the partial answer and then submits it to the mediator in order to get the complete answer. An example of a sequence for this model of interaction is $q, n_1, i_1, n_2, i_2, \ldots, n_k, i_k, a$. A different incremental query is used, in general, in each step of the sequence because the mediator makes partial progress towards the complete answer a depending on the sources that are available at each step.[1]

Incremental queries save work; in addition, the mediator state contains interesting information which may be useful for the user. The application program can extract information from the mediator state by submitting a secondary query, called a *parachute query* ρ. The answer to a parachute query, called a *parachute answer* α, can be computed given enough information in the mediator state. An example of a sequence of interaction is $q, n_1, \rho_1, \alpha_1, i_1, \ldots, n_k, i_k, a$. Note that parachute queries and incremental queries can be freely mixed. We call this model of interaction an *unconstrained evaluation model*. We use this term because the optimization of q is unconstrained by the knowledge of ρ.

The unconstrained evaluation model has several advantages: (i) it is easy to implement, (ii) parachute queries can be dynamically constructed by examining the partial answer, (iii) the plan used for the (original) query is always the optimal. However, this evaluation model has a disadvantage with respect to parachute queries because it cannot insure that the mediator state contains the information necessary to answer a parachute query.

We present in this paper a mediator which optimizes simultaneously the query and the parachute queries to insure that the mediator state contains the necessary information, assuming the appropriate data sources are available. An example of a sequence of interaction is $(q, \rho_1), (n_1, \alpha_1), (i_1, \rho_2), \ldots, a$. Note that parachute queries are submitted together with the original query. (This paper considers only a single parachute query.) The notification is followed by the parachute answers. Parachute queries can be submitted again with the incremental query. We call this model of interaction a *constrained evaluation model*. To help the intuition of the reader, we consider an example.

1.1 TPC-D example

Our example is based on the schema of the TPC-D benchmark. The schema consists of suppliers, parts, the relationship between suppliers and parts, nations, and regions. Consider a system where each base relation is located on a different data source. A possible conjunctive query over this schema, derived from the TPC-D query Q2, is *find all suppliers located in Europe which provide*

[1] This sequence is valid as long as the underlying data sources are not updated in a way that affects q. This assumption is a common one in mediator systems research and we use it throughout this paper.

a given part. In the following queries, attributes prefixed by S_ come from the SUPPLIER relation, N_ from the NATION relation, etc.

```
SELECT S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY,
       P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT
FROM SUPPLIER, PART, PARTSUPP, NATION, REGION
WHERE P_PARTKEY = PS_PARTKEY AND
      S_SUPPKEY = PS_SUPPKEY AND
      P_SIZE = 15 AND P_TYPE LIKE 'BRASS' AND
      S_NATIONKEY = N_NATIONKEY AND
      N_REGIONKEY = R_REGIONKEY AND
      R_NAME = 'EUROPE';
```

An interesting parachute query associated to this query is *all suppliers which provide a given part.*

```
SELECT S_ACCTBAL, S_NAME, P_PARTKEY, P_MFGR,
       S_ADDRESS, S_PHONE, S_COMMENT
FROM SUPPLIER, PART, PARTSUPP
WHERE P_PARTKEY = PS_PARTKEY AND
      S_SUPPKEY = PS_SUPPKEY AND
      P_SIZE = 15 AND P_TYPE LIKE 'BRASS';
```

Suppose the data sources containing the NATION or REGION relations are unavailable when the user asks the query. The system immediately notifies her that the query cannot be answered. The system however proceeds and obtains data from the other data sources for the SUPPLIER, PART and PARTSUPP relations. The system generates an incremental query that will efficiently compute the complete answer once the unavailable data sources are again available. (Reference [3] describes an incremental query for this example). The user submits the parachute query and the mediator returns the parachute answer. Clearly, this answer contains information that is interesting to the user. Once the unavailable data sources are again available, the user submits the incremental query. It retrieves data from these data sources and reuses data already obtained. Note that the incremental query and the parachute query are independent of each other.

1.2 Summary

In summary, we describe in this paper a novel approach to answering queries with unavailable data sources. Our approach is based on incrementally computing the answer to the query and permitting information to be extracted from the intermediate states of the computation. This approach leads to many interesting questions:

(1) What relevant information can be extracted from the mediator state, i.e., what are the interesting parachute queries? How to help the database programmer choose good parachute queries?

(2) How are queries evaluated in the constrained and unconstrained evaluation model? How are incremental queries constructed? How are parachute queries evaluated?

(3) How do the different evaluation models impact the availability of complete answers? What is the impact on performance? How likely is it that a parachute query can be answered?

In this paper, we attack these questions. In Section 2 we describe an intuitive class of parachute queries and demonstrate some interesting properties of this class. In Section 3 we detail the three evaluation models described above. In Section 4 we describe the algorithms which support the query processing shown in the example in the introduction. In Section 5 we describe our experimental framework (containing an analytical and a simulation model). In Section 6 we analyze the impact of our algorithms on the probability that an answer can be obtained given a sequence of interaction. In addition, we simulate the three evaluation models and analyze their performance characteristics. In Section 7 we discuss related work. Finally, in the last section we conclude the paper and discuss future work.

```
in: a conjunctive query Q1 with built-in
    predicates, a set of predicates L
out: a conjunctive query Q2

RemovePredicates(Q1, L) {
    Q2 := Q1

    for each l in L {
        if l appears in Q2 then
            Q2 := efface l from Q2
    }

    Efface from Q2 all built-in predicates where a
    non-range restricted variable appears.

    Efface from Q2 head all non-range restricted
    variables.

    Return Q2. }
```

Figure 1: The *RemovePredicates* function for Conjunctive Queries

```
in: a union query Q1, a set of predicate names L
out: a union query Q2

RemovePredicates(Q1, L) {
    Q2 := Q1

    for each l in L {
        if l appears in a rule, then
            Q2 := efface from Q2 the rule using l
    }
    Return Q2. }
```

Figure 2: The *RemovePredicates* function for Union Queries

2 Parachute Queries

The aim of a parachute query is to provide the user with relevant, useful data, in case the answer to a particular query cannot be computed. In Section 1 we showed an example of such a parachute query. However, not all parachute queries work well. To work well, first the mediator state must contain the necessary information for answering the parachute query. This problem is considered in detail in Section 4. Second, the set of sources needed to answer the parachute query must be different from the set of sources needed to answer the original query since, in the case that the set of sources are equal the system will simply answer the query and ignore the parachute query.

Given these restrictions, the application programmer is still faced with a daunting task: parachute queries must be semantically meaningful. To aid the programmer in the task of identifying interesting parachute queries, we define a *precipitate class* of parachute queries with respect to a query as follows: a parachute query is a (generalized) subset or superset of the original query. The intuitive connection is clear – the application programmer knows that the given parachute answer contains missing or extra tuples with respect to the complete answer.

If the parachute answer α is a subset or a superset of the query answer a for all possible databases, then containment [20] holds, i.e., $\rho \subseteq q$ or $\rho \supseteq q$. By *generalized* subset or superset, we mean that the projection of the parachute query and the original query are permitted to differ. More formally, let π_Q be the projection of the attributes of Q, then

Definition 1 (Generalized Subset) *Q' is a generalized subset of $Q \Leftrightarrow$ for any database D, $Q'(D) \subseteq \pi_{Q'}(Q(D))$.*

Definition 2 (Generalized Superset) *Q' is a generalized superset of $Q \Leftrightarrow$ for any database D, $Q'(D) \supseteq \pi_{Q'}(Q(D))$.*

Figure 3 shows an algorithm for the generation of parachute queries. All generated parachute queries belong to the precipitate class. The algorithm takes as input a query, a set of sources

```
in: a query Q, a mapping from predicate
    names to data sources M, a set of required
    sources S
out: a set of parachute queries PQ

pq-gen(Q, M, S) {
    V := use M to determine sources of Q
    for each configuration c in V − S {
        L := the available predicates derived
             from M, c and S
        PQ := PQ ∪ RemovePredicates(Q, L)
    }
    Return PQ. }
```

Figure 3: The *pq-gen* algorithm for generating parachute queries of a query.

required to be available, and a mapping from sources to predicate names. Given a set of sources available and unavailable (we call this set a *configuration of sources*) and the mapping, the set of available predicates can be identified.[2] The heart of the algorithm uses a function *RemovePredicates* that takes the set of available predicates and a query and generates a parachute query. This function is given in Figure 1 for conjunctive queries and Figure 2 for union queries. In these algorithms, a *range restricted variable* is a variable that appears in a non-built-in predicate in the body of rule. Given this algorithm, a tool which allows the application programmer to explore the precipitate class of parachute queries can easily be constructed. Investigation of other classes of parachute queries is future work.

Example 1 *Consider the query of employees, departments, and salaries greater than 10. Each predicate is mapped to a different data source.*

Query: $eds(X,Y,Z) \leftarrow e(X) \wedge ed(X,Y) \wedge es(X,Z) \wedge Z > 10$

Mapping: $\{e\} \rightarrow 1, \{ed\} \rightarrow 2, \{es\} \rightarrow 3$

Required Sources: $\{1\}$

The set of parachute queries are:

Available	*Parachute Query*
$\{1\}$	$pq_1(X) \leftarrow e(X)$
$\{1,2\}$	$pq_2(X,Y) \leftarrow e(X) \wedge ed(X,Y)$
$\{1,3\}$	$pq_3(X,Z) \leftarrow e(X) \wedge es(X,Z) \wedge Z > 10$
$\{1,2,3\}$	$pq_4(X,Y,Z) \leftarrow e(X) \wedge ed(X,Y) \wedge$ $es(X,Z) \wedge Z > 10$

3 Evaluation Models

In this section we describe in detail the three evaluation models mentioned in the introduction. The *classical* evaluation model represents existing systems which do not support parachute queries. This evaluation model requires almost no modifications to the mediator. The *unconstrained* evaluation model considers parachute queries after the evaluation of the query. This evaluation model requires only lightweight modifications to the interface and the run-time system of the mediator. The *constrained* evaluation model simultaneously optimizes the query and its associated parachute queries. This evaluation model requires modifications to the interface, optimizer and run-time system of the mediator.

Figure 4 shows the general evaluation model of these three systems. In the diagram for the classical evaluation model, (1) is the submission of the query, (2) is the notification that the query cannot be answered, (3) is the submission of the parachute query, (4) is the parachute answer, (5) is the re-submission of the original query, and (6) is the complete answer. This evaluation model represents existing mediator systems that do not support partial answers, or any form of materialization of intermediate results obtained when processing a query. Such a system has a classical cost based optimizer. It has no support for parachute queries. Parachute queries can be asked as follow-up queries and they are processed as all other queries. In case some data sources are unavailable, the original query is asked several times in order to obtain the complete answer.

[2] We assume that a predicate resides on a single source.

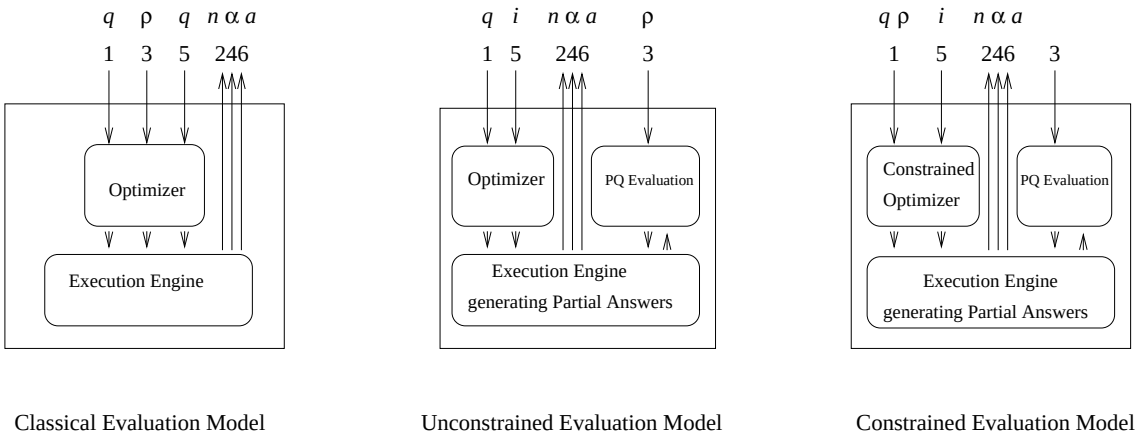

Figure 4: Evaluation Models of Three Representative Systems.

Queries and parachute queries are evaluated using the *evaluate* algorithm described in the next section.

The unconstrained system optimizes the query independently of the parachute queries. In the diagram for this evaluation model, (1) is the submission of the query, (2) is the partial answer, (3) is the submission of the parachute query, (4) is the parachute answer, (5) is the submission of the incremental query, and (6) is the complete answer. The optimizer is cost based, i.e. similar to the optimizer of the classical system. Queries and incremental queries are evaluated using the same evaluate algorithm. The incremental query is constructed by the mediator using the *construct* algorithm described in the next section. Parachute queries are evaluated using the *extract* algorithm described in the next section. This last algorithm only uses data materialized in the mediator.

The constrained evaluation model contains a constrained optimizer, described in the next section, that simultaneously optimizes a query and its parachute query. In the diagram for this evaluation model, (1) is the submission of the query and the parachute queries, (2) is the partial answer, (3) is the request for the evaluation of a particular parachute query, (4) is the parachute answer, (5) is the submission of the incremental query and its parachute queries, and (6) is the complete answer. The same algorithms as in the unconstrained evaluation model (evaluate, construct, and extract) are also used here.

4 Algorithms

4.1 Evaluate algorithm

The evaluate algorithm evaluates a query execution plan that has been generated by an optimizer for a query or an incremental query. The query execution plan is based on the Graefe iterator model[11]. The leaves of the plan are the sub-queries submitted to the data sources. The interior nodes are classical query processing operators such as join. In the case that all data sources are available, the algorithm computes the complete answer to the query. Otherwise it materializes part of the query execution plan in the mediator. The algorithm consists of two phases, a *sense* phase and an *execution* phase.

The sense phase is used to detect which data sources are available or unavailable. See [3] for an outline of the algorithm. This phase recursively descends the query execution plan in parallel along all sub-plans. When a sub-query is found, the corresponding data source is probed. If the data source responds within a *timeout* period, the source is considered available, otherwise it is unavailable. This phase examines all data source in parallel, thus overlapping the timeout wait on all data sources. This phase then recursively ascends the plan, marking as available an operator whose children are available, otherwise marking the operator as unavailable. After traversal of the plan finishes, the root operator of the plan has marked itself either available or unavailable.

For the execution phase, if the root operator is marked available, then all sources are available

and the final result is produced in the normal way. If at least one data source is unavailable, the root of the execution plan will be marked unavailable and the final result cannot be produced. In this latter case the execution phase proceeds via a second pass on the plan. This phase *materializes* some parts of the plan depending on a policy. Consider the plan $(A \bowtie B) \bowtie (C \bowtie D)$ where relations A, B and D are available. For the *nothing* policy, no materializations are performed. For the *maximal sub-plan* policy, each sub-plan rooted with an available operator materializes its result. Thus $(A \bowtie B)$ and D are materialized. For the *leaves* policy, each available leaf plan (containing the sub-query executed on the data source) materializes its result. Thus, A, B, and D are materialized. For the *shared component sub-query* policy, each sub-plan marked as a shared-component sub-query (cf. Section 4.4) is materialized. If $(B \bowtie C)$ is a parachute query, then the shared-component sub-queries are B and C. These materializations of sub-plans proceed in parallel. Note that this style of query execution is a form of query scrambling [2].

We assume in this paper that a data source which is marked available continues to operate in the execution phase. If an available data source becomes unavailable during the execution phase, an implementation would simply throw away the subplan which uses that data source. Also note that the parallel execution style is not critical to the issues of this paper – it simply results in better performance. However, as we shall see in Section 6, the materialization policy crucially affects several aspects of our work.

4.2 Construct algorithm

The construct algorithm constructs the incremental query from an execution plan and the mediator state. The execution plan is annotated with information such as the predicates used in joins, the attributes projected during a scan, etc. Each sub-plan that has been materialized, as described in the previous section, is annotated with the name of the temporary relation that stores the materialized data. The algorithm uses this information to construct a declarative query in a bottom-up fashion. This query is the incremental query. [3] shows the construct algorithm and a detailed example.

The incremental query that is constructed is equivalent to the original query. This ensures that the answer to the incremental query is exactly the same as the answer to the original query, under the assumption that no updates relevant to the query are performed on the data sources between the time the original query is submitted and the complete answer is computed. The incremental query, together with a handle to the execution plan, is returned as the partial answer to the query. The user interface is responsible for requesting the evaluation of the incremental query.

4.3 Extract Algorithm

The extract algorithm computes the answer to a parachute query using the materialized relations in the mediator state. The algorithm is a straightforward application of algorithms that answer queries using views (AQUV) [14]. These algorithms compute the answer to a query q using a set of views v. The result is a new query q' composed of some views in v and of a remainder query q'' that references base relations [3].

In our framework, we set q to be the parachute query ρ and the views v to be the sub-queries associated with the materialized relations in the mediator state. (The views are generated by applying the construct algorithm to the subplan associated to each materialized relation in the mediator state.) The AQUV algorithm is run to rewrite parachute query ρ into ρ' and ρ''. The remainder query ρ'' corresponds to accessing the data sources containing data that has not been materialized. Since the mediator does not access data sources during the processing of a parachute query, we permit the execution of the parachute query only if the remainder query ρ'' is empty. Thus, ρ' answers the parachute query using some combination of the materialized relations.

We have favored an approach where a parachute query is evaluated only against materialized data. Our primary reason is performance. As we shall see in Section 6, parachute queries evaluate

[3]The algorithm presented in [14] allows to find a minimal rewriting, i.e. the rewriting with the minimal number of literals

very quickly.[4] Permitting the evaluation parachute queries to access data sources, particularly in the case where the parachute query accesses data sources that are not involved in the original query, is future work.

4.4 Constrained Optimization

The constrained optimization algorithm takes as input a conjunctive query q, together with one associated parachute query ρ. The output of this constrained algorithm is an execution plan for q annotated with labels at the root of each shared component sub-queries (SCSQ), i.e., the sub-queries that are shared with the parachute query. These labels are used by the *evaluate* algorithm for the shared component sub-query materialization policy.

In our algorithm, the query q and the parachute query ρ are represented as conjunctive queries in Datalog [20]. The parachute query is thus:

$pq(Sacct, Sname, Ppk, Pmfgr, Sadd, Sphone, Scom) \leftarrow$
$s(Sacct, Sname, Ssk, Snk, Sadd, Sphone, Scom) \wedge$
$p(Ppk, Pkind, 15) \wedge$
$ps(Ppk, Ssk) \wedge$
$like(Pkind, \ 'BRASS')$

Our algorithm proceeds in four steps. The first step constructs the generalized shared sub-query (GSSQ) between the query and the parachute query. The GSSQ is essentially the most specific query whose body contains a subset of both the query and the parachute query. The second step constructs two groups of SCSQs based on the GSSQ. In the third step, the query and each group of SCSQs is used as an input to the AQUV algorithm to rewrite the query into a query q'. In the fourth step, a classical optimizer is invoked to determine the most efficient execution plan for each SCSQ and the associated rewritten query q'. The combination of plans with the lowest cost is chosen as the final plan. Each step is described in detail below.

Step 1 The GSSQ l is obtained as follows: (i) let l be the body of q; (ii) efface all literals in l whose predicate does not appear in a literal in the body of ρ; (iii) replace all variables or constants in l by new distinct variables. At this point, given the query and parachute query above, l is $p(X1, X2, X3) \wedge s(X4, X5, X6, X7, X8, X9, X10) \wedge ps(X11, X12) \wedge like(X13, X14)$.

We now refine l in order to obtain the GSSQ. Containment mapping c_q is constructed between the GSSQ and q, and c_ρ is constructed between the GSSQ and ρ. From these variable mappings, we deduce two sets of bindings b_q and b_ρ from c_q and c_ρ, respectively. The bindings are all equality relations and bindings of variables to constants. We obtain the GSSQ binding b as $b_q \cap b_\rho$. In our example, b_q is {X1 = X11, X2 = X13, X14 = 'BRASS', X3 = 15, X6 = X12} and $b_\rho = b_q$.[5] We apply b to l to obtain the GSSQ: $p(X1, X2, 15) \wedge s(X4, X5, X6, X7, X8, X9, X10) \wedge ps(X1, X6) \wedge like(X2, \ 'BRASS')$.

Step 2 From the GSSQ identified in the previous step, we construct two groups of shared component sub-queries (a shared component sub-query is a view composed of a head and a body). Note that considering only two groups of SCSQs is a heuristic. The first group of shared component sub-queries contains a single view whose body is the GSSQ and whose head is obtained with a new unique predicate symbol and the list of all variables that appear in the body and whose corresponding variables are needed to evaluate q. We obtain:

$\{scsq_1(X1, X4, X5, X7, X8, X9, X10) \leftarrow$
$p(X1, X2, 15) \wedge$
$s(X4, X5, X6, X7, X8, X9, X10) \wedge$
$ps(X1, X6) \wedge$
$like(X2, \ 'BRASS')\}$

The second group of SCSQ contains one view per literal (with any built-in predicate, if possible) appearing in the GSSQ. The body of each of these views is composed of one literal; their head is obtained with a new unique predicate symbol and the list of all variables that appear in the body needed to evaluate q. Thus, we obtain:

[4] Note that materialized data could also be used to evaluate subsequent queries. We consider that this is a separate area of research.

[5] The bindings are the same because the conditions and the joins in the parachute query appear in the query.

$\{scsq_2(X1) \leftarrow p(X1, X2, 15) \wedge like(X2,\ 'BRASS'),$
$scsq_3(X1, X2) \leftarrow ps(X1, X2),$
$scsq_4(X1, X2, X3, X4, X5, X6, X7) \leftarrow$
$s(X1, X2, X3, X4, X5, X6, X7)\}$

Step 3 For each group of shared component sub-queries, q is rewritten into a query q' that uses the group of SCSQ as views and a remainder query q''. The rewriting is accomplished using an AQUV algorithm [14]. For the first group, q is rewritten as

$q(Sacct, Sname, Nname, Ppk, Pmfgr, Sadd, Sphone, Scom) \leftarrow$
$scsq_1(Ppk, Sacct, Sname, Snk, Sadd, Sphone, Scom) \wedge$
$n(Snk, Nname, Nrk) \wedge$
$r(Nrk,\ 'EUROPE')$

Step 4 The optimizer is invoked once to generate the most efficient execution plan for each shared component sub-query in any group and the execution plan for the rewritten query q' of each group. The optimization of q' is done with respect to the *materialized* SCSQs in its group. The total cost of the group of SCSQ is the sum of the costs for computing the SCSQs and the costs for executing the associated q'. The group with the lower cost is chosen. The execution plan for q is obtained by *merging* the execution plan for q' and the execution plans for the corresponding SCSQ. The root of each shared component sub-query is labeled so that it can be recognized by the *evaluate* algorithm.

Thus, the constrained optimization algorithm identifies component sub-queries shared between q and ρ and generates the cheapest execution plan which contains either one SCSQ or a group of SCSQ, each of them being a leaf.

5 Experimental Environment

Our experiments are performed using an analytical model and a detailed simulation of the evaluation models introduced in Section 3 using a workload based on the query in the introduction. The analytical model is used to analyze the impact of the evaluation models on the likelihood that a query or a parachute query can be answered. The simulation is used to study classical response time and total work performance questions.

5.1 Analytical Model

As discussed in the previous sections, in the presence of unavailable data sources, a mediator needs several *trials* to obtain a complete answer. A trial corresponds to a mediator attempting to access several data sources. Each data source is either available or unavailable. An available data source can deliver data in a timely manner, an unavailable data source cannot. We model each trial to a data source as a uniformly random and independent event in which the data source is available with probability p and unavailable with probability $1 - p$. We model in this section, the three evaluation models of mediators introduced previously: classical, unconstrained and constrained.

We now express the probability that n sources are available simultaneously at least once in t trials. The probability that n data sources are available during a trial is p^n. The probability that not all n data sources are available during a trial is $1 - p^n$. For t trials, the probability that not all n data sources are available during a trial is $(1 - p^n)^t$. For t trials, the probability that, in at least one trial, all data sources are available is

$$1 - (1 - p^n)^t \tag{1}$$

Equation 1 represents the availability of complete answers in a classical evaluation model.

An unconstrained evaluation model materializes data from all available data sources in case some data sources are unavailable. When a query is issued, the mediator checks for t trials the availability of all n data sources and uploads the desired data from the *newly* available data sources at each trial. After t trials, a complete answer can be returned if data has been uploaded from all n sources. A data source only needs to be available once to participate in the complete answer.

We now express the probability that n sources are available at least once in t trials. The probability that a given source is never available in t trials is $(1-p)^t$ (since all trials are independent, we can consider either the data sources or the trials first). The probability that a given source is available at least once in t trials is $1-(1-p)^t$. The probability that all n sources are available at least once across t trials is

$$(1-(1-p)^t)^n \tag{2}$$

Equation 2 is the availability of complete answers in an unconstrained evaluation model. Note that Equation 1 and Equation 2 are equal if $p=0$ or $p=1$ or $t=1$ or $n=1$, as expected.

We have seen in Section 4.4 that the constrained optimizer identifies either one or several shared component sub-queries. This decision impacts the availability of complete answers. First, in case the constrained optimizer identifies one shared component sub-query, this shared component sub-query involves m of the n data sources contacted to obtain the complete answer. The shared component sub-query is materialized if the m data sources are available simultaneously.

When a query is issued to a constrained evaluation model, the mediator checks for t trials the availability of all n data sources. If on the t_0th trial, the shared component sub-query can be materialized, then a complete answer is returned whenever, in the remaining trials (including the current one, i.e. $t-t_0+1$ trials), the other $m-n$ data sources are available simultaneously.

$$\sum_{t_0=1}^{t}[(1-(1-p^m)^{t_0})-(1-(1-p^m)^{(t_0-1))}]\cdot(1-(1-p^{n-m})^{t-t_0+1}) \tag{3}$$

Equation 3 is the availability of complete answers in case a constrained evaluation model deals with one shared component sub-query. Details of this derivation are given in Reference [3].

In case the constrained optimizer identifies several shared component sub-queries, there are m shared component sub-queries involving one data source.

$$\sum_{t_0=1}^{t}[(1-(1-p)^{t_0})^m-(1-(1-p)^{t_0-1})^m]\cdot(1-(1-p^{n-m})^{t-t_0+1}) \tag{4}$$

Equation 4 is the availability of complete answers in case a constrained evaluation model deals with several shared component sub-queries, each involving one data source. Details of this derivation are given in Reference [3].

5.2 Simulation Environment

To study the performance of the algorithms producing partial answers, we have extended an existing simulator [9, 8] that models a peer-to-peer database system. We briefly describe, here, the simulator and present the extensions we have implemented to simulate the partial answers systems identified in Section 3.

5.2.1 Servers

Table 1 shows the main parameters for configuring the simulator and the settings used for this study. The mediator and the data sources are modeled as servers. A single mediator is connected to *NumSites* data sources. Each of the data source stores one base relation. The data sources are unloaded. (Delays from data sources are considered in [3].) The mediator can materialize temporary results on disk. Each server is characterized by a CPU whose speed is specified by the *Mips* parameter, *NumDisks* disks, and a main memory buffer pool of size *Memory*. Servers are connected via a network which is characterized by its bandwidth *NetBw*. The network is modeled as a FIFO queue. Although servers are configured with memory, base and materialized relations are always read from a server's disks, i.e., there is no caching across queries and relations are accessed once per query.

We extended this simulator by introducing the `evaluate` algorithm with different materialization policies as described in Section 4. In the sense phase, unavailable servers are modeled in a

Parameter	Value	Description
NumSites	3 or 4	number of data source servers
Mips	50	CPU speed (10^6 instr/sec)
NumDisks	1	number of disks per servers
DskPageSize	4096	size of a disk page (bytes)
NetBw	0.5	network bandwidth (Mbit/sec)
NetPageSize	4096	size of a network page (bytes)
Compare	4	instr. to apply a predicate
HashInst	25	instr. to hash a tuple
Move	2	instr. to copy 4 bytes
memory	2048	size of memory (disk pages)
time-out	10	time-out for sources (sec)

Table 1: Simulation parameters and main settings.

simple way. When a server representing a remote data source is contacted, it is either available or unavailable. If the server is available, it responds immediately. If the server is unavailable, mediator detects this fact after *time-out* seconds. For all the experiments, we have set the value of the time-out to 10 seconds. We use this value so that the behavior of the time-out can be clearly distinguished from other behavior in the simulation. An actual system would use a shorter time-out. In the execution phase, we assume that disk space is unlimited and that all intermediate results fit in memory and can be materialized on disk.

5.2.2 Query Optimizer

In the simulation of all three evaluation models a cost-based optimizer is used. Although the simulator implements hash join, where builds are done in parallel, we use a cost-based optimizer whose objective function is total work. This makes sense because we consider a slow network. The slow network essentially serializes the delivery of data from the data sources. The results we report in Section 6 do not include the time required for running the query optimizer, nor the time required for running the constrained optimization, nor the answering queries using views algorithm. Incremental queries are not re-optimized – they use the plan with the materialized results. This means that the incremental query results are more conservative than an actual system.

5.2.3 TPC-D Workload

We use the query and the parachute query presented in the introduction as the workload. In our experiment, each base relation is located on a separate data source. Select and Project operations are executed at the data sources and the mediator receives only the selected tuples.

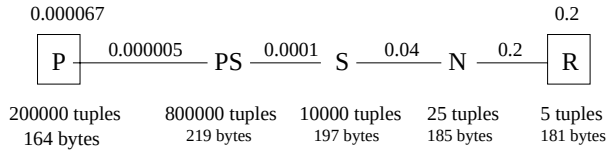

Figure 5: Query graph for query Q

The query is a 5-way join query, with selections on the `PART` and `REGION` relations. Figure 5 shows the query graph. Each relation is represented using the first letters of its name. The cardinality of a relation with the associated tuple size are listed below its abbreviated name. An edge between two relations indicates a join predicate between those relations in the query; the edge is labeled with the corresponding selectivity. Selection predicates are indicated by boxes containing the relation on which they are applied and the selectivity of the predicate is presented above the selection box.

The simulations use the parachute query given in the introduction. It belongs to the precipitate class and it is derived from the query by removing relations `NATION` and `REGION`.

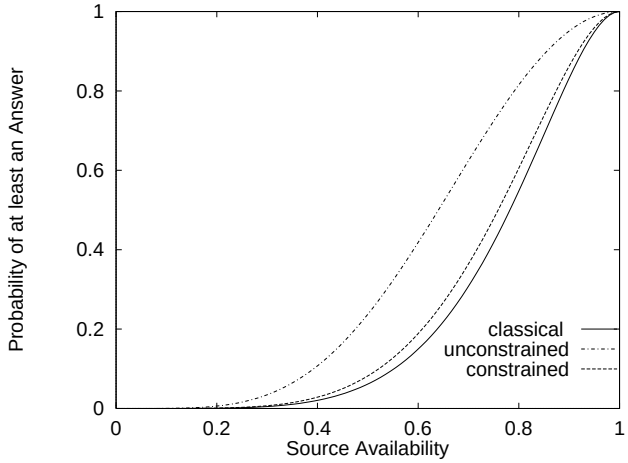

Figure 6: Probability of obtaining at least one complete answer in 2 trials for the TPC-D Workload

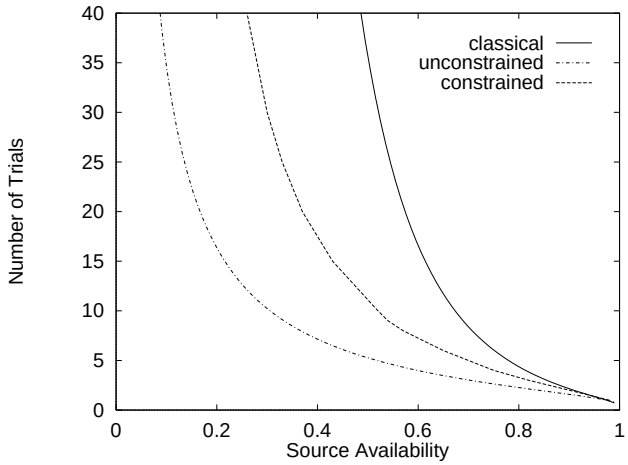

Figure 7: Numbers of trials required to have a probability 0.9 of obtaining a complete answer for the TPC-D Workload

6 Results

In this section, we study the influence of the optimization algorithms and of the materialization policies used in the `evaluate` algorithm on the availability of complete answers and response time.

6.1 Availability of Complete Answers

We use the analytical model of Section 5 to compare the availability of complete answers in the different evaluation models for the TPC-D Workload. This workload is characterized by a query which involves five data sources, and a parachute query which involves three of these data sources. Moreover, in the constrained evaluation model, an execution plan containing one shared component sub-query is chosen by the constrained optimizer.

Figure 6 plots the probability of obtaining at least one complete answer in two trials as a function of source availability. The curve for the constrained evaluation model is contained between the curve for the unconstrained evaluation model (upper bound) and the curve for the classical evaluation model (lower bound). The event of a classical evaluation model returning a complete answer in t trials is included in the event of a constrained evaluation model returning a complete answer in t trials which is itself contained in the event of an unconstrained returning a complete answer in t trials.

In Figure 6, the unconstrained evaluation model shows much better availability than the other

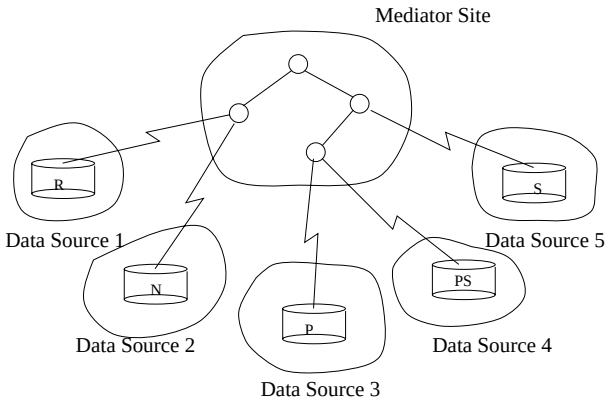

Figure 8: Execution plan for the query in the three evaluation models.

two evaluation models: for a source availability of 0.9, the probability of obtaining an answer is 0.96 in the unconstrained evaluation model, 0.89 in the constrained evaluation model and 0.83 in the classical evaluation model. The more data sources provide data which are materialized at each trial, the higher the availability of complete answer. As data is never materialized in the classical evaluation model, it has the lowest availability of complete answers. In the constrained evaluation model, data is only materialized if the three sources involved in the shared component sub-query are available. As a consequence the curve for the constrained evaluation model is close to the curve for the classical evaluation model. In the unconstrained evaluation model, as much data as possible is materialized at each trial. The probability of obtaining a complete answer is thus much higher in this evaluation model.

Figure 7 shows the number of trials that are required to get at least one complete answer with a probability of 0.9 as a function of source availability. When the source availability is low, say 0.6, the number of trials required to obtain a complete answer in the unconstrained evaluation model is still reasonable, almost 4 trials, while it is 7 trials in the constrained evaluation model and 16 trials in the classical evaluation model.

6.2 2 Trial Experiment

We use the simulator with the TPC-D workload to examine the influence of the optimization algorithm and of different materialization policies on response time using a realistic workload. Our experiment is based on enumerating all possible configurations of available and unavailable sources. For each configuration c, we use a sequence of interaction which submits the query, then the parachute query, and finally the incremental query. This sequence is achieved by (i) setting sources to be available or unavailable according to c, (ii) issuing the query, waiting for notification, issuing the parachute query, waiting for the parachute answer, (iii) changing all unavailable sources to available sources after the parachute answer is returned, and (iv) issuing the incremental query and waiting for the complete answer. Thus, the mediator attempts twice to answer the query and once to answer the parachute query. In the case that all sources are available, we use the sequence of interaction q, a.

To measure our experiments, we introduce several metrics. The *time to first answer* is the time between q and n, i.e., the time to execute the evaluate algorithm. The *time to incremental answer* is the time between i and a, i.e., the time to execute the evaluate algorithm on the incremental query. The *time to parachute answer* is the time between ρ and α, i.e., the time to execute the extract algorithm.

Classical Evaluation Model The sequence of interaction for the classical evaluation model experiment is q, n, ρ, α, i, a. When the query is submitted to the classical evaluation model, it is optimized. Figure 8 shows the execution plan which is chosen.[6] When the parachute query is

[6]We have also experimented with an optimizer whose objective function is to minimize response time. The execution plan chosen by this optimizer is a right linear tree, where the build phase of each hash join operator is performed in parallel. In our experiment, however, network bandwidth is low. As a result, the parallelism that appears in right linear trees cannot be exploited because the network serializes data access.

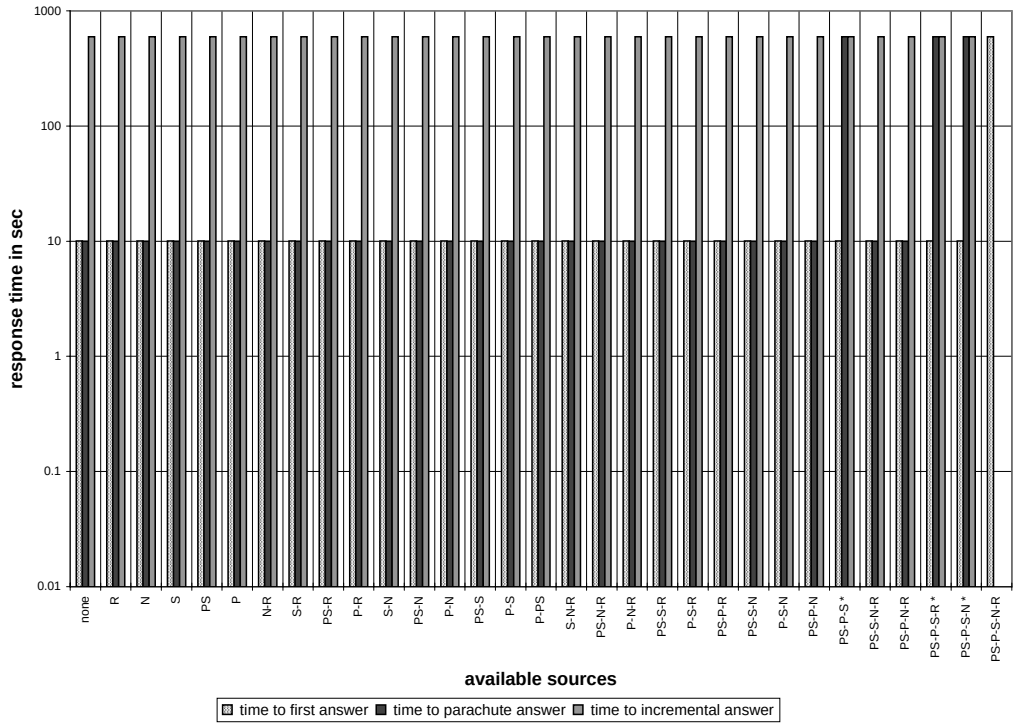

Figure 9: TPC-D Workload – Classical Evaluation Model

submitted, it is also optimized – its execution plan is $(P \bowtie PS) \bowtie S$. The classical evaluation model does not materialize relations. Thus, unless all sources are available, the time taken by the evaluate algorithm is essentially equal to the sense phase of this algorithm.

Figure 9 shows the results for the TPC-D workload with a classical evaluation model for each possible configuration of sources in the first trial. The x-axis indicates the configuration of available sources in the first trial and the y-axis indicates the query response time (on a logarithmic scale between 0.1 and 1000 seconds). In case all sources are available (P-PS-S-R-N), the query runs to completion and the first answer is the complete answer. The parachute query is not submitted. The time to first answer is 597.2 seconds.

The time to incremental answer is identical in all configurations where some sources are unavailable and it is equal to the time to complete answer, i.e. 597.2 seconds. Since no relations are materialized in the classical evaluation model, the incremental query is identical to the query and thus no work is saved between consecutive executions.

The time to first answer is identical, 10 seconds, in all configurations where some sources are unavailable. It corresponds to the time-out value in the simulator required to recognize a data source is unavailable. Since data sources are contacted in parallel, all time-outs are overlapped with each other. Note that this measurement is liberal since many mediator systems do not contact sources in parallel.

A parachute query is submitted in all configurations, except one where the complete answer is immediately returned. In the case that a parachute answer cannot be obtained, we report the time to parachute answer as the notification of this event. This time is equal to the time to first answer since exactly the same mechanism is used. A parachute answer is obtained for the three configurations where P, PS and S are available (these configurations are marked with a star on the x-axis). These answers are obtained because configuration of sources remains the same between the first trial and the execution of the parachute query. In these cases, the time to parachute answer is 597 seconds – slightly less than the time to complete answer because the tiny relations R and N do not participate in the parachute query.

Unconstrained Evaluation Model We use the same sequence of interactions with the unconstrained evaluation model that we used with the classical evaluation model: q, n, ρ, α, i, a. The

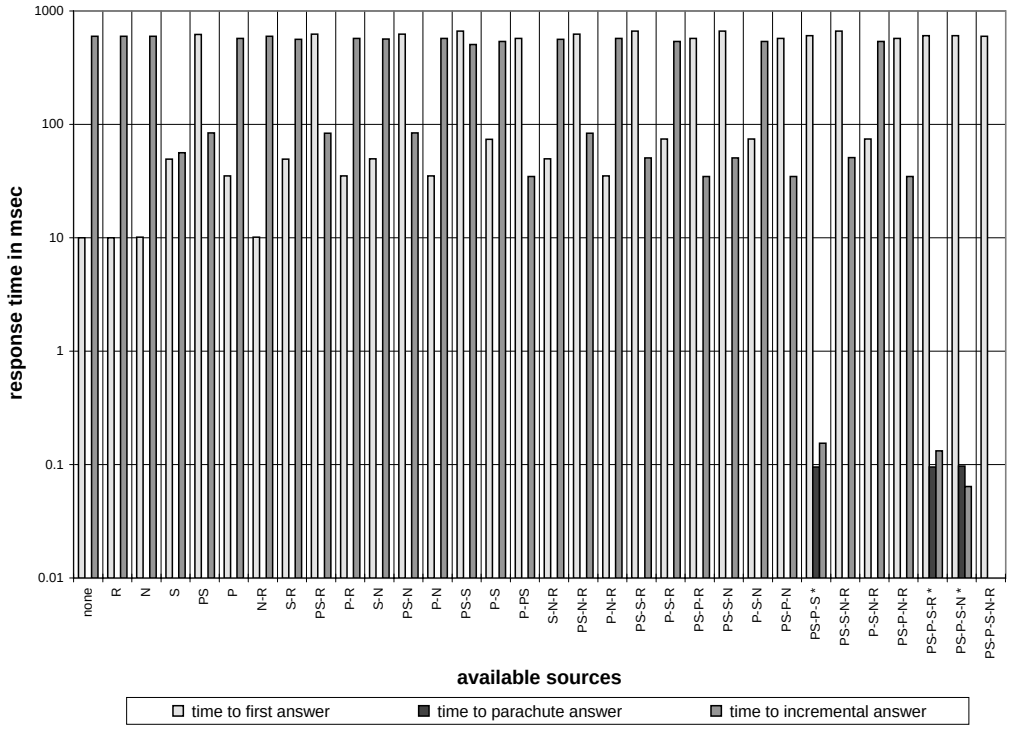

Figure 10: TPC-D Workload – Unconstrained Evaluation Model

execution plans chosen by the optimizer for q and ρ are the same as in the classical evaluation model (see Figure 8). However, the notification n returned by the mediator is a partial answer in case some sources are unavailable. Thus, the incremental query i is based on the mediator state. In particular i depends on the materialization policy.

For these simulations, the unconstrained evaluation model uses the *maximal available sub-query* policy (see Section 4). When evaluating a query tree, the *evaluate* algorithm first marks all available nodes and in a second pass materializes the maximal available subtrees into temporary relations.

Figure 10 shows the results for the TPC-D workload with an unconstrained evaluation model. In case all sources are available, the first answer is the complete answer, obtained in 597.2 seconds. The unconstrained evaluation model operates in the same way as the classical evaluation model for this case.

The time to first answer is dominated by the access to relation PS, which takes approximately 540 seconds. In cases where relation PS is unavailable in the first trial, the time to first answer is low (just above the *time-out* boundary of 10 seconds). In case relation PS is available in the first trial, the time to first answer is high (above 540 seconds). This time is even higher than the time to compute the complete answer in configurations PS, PS-R, PS-N, PS-S, PS-N-R, PS-S-R, PS-S-N, PS-S-N-R. In these cases, the time to first answer is the sum of the *time-out* required to recognize a source is unavailable, the time to access PS and other relations, plus the time to materialize PS and the other relations (PS is not joined in these configurations). In cases where PS and P are available together in the first trial, the join $P \bowtie PS$ can be performed with the *maximal sub-query* materialization policy that we have chosen. As a result relation PS is reduced and the time it takes to perform the join and materialize the result is lower than the time to materialize relation PS.

For the time to incremental answer, generally it holds an inverse relationship with the time to first answer. The materialization work done during the time to first answer makes the incremental answer cheaper to evaluate. The size of this inverse relationship depends on exactly how much work can be accomplished via joins and how many intermediate results must be materialized.

Parachute answers are provided in the configurations PS-P-S, PS-P-S-N and PS-P-S-R (these configurations are marked with a star on the x-axis). In all other cases, the algorithm for the evaluation of parachute queries, based on answering queries using views detects that the parachute

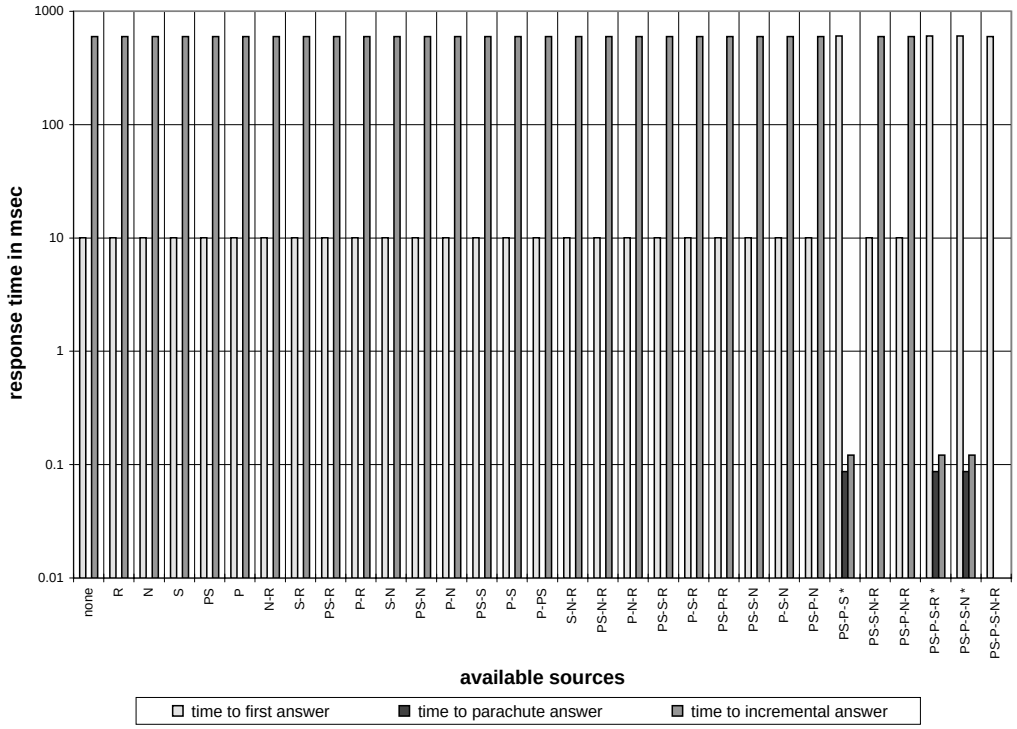

Figure 11: TPC-D Workload – Constrained Evaluation Model

query cannot be evaluated given the current mediator state. In those cases the time to parachute answer is reported as zero. The time to parachute answer is approximatively 0.09 seconds in all configurations where a parachute answer is provided. In those cases, the parachute query evaluation algorithm has recognized that $(P \bowtie PS) \bowtie S$ has been materialized. This time to parachute answer is thus the time to read a local relation of 30 pages. This time is very fast compared to the classical evaluation model which must evaluate the parachute query from scratch.

Constrained Evaluation Model The sequence of interactions for the constrained evaluation model is: $(q, \rho), n, r, \alpha, \alpha, (i, \rho), a$ since queries and parachute queries are issued together in this evaluation model. The symbol r represents the request for the parachute answer. In terms of timing, this request functions in a manner similar to the parachute query submission of the other two evaluation models. In our simulations, the time between notification and the request for the parachute answer is zero.

When (q, ρ) is submitted, the constrained optimization algorithm is applied. After the SCSQ and the corresponding remaining queries are optimized and their costs added, the execution plan chosen for the parachute query uses group containing the shared component sub-query. Surprisingly, the *same* execution plan as the classical and unconstrained evaluation model results from this optimization. This coincidence occurs because (i) the parachute query is contained in the query and (ii) the unconstrained optimizer joins exactly the three relations in the parachute query and in the same way as the constrained optimizer.

The materialization policy we have chosen to illustrate the constrained evaluation model is the *shared component sub-query* policy (cf. Section 4). When evaluating a query tree, the evaluate algorithm first marks all available nodes in the sense phase and then in the execution, materializes the shared component sub-query if it is available. If one of the relations involved in the shared component sub-query is unavailable, then the shared component sub-query is not materialized.

Figure 11 shows the results for the TPC-D workload with a constrained evaluation model. In the configuration where all sources are available in the first trial (P-PS-S-N-R), the first answer, which is the complete answer is obtained in 597.2 seconds, as in both previous evaluation models (the execution plans being the same). This situation is actually rare – typically the constrained optimizer does not generate the same execution plan as the unconstrained optimizer.

In configurations PS-P-S, PS-P-S-R and PS-P-S-N, the shared component sub-query can be materialized (it involves relations P, PS and S). As a consequence, the time to first answer is the time it takes to recognize a data source is unavailable plus the time to process and materialize the shared component sub-query.

In all other configurations, the shared component sub-query cannot be materialized, because one of the relations it involves is unavailable. In these cases, nothing is materialized. The time to first answer is thus the *time-out* value. As a consequence the incremental query which is constructed is identical to the original query. The time to incremental answer is thus similar to the time to complete answer. In these cases, the parachute query evaluation algorithm recognizes that the parachute query cannot be answered using the mediator state. We report the time to parachute answer in this case as zero.

7 Related Work

An alternative to our techniques in dealing with unavailable data sources is *replication*. Replication can increase the availability of all data sources to the point that queries almost always execute. However, note that parachute queries are completely compatible with replication – in the case that a data source is replicated, the probability that it will be unavailable is simply smaller.

Multiplex [17] tackles the issue of unavailable data sources in a multidatabase system and APPROXIMATE [21] tackles the issue of unavailable data in a distributed database. Both systems propose an approach based on approximate query processing. In presence of unavailable data, the system returns an approximate answer which is defined in terms of subsets and supersets sandwiching the exact answer. Approximation has been notably developed in [5], [6], [15].

Multiplex uses the notions of subview and superview to define the approximate answer. A view V1 is a subview of a view V2 if it is obtained as a combination of selections and projections of V2; V2 is then a superview of V1. These notions can be a basis to define the relationship between a query and its associated parachute queries. APPROXIMATE uses semantic information concerning the contents of the database for the initial approximation. In our context, we do not use any semantic information concerning the data sources. None of these system produce an incremental query for accessing efficiently the complete answer.

References [10] and [16] survey cooperative answering systems. These systems emphasize the interaction between the application program and the database system; they extend the basic scheme where the application program asks a precise query that the database system answers. Reference [16] identifies two classes of cooperative answering techniques. The first class of techniques aims at assisting users in the formulation of precise queries. The second class of techniques aims at providing meaningful answers in presence of incomplete or empty results. Parachute queries can be considered as a technique that aim at providing meaningful answers in presence of unavailable data sources.

Reference [13] attacks the problem of obtaining a complete answer from an incomplete database. A query is asked on a set of virtual relations. To each virtual relation R that contains all the tuples that should be in a relation, corresponds an available relation R' which contains the tuples that are actually in the relation. A constraint expresses the relationship between relations R and R'. If we consider that a virtual relation denotes a complete answer and that an available relation denotes a parachute answer, we can use the formalism introduced in [13] to refine the definition of relevant parachute queries. This only concerns parachute queries which are a subset of the original query.

The constrained optimization algorithm we have introduced in Section 4.4 is a multiple-query optimization algorithm. This problem has been studied in [18]. The author formulates the problem of multiple-query optimization as follows: given n sets of access plans (each set corresponds to all possible plans to evaluate a query) find a global access plan by *merging* n local access plans (one out of each set) such that the cost of this global plan is optimal. Our algorithm is constructed in order to maximize the probability of answering parachute queries and to minimize total work once the shared component sub-queries are materialized.

8 Conclusion

In this paper we have presented a novel method for dealing with queries in distributed heterogeneous database systems (mediators) which may access unavailable sources. The method is based on a combination of techniques. In the case that all sources are available, queries are evaluated in the normal way. In the case that some sources are unavailable, queries are evaluated in a way which obtains the maximum amount of information from available data sources. A representation of this work materialized in the mediator state, called the partial answer, is returned to the user. The user can then extract information from the mediator state using another query, called the parachute query. The parachute query is submitted to the mediator and the parachute query answer is extracted. In addition, the mediator constructs an incremental query using its state. The incremental query is resubmitted to the mediator to obtain the answer to the original query, assuming that the unavailable data sources are now available.

In this paper we have shown several results. We defined a sequential model of interaction with the database programmer. This model modifies the interface between the database system and the user program. We then gave a definition of a precipitate class of parachute queries. We described an algorithm for the generation of parachute queries that belong to the precipitate class. This algorithm is the basis for a tool which permits the database programmer to explore the precipitate class of parachute queries for a given configuration and query. (Required because there are, in the worst case, an exponential number of parachute queries in the precipitate class.)

We then proceeded to describe a collection of algorithms for dealing with queries and parachute queries in this environment. We described an evaluate algorithm which evaluates queries in two phases. The first phase senses the collection of available sources and the second phase evaluates the query according to some materialization policy. We described a construct algorithm which gives the incremental query for a partial answer. This algorithm translates algebraic representation of query execution into an equivalent declarative representation. We described an extract algorithm which computes the parachute query answer. This algorithm matches (via query sub-query matching) the parachute queries with the sub-queries representing the materialized relations in the partial answer.

To test the viability of our work, we defined three evaluation models for implementing parachute queries. The classical evaluation model implements parachute queries in the user interface. This evaluation model requires no modifications to the mediator. The unconstrained evaluation model implements parachute queries on partial answers. This evaluation model requires only lightweight modifications to the interface and the run-time system of the mediator. The constrained evaluation model simultaneously optimizes a query and its associated parachute queries. This evaluation model requires modifications to the interface, optimizer and run-time system of the mediator.

We then analytically analyzed the availability of query and parachute query answers in the three evaluation models. We showed that availability of the query answer depends on the probability that a source is available, the number of sources accessed by the query, the materialization policy and the evaluation algorithm.

To show the performance impact of our work, we simulated the three evaluation models. We defined several new performance metrics to compare the performance of the three evaluation models. These performance metrics are based on the classical query response time metric. We simulated the classical evaluation model as a baseline for comparison to the other evaluation models.

We simulated the unconstrained evaluation model and demonstrated that parachute query extraction and incremental query evaluation response times are much faster than in the classical evaluation model. This performance improvement is due to the materialization policy.

We simulated the constrained evaluation model and demonstrate that query evaluation, parachute query extraction and incremental query evaluation are nearly as fast as in the unconstrained evaluation model, and that the performance penalties are small in most cases. The constrained evaluation model may have a negative impact on the availability of complete answers. This impact results from the materialize shared component sub-queries policy. Thus, we conclude that there is a trade-off between performance and availability of queries and parachute queries.

The reader is encouraged to consult reference [3]. In this reference we show the impact of the materialization policy on the availability of the parachute answer and the impact of delays from data sources. We also run a collection of experiments on a synthesized workload where the

constrained optimizer chooses a different execution plan than the unconstrained optimizer, thus we show the impact of parachute queries on the optimizer.

Acknowledgments The authors wish to thank Laurent Amsaleg, Stéphane Bressan, Mike Franklin, Rick Hull, Tamer Özsu, Louiqa Raschid and Dennis Shasha for interesting discussions on the subject of this paper. Helena Galhardas, Olivier Lobry and João Pereira helped debug versions of this paper.

References

[1] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD International Conference on Management of Data*, pages 137–148, Montreal, Canada, 1996.

[2] L. Amsaleg, Ph. Bonnet, M. J. Franklin, A. Tomasic, and T. Urhan. Improving responsiveness for wide-area data access. *Bulletin of the Technical Committee on Data Engineering*, 20(3):3–11, 1997.

[3] Ph. Bonnet and A. Tomasic. Parachute queries in the presence of unavailable data sources. INRIA Technical Report, 1998. In preparation.

[4] S. Bressan, C.H. Goh, et al. The COntext INterchange mediator prototype. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[5] P. Buneman, S. Davidson, and A. Watters. Querying independant databases. *Information Sciences*, nov 1989.

[6] P. Buneman, S. Davidson, and A. Watters. A semantics for complex objects and approximate answers. *Journal of Computer and System Sciences*, 43, 1991.

[7] C.M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proceedings of the 4th International Conference on Extending Database Technology*, 1994.

[8] S. Dar, M.J. Franklin, B.T. Jònsson, D. Srivasta, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22nd International Conference on Very Large Databases*, Bombay, India, 1996.

[9] M.J. Franklin, B.T. Jònsson, and D. Kossmann. Performance tradeoffs for client-server query processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montréal, Canada, 1996.

[10] T. Gaasterland, P. Godfrey, and J. Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992.

[11] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[12] A. Y. Levy, A. Rajaraman, and J. J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proceedings of the 22nd International Conference on Very Large Databases*, Bombay, India, 1996.

[13] A.Y. Levy. Obtaining complete answers from incomplete databases. In *Proceedings of the 22nd International Conference on Very Large Databases*, Bombay, India, 1996.

[14] A.Y. Levy, A. Mendelzon, Y. Sagiv, and D. Srivasta. Answering queries using views. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, PODS-95*, San Jose, California, 1995.

[15] L. Libkin. Approximation in databases. In *Proceedings of the International Conference on Database Theory*, 1995.

[16] A. Motro. Cooperative database systems. In *Proceedings of the 1994 Workshop on Flexible Query-Answering Systems (FQAS '94)*, pages 1–16, 1994.

[17] A. Motro. Multiplex: A formal model for multidatabases and its implementation. Technical Report ISSE-TR-95-103, George Mason University, 1995.

[18] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, March 1988.

[19] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous database and the design of DISCO. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS-16)*, pages 449–457, Hong Kong, May 1996. IEEE Computer Society Press.

[20] J. D. Ullman. *Principals of Database and Knowledge-Base Systems*, volume 2. Computer Science Press, 1989.

[21] S. V. Vrbsky and J. W. S. Liu. APPROXIMATE: A query processor that produces monotonically improving approximate answers. *Transactions on Knowledge and Data Engineering*, 5(6):1056–1068, December 1993.

[22] G. Wiederhold. Intelligent integration of information. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 434–437, Washington, D.C., 1993.