Parachute Queries in the Presence of Unavailable Data Sources

Philippe Bonnet, Anthony Tomasic

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Parachute Queries in the Presence of Unavailable Data Sources

Philippe Bonnet and Anthony Tomasic

No 3429

Mai 1998

THÈME 3

ISSN 0249-6399

Parachute Queries in the Presence of Unavailable Data Sources

Philippe Bonnet* and Anthony Tomasic†

Thème 3 — Interaction homme-machine,
images, données, connaissances
Projet Rodin

Rapport de recherche n ° 3429 — Mai 1998 — 61 pages

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : 01 39 63 55 11 - International : +33 1 39 63 55 11
Télécopie : (33) 01 39 63 53 30 - International : +33 1 39 63 53 30

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. In either case, to obtain the complete answer, the query is reprocessed from scratch. 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 a *partial answer*. The partial answer represents the state of the mediator at the end of query processing, i.e., materialized data. This state is used to construct an *incremental query*. The answer to the incremental query is the same as the complete answer, but it is more efficient to evaluate than the original query. In addition, information can be extracted from the mediator state through the use of secondary queries, called *parachute queries*. We describe an intuitive class of parachute queries and an algorithm which generates it. We define two new evaluation models for partial answers, incremental and parachute queries 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. In addition, we show that there is a trade-off between the cost of query execution and the probability of obtaining a complete or parachute query answer.

Key-words: Heterogeneous databases, Query Processing, Partial Evaluation, Unavailable Data

(Résumé : tsvp)

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

* Bull, GIE Dyade - Address: INRIA Rhone Alpes, 655 Av de l'Europe, 38330 Montbonnot, France. e-mail: Philippe.Bonnet@inrialpes.fr, http://sirac.inrialpes.fr/ pbonnet

† Address: INRIA Rocquencourt, 78153 Le Chesnay, France. e-mail: Anthony.Tomasic@inria.fr, http://rodin.inria.fr/person/tomasic

Requêtes Parachutes en présence de sources de données indisponibles

Résumé : Des systèmes de bases de données hétérogènes sont aujourd'hui deployés dans un grand nombre d'environnement. Lorsqu'un médiateur traîte une requête, il est probable qu'il tente d'accéder à une source de données qui est indisponible. Dans cette situation, les systémes existants génèrent une erreur ou ignorent les sources de données indisponibles. Dans les deux cas, pour obtenir la réponse complète, le traitement de la requête est effectué entièrement à nouveau. 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 où, en présence de sources de données indisponibles, la réponse à une requête est une *réponse partielle*. Une réponse partielle représente l'état du médiateur à la fin du traitement de la requête, i.e. un ensemble de données matérialisées. Le médiateur utilise son état pour construire une *requête incrémentale*. La réponse à la requête incrémentale est identique à la réponse complète, et elle est plus efficace à évaluer que la requête initiale. De plus, des données peuvent être extraites de l'état du médiateur grâce à l'utilisation de requêtes secondaires, appelées *requêtes parachutes*. Nous décrivons une classe intuitive de requêtes parachutes et un algorithme pour la générer. Nous définissons deux nouveaux modèles d'évaluation pour supporter les réponses partielles, les requêtes incrémentales et les requêtes parachutes. Nous décrivons un modèle analytique qui pour chacune de ces modèles d'évaluation 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 simultation ; nous montrons que, dans le cas où toutes les sources de données sont disponibles, les pertes de performances imputables à notre approche sont négligeables. De plus, nous montrons l'existence d'un compromis entre le coût de l'exécution d'une requête et la probabilité d'obtenir une réponse à la requête initiale ou à une requête parachute.

Mots-clé : Bases de Données hétérogènes distribuées, traitement de requêtes, évaluation partielle, données indisponibles

1 Introduction

Many current application environments use mediators (e.g., [23, 8, 1, 16, 6, 26]) 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*.

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. When the original query cannot be answered, a parachute query makes up for it by allowing the user to retrieve useful information. [2] 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*. We use this term because the optimization of q is unconstrained by the knowledge of ρ.

[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.

[2] In the case that insufficient information is available to answer the parachute query, we return an error. We do not consider incremental parachute queries in this paper.

The unconstrained evaluation 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 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*. To help the intuition of the reader, we consider two examples.

1.1 Examples

1.1.1 Resource Management

Consider a consulting company which operates throughout the world. Each country contains an independent data source which records information about the current consulting jobs, the number of persons involved, etc. At the headquarters, this information is needed regularly to monitor risk and resource management. The organization of the schema of each data source is different. For each data source i, the mediator contains a view v_i which computes a horizontal partition of a virtual relation r. To compute r, the mediator computes a union query $q = \bigcup_i v_i$ over all the views.

In the classical evaluation, to compute the answer to this query requires resubmitting q until all data sources are available. In either the unconstrained or the constrained evaluation, when some data sources are unavailable the v_j that access available data sources can be processed and materialized in the mediator state. The user is notified that the complete answer could not be obtained; the notification indicates which data sources were unavailable. The incremental query which is constructed is a union over the materialized views and the views that access the data sources that have been unavailable so far. All views that have been materialized are interesting for the user in charge of risk and resource management. Interesting parachute queries ρ are either independent views or the union over all materialized views.

1.1.2 TPC-D example

Our second 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 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 *find 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';
```

We consider, in this example, an unconstrained evaluation. Suppose the data sources containing the NATION or REGION relations are unavailable when the user asks the query. The system immediately recognizes that the query cannot be answered. It 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. (Section 4.2 describes an incremental query for this example). A notification is sent to the user to inform her that the complete answer could not be computed; this notification contains a handle on the incremental query. After the user has received the notification, she 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, which are now available, 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? 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 a precipitate 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. As an aid to the reader, Appendix A summarizes the terms introduced in this paper.

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 [24] 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, $Q'(D) \supseteq \pi_{Q'}(Q(D))$.*

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

in: a conjunctive query Q_1 with built-in
predicates, a set of predicates L
out: a conjunctive query Q_2

```
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 2: The *RemovePredicates* function for Conjunctive Queries

Figure 1 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 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.[3] 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 2 for conjunctive queries and Figure 3 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.

In Appendix B, we show (i) that the set of parachute queries generated using the *RemovePredicates* function for conjunctive queries forms a lattice together with the generalized superset relation, and that the set of parachute queries generated using the *RemovePredicates* function for union queries forms a lattice together with the subset relation. This means that the query generated by *pq-gen* given a conjunctive query (resp. a union query) and a fixed configuration of sources is maximal in the generalized superset ordering (resp. subset ordering), i.e. the query generated by *pq-gen* is as close as possible from the original query in the generalized superset ordering (resp. the subset ordering).

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

Interestingly, the *combination* of the ideas of *RemovePredicates* for conjunctive queries and union queries does not stay within the precipitate class, i.e., applying the *RemovePredicates* function for conjunctive queries to each rule of a union query.

Example 2 *The incompatibility of conjunctive queries and union queries.*

Query q: $p(X) \leftarrow q(X) \wedge r(X)$ *union* $p(X) \leftarrow q(X) \wedge s(X)$

Mapping: $\{q\} \rightarrow 1$, $\{r\} \rightarrow 2$, $\{s\} \rightarrow 3$

Configuration: $\{2\}$ *available,* $\{1,3\}$ *unavailable*

The resulting parachute query ρ *is:* $p(X) \leftarrow r(X)$.

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

in: a union query Q_1, a set of predicate names L
out: a union query Q_2

```
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 3: The *RemovePredicates* function for Union Queries

For the database $\{q(a), r(a), s(a)\}$, $q \subseteq \rho$, *but for the database* $\{q(a), s(a)\}$, $q \supseteq \rho$.
q: $q(X) \leftarrow redcar(X) \wedge price(X, 10000)$ *union* $q(X) \leftarrow whitecar(X) \wedge price(X, 10000)$
Mapping: $\{redcar\} \rightarrow 1$, $\{whitecar\} \rightarrow 2$, $\{price\} \rightarrow 3$
Configuration: $\{1\}$ *available,* $\{2, 3\}$ *unavailable*
The resulting ρ *is:* $pq(X) \leftarrow redcar(X)$.
Neither $\rho \subseteq q$ *nor* $\rho \supseteq q$ *hold.*

Thus, in this last example, the parachute query does not belong to the precipitate class, by our definition.

3 Evaluation Models

In this section we describe in detail the three evaluation models mentioned in the introduction. The *classical* evaluation represents existing systems which do not support parachute queries. This evaluation model requires almost no modifications to the mediator. The *unconstrained* evaluation 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 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 models of these three systems. In the diagram for the classical evaluation, (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.

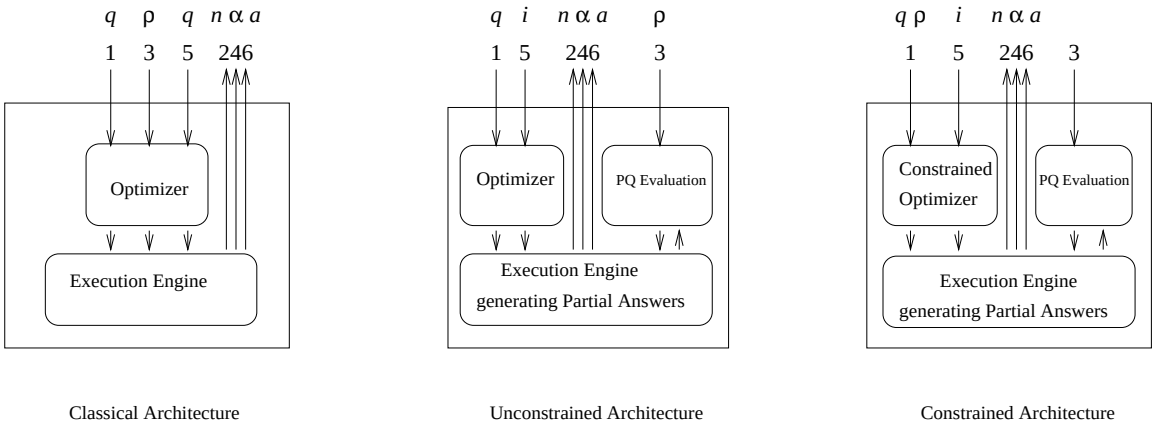

Figure 4: Evaluation Models of Three Representative Systems.

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. 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 is based on 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 for the unconstrained evaluation (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[14]. 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 Figure 5 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 sub-plan 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.

in: a query execution plan rooted at *operator*
out: an annotated query execution plan

```
sense(operator) {
    for all subplan in children of operator {
        sense(subplan)
    }
    if (source is available) or
        (all children are available) then {
        mark operator available
    } else {
        mark operator unavailable
    } }
```

Figure 5: The sense phase of the evaluate algorithm. The **for all** expression evaluates its body in parallel.

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. The algorithm uses this information to construct a declarative query in a bottom-up fashion.

Figure 6 shows the construct algorithm. The algorithm returns a 4-tuple of the projection list, the from list, the where list, and a mapping. From the first three lists an SQL or OQL query can readily be constructed. This query is the incremental query. The mapping function *map* in the algorithm is used to rename variables so that the resulting expression references the materialized relations instead of the original relations. Materialized sub-plans generate a declarative expression that accesses the materialized intermediate result. It is an expression of the form r where r is the name of the temporary relation holding the materialized intermediate result. [4]

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] An alternative implementation generates the expression `bag(`t`)`, where t is the list of tuples in the materialized intermediate result. If t is small, some optimizers may perform better with this implementation when optimizing the incremental query.

in: an annotated execution plan rooted at *operator*
out: a 4-tuple (P, F, W, M)

```
construct(operator) {
    P := the projection of operator
    W := the selection of operator
    if operator is materialized then {
        x := a unique variable
        X := the materialized relation identifier
        V := the set of variables appearing in the
            sub-query for operator
        return (P, {(x, X)}, ∅, (V, x))
    } else if operator is a join then {
        (P1, F1, W1, M1) := construct(left(operator))
        (P2, F2, W2, M2) := construct(right(operator))
        M := M1 ∪ M2
        return (map(M, P1 ∪ P2), F1 ∪ F2,
            map(M, W) ∪ W1 ∪ W2, M)
    } else if operator is a project then {
        (P1, F1, W1, M1) := construct(child(operator))
        return (P, F1, W1, M1)
    } else if operator is a select then {
        (P1, F1, W1, M1) := construct(child(operator))
        return (P1, F1, W ∪ W1, M1)
    } else operator is a scan then {
        x := a unique variable
        X := the relation identifier of operator
        V := the variable appearing in the scan operator
        return (P, (x, X), ∅, (V, x))
        } } }
```

Figure 6: Construction of the incremental query. The *map* function replaces variable occurrences according to the given mapping.

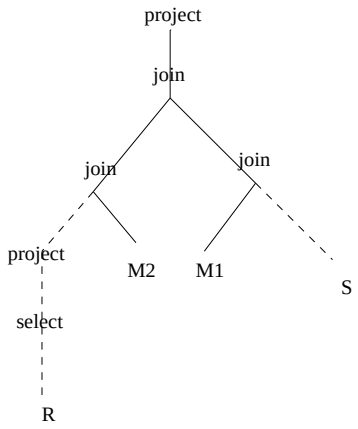

Figure 7: Annotated Execution Plan for the query

Example 3 *Consider the execution plan of Figure 24 (presented in Section 6) for the query presented in the introduction and the configuration of sources P-PS-N, where sources P, PS and N are available and sources S and R are unavailable.*

The annotated execution plan resulting from the sense phase of the evaluate algorithm is used as input for the construct algorithm. Figure 7 shows this annotated execution plan. The solid lines represent links between operators within the mediator, dashed lines represent links between operators executed on the data sources. M1 and M2 are the materialized relations that store N and $P \bowtie PS$ *respectively.*

The result of the construct algorithm applied to this annotated execution plan is a 4-tuple composed of a projection list, a list of relation (from clause), a list of conditions (where clause) and a mapping. The projection list is obtained from the project operator at the root of the annotated execution plan: {S_ACCTBAL, S_NAME, N_NAME, P_PARTKEY, P_MFGR, S_ADDRESS, S_PHONE, S_COMMENT}. *The list of relations is constructed recursively: each relation name is obtained from a scan operator or a materialized relation, a union is performed for each join operator in the execution plan. The list of relations is (*{(x1, REGION)} ∪ {(x2, M1)}*)* ∪ *(*{(x3, M2)} ∪ {(x4, SUPPLIER)}*). The list of conditions is also obtained recursively from the select and join operators:* {N_NATIONKEY = S_NATIONKEY} ∪ *(*{R_NAME = "EUROPE"} ∪ {R_NATIONKEY = N_NATIONKEY}*)* ∪ {PS_SUPPKEY = S_SUPPKEY}. *The mapping information is initialized in the materialized or scan operators. In the materialized operators a new variable is mapped to one or several original variable names depending on the number of relations that have been processed to produce the materialized relation. In the scan operator a new variable is mapped to one original relation. In our example, the mapping list is* {(x1,REGION), (x2, NATION), (x3, {PART, PARTSUPP}), (x4, SUPPLIER)}.

The incremental query that is constructed using the 4-tuple produced by construct is

```
SELECT x4.S_ACCTBAL, x4.S_NAME, x2.N_NAME,
       x3.P_PARTKEY, x3.P_MFGR, x4.S_ADDRESS,
       x4.S_PHONE, x4.S_COMMENT
FROM x1 REGION, x2 M1, x3 M2, x4 SUPPLIER
WHERE x2.N_NATIONKEY = x4.S_NATIONKEY AND
      x1.R_NAME = 'EUROPE' AND
      x1.R_NATIONKEY = x2.N_NATIONKEY AND
      x3.PS_SUPPKEY = x4.S_SUPPKEY;
```

Note that the mapping information is needed in case of conflicts between attribute names.

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) [18]. 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.

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 sub-plan 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 very quickly.[5] 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 or a union query q, together with a set of associated parachute queries $\rho_0 ... \rho_n$. We assume for this algorithm that the parachute queries are part of the precipitate class. The output of this constrained algorithm is an execution plan for q annotated with labels at the root of each materializable shared component sub-queries (MSCSQ), i.e., the sub-queries (i) that are shared between the query and the parachute queries and thus (ii) can be materialized. These labels are used by

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

the *evaluate* algorithm when implementing the shared component sub-query materialization policy.

In our algorithm, the query q and the parachute queries $\rho_0 ... \rho_n$ are represented in Datalog [24]. We consider as an example, the query and the parachute query from the introduction, i.e. a conjunctive query and one associated parachute query ρ:

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

Our algorithm proceeds in four steps. The first step constructs the generalized shared subqueries (GSSQs) between the query and the parachute queries. The GSSQs are essentially the most specific sub-queries shared between the query and the parachute queries. The second step constructs two groups of MSCSQs based on the GSSQs. In the third step, the query and each group of MSCSQs 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 MSCSQ 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 identification of the generalized shared component sub-queries between the query and all parachute queries is based on a basic algorithm, called *2-queries-GSSQ*, for recognizing the most specific sub-query which contains a subset of two queries. There are two variants of this algorithm, one for conjunctive queries, presented in Figure 8, and one for union queries, presented in Figure 9.

The GSSQs for the query and its associated parachute queries are obtained as follows:

- **Step 1.1** For each parachute query pq_i, generate its generalized shared sub-query with the original query q. This generalized shared sub-query is noted $shared_i^0$ and is obtained using the algorithm 2-conjunctive-queries-GSSQ if q is a conjunctive query, or the algorithm 2-union-queries-GSSQ if q is a union query.

- **Step 1.2** If $shared_i^0$ is not empty, then it is renamed sq_k^1. The next steps construct recursively the intersection between the shared sub-queries already identified. If there is only one element in sq^1 or if this set is empty then go to step 1.5, otherwise go to step 1.3.To start with l = 1.

- **Step 1.3** For each pair of queries in sq^l, sq_i^l and sq_j^l, generate $shared_{ij}^l$, their generalized shared component sub-query. This is done using the same GSSQ generation algorithm as in step 1.1.

- **Step 1.4** If $shared_{ij}^l$ is not empty, it is renamed sq_k^{l+1}. l = l+1. If there is only one element in sq^l or if this set is empty then go to step 1.5, otherwise go to step 1.3.

in: two conjunctive queries q_i and q_j
out: the generalized shared sub-query of q_i and q_j

`2-conjunctive-queries-GSSQ`(q_i, q_j) {
    let x be the body of q_i
    efface all literals in x whose predicate does not appear in a literal in the body of q_j
    replace all variables in x or constants in x by new variables
    containment mappings are constructed between (1) x and q_i and (2) x and q_j
    the intersection of these containment mappings forms a binding b,
    i.e. bindings from x variables to constants or equality relations between variables in x
    we apply b to x to obtain the generalized shared sub-query of q_i and q_j
}

Figure 8: The construction of the generalized shared sub-query between two conjunctive queries.

in: two union queries q_i and q_j
out: the generalized shared sub-query of q_i and q_j

`2-union-queries-GSSQ`(q_i, q_j) {
    let x be the body of q_i
    efface all rules in x who do not appear in q_j
    x is the generalized shared sub-query of q_i and q_j
}

Figure 9: The construction of the generalized shared sub-query between two union queries.

- **Step 1.5** if sq^l is empty then l_{max} = l-1, else l_{max} = l. V = $sq^{l_{max}}$
- **Step 1.6** if $l_{max} = 1$ then return V; else $l_{max} = l_{max}$ - 1. For each $sq_k^{l_{max}}$ in $sq^{l_{max}}$, rewrite $sq_k^{l_{max}}$ using views in V: $sq_k^{l_{max}} = V_t ... V_{t'} R_k^{l_{max}}$, where $R_k^{l_{max}}$ is the remainder query. if $R_k^{l_{max}}$ is not empty then V = V $\cup$ $\{R_k^{l_{max}}\}$. $l_{max} = l_{max}$ - 1. Repeat step 1.6.

The set V contains the generalized shared component-subqueries between q and its associated parachute queries $pq_0 ... pq_n$.

In our example, given the query and parachute query above, the algorithm 2-conjunctive queries is applied in step1.1: x is $p(X1, X2, X3) \wedge s(X4, X5, X6, X7, X8, X9, X10) \wedge ps(X11, X12) \wedge like(X13, X14)$. The bindings that are identified between x and q is b_q is {X1 = X11, X2 = X13, X14 = 'BRASS', X3 = 15, X6 = X12} ; the binding identified between x and ρ is $b_\rho = b_q$.[6] We apply b to x to obtain $shared_0^0$: $p(X1, X2, 15) \wedge s(X4, X5, X6, X7, X8, X9, X10) \wedge ps(X1, X6) \wedge like(X2, 'BRASS')$. In step 1.2, $shared_0^0$ is renamed into sq_0^1 and l is initialized to 1. As there is only one element in sq^1 we go to step 1.5. There l_{max} is initialized to 1 and V contains sq_0^1. In step 1.6, as l_{max} is equals to 1, V is returned. The set og generalized shared component sub-queries between q and ρ thus contains one element: $p(X1, X2, 15) \wedge s(X4, X5, X6, X7, X8, X9, X10) \wedge ps(X1, X6) \wedge like(X2, 'BRASS')$.

Step 2 From the set of GSSQs identified in the previous step, we construct two groups of materializable shared component sub-queries (a MSCSQ is a view composed of a head and a body). Note that considering only two groups of MSCSQs is a heuristic. Each combination of views that covers the set of GSSQs could be considered as a group of materializable shared component sub-query.

The first group of shared component sub-queries contains one view per GSSQ identified in step1. It is a 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 in our example:

$\{mscsq_1(X1, X4, X5, X7, X8, X9, X10) \leftarrow$
$p(X1, X2, 15) \wedge$
$s(X4, X5, X6, X7, X8, X9, X10) \wedge$
$ps(X1, X6) \wedge$
$like(X2, 'BRASS')\}$

The second group of MSCSQ contains one view per literal (with any built-in predicate, if possible) appearing in the set of GSSQs. 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:

$\{mscsq_2(X1) \leftarrow p(X1, X2, 15) \wedge like(X2, 'BRASS'),$
$mscsq_3(X1, X2) \leftarrow ps(X1, X2),$

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

$mscsq_4(X1, X2, X3, X4, X5, X6, X7) \leftarrow$
$s(X1, X2, X3, X4, X5, X6, X7)\}$

Step 3 For each group of materializable 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 [18]. For the first group, q is rewritten as

$q(Sacct, Sname, Nname, Ppk, Pmfgr,$
$Sadd, Sphone, Scom) \leftarrow$
$mscsq_1(Ppk, Sacct, Sname, Snk, Sadd, Sphone, Scom) \wedge$
$n(Snk, Nname, Nrk) \wedge$
$r(Nrk, \ 'EUROPE')$

Step 4 The optimizer is invoked once to generate the most efficient execution plan for each materialized 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 materializable component sub-queries shared between q and $\rho_0 ... \rho_n$ and generates the cheapest execution plan for q which contains them.

5 Experimental Environment

Our experiments are performed using an analytical model and a detailed simulation of the evaluation models introduced in Section 3 using two workloads, a first one based on a synthetic query and a second one 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 introduced previously: classical, unconstrained and constrained.

The *classical* evaluation can be modeled as follows. When a query is issued to a *classical* mediator, the mediator checks for t trials whether all n data sources are available. It returns a complete answer if all data sources are available during at least one trial. Figure 10 illustrates four sources during three trials. Each trial is represented horizontally, as a concatenation of n circles or crosses. Circles represent unavailable sources, and crosses available sources. On the figure, the *classical* mediator returns a complete answer (represented as a straight line between crosses) on the second trial: all data sources are available simultaneously.

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.

An unconstrained evaluation 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. Figure 11 illustrates again four sources during three trials for the unconstrained evaluation. On the figure, the unconstrained mediator returns a complete answer (represented as a broken line between crosses) on the second trial: all data sources have been available at least once in the previous trials.

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. 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. Figure 12 illustrates a constrained evaluation processing a query

on four data sources with a shared component sub-query concerning two data sources. On the figure, the data sources involved in the shared component sub-query are both available during the second trial. On the third trial, both remaining data sources are available simultaneously; the complete answer (represented as the combination of two straight lines) can thus be returned.

When a query is issued to a constrained evaluation, 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. The probability that $m - n$ data sources are available simultaneously at least once in $t - t_0 + 1$ trials derives directly from Equation 1. In case there is one shared component sub-query involving m data sources, the probability of materializing this shared component sub-query in exactly t_0 trials can also be derived from Equation 1: it is the probability that all m data sources are available simultaneously at least once in t_0 trials minus the probability that all m data sources are available simultaneously at least once in $t_0 - 1$ trials. These two probabilities can be multiplied and summed over all t_0 between 1 and t to obtain the probability that a *constrained* mediator returns at least one complete answer in t trials

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

Equation 5.1 is the availability of complete answers in case a constrained evaluation deals with one shared component sub-query.

In case the constrained optimizer identifies several shared component sub-queries, there are m shared component sub-queries involving one data source. All m shared component sub-queries are materialized if the m data sources are available at least once. Once the shared component sub-queries are materialized, the complete answer is returned whenever the remaining $m - n$ data sources are all available simultaneously. Figure 13 illustrates a *constrained* mediator processing a query on four data sources with two shared component sub-queries each concerning one data source. On the figures, the first shared component sub-queries is available on the first trial, and the second is available on the second trial. On the third trial, both remaining data sources are available simultaneously; the complete answer (represented as the combination of a broken line and a straight line) can thus be returned.

Again, when a query is issued to a *constrained* mediator, the mediator checks for t trials the availability of all n data sources. If on the t_0th trial, all shared component sub-queries 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.

For Equation 5.1, the probability of materializing all shared component sub-queries in exactly t_0 trials can be derived from Equation 2: it is the probability that m data sources are available at least once in t_0 trials minus the probability that m data sources are available at

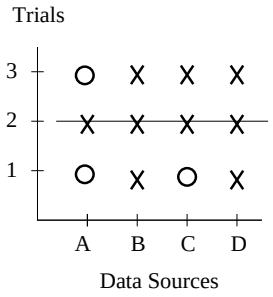

Figure 10: Classical Mediator

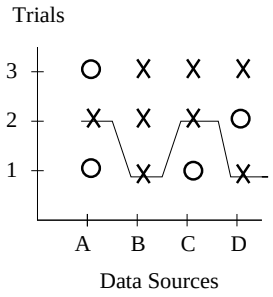

Figure 11: Unconstrained Mediator

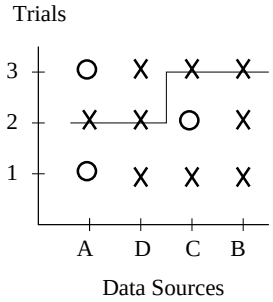

Figure 12: Constrained Mediator (AD: Shared Component Sub-query)

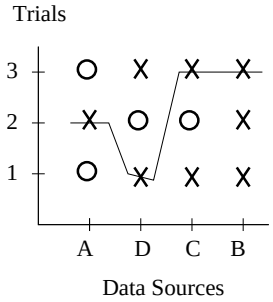

Figure 13: Constrained Mediator (A and D: Shared Component Sub-queries)

least once in t_0-1 trials. The probability of $m-n$ data sources being available simultaneously at least once in $t - t_0 + 1$ trials has been described previously. These two probabilities can be multiplied and summed over all t_0 between 1 and t to obtain the probability that a *constrained* mediator returns at least one complete answer in t trials.

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

Equation 5.1 is the availability of complete answers in case a constrained evaluation deals with several shared component sub-queries, each involving one data source.

5.2 Simulation Environment

To study the performance of the algorithms producing partial answers, we have extended an existing simulator [12, 10] that models a peer-to-peer database system. We briefly des-

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.

cribe, here, the simulator and present the extensions we have implemented to simulate the evaluation models 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 section 6.2.4.) 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.

The CPU is modeled as a FIFO queue and the simulator charges for all the functions performed by query operators like hashing, comparing and moving tuples in memory. The disk model includes costs for random and sequential physical accesses and also charges for software operations implementing I/Os. The unit of disk I/O is a page of size *DskPageSize*. The unit of transfer between servers are pages of size *NetpageSize*. The network is modeled simply as a FIFO queue with a specified bandwidth (*NetBw*); the details of a particular technology are not modeled. The simulator also charges CPU instructions for networking protocol operations.

In our experiments, pages are accessed from the data sources in a page-at-a-time approach, rather than a streaming approach. When an operator requests a data page, this operator has to wait until the page is received before requesting the next one.

We extended this simulator by introducing the `evaluate` algorithm with different materialization policies as described in Section 4. Materialization is performed as follows. When a query tree is executed, an open call is issued on its root operator. When the open call finishes, all nodes in the query tree are marked available or unavailable. If the root of the execution tree is marked available, then the complete answer is produced by calling the get-next method and the execution tree is closed. If the root of the execution tree is marked unavailable, then depending on the materialization policy, materialization operators are included in the execution tree. Such operators call repeatedly get-next on their child operator and materialize locally the intermediate result which is produced. In the sense phase, unavailable servers are modeled in a 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 Synthesized Workload

The synthesized workload consists of a 4-way join query (i.e. a conjunctive query) and its associated parachute query.

```
Q: SELECT A.att1, B.att1, C.att1, D.att1
   FROM A, B, C, D
   WHERE A.att2 = B.att2 and B.att2 = C.att2 and C.att2 = D.att2;

PQ: SELECT A.att1, D.att1
    FROM A, D
    WHERE A.att2 = D.att2;
```

Base relations (A, B, C and D)) are stored on separate data sources. Each base relation contains 10,000 tuples of 100 bytes each (i.e. 1MB).

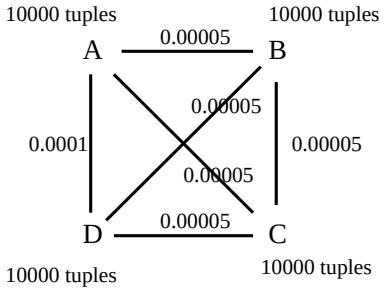

Figure 14: Query graph for query Q

Figure 14 shows the query graph for query Q. Each base relation is represented by its name and is labeled with its cardinality. Each edge between two relations is labeled with the selectivity of the corresponding join operation. This query graph is fully connected. Selectivities are set so that A ⋈ D is the most expensive join. It produces a relation of the same size as the base relations. All other selectivities are equals and inferior to the selectivity of A ⋈ D.

The parachute query PQ belongs to the precipitate class. It is derived from Q by removing relations B and C.

This query and its associated parachute query were chosen mainly for simplicity reasons. First, a 4-way join query and a 2-way join parachute query are sufficient to illustrate the availability and performance characteristics of the three evaluation models we consider. Second, considering base relations with similar sizes simplifies significantly the interpretation of the experiments. A synthesized query is however limited, in the sense that it does not represent realistic join graphs, cardinalities or selectivities. We use a modified TPC-D query to overcome this limitation.

5.2.4 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.

The query is a 5-way join query, with selections on the PART and REGION relations. Figure 15 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

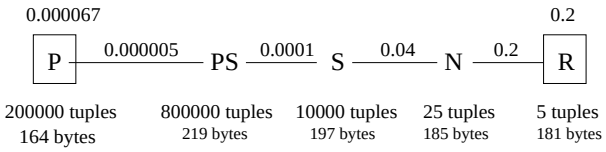

Figure 15: Query graph for query Q

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`.

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, an execution plan containing one shared component sub-query is chosen by the constrained optimizer.

Figure 16 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 is contained between the curve for the unconstrained evaluation (upper bound) and the curve for the classical evaluation (lower bound). The event of a classical evaluation returning a complete answer in t trials is included in the event of a constrained evaluation 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 16, the unconstrained evaluation shows much better availability than the other two evaluation models: for a source availability of 0.9, the probability of obtaining an answer is 0.96 in the unconstrained evaluation, 0.89 in the constrained evaluation and 0.83 in the classical evaluation. 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, it has the lowest availability of complete answers. In the constrained evaluation, 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 is close to the curve for the classical evaluation. In the unconstrained evaluation, as much data as

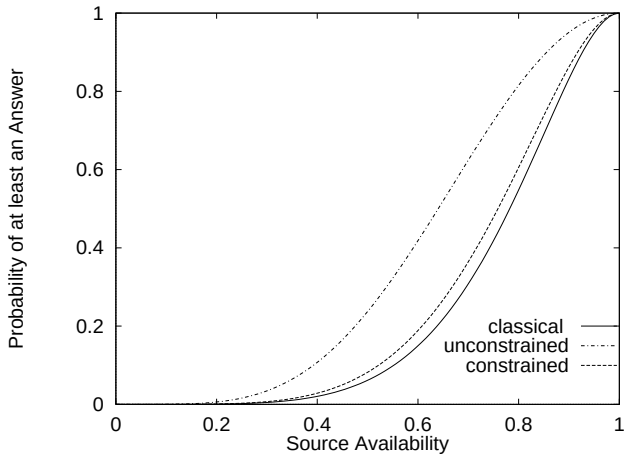

Figure 16: Probability of obtaining at least one complete answer in 2 trials for the TPC-D Workload

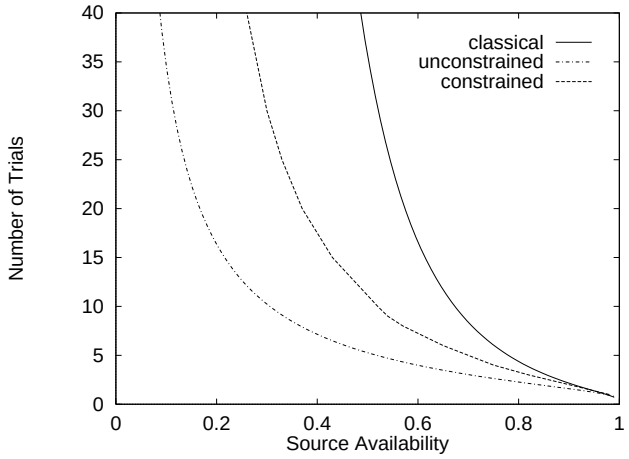

Figure 17: Numbers of trials required to have a probability 0.9 of obtaining a complete answer for the TPC-D Workload

possible is materialized at each trial. The probability of obtaining a complete answer is thus much higher in this evaluation model.

Figure 17 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 is still reasonable, almost 4 trials, while it is 7 trials in the constrained evaluation and 16 trials in the classical evaluation.

6.2 2 Trial Experiment - Synthesized Workload

We use the simulator with the synthesized workload to examine the influence of the optimization algorithm and of different materialization policies on response time.

6.2.1 Classical Evaluation

We consider the following settings for unavailable sources: in a first trial some sources are unavailable (we examine all combinations of unavailable sources), in a second trial all sources are available. This corresponds to the following sequence of interactions between the application program and the mediator: q, fa, pq, pqa, q, a, where q is the query Q presented in Section 5.2.3, fa is the first answer returned by the mediator (an error), pq is the parachute query PQ, pqa is the corresponding parachute answer, a is the answer to query Q which is provided computed in the second trial because all sources are available (we call it incremental answer for homogeneity with the other evaluation models). Note that in case no source is unavailable, the sequence of interaction is then q, a: the first answer returned by the mediator is the complete answer.

When query Q is submitted to the system, it is optimized. The optimizer is cost based; its objective function is to minimize total work. We obtain the execution plan presented in Figure 18. In case some sources are unavailable, an error is returned and parachute query PQ is asked. The parachute query is optimized, its execution plan is $A \bowtie$ D. We assume that the configuration of sources remains the same between the execution of Q and PQ. The mediator returns either the parachute answer or an error. Finally, all data sources are turned on. Query Q is re-submitted to the mediator and the complete answer can be computed.

We run the sequence of queries described above for each possible configuration of sources in the first trial. We measure the time to first answer, the time to parachute answer and the time to complete answer.

Figure 19 shows the results for the synthesized workload with a classical system. The x-axis indicates the configuration of available sources in the first trial and the y-axis indicates the query response time (in seconds).

In case all data sources are available in the first trial (ABCD), the first answer, which is the complete answer, is obtained in 68.5 seconds.

The time to first answer and the time to incremental answer are identical in all configurations of sources where some sources are unavailable. The time to first answer is 10 seconds. It corresponds to the *time-out* value in the simulator: as soon as a data source is

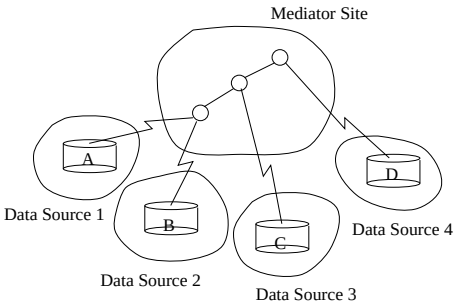

Figure 18: Execution plan for the synthesized query produced by a cost based optimizer

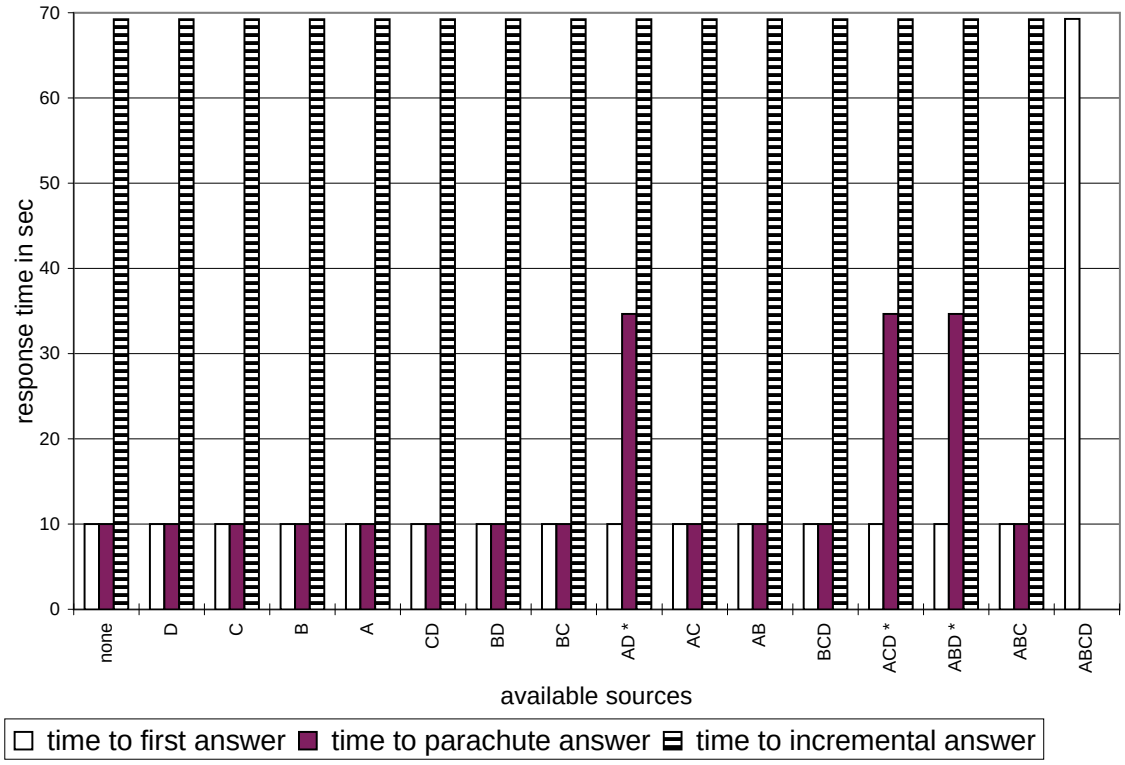

Figure 19: Synthesized Workload on a Classical System

recognized unavailable an error is returned. It always takes *time-out* seconds to recognize

that a source is unavailable. The time to incremental answer is 68.5 seconds. It is equals to the time to first answer in case all sources are available. No relations are materialized in the constrained evaluation, as a result there is no incremental query (Q is simply re-submitted) and no work is saved between consecutive executions.

A parachute answer is obtained for configurations AD, ACD and ABD (whenever A and D are available – these configurations are indicated with a star on the x-axis). This is a consequence of our assumptions that the configuration of sources remains the same between the first trial and the execution of the parachute query. In this case, the time to parachute answer is 34.5 seconds. It is the time to access relations A and D and perform the join. In all other cases, the time to parachute answer is 10 seconds. It is here also the value of the *time-out* necessary to recognize that one of the relations involved in the parachute query is unavailable.

6.2.2 Unconstrained Evaluation

We use the same sequence of interactions with the unconstrained evaluation that we used with the classical evaluation: q, fa, pq, pqa, i, a. Here however, fa the first answer returned by the mediator is a partial answer in case some sources are unavailable and we do not re-submit the original query Q (see Section 5.2.3) but the incremental query i based on the mediator state.

The execution plans chosen by the optimizer for Q and PQ are the same as in the classical evaluation (see Figure 18).

The materialization policy we have chosen to illustrate the unconstrained evaluation is the *materialize 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. We assume here that disk space is unlimited.

Figure 20 shows the results for the synthesized workload with an unconstrained system.

In case all sources are available (ABCD), the first answer is the complete answer. It is obtained in 68.5 seconds. In this case no parachute query or parachute query are submitted.

Because all relations have the same size, the more relations are available in the first trial, the higher the time to first answer and the lower the time to incremental answer. The more relations are available in the first trial, the more work can be done with available sources and the longer it takes to generate the partial answer. Besides, the more data have been materialized in the mediator state, the less work needs to be done for the incremental query.

In case no data source is available (configuration *none*), nothing is materialized in the mediator state. The time to partial answer is the time it takes to recognizes that a data source is unavailable: 10 seconds, i.e. the *time-out* value. The incremental query is here identical to the original query Q and the time to incremental answer is also identical to the time to compete answer in configuration ABCD, i.e., 68.5 seconds.

In case one data source is available, the time to first answer is 29.8 seconds. This time corresponds to the time it takes to recognize a data source is unavailable plus the time to access and materialize the available relation. In those cases, the time to incremental answer

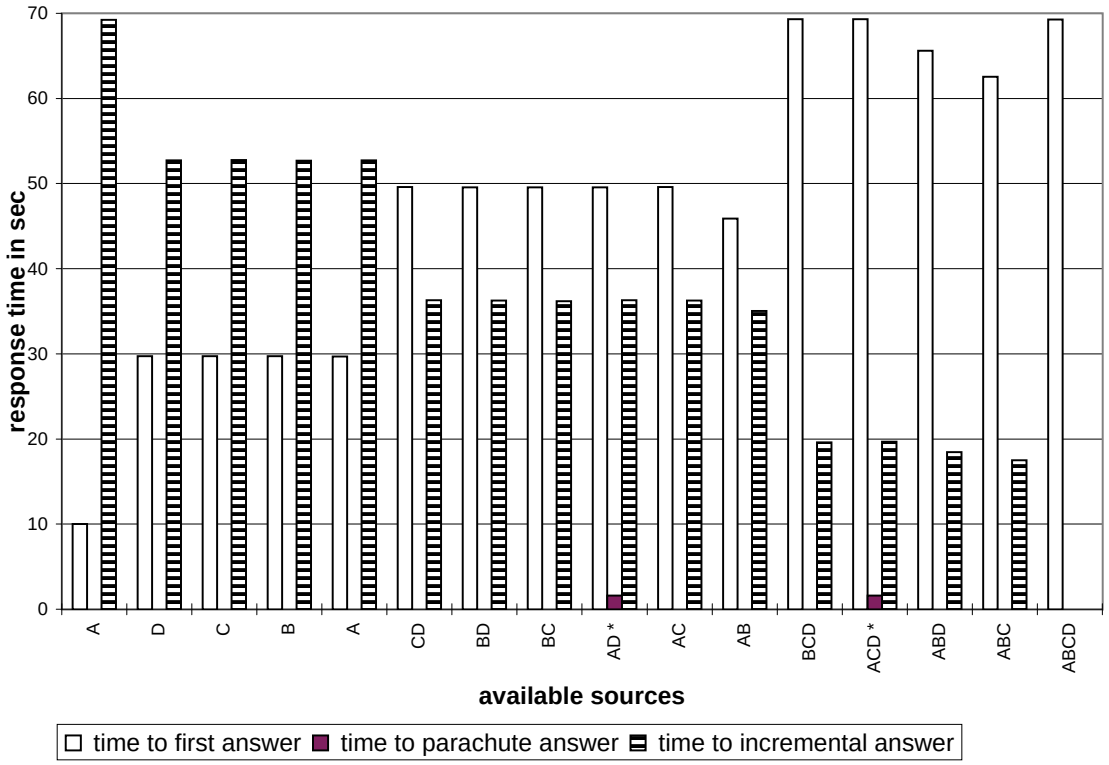

Figure 20: Synthesized Workload on an Unconstrained System

is 51.4 seconds. This is the time to access the three sources that were unavailable in the first trial, reading the materialized relation and performing the join operations. These times are equal for all configurations where one data source is available because all relations have the same size.

In case CD, BD, BC, AD and AC are available, the time to first answer is 49.5 seconds. This is the time it takes to recognize a data source is unavailable plus the time it takes to access and materialize the two available relations. In these cases, the time to incremental answer is 34.7 seconds. This is the time it takes to access the two data sources that were unavailable in the first trial, read the materialized relations and perform the join operations. Here, the time to incremental answer is lower than the time to first answer. This is because the *time-out* penalty and the cost of writing to the materialized relations are not paid when answering the incremental query. Besides, the time to first answer is higher in theses cases than it is in the configurations where only one data source is available. This is because more

work is done when generating the partial answer (two relations are accessed and materialized instead of one and all relations have the same size). The time to incremental answer is lower in these cases than it is in the cases where one data source is available. This is because the price of contacting one more data source is higher than the price of reading one more materialized relation. In case AB are available, the time to first answer is 45.8 seconds. This is the time it takes to recognize a data source is unavailable plus the time it takes to access relations A and B, perform the join and materialize the result. This time is slightly inferior to the time to first answer in the other cases where two data sources are available because it is cheaper to materialize the relation produced by $A \bowtie B$, which is half the size of a base relation and perform the join, than it is to materialize two base relations. The hash join operator introduces parallelism between the accesses to both relations. This parallelism is however not very sensitive because of the low network bandwidth. In case A and B are available, the time to incremental answer is 34.5 seconds.

In case data sources BCD and ACD are available, the time to first answer is 69.2 seconds, i.e. it is slightly higher than the time to complete answer 68.5 seconds (case where ABCD are available). In cases ABC and ABD are available, a join is performed (respectively $A \bowtie B$ and $A \bowtie B \bowtie C$) and the time to first answer, respectively 65.6 and 62.6 seconds, is thus inferior to the time obtained in cases where no joins are performed. As we have seen before, the joins reduce the size of the relations to be materialized. In all these cases, the time to incremental answer is inferior to 20 seconds.

Parachute answers are provided in the configurations AD and ACD (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 query cannot be evaluated given the current mediator state. In those cases the time to parachute answer is null. The time to parachute answer is 1.5 seconds in both configurations where a parachute answer is provided. In those cases, the parachute query evaluation algorithm has recognized that relations A and D where available in the mediator state and that they have to be joined in order to compute the parachute answer.1.5 seconds is the time required to read, locally, the materialized relations A and D and to perform the join between them. In configuration ABD, A and D are available. However, the parachute answer cannot be computed. This is because with the *maximal available sub-query* materialization policy, in this case, $A \bowtie B$ is performed and the parachute query PQ involving A and D can not be evaluated.

6.2.3 Constrained Evaluation

In the constrained evaluation, Q and PQ are submitted together. The sequence of interactions between the application program and the mediator is now: $(q, \rho), (pa, \alpha), i, a$, where q and ρ are the synthesized query and its associated parachute query, fa is the partial answer, α is the parachute answer, i is the incremental query and a is the incremental answer.

When Q and PQ are submitted, the constrained optimization algorithm is applied. The least general shared sub-query which is identified is *a(X1, X2), d(X4, X2)*. The two possible groups of shared component sub-queries are, on the one hand, *scsq1(X1, X2, X3) :- a(X1, X2), d(X4, X2)*, and on the other hand, *scsq2(X1, X2) :- a(X1, X2).* and *scsq3(X1, X2) :-*

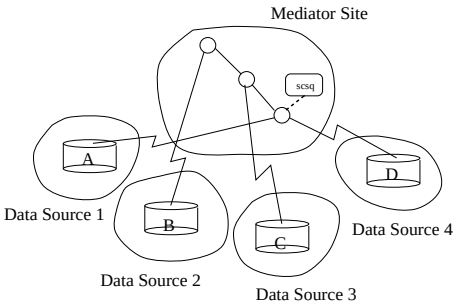

Figure 21: Execution plan for the synthesized query produced by the constrained optimization algorithm

d(X1, X2).. After these queries and the corresponding remaining queries are optimized and their costs added, the execution plan chosen for Q uses the shared component sub-query scsq1. Figure 21 presents this execution plan.

The materialization policy we have chosen to illustrate the constrained evaluation is the *materialize shared component 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 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. Again, we assume here that disk space is unlimited.

Figure 22 shows the results for the synthesized workload with a constrained system.

In case all sources are available (ABCD), the first answer, which is the complete answer is obtained in 68.5 seconds. The execution plan chosen by the constrained optimization algorithm is here almost as efficient as the query plan chosen in the unconstrained or classical evaluations. The objective function of the constrained optimization algorithm is to minimize total work for computing and materializing the shared component sub-query and returning the complete answer. The objective function of the optimizer in the other evaluation models is to minimize total work for producing the complete answer. As a result, with other parachute queries, selectivities and cardinalities, the execution plans chosen by the constrained algorithm could much more inefficient than it is in our experiment.

In cases where A and D are available, the shared component sub-query is materialized and the parachute answer can be computed. In those cases, time to first answer is 47 seconds. It is the time to recognize a source is unavailable plus the time to access the relations A and D, perform the join and materialized the produced relation. Time to parachute answer is 0,8 seconds. It is the time to read the relation materializing $A \bowtie D$. The time to incremental answer is 35.4 seconds. It is the time to access the two relations that have not been materialized during the first trial, and compute the complete answer.

All other cases are similar to the classical evaluation. The time to first answer is 10 seconds, the value of the *time-out*. The time to incremental answer is 69.3 seconds, i.e.

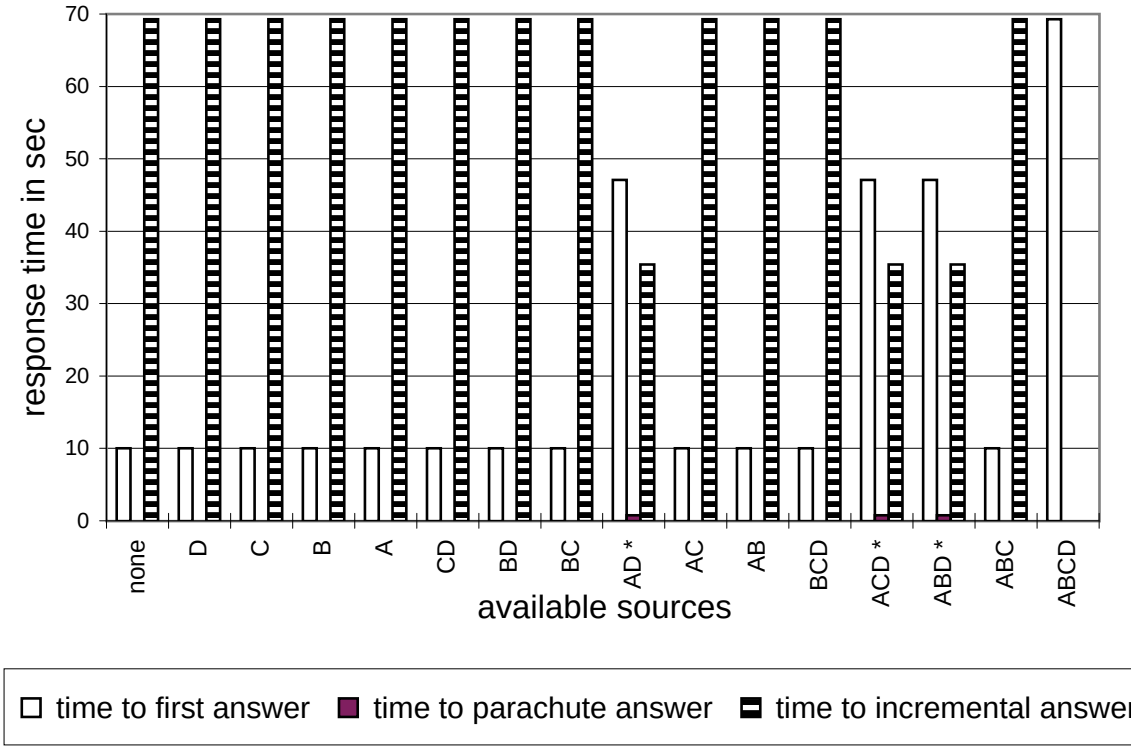

Figure 22: Synthesized Workload on a Constrained System

the time to complete answer in case all data sources are available. Here also no parachute answers are obtained. However, the time to parachute answer is null because the parachute query evaluation algorithm recognizes that PQ cannot be evaluated given the mediator state.

6.2.4 Influence of Delays

The previous graphs show response time in a sequence of two trials. We assume that when the incremental query is re-submitted all data sources are available. The time it takes for all sources to be available again is off course variable.

In this experiment, we still consider that the complete answer is obtained in two trials. We further assume that the incremental answer is submitted as soon as all data sources are available. In a first trial, source A is unavailable. Source A becomes available after a variable delay. The incremental answer is submitted as soon as the delay during which source A is

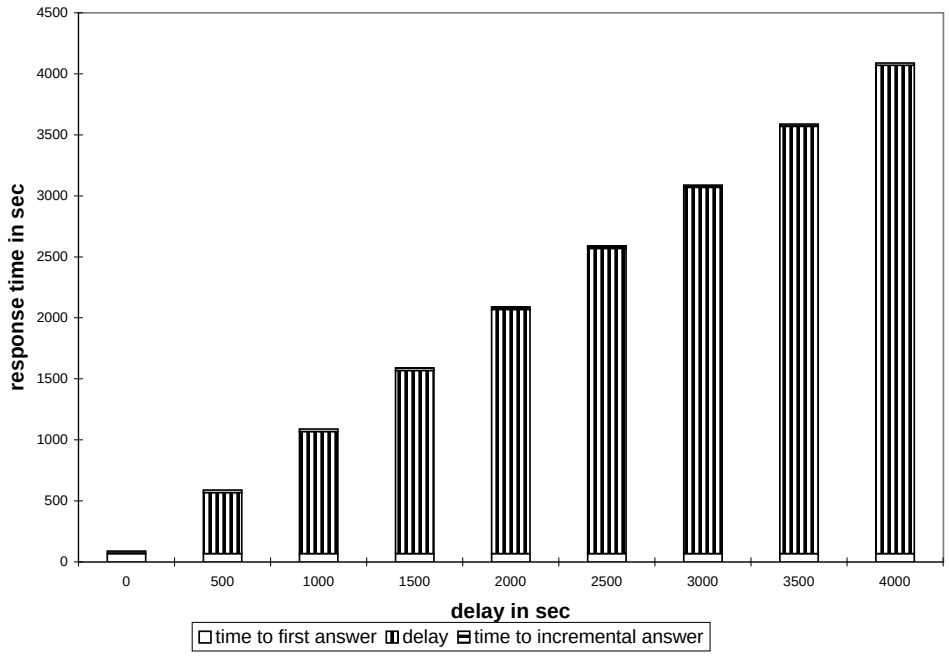

Figure 23: Unconstrained System - Source A unavailable

unavailable is terminated. This simulates a system where the application program is notified that a given source becomes available after a period during which it was unavailable.

Figure 23 shows the time to complete answer (composed of the time to first answer, a delay and the time to incremental answer) in case source A is unavailable for a variable delay in an unconstrained evaluation.

The time to complete answer is dominated by the delay, which varies here between 0 and 4000 seconds. The time to first answer and time to incremental answer are the same as they were in Figure 20 for the case where A is unavailable in the first trial (case BCD).

6.3 2 Trial Experiment - TPC-D Workload

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

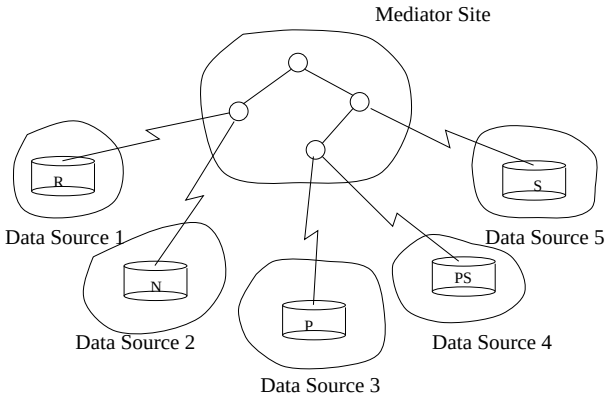

Figure 24: Execution plan for the query in the Three Evaluation Models.

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.

6.3.1 Classical Evaluation

The sequence of interaction for the classical evaluation experiment is q, n, ρ, α, i, a. When the query is submitted to the classical evaluation, it is optimized. Figure 24 shows the execution plan which is chosen.[7] When the parachute query is submitted, it is also optimized – its execution plan is $(P \bowtie PS) \bowtie S$. The classical evaluation 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 25 shows the results for the TPC-D workload with a classical evaluation 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-

[7] 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.

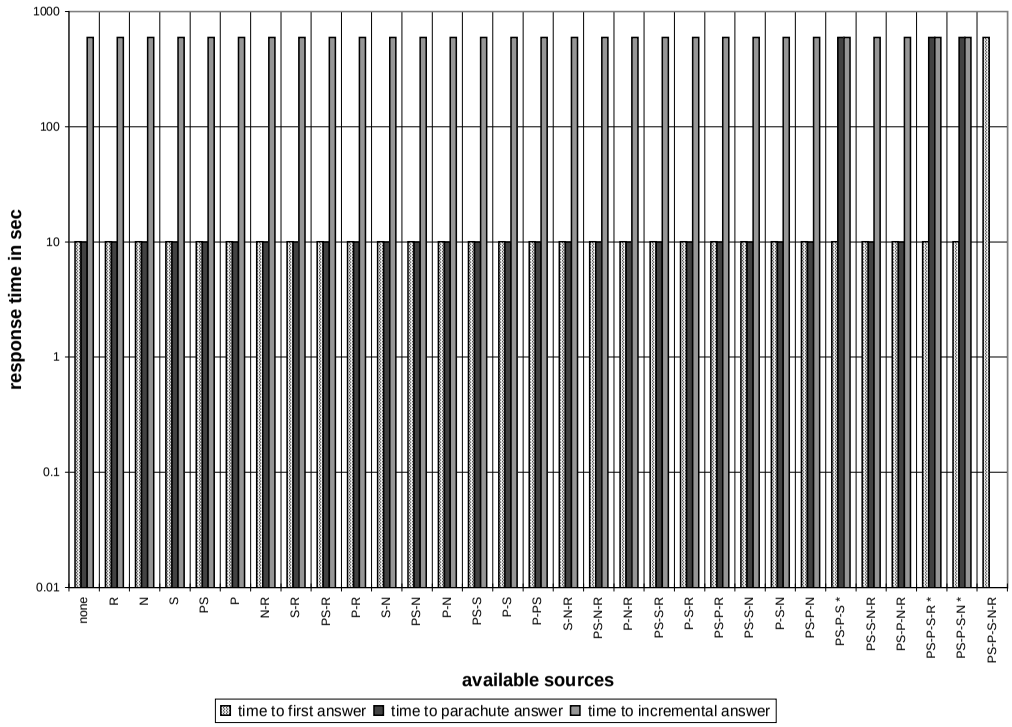

Figure 25: TPC-D Workload – Classical Evaluation

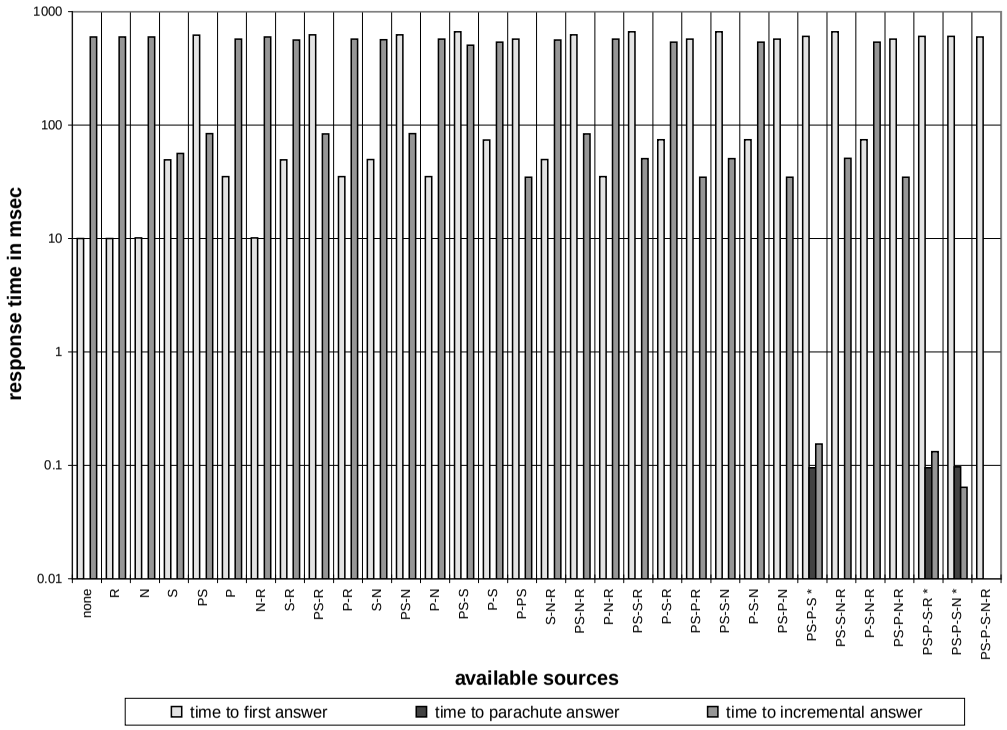

Figure 26: TPC-D Workload – Unconstrained Evaluation

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, 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.

6.3.2 Unconstrained Evaluation

We use the same sequence of interactions with the unconstrained evaluation that we used with the classical evaluation: q, n, ρ, α, i, a. The execution plans chosen by the optimizer for q and ρ are the same as in the classical evaluation (see Figure 24). 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 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 26 shows the results for the TPC-D workload with an unconstrained evaluation. In case all sources are available, the first answer is the complete answer, obtained in 597.2 seconds. The unconstrained evaluation operates in the same way as the classical evaluation 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 configuration PS for instance, the time to first answer is 622.3 seconds. It is approximately the sum of 10 seconds delay, 540 seconds to access PS and 70 seconds to materialize this relation (7800 pages). 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 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 which must evaluate the parachute query from scratch.

6.3.3 Constrained Evaluation

The sequence of interactions for the constrained evaluation 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 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 is the *shared component sub-query* policy (cf. Section 4). When evaluating a query tree, the eva-

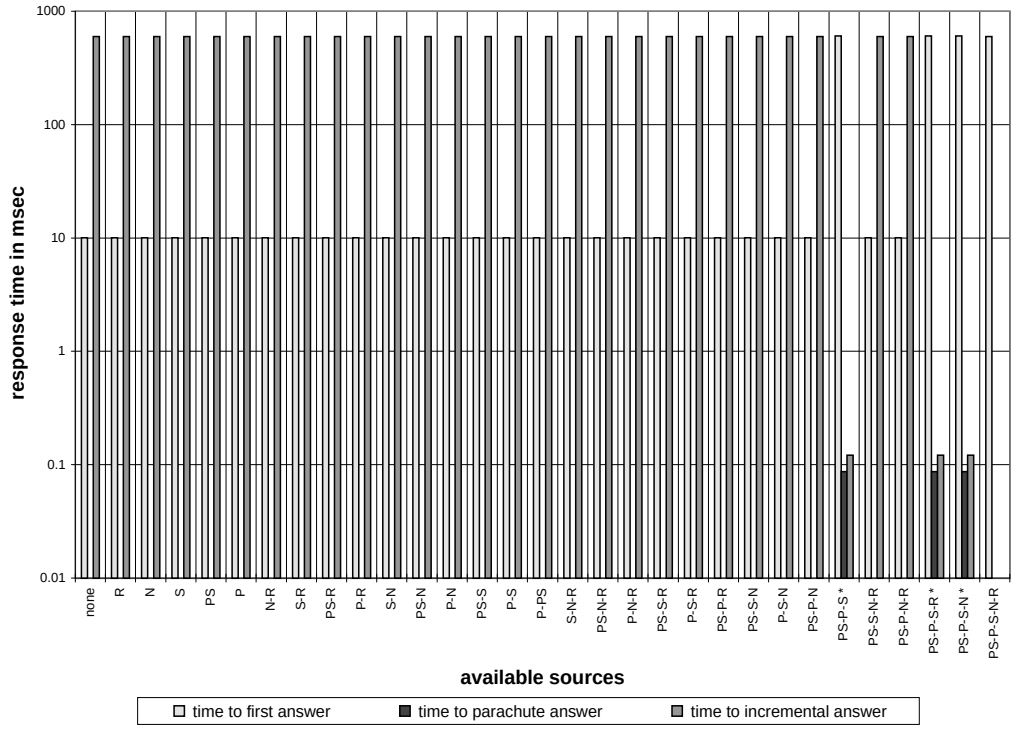

Figure 27: TPC-D Workload – Constrained Evaluation

luate 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 27 shows the results for the TPC-D workload with a constrained evaluation. 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. This time to first answer is 600.7 seconds. In these cases, a parachute answer can be obtained. The time to parachute answer is again the

time it takes to read a local relation of 30 pages, i.e. 0.09 second. The time to incremental answer is the time to access the remaining relations (either R, N or R and N). This time is 0.12 second.

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.

6.4 Availability of Parachute Answers

In this section, we study the influence of the optimizer and of the materialization policies on the availability of parachute answers. The availability of a parachute answer is the probability that a parachute answer can be evaluated using the mediator state.

For this experiment, we consider all possible configurations of sources for a given query: for each optimizer and materialization policy, we count the number of configurations where the associated parachute query can be evaluated. The availability of parachute answers is then computed as the number of configurations of sources where the parachute query can be answered divided by the total number of configurations of sources. We slightly modify the workload presented in Section 5.2.3. We still use the query and the parachute query described there, and we introduce a new factor: relations may be co-located. Relations A, B, C and D are not necessarily located on different sites, they may be co-located on a same data source. In our experiment, relations A and B may be co-located or not.

In the case all relations are on separate sites, the plans produced by the unconstrained optimizer and the constrained optimizer have been presented in Section 6.2. Figure 18 presents the execution plan generated by the unconstrained optimizer and Figure 21 presents the execution plan generated by the constrained optimizer. The constrained optimizer identifies A and D as the shared component sub-queries.

In the case A and B are co-located, the unconstrained and constrained optimizer produce different execution plans. Figure 28 presents the execution plan produced by the unconstrained optimizer. This is the cheapest execution plan where the join A $\bowtie$ B is pushed to data source 1. Figure 29 presents the execution plan produced by the constrained optimizer. A and D are again the shared component sub-queries, the join A $\bowtie$ B is thus not pushed to data source 1.

The first row in Table 30 presents the availability of parachute answers in case all relations are on separate data sources for an unconstrained optimizer. There are four data sources and thus 16 different configurations of sources. One of these configurations corresponds to the case where A, B, C and D are available. In this case the complete answer is produced and no parachute query is asked. There are thus 15 configurations of sources where a parachute query can be asked. As a result, the probability of a parachute answer is the number of configurations in which a parachute query can be evaluated divided by 15. In the case of

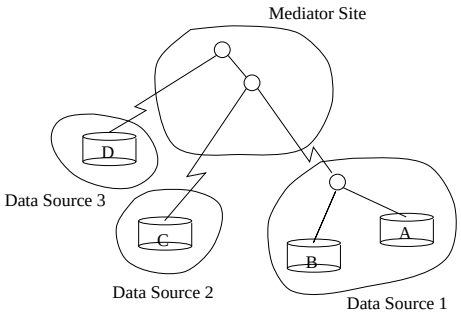

Figure 28: Execution plan for the synthesized query produced by the unconstrained optimizer in case relations A and B are co-located

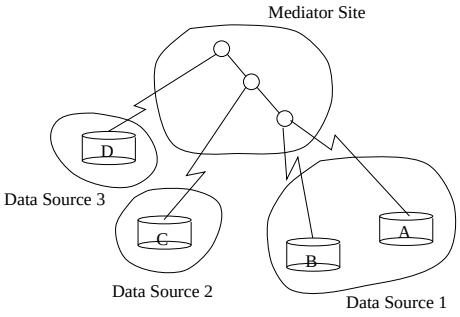

Figure 29: Execution plan for the synthesized query produced by the constrained optimizer (A and D are the shared component sub-queries) in case relations A and B are co-located

	Materialization Policy			
	nothing	leaves	max	scsq
unconstrained	x	3/15	2/15	n.a.
constrained	x	3/15	2/15	3/15

Figure 30: Probability of Parachute Answers in the case all Relations are on Located on Separate Data Sources

	Materialization Policy			
	nothing	leaves	max	scsq
unconstrained	x	0/7	0/7	n.a.
constrained	x	1/7	0/7	1/7

Figure 31: Probability of Parachute Answers in the case Relations A and B are co-located

the materialize nothing policy, the parachute query cannot be evaluated using the mediator state. We consider, in this case, that the parachute query is evaluated using remote data sources. The number of parachute queries that can be evaluated depends on the availability of the sources when these query are asked; we therefore cannot compute the probability of parachute answer [8]. In the case of the materialize leaves policy, the relations A and D are materialized on the mediator site in all configurations of sources where both are available: data sources 1 and 4 are available, data sources 1, 2 and 4 are available and data sources 1, 3 and 4 are available. The number of configurations where a parachute query can be evaluated is thus three in the case of a materialize leaves policy. The case of the materialize maximal available sub-query policy is a bit different. In the configuration where data sources 1, 2 and 4 are available, the maximal available sub-query is $((A \bowtie B) \bowtie D)$. This sub-query is evaluated and materialized. As a result, the parachute query $(A \bowtie D)$ cannot be evaluated. In the other two configurations of sources where A and D are available, the parachute query can be evaluated. The probability of parachute answer is thus 2/15 for the materialize maximal available sub-query policy. The materialize shared component sub-query is not applicable (n.a. in the table) with an unconstrained optimizer. The second row of this table presents the availability of parachute answers in case all relations are on separate data sources for a constrained optimizer. The execution plans produced by the constrained optimizer is presented in Figure 21). This execution plan is different than the one produced by the classical and unconstrained optimizer: $(A \bowtie D)$ is present in the execution plan. The materialize shared component sub-query policy is specific to the constrained optimizer. In this case relations A and D are materialized in all configurations of sources where they are available. The availability of parachute answer is thus 3/15 for the materialize shared component sub-query policy. When considering the materialize leaves policy, the shape of the execution plan is irrelevant; as a result the availability of parachute answer is 3/15, as in the unconstrained case 3/15. When considering the materialize maximal available sub-tree policy, the parachute answer can be produced when A, D and B are available, or when only A and D are available. However, when A, D and C are available $(A \bowtie D) \bowtie C$ is materialized and as a result the parachute query $(A \bowtie D)$ cannot be answered. The availability of parachute answers is 2/15 for the materialize maximal available subtree policy, as in the unconstrained case. Whatever the shape of the execution plan, there is a risk of materializing too much data with the materialize maximal available subtree policy.

[8] note that we assume in the simulation experiment the same configuration of sources for the execution of the query and the parachute query.

Table 31 presents the availability of parachute answers in case relations A and B are co-located. There are now three data sources and thus 8 different configurations of sources. As discussed before, there are 7 configurations where parachute queries are asked. The first row in this table, presents the results obtained with the execution plan produced by the unconstrained optimizer (see figure 28). Here, both applicable materialization policies do not allow to evaluate any parachute query. In all configurations of available data sources, the sub-query (A $\bowtie$ B) is a leave pushed on data source 1. Therefore relation A alone cannot be materialized on the mediator site, and the parachute query (A $\bowtie$ D) can never be evaluated. The second row in this table, presents the results obtained with the execution plan produced by the constrained optimizer (see figure 29). A and D are identified as shared component sub-queries. They are thus leaves in the execution plan, and they can be materialized. Relations A and D are both available in only one configuration: data sources 1 and 3 are available. The materialize leaves policy and the materialize shared component sub-queries allow to materialize A and D in these cases and thus the parachute query can be answered. In the case of the maerialize maximal available sub-query policy, when A is available, then B is also available and the sub-query (A $\bowtie$ B) is computed and materialized; A alone is never materialized. The parachute query is, in this case, never answered.

6.5 Discussion

Complete Answers The classical evaluation always pays the least amount in terms of query processing. This results from the use of the sense phase of the evaluate algorithm. Without the sense phase, the classical evaluation would perform *very* poorly since it would start query processing before knowing which data sources were available. Any work accomplished would be thrown away if the complete answer was not produced. We have chosen not to study this obvious case. Contrasting this evaluation model with the other two in terms of the probability of receiving a complete answer, we note again that the classical evaluation performs poorly by this metric. Thus, to achieve better availability of the complete answer, a performance price must be paid.

In contrasting the unconstrained evaluation to the constrained evaluation, we note that the performance cost is generally higher. This cost is due to the arbitrary materialization of all data that arrives from data sources. However, this evaluation model has better chance of returning a complete answer earlier.

In terms of complete answers, the constrained evaluation operates in a range between the two other evaluation models. The time to a complete answer is generally higher than in the classical case. It may be lower than the unconstrained case in some situations due to avoiding unnecessary materializations, however it may be higher due to the fact that the parachute queries *distort* the query execution plan (see Section 6.2. The probability of returning a complete answer is in-between the other two evaluation models.

Parachute queries The probability of receiving the answer to a parachute query follows similar curves as those shown in Figures 16. However, for parachute queries the constrained evaluation has a distinct advantage. First, it will materialize exactly the information needed to answer the parachute queries (see Section 6.4). No additional materializations

will be done. Second, it will insure that the execution plan contains the correct joins for the parachute queries. In contrast, the best competing alternative to this solution is the unconstrained evaluation with a materialize leaves policy.

7 Related Work

Replication An alternative to parachute queries in dealing with unavailable data sources is their *replication*. Replication can increase the availability of all data sources to the point that queries almost always execute. Replication suffers from economic disadvantages – first, the hardware for each data source must be replicated at a different physical site (to avoid communication failures) and second the software has higher cost because replication impacts the complexity of software. Replication also (usually) impacts update performance. In addition, in an environment with autonomous data sources, replication may not be possible simply because the data source forbids it. In any case, 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.

The generation of parachute queries, that we have presented above, can be classified as a *cooperative answering* technique; it can be compared to existing cooperative answering techniques such as approximations and query relaxation.

Cooperative Answering Systems References [13] and [20] 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. [20] 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 incomplete or empty results; parachute queries allow the application program to retrieve *relevant* data in case some tables involved in the original query are unavailable. We can compare parachute queries with two existing techniques whose goals are comparable: approximation and query relaxation.

Approximations have been notably developed in [7], [19], [21] and [25]. In case a precise answer to a query cannot be computed, the database system may return an answer composed of a *lower approximation* whose objects belong with certainty to the precise answer, and of an *upper approximation* whose objects may belong to the precise answer. Such a pair of approximation is called a *sandwich*.

Reference [7] describes the Hoare ($\sqsubseteq^\flat$) and Smyth ($\sqsubseteq^\sharp$) orderings: the lower approximation is inferior to the precise answer in the Hoare ordering, and the upper approximation is inferior to the precise answer in the Smyth ordering. [7] introduces an ordering on sandwiches: $(U, L) \sqsubseteq (U', L') \equiv$ U $\sqsubseteq^\flat$ U' and L $\sqsubseteq^\sharp$ L'. Any relational query can be encoded as a set of rules; these rules can then be used to infer a sandwich approximation of the answer

starting from sandwich approximations of the base tables. The sandwich approximation of the answer is guaranteed to be a unique minimal model.[9]

Reference [21] studies the problem of unavailable data sources in a multi-database system: Multiplex. This system uses the notions of subview and superview to define the approximate answer which is returned in case a data source is unavailable. 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 are close to the relations we have used to define the precipitate class. V1 is a subview of V2 is equivalent to V1 is a generalized subset of V2. The notions of superviews and generalized superset are however not equivalent. If we consider the TPC-D example from the introduction, with the data sources containing NATION and REGION which are unavailable. If we consider that materialized views are not reused between the processing of consecutive queries and can only be obtained by materialzing data from the available data sources, then there is no subview or superview defined for the original query. The approximation that would be returned by Multiplex in this case is thus trivial.

Reference [25] describes a system, called APPROXIMATE, that implements a system dealing with sandwich approximations. The sandwich approximations in APPROXIMATE are simpler than in [7] for two reasons. First, the lower and upper approximations are ensured to have a similar schema, and second, there are no null values appearing in the approximations. With these simplifications the Hoare and Smyth orderings are not needed, the subset and superset relationships can be used. APPROXIMATE implements a special kind of data model and query processor which ensures that the approximate answer improves when the approximations of the base relations improve. APPROXIMATE is used for applications where the approximation of the base relations can be refined over time. An assumption of the system is thus that a superset of each base relation can always be obtained. This requires that the system manipulates semantic information about base relations.

Approximations are well suited when relations contain null values or when independent relations are combined, however they are not well suited when relations are unavailable. Both available and unavailable relations are represented by trivial approximations. The sandwich approximation for an unavailable relation, i.e.a relation whose schema is known but whose contents may be arbitrary, is a pair composed of the minimal elements of the Smyth and Hoare orderings: $(\bot, \emptyset)$, where $\bot$ represents a tuple of the unavailable relation filled with null values. On the contrary, the sandwich approximation for an available relation R is (R,R). The sandwich answer to a parachute query is also a trivial sandwich where either the lower or the upper approximation is an approximation of the form ($\bot$, S), or (S, $\emptyset$).

We have used the framework underlying approximations. We have however opted for a different representation where instead of providing an approximate answer to the original query of the form ($\bot$, S), or (S, $\emptyset$), we generate a parachute query whose precise answer is S, S being a generalized subset or superset of the answer to the original query.

Query relaxation has been implemented in the CoBase system [9]. A precise query is asked, if the answer to this query is empty, then the query is modified, i.e.one or several

[9] I don't understand this semantic point, but I guess it is important.

conditions are removed or modified, in order to be more general. The query is modified until the answer that the database system returns is not empty. In Cobase, query relaxation is based on a dynamically constructed type abstraction hierarchy. Each base table is partitioned into a set of first level abstractions. A hierarchy that groups these abstractions can then be built, where the higher levels of the hierarchy provide a more abstract data representation than the lower. Such type abstraction hierarchies are comparable to the dimension hierarchies in the OLAP models. The operations on the type abstraction hierarchy are *generalization* (moving up the hierarchy, i.e.roll-up), or *specialization* (moving down the hierarchy, i.e.drill-down).

Queries to the Cobase system are expressed in a modified version of SQL. This query language provides, first, additional operators for the WHERE clause that allow to express the notion of proximity using the type abstraction hierarchy, i.e.instead of specifying a value it is possible to specify a range where this value can occur. Second, a WITH clause has been added that allow to control the query relaxation mechanism, i.e.it is possible to indicate that query relaxation should not be performed on some attributes, or to order the attributes on which relaxation can be performed.

Here is an example of query relaxation in Cobase. Let us consider an airport table that associates the location of the airport and its runway_length. The following query

```
SELECT a_location
FROM airport
WHERE runway_length > 7500;
```

has an empty answer. Using *generalization* in the type abstraction hierarchy, this query can be rewritten into:

```
SELECT a_location
FROM airport
WHERE runway_length >= 4000;
```

whose answer is not empty.

In a sense, parachute queries are a form of relaxation of the original query. If one or several tables are unavailable, then a parachute query is picked, if it is defined, that only involves the available relations. In our case, however, the notion of relaxation concerns the tables in the FROM clause and not, as it is the case in Cobase for instance, the conditions in the WHERE clause.

Incomplete Databases Some work considers incomplete databases, i.e. databases where tuples from some relations are missing.

[17] 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. The available relation is expressed with a constraint

on the virtual relation. This constraint C defines the local completeness of an available relation with respect to the virtual relation: LC(R, R', C).

This paper describes the answer completeness problem, which can be expressed as follows. Given a set of local completeness statements, a query is said to be answer-complete if for any database instance, the query expressed on the virtual relations and the query expressed on the available relations denote the same answer. This problem can also be expressed in terms of independence of queries from updates.

The article describes an algorithm for deciding answer completeness. It also provides an algorithm for deciding whether in a given database state, the answer to a query is complete.

A local constraint defined between a virtual relation (denoted by the original query) and an available relation (denoted by the parachute query) can be a complementary notion of precipitate parachute query. Using this formalism, the parachute query denotes all the tuples denoted by the original query that satisfy a given constraint. This notion of precipitate parachute queries is more restrictive than ours; it allows to define more precisely the class of parachute queries that are a subset of the original query.

In [11], the author tackles the problem of unavailable data in a data cube. He considers the cases where some points in the multidimensional space are missing and cannot be computed. Here queries are aggregations over a union query. A measure, based on precision and cover, is introduced to compare different queries. The semantic relevance of query can be reduced to a distance problem here because, in a data cube, the multidimensional hierarchy is fixed.

An algorithm is provided for suggesting queries at a lower query measure than the given one, in case this query cannot be evaluated. An algorithm is also provided that computes the partial result to a query as the aggregate over a subset of a union.

The measure introduced in this paper is an alternative relation for defining precipitate parachute queries in the case of aggregates over union queries.

Multiple Query Optimization Reference [22] studies the problem of multiple query optimization. He 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.

The major issue in multiple query processing is the redundancy introduced by accessing the same data multiple times in different queries. A global access plan is derived based on the idea that temporary result sharing should be less expensive compared to a serial execution of queries.

Reference [22] presents two classes of algorithms. In the first class of algorithms, only locally optimal access plans are considered. The AS algorithm (Arbitrary Serial Execution) simply executes the locally optimal plans in an arbitrary order. The IE algorithm (Interleaved Execution) allows queries to be decomposed in smaller sub-queries that are the unit for execution: some sub-queries are shared between different queries.

In the second class of algorithms, all access plans are considered (even the plans that are not locally optimal). The HA (Heuristic Algorithm) is proposed. This algorithm consists in searching among local (not necessarily optimal plans) and building a global access plan by choosing one local plan per query. HA uses some heuristics (A* with a particular cost function) to restrict the number of permutations of plans it has to examine in order to find the optimal global access plan.

There are two main differences between our constrained algorithm and the HA algorithm. First, our objective we do not construct a global execution plan for several queries but we annotate the materializable shared component sub-queries in the execution plan for the original query. Second, our objective is not only to minimize I/Os; our objective is twofold: we aim at increasing parachute queries availability while limiting the impact on total work.

Query Scrambling Query scrambling is a dynamic optimization technique described in [4], [3], and [2]. It addresses the issue of unpredictable delays (initial delays or bursty arrivals) when processing queries in a wide-area network environment. Query scrambling is based on the idea that useful work can be done while one or several sources are delayed. Useful work is (1) materialization of base relations or intermediate results and (2) synthesis of new operators to process available data. When all work that could be done has been achieved, the query processing system simply stalls and waits for the delayed sources to produce their data.

Our technique is based on a similar idea that useful work can be done in case some sources are unavailable. One major difference is that we consider that, after a certain time, a data source is unavailable. We do not wait for this data source to produce a result. We thus introduce the sequential model of interaction to produce a complete answer. Returning a partial answer to the user representing the mediator state allows us to introduce parachute queries where the mediator state is exploited to provide the user with useful data before the complete answer is produced.

On-line Processing In [15], the notions of on-line aggregation and on-line processing are introduced. The objective of this work is to give to the user control over query processing. In the case of aggregations this is done by trading the accuracy of the answer with the time to obtain this answer.

The idea underlying parachute query is based on the same principle of giving information to the user as soon as possible. The contexts of our works, and the techniques we use are however completely different.

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. We have introduced our approach in [5].

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 implements parachute queries in the user interface. This evaluation model requires no modifications to the mediator. The unconstrained evaluation implements parachute queries on partial answers. This evaluation models requires only lightweight modifications to the interface and the run-time system of the mediator. The constrained evaluation 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 as a baseline for comparison to the other evaluations.

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

We simulated the constrained evaluation and demonstrate that query evaluation, parachute query extraction and incremental query evaluation are nearly as fast as in the unconstrained evaluation, and that the performance penalties are small in most cases. The constrained evaluation may have a negative impact on the availability of query 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.

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. We also wish to thank Bjorn Jònsson for his help concerning the simulator. 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] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic query operator scheduling for wide-area remote-access. Technical Report 3283, INRIA, 1997.

[4] L. Amsaleg, M. J. Franklin, A. Tomasic, and T. Urhan. Scrambling query plans to cope with unexpected delays. In *International Conference on Parallel and Distribution Information Systems (PDIS)*, Miami Beach, Florida, 1996.

[5] Ph. Bonnet and A. Tomasic. Partial answers for unavailable data sources. In *Proceedings of the Conference on Flexible Query Answering Systems*, Roskilde, Denmark, 1998.

[6] 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.

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

[8] 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.

[9] W. Chu, H. Yang, K. Chian, M. Minock, G.. Chow, and C. Larson. Cobase: A scalable and extensible cooperative information system. *Journal of Intelligent Information Systems*, 6(1), 1996.

[10] 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.

[11] C. Dyreson. Information retrieval from an incomplete data cube. In *Proceedings of the 22nd International Conference on Very Large Databases*, Bombay, India, 1996.

[12] 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.

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

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

[15] J. M. Hellerstein, P. J. Haas, and H.J. Wang. Online aggregations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997.

[16] 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.

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

[18] 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.

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

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

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

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

[23] 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.

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

[25] 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.

[26] 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.

A Glossary

answering queries using views (AQUV) An algorithm which takes a query and set of views and rewrites the query to use the views, if possible.

available A source is available if it returns sub-answers to a sub-query (cf. 5.2.1).

availability of complete answer The probability that a complete answer can be obtained from the evaluation of a query or an incremental query.

availability of a parachute answer The probability that a parachute query answer can be extracted from a mediator state.

classical evaluation An evaluation model where queries and parachute queries are treated independently and an incremental query is identical to its corresponding query.

complete answer The answer to the query.

configuration of sources A set of available and unavailable sources.

constrained evaluation An evaluation model where a query and its parachute queries are simultaneously optimized in the mediator.

construct The algorithm for constructing the incremental query from the mediator state.

evaluate The algorithm for evaluation of queries. This algorithm materializes available sub-queries according to a given materialization policy.

extract The algorithm which extracts the parachute query answer from the mediator state.

generalized shared sub-query (GSSQ) The most specific query whose body contains both the query and the parachute query

incremental answer The answer to the incremental query. This answer is the same as the complete answer if the required sources are available.

incremental query A query which computes the query answer from the mediator state and the sources.

leaves The leaves of the query execution plan in the mediator (*not* the leaves of the query execution plan).

materialized relation A relation in the mediator state which is the answer to a sub-query.

materialization policy The policy for determining which answer to a sub-query is materialized. This paper considers the following policies: materialize nothing, materialize leaves, materialize maximal sub-query, and materialize shared component sub-query.

maximal parachute query The unique parachute query in the precipitate class defined by a given query and a configuration of sources.

maximal available sub-query The largest sub-query which only accesses available sources in a given configuration of sources.

mediator The system for processing the answers to queries, parachute queries and incremental queries.

mediator state The state of the mediator after the execution of the evaluate algorithm.

parachute answer The answer to a parachute query.

parachute query A query used to extract information from the mediator state.

parachute query generation algorithm An algorithm which generates a subclass of the precipitate class given a query and a set of configuration of sources.

partial answer A handle to the mediator state generated in the case that the complete answer cannot be produced.

precipitate class The class of parachute queries defined by the precipitate relation to a query (subset, superset, generalized subset or generalized superset).

query A select-project-join (SPJ) relational query or a union of SPJ queries.

range restricted variable A variable which appears in a non-built-in predicate in the body of a query.

shared component sub-query (SCSQ) A sub-query shared between a query and a parachute query.

source An autonomous source of data.

sub-answer The answer to a sub-query.

sub-query A part of a query which results from decomposing a query. Sub-queries are sent to the respective data sources to produce sub-answers. Also, sub-queries represent materialized relations in the mediator state. Every subtree of an execution plan has an associated sub-query.

time to first answer The time required to execute the evaluate algorithm for a query. In the case that all sources are available, this is the time to the complete answer, otherwise it is the time to the partial answer.

time to complete answer The time required to execute the evaluate algorithm when all sources are available.

time to incremental answer The time required to execute the evaluate algorithm on the incremental query and the mediator state.

time to parachute answer The time required to execute the extraction algorithm.

time to partial answer The time required to execute the evaluate algorithm when some sources are unavailable.

unconstrained evaluation An evaluation model where a query is evaluated in the mediator and then the parachute query answers are extracted from the mediator state.

B Precipitate Class of Parachute Queries

B.1 Conjunctive Queries

We show, here, that the set of parachute queries obtained from a conjunctive query Q by removing all possible combinations of predicates in the body of Q and the generalized superset relation form a lattice.

More formally the set of parachute queries obtained from a conjunctive query Q by removing all possible combinations of predicates in the body of Q, that we note here pqgen(Q), can be expressed as follows:

> pqgen(Q) = {`RemovePredicates`(Q, P) | P $\in \mathcal{P}$(Pred(Q)), where $\mathcal{P}$(Pred(Q)) denotes the powerset of Pred(Q) and Pred(Q) is the set of predicates in the body of Q}.

We show that a parachute query obtained with `RemovePredicates` is the generalized superset of a second one if and only if the set of predicates removed from Q to obtain the second parachute query is included in the set of predicates removed from Q to obtain the first one.

[24] defines theorems where containment mappings are used to show containment of conjunctive queries. We extend these theorems and use body containment mappings to show that a generalized superset relations holds between two conjunctive queries. In the following, Q and Q' are conjunctive queries.

Theorem 1 *Q is a conjunctive query, Pred1 and Pred2 are subsets of Pred(Q).* $Q \xrightarrow{\supseteq} Q'$ $\Leftrightarrow$ *(1) there exists a body containment mapping from Q to Q', (2) all variables in head(Q) corresponds to a variable in head(Q') with this mapping and (3) the head(Q) and head(Q') have the same predicate symbol*

Proof The proof consists of two parts: (if) We consider a body containment mapping μ from Q to Q' such that all variables in head(Q) corresponds to a variable in head(Q') with this mapping and the head(Q) and head(Q') have the same predicate symbol. Let D be any database.

- Every tuple t' in Q'(D) is produced by some substitution σ that makes Q' subgoals all become facts in D;
- the substitution $\sigma \circ \mu$ is a transformation for variables of Q that produces t' such that :
  - $\sigma \circ \mu(\text{subgoal(Q)}) = \sigma(\text{subgoal(Q')})$. Therefore it is in D;
  - $\sigma \circ \mu(\text{head(Q)}) = \sigma(\pi_Q(\text{head(Q')})) = \sigma(\pi_Q(\text{t'}))$.
- thus every t' in Q'(D) can be projected on the attributes of Q to obtain a tuple in Q(D); i.e. $Q \xrightarrow{\supseteq} Q'$.

(Only If) We consider $Q \xrightarrow{\supseteq} Q'$.

- D is frozen Q', i.e. the database consisting of all the subgoals of Q', with a unique constant chosen for each variable in Q';
- Q'(D) contains t';
- from our hypothesis $Q \supseteq \pi_Q(Q')$ and thus Q(D) contains $\pi_Q(t')$;
- σ is a transformation that maps each subgoal of Q to a tuple of D and yields $\pi_Q(t')$;
- σ' is a transformation that maps each variable X of Q to the variable of Q' that corresponds to $\sigma(X)$;

- σ' is a body containment such that all variables in head(Q) corresponds to a variable in head(Q') with this mapping and the head(Q) and head(Q') have the same predicate symbol:
  - each subgoal of Q is mapped by σ to some tuple in D, which is by definition a frozen version of some subgoals of Q'. Thus σ' maps each subgoal of Q to a subgoal of Q'. σ' is a body containment mapping.
  - The head of Q is mapped by σ to π_Q(t'), which is the frozen version of π_Q(Q'). As a result the mapping σ' maps the head predicate of Q into the head predicate of Q', and each variable in head(Q) to a variable in head(Q').

Theorem 2 *Q is a conjunctive query, Pred1 and Pred2 are subsets of Pred(Q). Pred1* $\subseteq$ *Pred2* $\Leftrightarrow$ `RemovePredicates`*(Q, Pred2)* $\xrightarrow{\supseteq}$ `RemovePredicates`*(Q, Pred1).*

Proof Identity variable mapping transforms `RemovePredicates`(Q, Pred2) into itself and thus into all conjunctive queries whose body contains at least all the predicates of `RemovePredicates`(Q, Pred2). It also transforms its head variables into the head variables of all conjunctive queries whose head contains at least all its head variables. By construction, `RemovePredicates`(Q, Pred1) (1) contains at least all the predicates of `RemovePredicates`(Q, Pred2) and (2) contains at least all head variables of `RemovePredicates`(Q, Pred2) if and only if Pred1 $\subseteq$ Pred2. All parachute queries obtained from the same query Q with `RemovePredicates` have the same head predicate symbol.

If `RemovePredicates`(Q, Pred2) is the empty query, then `RemovePredicates`(Q, Pred2) $\xrightarrow{\supseteq}$ `RemovePredicates`(Q, Pred1) for all Pred1.

We now show that {`pqgen`(Q, M, $\emptyset$), $\xrightarrow{\supseteq}$} is a superior semi-lattice. To do that we need to show that, for any two queries in `pqgen`(Q), there exists a unique minimum to the set of queries that are contained in both of them. According to the definition of `pqgen`, `RemovePredicates`(Q, Pred1) and `RemovePredicates`(Q, Pred2) (where Pred1 and Pred2 are subsets of Preds(Q)) are two elements in `pqgen`(Q). The set of elements that are superior to any two elements in `pqgen`(Q, M, $\emptyset$) is defined as follows

$$\sup(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2)) = \{\ \texttt{RemovePredicates}(Q, L) \mid \texttt{RemovePredicates}(Q, Pred1) \xrightarrow{\supseteq} \texttt{RemovePredicates}(Q, L) \text{ and } \texttt{RemovePredicates}(Q, Pred2) \xrightarrow{\supseteq} \texttt{RemovePredicates}(Q, L)\}$$

We apply Theorem 2 and we obtain a new equation that defines the set of elements that are superior to any two elements in `pqgen`(Q, M, $\emptyset$).

$$\sup(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2)) = \{\texttt{RemovePredicates}(Q, L) \mid L \subseteq Pred1 \text{ and } L \subseteq Pred2\}$$

which is equivalent to the following equation (because sets with inclusion forms a lattice)

$\sup(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2)) = \{\ \texttt{RemovePredicates}(Q,$ L) $\mid$ L $\in$ inf(Pred1,Pred2) $\}$

We can now define the unique minimal element in the set of elements that are superior to any two elements in pqgen(Q, M, $\emptyset$) as follows

$\min(\sup(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2))) = \texttt{RemovePredicates}(Q,$ L) such that RemovePredicates(Q, L) $\xrightarrow{\supseteq}$ RemovePredicates(Q, L') for all L' $\in$ inf(Pred1,Pred2)

and thus

$\min(\sup(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2))) = \texttt{RemovePredicates}(Q,$ L) such that L $\supseteq$ L' for all L' $\in$ inf(Pred1, Pred2).

and finally

$\min(\sup(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2))) = \texttt{RemovePredicates}(Q,$ L) such that L = max(inf(Pred1,Pred2)) = Pred1 $\cap$ Pred2.

Pred1 $\cap$ Pred2 is unique. We have thus shown that the minimum element of the set of elements that are superior to any two elements in pqgen(Q, M, $\emptyset$) exists and is unique. As a result, {pqgen(Q, M, $\emptyset$), $\xrightarrow{\supseteq}$} is a superior semi-lattice.

We now prove that {pqgen(Q, M, $\emptyset$), $\xrightarrow{\supseteq}$} is also an inferior semi-lattice. We have to show that there exists a unique maximal element in the set of elements that are inferior to any two queries in pqgen(Q, M, $\emptyset$). With the same demonstration as above we obtain

$\max(\inf(\texttt{RemovePredicates}(Q, Pred1), \texttt{RemovePredicates}(Q, Pred2))) = \texttt{RemovePredicates}(Q,$ L) such that L = min(sup(Pred1, Pred2)) = Pred1 $\cup$ Pred2.

Pred1 $\cup$ Pred2 is unique. This shows that the maximal element in the set of elements that are inferior to any two elements in pqgen(Q, M, $\emptyset$) exists and is unique. As a result{pqgen(Q, M, $\emptyset$), $\xrightarrow{\supseteq}$} is an inferior semi-lattice. We can thus conclude that for a conjunctive query Q, {pqgen(Q, M, $\emptyset$), $\xrightarrow{\supseteq}$} being an inferior and a superior semi-lattice is a lattice.

B.2 Union Queries

We show, here, that the set of parachute queries obtained from a union query Q by removing all possible combinations of rules in Q and the subset relation form a lattice.

More formally the set of parachute queries obtained from a union query Q by removing all possible combinations of rules in Q, that we note here pqgen(Q), can be expressed as follows:

pqgen(Q) = $\{\texttt{RemovePredicates}(Q, R) \mid R \in \mathcal{P}(\text{Rules}(Q))$, where $\mathcal{P}$(Rules(Q)) denotes the powerset of Rules(Q) and Rules(Q) is the set of rules in Q$\}$.

We show that a parachute query obtained with `RemovePredicates` is the generalized subset of a second one if and only if the set of rules removed from Q to obtain the second parachute query is included in the set of rules removed from Q to obtain the first one.

Theorem 3 *Q is a union query, Rules1 and Rules2 are subsets of Rules(Q). Rules1* $\subseteq$ *Rules2* $\Leftrightarrow$ `RemovePredicates`*(Q, Rules2)* $\subseteq$ `RemovePredicates`*(Q, Rules1).*

Proof Identity is a containment mapping from each rule of `RemovePredicates`(Q, Rules2) into a rule in `RemovePredicates`(Q, Rules1), if and only if Rules2 is a subset of Rules1. It follows from [24] that `RemovePredicates`(Q, Rules2) is a subset of `RemovePredicates`(Q, Rules1).

Using theorem 3 and the same mechanisms as in the previous section, we can show that for a union query Q, {`pqgen`(Q, M, $\emptyset$), $\subseteq$} is a lattice.

Unit´e de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unit´e de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unit´e de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unit´e de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unit´e de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
http://www.inria.fr
ISSN 0249-6399