INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Partial Answers for Unavailable Data Sources

Philippe Bonnet and Anthony Tomasic

N° 3127

Mars 1997

THÈME 3

ISSN 0249-6399

Partial Answers for Unavailable Data Sources

Philippe Bonnet* and Anthony Tomasic†

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

Rapport de recherche n ° 3127 — Mars 1997 — 24 pages

Abstract: Many heterogeneous database system products and prototypes exist today; they will soon be deployed in a wide variety of environments. All existing systems suffer from an *Achilles' heel*: if some sources are unavailable when accessed, these systems either silently ignore them or generate an error, i.e. they *ungraciously fail*. This behavior is improper in environments where there is a non-negligible probability that data sources cannot be accessed (e.g., Internet). In this paper, we propose a novel approach to this issue where, in presence of unavailable data sources, the answer to a query is a *partial answer*. A partial answer is itself a query that results from theo partial evaluation of the original query; it is composed of the data that have been obtained and processed during the evaluation and of a representation of the unfinished work to be done. Partial answers can be resubmitted to the system in order to obtain the final answer to the original query, or another partial answer. Additionally, the application program can extract information from a partial answer through the use of a secondary query. This secondary query is called a *parachute query*. In this paper we give a taxonomy of partial answers and parachute queries. We present algorithms for the evaluation of queries in presence of unavailable data sources, and we describe an implementation.

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

Réponses partielles pour sources de données indisponibles

Résumé : De nombreux systèmes de bases de données hétérogènes existent aujourd'hui, sous forme de prototype ou de produits; ils seront sous peu deployés dans un grand nombre d'environnements. Tous ces systèmes existants souffrent cependant d'un *talon d'Achille*: si certaines sources de données sont indisponibles lorsqu'elles sont accédées, les systèmes les ignorent ou génèrent une erreur. Ce comportement n'est pas satisfaisant dans des environnements où il existe une probabilité non négligeable qu'un site soit indisponible (e.g., Internet). Dans cet article, nous proposons une approche originale où, en présence de sources de données indisponibles, la réponse à une requête est une réponse partielle. Une réponse partielle est elle même une requête qui résulte de l'évaluation partielle de la requête initiale; elle est composée des données obtenues lors de l'évaluation et d'une représentation du travail restant à accomplir. Les réponses partielles peuvent être soumises au système pour obtenir la réponse finale la requête initiale, ou bien une autre réponse partielle. Par ailleurs, une application peut extraire de l'information d'une réponse partielle en utilisant une requête secondaire. Nous appelons cette requête secondaire une /em requête parachute. Dans ce papier, nous présentons une taxonomie des réponses partielles et des requêtes parachutes. Nous présentons des algorithmes pour l'évaluaiton de requêtes en présence de sources de données indisponibles, et nous décrivons notre implémentation.

Mots-clé : Bases de Donnes htrognes distribues, traitement de requtes, valuation partielle, donnes indisponibles

1 Introduction

Many current application environments provide declarative access to a wide variety of heterogeneous data sources. Research into improving these systems has produced many new results [8, 7, 17, 1, 4, 12, 13, 3, 15, 11], which are being incorporated into prototypes and commercial products. However, to the best of our knowledge, these systems all fail ungraciously in the presence of unavailable data sources. They either assume that all data sources are available, report error conditions, or silently ignore unavailable sources.

In this paper we present a novel solution to this problem of ungracious failure. Our solution relies, first, on the *partial evaluation of queries* to produce a *partial answer*, and second, on the ability to *extract information from the partial answer of a query.* By partial evaluation we mean that part of the work in processing a query is accomplished and then query processing stops due to an *interrupt condition.* In this paper, the interrupt condition is based on the unavailability of a data source, but it can be more general. It could be, for instance, an alarm that is triggered after a certain amount of time. When the interrupt condition is reached and query processing stops, a partial answer is generated. The partial answer represents both the work accomplished during the partial evaluation and the work that remains to be accomplished to provide the final answer to the original query.

The partial answer contains a query that can be *resubmitted* to the query processing system and query processing will continue until either the final answer is produced or the interruption condition again occurs, generating another partial answer. Under our working assumptions, the final answer will always equal the answer to the original query as if the interrupt condition and the partial evaluations never occurred. Given a partial answer we can extract useful information from it and return this partial information to the user.

1.1 Example

To make our solution more concrete, let us consider the following example, executed on the Disco [18] heterogeneous distributed database system. Suppose we have a mediator [22] that accesses two distributed data sources, `companies` for companies, `bids` for bids on work, and a local relation `regions` that classifies regions into districts. The schemas of these data sources are given in Appendix A. We ask the query *select the companies and the bids for work located in the Rhne-Alpes region and return the name and activity of the companies with the markets of bids on work.* In Disco this query is expressed in an OQL-like syntax as

```
from x in companies, y in bids, z in regions
select x.name, x.activity, y.market
where x.district = y.district and z.region = "Rhone Alpes"
  and x.district = z.district;
```

The query starts execution. The interrupt condition occurs if *a data source is unavailable.* If all sources are available, the query completes and returns the answer

```
{["ATELIER SOIXANTE QUATORZE","Bureaux
d'tudes - btiments et travaux publics"," Objet du march :
construction d'un groupe scolaire. Lieu d'excution : avenue
du Stade, 74970 Marignier. "]
```

[12 more tuples not shown]

```
 ["BALTHAZARD ET COTTE","Matires premires pour
le btiment
et les travaux publics"," Objet du march : extension et
rhabilitation du collge Les Mntriers. Lieu d'excution : au
collge Les Mntriers, 21, rue de Landau, 68150 Ribeauvill. "]}
```

exactly as in any heterogeneous database system. However, for example, if the `bids` data source is unavailable, Disco responds with an object, say `o1` that represents the partial answer and its environment. This object represents the partial answer interface to the user. The `unavailable` method of this object returns the list of sources that are unavailable. Thus o1.unavailable() returns the set `(bids)`.

Another method of this object returns a query that can be re-submitted as a new query. The following string is obtained, with some details suppressed:

```
from x0 in (from x1 in (from x3 in (from x4 in (bids) select x4)
select x3), x2 in (from x5 in ({["ATELIER SOIXANTE QUATORZE","Bureaux
d'tudes - btiments et travaux publics","74","1 r Jean Jaurs BP362
74012  Annecy Cedex ","0450516945","0450528088","Rhone
Alpes","74"],
```

[7 more tuples not shown]

```
) select x5) select
x1.parution, x1.valid, x1.district, x1.market, x2.name, x2.activity,
x2.district, x2.address, x2.tel,x2.fax,x2.region,x2.district
where x1.district = x2.district) select x0.name, x0.activity,
x0.market;
```

1.2 Discussion

An alternative to dealing with partial answers in applications is *replication* of data sources to increase the availability of all data sources to the point that queries almost always execute. Replication suffers from an economic disadvantage – 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 and update performance. In addition, in an environment with autonomous data sources, replication may not be possible simply because the data source forbids it. Finally, partial answers are completely compatible with replication – in the case that a data source is replicated, the probability that it will trigger our example interrupt condition is simply smaller.

In the example, the user re-submits the partial answer as a new query to continue partial evaluation. However, if the interrupt condition still holds (e.g., data source `bids` is still unavailable), the re-submission will accomplish little. The new query will be re-optimized, but its execution will stop immediately. Thus, in this application of partial answers, the *notification* to the user of the availability of a data source would improve matters. This issue is future work.

Finally, in our example we state that, under our working assumptions, the re-submission of the partial answer as a new query will result in the same answer as the original query. This statement holds under the assumption that *no update relevant to the query occurs between the start of query processing and the final answer.* By a relevant update, we mean an update that changes the answer to the query. For the rest of this paper, we keep this working assumption. Handling updates is also future work.

1.3 Contributions

In summary, this paper describes a novel approach to handling unavailable data sources during query processing in heterogeneous distributed databases. We describe in Section 2 a framework of related technical problems and solutions that partial answers induces. In particular, we define *parachute queries*, a general method for extracting information from partial answers. In Section 3 we describe an algorithm for generating partial answers and discuss implementation trade-offs for this algorithm. Section 4 describes an implementation of one collection of these trade-offs. Section 5 describes a query processing example which involves partial answers. In Section 7 we conclude the paper by summarizing our results and discussing future work.

2 Framework

Our approach to the problem of unavailable data sources offers two aspects. First, we consider the evaluation of queries in presence of unavailable data sources. Second, in case some data sources are unavailable, we study the problem of extracting information gathered during the evaluation.

Let us study the behavior of a query processing system that integrates our approach. An application program submits a query **Q**, which involves several data sources. If all data sources are available, then the system returns an answer **A(Q)**. If one or several sources are unavailable, the system returns a partial answer **P(Q)**.

We define a partial answer **P(Q)** as a pair **(Q',E)**. **Q'** is a query that results from the partial evaluation of the original query and **E** is the environment of the partial answer. We provide an algorithm in Section 3 that constructs **Q'**. Generally, **Q'** is composed of two parts: a part that contains the data accessed from data

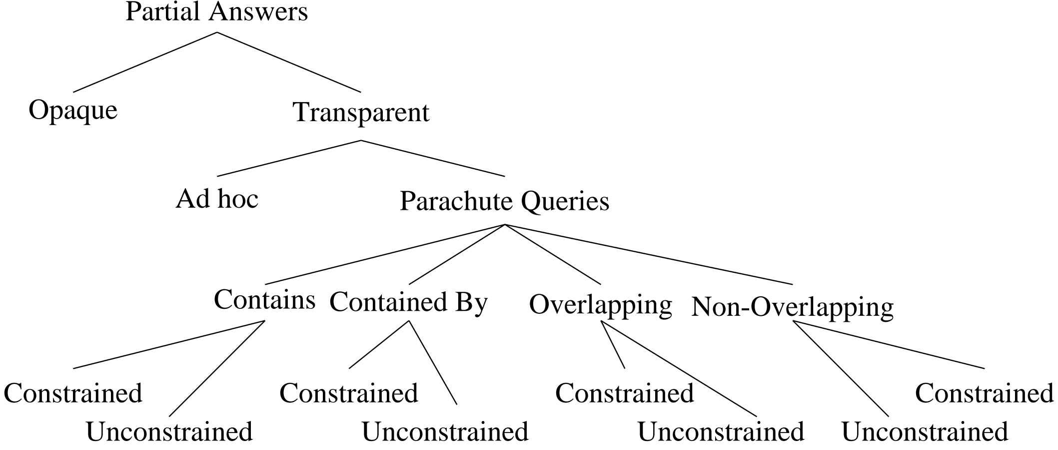

Figure 1: Taxonomy of Partial Answers and Parachute Queries

sources, and a part that is a query on the data sources that were unavailable. **E** contains additional information that has been gathered during the evaluation, e.g., the list of data sources that were available or unavailable.

At a later point in time, **Q'** can be re-submitted to the system. Our algorithm for the generation of **Q'** only involves the data sources that were unavailable when **Q** was evaluated. The system returns either an answer **A(Q')**, or another partial answer **P(Q')**, depending on the interrupt condition, i.e., the availability of the data sources involved in **Q'**. When **Q'** is submitted to the system, the query processor considers it in the same way as a plain query, and it is optimized. The execution plan that is used for **Q'** is generally different from the execution plan used for **Q**.

If the sources that were unavailable during the evaluation of **Q** are now available, then an answer **A(Q')** is returned. Under our working assumption that no updates are performed on the data sources **A(Q')** is the answer to the original query: **A(Q') = A(Q)**.

If some of the sources that were unavailable during the evaluation of **Q** are still unavailable, then another partial answer is returned, **P(Q') = (Q",E')**. Possibly successive partial answers are produced before the final result can be obtained.

For example, consider the original query given in the introduction. It involves two data sources (companies and bids), and a local class (region). Originally, both remote sources are unavailable; a partial answer is generated. This partial answer contains a query that still involves the companies and bids data sources. It is later re-submitted to the system, optimized and evaluated. During the second evaluation, the bids data source is available; the companies data source is still unavailable. The system returns another partial answer, which contains a query involving only the companies data source. It is later resubmitted; this time, the companies data source is available and the system returns the final result.

Using successive partial answers, there is no need to have every data sources available simultaneously to produce a result. It suffices that a data source is available during the evaluation of one of the successive partial answers to ensure that the data from this source is used for the final result. The system can produce the final result if each source is available at least once during the successive evaluations. This is an increase in reliability with respect to the systems that can produce the final result only if all sources are available simultaneously.

When an application program receives a partial answer **P(Q)**, it can resubmit **Q'**, or extract information from **E**. However, the information extracted from **E** is essentially specific information coded into the interface, such as the list of available or unavailable data sources. Intuitively, there is more information contained in a partial answer. Considering the example in the introduction, in the case that the `bids` data source is unavailable, the set of `companies` that appear in the query *is* available, and at least some subset of it may be contained in the partial answer. We introduce a parachute query **C** to denote data an application would like to extract from a partial answer. There is a function **M(P(Q),C)** that attempts to return the answer to a parachute query with respect to a partial answer. If the data denoted by **C** can be obtained, then it is returned, otherwise the answer to **C** is null. The application program may ask several parachute queries.

In this section, we present a taxonomy of the issues related to partial answers and parachute queries, Figure 1. This classification of the different types of partial answers and parachute queries that we have identified includes some implementation considerations; it gives us a framework for the different possible research directions, and it provides an overview of the problems raised by partial answers and parachute queries.

2.1 Partial Answers

We make a primary division between *opaque* partial answers, that are simply a step towards the final result, and *transparent* partial answers, from which data can be extracted.

2.1.1 Opaque Partial Answers

When an application program receives an opaque partial answer, it knows that the original query could not be completely evaluated because the interrupt condition occurred. The application program can re-submit the opaque partial answer at a later point in time to obtain the result of the original query. It cannot extract information. Even this minimal functionality is very useful, since an answer can be returned even if all data sources are not available simultaneously. The probability that an application program obtains an answer is thus increased.

The central issue concerning opaque partial answers concerns the *amount of work accomplished during partial evaluation*. (In fact, this issue also concerns transparent partial answers, but we discuss the issue here.) The amount of work depends on the interrupt condition. In our application, the work accomplished is strongly related to the issues raised in the work on *query scrambling* [2]. Query scrambling tackles the problem of delays in the responses of the data sources by *rescheduling operations* in response to delays. In a simple run-time system, each data source involved in query is contacted sequentially. When a data source delays, the run-time system blocks. In a query scrambling run-time system, the delayed source is *skipped* and other parts of the run-time system are executed. We use this technique to increase the amount of work accomplished during partial evaluation, as detailed in Section 3.

The main advantage of opaque partial answers is that the system is free to generate and maintain partial answers in the most efficient way. The obvious disadvantage is that a partial answer may contain useful information that cannot be exploited, as it is not possible to extract it.

2.1.2 Transparent Partial Answers

Transparent partial answers permit the extraction of information. We classify information as *ad hoc* or as *specific*. We use parachute queries to specify specific information.

The nature of ad hoc information depends on the general system in which partial answers are used. In our application, partial answers are embedded in a heterogeneous distributed database with multiple data sources, so we provide various functions related to this system. We provide a function to extract the set of sources available during query processing and the set of sources unavailable during query processing. The application may track the sources that are often unavailable or available and take decisions on whether to submit the query again, or not, based on this information.

A problem is the interpretation an application program gives to a data source name, since the application program actually deals with global schema that does not explicitly state data sources; the connection between the global schema and data sources is provided by the database administrator. We leave this problem for future work.

Most importantly, the users want to extract data that have been obtained from a data source and processed during partial evaluation. This presents a problem, since the execution plan for the query, and therefore the structure of the intermediate results during execution of the query, are determined by the query optimizer.

Consider the three data sources of the introduction: bids, companies, and regions; and the query of the introduction.

Now, let us consider that the bids data source is unavailable when the query is evaluated; a partial answer is returned. Depending on the execution plan, intermediate results will either contain the materialization of the data obtained from the company and region relations, or the result of a join between these two relations. Linking application programs to the result of query optimization is clearly undesirable, since the application program will have to cope with both possible materializations.

To solve this problem the application program specifies a parachute query $\mathbf{C}$ and an algorithm $\mathbf{M}$ that *may* extract the answer to $\mathbf{C}$ from the partial evaluation $\mathbf{P(Q)}$. The likelihood that $\mathbf{M}$ succeeds in extracting the answer to $\mathbf{C}$ depends on first the relationship between $\mathbf{C}$ and $\mathbf{Q}$ (cf. Section 2.2) and second on the implementation of $\mathbf{M}$ (cf. Section 2.2.1.)

The main advantage of transparent partial answers is that useful and meaningful information can be extracted part way through the computation of a query. (For example, the interrupt condition could simply be an alarm that is triggered after a certain amount of time.) The disadvantage is that, in order to provide this information, general extra data structures must be maintained and the run-time system is more complicated.

In addition, some query execution plans may be rejected because they prevent the evaluation of **C** by **M**. This may restrict the efficiency and the simplicity of the system.

2.2 Parachute Queries

We distinguish four different kinds of parachute queries **C** depending on how the answers to the parachute queries relate to the answer of the original query **Q** *independently of the state of the underlying data sources.* We call the four kinds of parachute queries *contains*, *contained-by*, *overlapping* and *non-overlapping. Contains* means that for all states of the data $\mathbf{A(C)} \supset \mathbf{A(Q)}$, i.e., the answer to the parachute query contains the answer to the initial query (in the sense of containment defined in [20]). *Contained-by,* $\mathbf{A(C)} \subset \mathbf{A(Q)}$ means the opposite. *Overlapping* , $\mathbf{A(C)} \cap \mathbf{A(Q)} \neq \emptyset$, means that the answers overlap but do not contain each other. *Non-overlapping* ($\mathbf{A(C)} \cap \mathbf{A(Q)} = \emptyset$) means that the answer to the parachute query and answer to the original query never overlap, regardless of the data in the underlying data sources. The equality case of $\mathbf{A(C)}=\mathbf{A(Q)}$ reduces to the case of opaque partial answers.

Let us take again the example of the introduction. An example of a contains parachute query is *select the companies and the bids of work located in any region and return the name and activity of the company with the markets of bids on work.* An example of parachute query contained-by the original query is *select the company* Balthazard et Cotte *and the bids of work located in the Rhne-Alpes region and return the name and the activity of this company with the markets of bids on work.* Overlapping parachute queries are difficult to generate in our example. Consider a query *select the names of employees with a salary* < 5. An overlapping parachute query is *select the names of employees with a salary* > 3. Returning to the query in the introduction, an example of a non-overlapping parachute query for the query is *select the companies located in region Rhne-Alpes.* This example is particularly interesting because the parachute query corresponds to an intermediate result produced by some query execution plans. Parachute queries can thus be a natural way for the application program to specify and use intermediate results that have been produced during the partial evaluation.

2.2.1 Implementation Issues

Until now, we have defined parachute queries with respect to partial answers. That is, a parachute query is asked after partial evaluation has taken place. In this case, the parachute query was not known during the optimization. The information that is gathered during partial evaluation is not targeted at a particular parachute query.

However, a parachute query can be asked together with the original query. We define the former case as *unconstrained optimization* and the latter case as *constrained optimization.* In the latter case, the parachute query may be considered when optimizing the original query. Thus, a new, additional, goal of the optimizer, beyond optimizing the query, is to increase the likelihood that an answer to the parachute query is produced in the case that the interrupt condition occurs.

Constrained Optimization A parachute query is submitted together with the original query. Both queries are simultaneously optimized. Depending on the relationship between the parachute query and the original query, possibly a unique execution plan is generated, where the execution plan for the parachute query corresponds to a subtree of the execution plan for the original query. In other cases, two entirely independent plans are produced. Optimization of parachute queries is related to the problem of simultaneous optimization of multiple query [16].

Providing the parachute query together with the original query has the following advantage. It increases the likelihood that the system can return an answer to the parachute query. The disadvantage is that the *cost function of the optimizer has changed*: it now includes computations based on the parachute query and the likelihood that the interrupt condition will occur. For instance, if the optimizer predicts that the interrupt condition will occur with high probability, it will favor the plan that optimizes the execution of the parachute query.

Unconstrained Optimization The parachute query is submitted once the original query has been evaluated. Query optimization proceeds classically, and only concerns the original query. If the interrupt condition occurs, a partial answer is produced. The application program can extract information from this partial answer by asking one or several parachute queries.

If the optimizer is not modified, the likelihood of $\mathbf{M(P(Q),C)}$ producing an answer is very low. The parachute query **C** must correspond to an intermediate result that is in the environment **E** at the exact moment

of the interrupt condition being satisfied. For instance, the parachute query may correspond to an intermediate result that has been produced but destroyed after it has been used to produce another intermediate result. In this case, the system cannot give an answer to the parachute query.

The unconstrained optimization does not impact the way execution plans are produced, it is thus simpler to implement than the constrained optimization that impacts the run-time system and the optimizer. Getting good performance is a problem that we can handle in the classical optimization framework. However, the likelihood of getting an answer to a parachute query depends very much on the output of the optimizer, and is thus restricted. The optimizer can easily be extended to increase the likelihood of matching a parachute query, for instance by retaining all intermediate results. However, this heuristic is completely unguided by any knowledge of the nature of the parachute query.

2.2.2 Non-Deterministic Set of Queries

A generalization of the idea of parachute query consists in asking to the system a set of queries, instead of a single query. The system answers at least one of the queries. The behavior of the system is non-deterministic: the evaluation stops when an answer is found, and this answer is returned to the application program. Non-deterministic queries for handling unavailable data sources is an area of future research.

3 Algorithms

In this section, we describe an unconstrained system for producing partial answers. Our system is oriented towards the application described in the introduction: the architecture is based on a mediator architecture for accessing heterogeneous data sources; however, the algorithms we describe have broad application.

Our system consists of a run-time system for the evaluation of queries, and of a component for the extraction of information from partial answers. The run-time system is based on the iterator model [9] that is slightly modified to handle partial evaluation. First, the run-time system extends the iterator execution model with a simple form of query scrambling to permit execution to proceed in the presence of unavailable data sources. Second, the run-time system implements an interrupt condition to trigger partial evaluation. Third, it includes an algorithm for the construction of partial answers after the interrupt condition has been satisfied.

The component for the extraction of information contains an algorithm for the processing of parachute queries. These aspects are described below. They are followed by a discussion on the implementation trade-offs.

3.1 Architecture

For our algorithms, we consider an architecture that involves an application program, a mediator, wrappers, and data sources. During query processing, the application program issues a query to the mediator. The mediator transforms the query into *any* valid execution plan consisting of subqueries and of a composition query. The mediator then evaluates the execution plan. Evaluation proceed by issuing subqueries to the wrappers. Each wrapper that is contacted process the subqueries by communicating with the associated data source and returning subanswers. If all data sources are available, the mediator combines the subanswers by using the composition query and returns the answer to the application program. In case one or several data sources are unavailable, the mediator returns a partial answer to the application. The application extracts information from the partial answer by asking a parachute query.

3.2 Query Evaluation

The algorithm for query evaluation follows the iterator model. The query optimizer generates a tree of operators that computes the answer to the query. The operators are relational-like, such as `project`, `select`, etc. (For clarity, in this paper we describe operators in terms of operations instead of the corresponding physical operator algorithms. However our implementation uses physical operators.) Each operator supports three procedures: `open`, `get-next`, and `close`. The procedure `open` prepares each operator for producing data. Each call to `get-next` generates one tuple in the answer to the operator, and the `close` procedure performs any clean-up operations.

The operator `submit` contacts a wrapper to process a subquery. During the `open` call to `submit` a network connection to the wrapper is opened. In this paper, we make a working assumption about the behavior of wrappers and data sources: if the `open` call to the wrapper succeeds, then the corresponding data source is available and will deliver its subanswer without problems. If the `open` call fails, then the corresponding data

source is unavailable. This behavior implies that each data source can be classified as *available* or *unavailable* according to the result of the `open` call.

A second working assumption is that, during execution, only the mediator sends subqueries to wrappers. Two wrappers cannot communicate directly with each other.

Third, we assume that no updates relevant to a query are performed between the moment the processing of this query starts and the moment where the processing related to this query ends, because the final answer is obtained, or because the user does not resubmit a partial answer.

We describe a two steps evaluation of queries. The first step, the `eval` algorithm, performs a partial evaluation of the execution plan with respect to the available data sources. If all the sources are available, the result of the first step is the answer to the query (a set of tuples). If at least one source is unavailable, the result of the first step is an annotated execution plan. The second step, the `construct` algorithm, constructs a partially evaluated query from the annotated execution plan. A partial answer built from the partially evaluated query and the annotated execution plan can then be returned. See Figure 2 for an example.

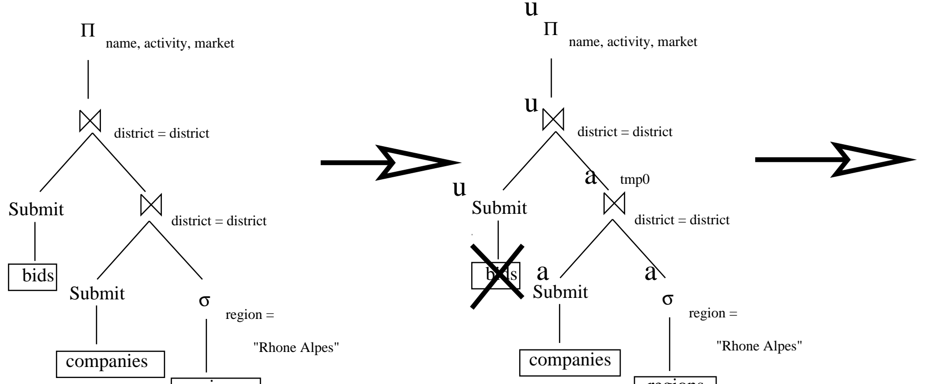

Figure 2: Step 1 - partial evaluation of the execution plan, Step 2 - construction of the partially evaluated query

3.2.1 Eval algorithm

The eval algorithm is encoded in the `open` call to each operator. The implementations of `get-next` and `close` are generally unchanged from the classical implementations.[1] Evaluation commences by calling `open` on the root operator of the tree. Each operator proceeds by calling `open` on its children, waiting for the result of the call, and then returning to its parent. We consider two cases that can result from calling *open* on all the children of an operation. Either all the calls succeed, or at least one call fails. In the former case, the operator marks itself as *available* and returns success to its parent. In the latter case, the operator marks itself as *unavailable* and returns failure to its parent. The traversal of the tree continues until all operators are marked either available or unavailable. Note that by insisting that each operator open all its children, instead of giving up with the first unavailable child, we implement a simple form of query scrambling. See Figure 3 for an outline of the algorithm.

After traversal of the tree for the open calls finishes, the root operator of the tree has marked itself either available or unavailable. If it 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 the latter case the tree is processed on a second pass. Each subtree rooted with an available operator *materializes* its result. Materialization is accomplished by the root operator of the subtree repeatedly executing its *get-next* call and storing the result. The resulting tree is passed to the construct algorithm.

3.2.2 Construct algorithm

We construct a declarative query from an annotated execution plan by first constructing a declarative expression for each operator in the tree in a bottom-up fashion. The declarative expressions are nested to form the partially evaluated query. This algorithm is based on two working assumptions. We assume, first, that each (physical)

[1] A complication arises from operators that call `open` during the execution of a `get-next` call. We do not consider these operators here.

```
eval(operator) {
  for each subtree in children of operator {
    eval(subtree)
  }
  if source is available or all subtrees are available then {
    produce result
    mark operator available
  } else {
    mark operator unavailable
  }
}
```

Figure 3: The evaluation algorithm.

```
construct(execution_plan) returns Partially EvaluatedQuery {
  if available() then {
    return the query containing the intermediate result
  } else {
    S := ∅
    for each subtree in children(execution_plan) {
      S := S ∪ construct( subtree )
    }
    return the query for execution_plan using S
  }
}
```

Figure 4: Construction of the partially evaluated query.

operator has a corresponding declarative expression, and second, that declarative expressions can be composed in the query language. The first assumption holds for a broad range of operations including all relations operations and operations for sorting, materialization, etc. The second assumption holds for database query languages, including SQL and OQL.

The Codd's theorem [10] guarantees that every relational algebra expression can be transformed into an equivalent relational calculus expression. The partially evaluated query that is constructed is thus equivalent to the tree of operators obtained from the eval algorithm, i.e., a partial evaluation of the original execution plan. This ensures that the answer to the partially evaluated query is similar to the answer to the original query, under the assumption that no updates are performed on the data sources.

Operators marked available generate a declarative expression that accesses the materialized intermediate result. It is an expression of the form `select` x `from` x `in` r, where x is new unique variable and r is the name of the temporary relation holding the materialized intermediate result.[2]

Operators marked unavailable generate a declarative expression corresponding to the operator. For example, a `project` operator generates an expression of the form `select` p `from` x in arg, where p is the list of attributes projected by the operator, x is a unique variable, and arg is the declarative expression that results from the child operator of the project operation. The association between the operators we consider and declarative expressions is straightforward.

The construction of the partially evaluated query, see Figure 4, consists in traversing recursively the tree of operators, stopping the traversal of a branch when an available operator is encountered (there is an intermediate result), or when an unavailable leaf is reached (a `submit` operator associated to an unavailable data source), and in nesting the declarative expression associated to each traversed node.

The partially evaluated query, together with the annotated execution plan is used to return a partial answer.

[2]An alternative implementation generates the expression `select` x `from` x `in bag(`t`)`, where t is the list of tuples in the materialized intermediate result. This implementation advantageously encodes the entire state of the partial answer in the query. The disadvantage is the size of the resulting partial answer query.

```
extract(execution_plan, parachute_query) returns Answer {
  S := materialized_subqueries(execution_plan)
  for each subquery in S {
    if parachute_query ⊆ query then
      return parachute_query evaluated on intermediate result of subquery
  }
  return null
}
```

Figure 5: The extraction algorithm.

3.3 Extraction Algorithm

We present an algorithm for extracting information from a transparent partial answer generated by the `eval` algorithm, using a parachute query. The algorithm traverses the annotated execution plan searching for an intermediate result that *matches* the parachute query.

The algorithm proceeds as follows, see Figure 5. First, a query is generated for each intermediate result materialized in the annotated execution plan. We obtain a set of queries whose result is materialized in the annotated execution plan. Then, we compare the parachute query to each of these queries. If the parachute query is contained by one of these queries, then we can obtain the answer to the parachute query: it is the result of the evaluation of the parachute query on the materialized result of the associated subquery in the annotated execution plan. (This problem is exactly the same as matching a query against a set of materialized views.) Otherwise, we cannot return any answer to the parachute query. Query containment is defined in [20].

3.4 Trade-offs in the Implementation

There are a few trade-offs or design decisions that concern the implementation of the evaluation, construction, and extraction algorithms. We use a heuristic for the traversal of the execution plan in the evaluation algorithm. This heuristic is *evaluate as much as we can during the evaluation of a query.* We assume that the user prefers producing as much of the final result as possible with the sources that are available. An alternative would be to stop the evaluation as soon as a source is found unavailable. The former solution improves the availability of the final answer, the latter solution informs the user more quickly that only a partial answer is possible.

The algorithms described in this section are based on a pipelined execution model. In particular we assume that once a data source responds to a request, its pipeline functions properly. Similar algorithms for non-pipelined execution can be readily constructed to avoid this assumption.

The traversal of the execution plan can be performed either sequentially or in parallel. If the execution plan is traversed sequentially, each unavailable data source must time-out, thus lengthening the total execution time. (A data source is found unavailable if it has not returned its complete response within a given time.) Parallel evaluation reduces this cost considerably since all data sources are contacted in parallel.

There are different possibilities concerning the materialization of the results produced during the evaluation. A first possibility is to systematically materialize the results obtained from the data sources and record them in the partial answer. This approach has two advantages. First, if the partial answer is resubmitted to the system, the sources that have already been available are not concerned by the query anymore. This increases the likelihood of obtaining the final result. Second, in the case of transparent partial answers, there are intermediate results that can be extracted. This heuristic increases the cost of processing and storage. An alternative heuristic, in the case of a partial evaluation, chose to leave unmaterialized the intermediate results of some *available* data sources, thus forcing the data source to be contacted again when the partial answer is resubmitted. This heuristic trade-offs between the amount of information contained in partial answers and work required to evaluate a re-submitted query.

When constructing the partially evaluated query, declarative expressions containing the intermediate results are generated. One solution includes the contents of the intermediate result in the query. Since the size of a partially evaluated query can be very large, depending on the sources that are available and on the intermediate results that are produced, the partially evaluated query may be very large; it may occupy several megabytes. This partially evaluated query may be submitted again to the database system. However, database optimizers have not been designed for handling queries of this size. An alternative consists in including a reference to each

temporary relation that stores an intermediate result in the partially evaluated query. Such a solution has the advantage of producing partially evaluated queries of a reasonable size. However, the query interface has to be modify to accept queries containing these references and the temporary relations must be garbage collected at some point in time.

4 Implementation

We have implemented the evaluation and construct algorithms in the framework of the Disco mediator prototype [18]. We have made some choices concerning the trade-offs discussed in Section 3.4. First, the execution model for our run-time system is non-pipelined, and the execution plan is traversed sequentially. Second, we materialize systematically the intermediate that are produced into temporary files. Third, we use exceptions for the upward exchange of information during the recursive traversal of the tree. Finally, we use temporary files to store intermediate results of a partially evaluated query.

We have implemented ad hoc extraction primitives that allow access to the set of data sources that were involved in a query, the set of data sources that were available, and the set of sources that were unavailable. It is also possible to retrieve the intermediate results that have been materialized, together with the list of data sources that participated in this result. We assume that the application program can manipulate the intermediate result, i.e. it knows the type and the meaning of some intermediate result. In Section 3 we discuss the implicates of this assumption.

Transparent partial answers include the annotated execution plan produced by the `eval` function and the partially evaluated query produced by the `construct` function. The implementation of the eval algorithm takes into account the requirements introduced by the extraction primitives we support. Each node in the annotated execution plan maintains a list of the sources that participate in the result it produces, and whether these sources are available or not during the evaluation.

The Disco mediator prototype accepts a query language that includes composition of declarative statements (nested select statements). We thus use this language to represent the original queries, the partially evaluate queries and the parachute queries. Our implementation is currently restricted to use a purely relational model, and ask conjunctive queries involving select, project, and join.

The status of the implementation allows us to validate the algorithms we have presented. We have set up an example with the company, bids and region data sources as described in Section 5. The prototype has some limitations: it is not integrated with the query compiler and optimizer, it accepts only restricted forms of join and select predicates, and it does not implement the extract algorithm. We are currently working on the extract algorithm and on the integration with the Disco query compiler. This work will allow us to experiment with parachute queries.

5 Application Scenario

We use the DISCO infrastructure in the framework of the project *Bourse d'Affaires Electroniques.* The goal of this project is to provide an electronic commerce platform in the domain of public construction. The DISCO infrastructure provides a uniform view of a set of heterogeneous data sources, spread in France over the Internet, to different actors in the life of a construction project.

We have in particular built wrappers for a data source that contains administrative information about companies, and for a data source that contain bids for work concerning public markets. These wrappers support the *select* and *scan* operators.

The wrappers export the schemas described in appendix A. A bid for work concerns a `market` in a `district`; it has been issued on a `parution` date and it is valid until a `validity` date. A company has a `name`, an `address`, a `phone` and a `fax` number; it is located in a given `district`, and it works in a given `activity` domain. Both wrappers support scan and select operations.

Wrappers are located on the same host as the mediator, and it is possible to turn them on and off. We can thus easily experiment with unavailable data sources, in the context of the architecture we described in section 3.1. Our prototype performs the evaluation and sends sub-queries to the wrappers, that are available or not, simulating the data sources that are available or not.

Our prototypes evaluate an execution plan that corresponds to a query written against the global schema. The global schema we consider contains two remote classes and a local class. The remote classes correspond to the classes exported by the wrappers. The local class `region` associates regions and districts. We can note that we use a purely relational model, as the mediator only supports relational operations for the time being.

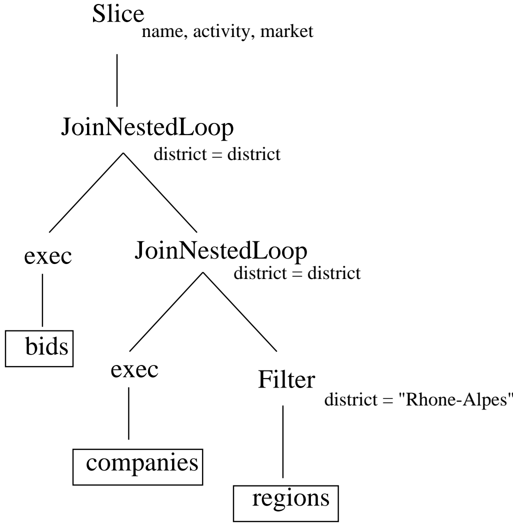

Figure 6: Example execution plan

We consider in this section an original query asked of the system, and we study various cases corresponding to the availability of the different data sources. We also discuss the information that can be extracted from the different transparent partial answers that are produced.

The original request is : *select the companies and the bids for work located in the Rhne-Alpes region and return the name and activity of the companies with the markets of bids on work.*. This can be expressed with the following query:

```
from x in companies, y in bids, z in regions
select x.name, x.activity, y.market
where x.district = y.district and z.region = "Rhone Alpes"
  and x.district = z.district;
```

The mediator compiler generates an execution plan for this original query, see fig 6. A query execution plan is a tree of physical operators. Each physical operator corresponds to the implementation of a logical operator, i.e. an operator defined in the realtional algebra [10]. *scanFile* corresponds to *scan*; this operator performs a scan on a collection of data. *slice* corresponds to *project*; this operator projects out a list of attributes from the result of its input operator. *filter* corresponds to *select*; this operator uses a given predicate to filter the result of its input operator. *nestedJoin* corresponds to *join*; this operator implements the nested loop algorithm for joining its two input operators (that are designated as the right and left input operators). *exec* corresponds to *submit*; this operator sends a subquery (a tree of logical operators) to a data source.

The execution plan in figure 6 is the input of our prototype. The operator *slice* is the root of the tree. It has two inputs, a *scanFile* on the bids and a *JoinNestedLoop*. The *JoinNestedLoop* operator has in turn two inputs, a *scanFile* on the companies and a *filter* operator, which has one input, a *scanFile* on the regions. A projection list is defined for the *slice* operators; predicates are defined for the *nestedJoin* and *filter* operators.

We experiment by evaluating this execution plan in presence of unavailable data sources. There are two data sources involved, there are thus four possible cases: companies data source is unavailable, bids data source is unavailable, both bids and companies data sources are unavailable, all data sources are available.

In the following sections, we discuss the results of the `eval` and `construct` algorithms in the four cases we have identified. We also describe the output of the extraction primitives.

When our prototype is integrated with the DISCO compiler, we can show that we can obtain the final result if we resubmit the partially evaluated query to the system. We can also discuss the execution plans that are generated by the compiler for the partially evaluated queries.

5.1 Companies Data Source is Unavailable

The companies data source is unavailable during the evaluation of the execution plan. The output of `eval` is an annotated execution plan (see fig 7). None of the join operations can be performed. Intermediate results

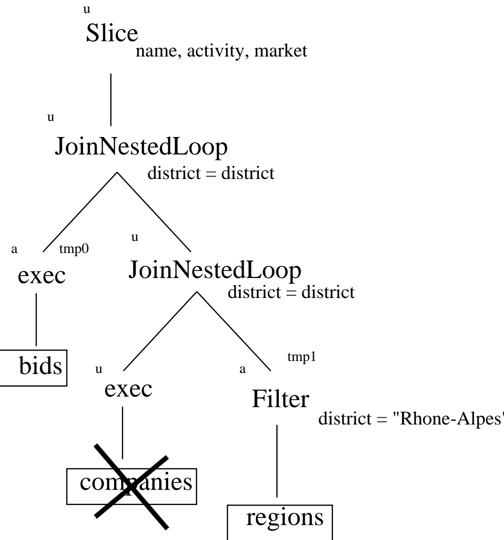

Figure 7: Case 1: companies data source is unavailable

are materialized that correspond to the extraction from the bids data source, and to the result of the filter operation applied to the region relation.

The `construct` function takes the annotated execution plan and constructs the following partially evaluated query.

```
from x0 in (from x1 in (from x3 in ({["23/11/96","23/12/96","35","
Amnagement avec extension de la cuisine et des locaux annexes de la
salle polyvalente. "],
```

[52 more tuples not shown]

```
["23/11/96","13/12/96","75"," Objet du march : restructuration du
btiment Colonie aux chlets du Prariand. Lieu d'excution :  Megve
(74). "]}) select x3), x2 in (from x4 in (from x6 in (from x7 in
(companies) select x7) select x6), x5 in (from x8 in ({["Rhone
Alpes","69"],["Rhone Alpes","38"],["Rhone Alpes","74"],["Rhone
Alpes","73"],["Rhone Alpes","42"],["Rhone Alpes","1"],["Rhone
Alpes","7"],["Rhone Alpes","26"]}) select x8) select
x4.name,x4.activity,x4.district,x4.address,x4.tel,x4.fax,x5.region,x5.district
where x4.district = x5.district) select
x1.parution,x1.valid,x1.district, x1.market,x2.name,x2.activity,
x2.district,x2.address,x2.tel,x2.fax,x2.region,x2.district
where x1.district = x2.district) select x0.name, x0.activity,
x0.market;
```

We can note that the intermediate results are materialized in this query. We use a relatively small data set for our experiments, so it is possible to print this partially evaluated query. With a realistic data set, say with 100 times more bids, the print out of this partially evaluated query alone would take many pages!

A first extraction function returns the sources that were available

```
available sources: ( (rmi://dyade.inrialpes.fr/bids) (local) )
```

A second extraction function returns the sources that were unavailable

```
unavailable sources: ( (rmi://dyade.inrialpes.fr/companies) )
```

A third extraction function returns the intermediate results, together with the sources that participated in these results. Here, the intermediate results are obtained from a single source, and the operations that are performed preserve the type which can be derived from the global schema. In this case, an application program may know the type of the intermediate results.

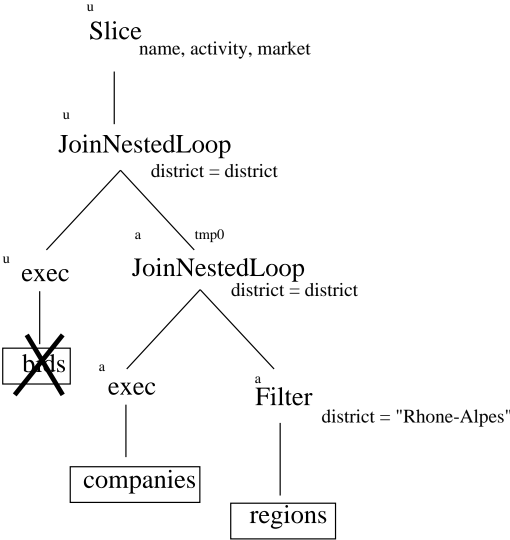

Figure 8: Case 2: bids data source is unavailable

```
source: ( (rmi://dyade.inrialpes.fr/bids) )
intermediate result: rto(set( bids ),(
(tuple("23/11/96","23/12/96","35"," Amnagement
avec extension de la cuisine et des locaux annexes de la salle
polyvalente. ")) (tuple("23/11/96","13/12/96","75"," Objet du march :
restructuration du btiment Colonie aux chlets du Prariand. Lieu
d'excution :  Megve (74). ")),
```

[51 more tuples not shown])

```
source: ( (local) )
intermediate result: rto(set( region ),(tuple("Rhone Alpes","69"))
(tuple("Rhone Alpes","38")) (tuple("Rhone Alpes","74")) (tuple("Rhone
Alpes","73")) (tuple("Rhone Alpes","42")) (tuple("Rhone Alpes","1"))
(tuple("Rhone Alpes","7")) (tuple("Rhone Alpes","26") ))
```

5.2 Bids Data Source is Unavailable

The bids data source is unavailable during the evaluation of the execution plan. The output of `eval` is an annotated execution plan (see fig 8). A join operation can be performed between the relation extracted from the companies data source and the result of the filter applied to the region relation. As a result , there is only one intermediate result produced, which corresponds to the result of this join operation.

The `construct` function takes the annotated execution plan and constructs the following partially evaluated query.

```
from x0 in (from x1 in (from x3 in (from x4 in (bids) select x4)
select x3), x2 in (from x5 in ({["ATELIER SOIXANTE QUATORZE","Bureaux
d'tudes - btiments et travaux publics","74","1 r Jean Jaurs BP362
74012  Annecy Cedex ","0450516945","0450528088","Rhone
Alpes","74"],
```

[6 more tuples not shown]

```
["AFREM","Bureaux d'tudes - btiments et travaux
publics","69","12 quai Commerce  69336 Lyon", "0478434343",
"0478648777", "Rhone Alpes","69"]})
```

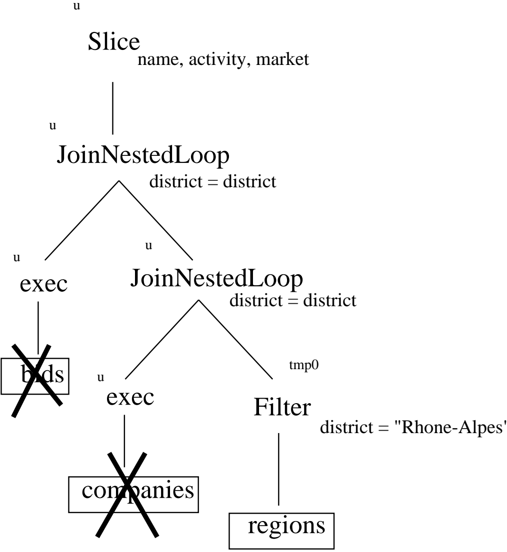

Figure 9: both bids and companies data sources are unavailable

```
select x5) select
x1.parution, x1.valid, x1.district, x1.market, x2.name, x2.activity,
x2.district, x2.address, x2.tel,x2.fax,x2.region,x2.district
where x1.district = x2.district) select x0.name, x0.activity,
x0.market;
```

The extraction function returns the intermediate result, together with the name of the two sources that participated in this result. Here, the intermediate result does not directly correspond to a class in the global schema. Therefore, there is no possibility that the application program can know the type or the meaning of this result without specifying a parachute query. We also see that in this case there is only one intermediate result, compared to the two that were available in the previous case. The intermediate results corresponding to the extraction from the companies data source, and to the result of the filter operation have been destroyed when the result of the join operation has been produced. We can increase the number of intermediate result by not destroying them when they have been used to produce another result. This is at the cost of an increased space occupation.

```
source: ( (rmi://dyade.inrialpes.fr/companies) (local) ) )
valeur: rto(set( foobar ),( (tuple("ATELIER
SOIXANTE QUATORZE","Bureaux d'tudes - btiments et travaux
publics","74","1 r Jean Jaurs BP362  74012  Annecy Cedex
","0450516945","0450528088","Rhone Alpes","74"))
(tuple("AFREM","Bureaux d'tudes - btiments et travaux
publics","69","12 quai Commerce  69336  Lyon
","0478434343","0478648777","Rhone Alpes","69"))
```

[6 more tuples not shown]))

5.3 Both Bids and Companies Data Sources are Unavailable

Both the bids and the companies data sources are unavailable during the evaluation of the execution plan. The output of `eval` is an annotated execution plan (see fig 9). Only the local operation can be performed, there are no data extracted from the data sources.

The `construct` function takes the annotated execution plan and constructs the following partially evaluated query.

```
from x0 in (from x1 in (from x3 in (from x4 in (bids) select x4)
select x3), x2 in (from x5 in (from x7 in (from x8 in (companies)
```

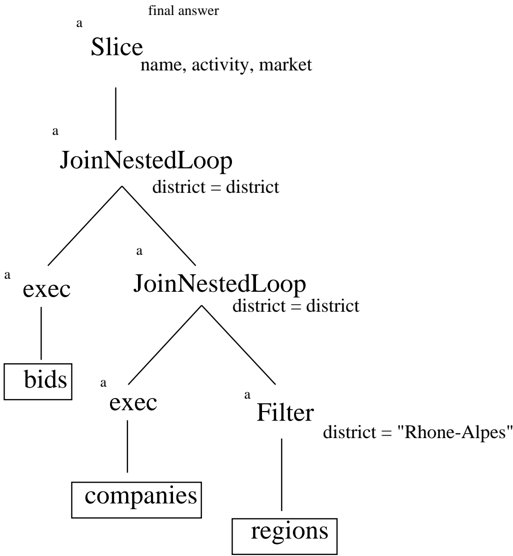

Figure 10: all sources are available

```
select x8) select x7), x6 in (from x9 in ({["Rhone Alpes","69"],["Rhone
Alpes","38"],["Rhone Alpes","74"],["Rhone Alpes","73"],["Rhone
Alpes","42"],["Rhone Alpes","1"],["Rhone Alpes","7"],["Rhone
Alpes","26"]}) select x9) select x5.name,x5.activity,x5.district,
x5.address,x5.tel,x5.fax,x6.region,x6.district where
x5.district = x6.district) select x1.parution,x1.valid,x1.district,
x1.market,x2.name,x2.activity,x2.district,x2.address,x2.tel,x2.fax,
x2.region,x2.district where x1.district = x2.district) select x0.name,
x0.activity, x0.market;
```

5.4 All sources are Available

Both the bids and the companies data sources are available during the evaluation of the execution plan (see fig 10). The output of `eval` is the final result. The overhead in the computation of the final result is limited to a variable affectation each time an operator produces its result. We use the exception mechanism to encode the behavior of the system in case sources are unavailable, this also introduces an overhead in the case where all sources are available. We however claim that the use of the `eval` algorithm in the run-time system does not affect performances. [3]

```
({["BALTHAZARD ET COTTE","Matires premires pour le
btiment et les travaux publics"," Objet du march : construction d'un
centre de loisirs et du spectacle avec cantine scolaire. Lieu
d'excution : Z.A.C. de la Bovagne. "], ["ATELIER SOIXANTE QUATORZE","Bureaux
d'tudes - btiments et travaux publics"," Objet du march :
construction d'un groupe scolaire. Lieu d'excution : avenue
du Stade, 74970 Marignier. "],["AFREM","Bureaux d'tudes - btiments
et travaux publics"," Objet du march : cration de 8 salles de cours
dans le hall du btiment 307. Lieu d'excution : btiment 307
(I.N.S. A.). "],["MORTAMET VIDAL MANHES (STE)","Bureaux d'tudes -
btiments et travaux publics"," Objet du march : cration de 8 salles
de cours dans le hall du btiment 307. Lieu d'excution : btiment 307
(I.N.S. A.). "],["MORILLON-CORVOL","Matires premires pour le
btiment et les travaux publics"," Objet du march : construction d'un
btiment  usage de salle polyvalente, mairie et restaurant
```

[3] The overhead obviously depends on the number of nodes in the execution plan.

```
scolaire. Lieu d'excution :  Bny. "],["SEDIME
DIFFUSION","Import-export - btiment et travaux publics"," Objet du
march : construction d'un btiment  usage de salle polyvalente,
mairie et restaurant scolaire. Lieu d'excution :
Bny. "],["BALTHAZARD ET COTTE","Matires premires pour le btiment
et les travaux publics"," Objet du march : extension et
rhabilitation du collge Les Mntriers. Lieu d'excution : au
collge Les Mntriers, 21, rue de Landau, 68150 Ribeauvill. "]})
```


6 Related Work

The problem of unavailable sources has, so far, been considered as an implementation issue. To our knowledge, various straightforward solutions, where an unavailable source either returns the empty set or generates an error, are not documented.

APPROXIMATE, [21], tackles the issue of unavailable data. They propose an approach based on approximate query processing. In presence of unavailable data, the system returns an approximate answer which is defined in terms of the subsets and supersets sandwiching the exact answer. The system uses semantic information concerning the contents of the database for the initial approximation. In our context, we do not use any semantic information concerning the data sources. The hypothesis underlying APPROXIMATE and our system are thus different.

References [6] and [14] survey cooperative answering systems. These systems emphasize the interaction between the application program and the database system. They aim at assisting users in the formulation of queries, or at providing meaningful answers in presence of empty results. Reference [14] introduces a notion of partial answer. When the result of a query is empty, the system anticipates follow-up queries, and returns the result of broader queries, that subsume the original query. These answers are offered in partial fulfillment of the original query. This notion of partial answer is different from the one we have introduced. For [14], a partial answer is an answer to a query subsuming the original query. For us, a partial answer is the partial evaluation of the original query.

The approach we have developed in this paper was originally proposed in [19]. In that paper, partially evaluated queries, i.e. opaque partial answers, are introduced. They are proposed as a step in query evaluation with unavailable data sources. In this paper, we have introduced transparent partial answers and parachute queries.

In the area of research on programming languages, partial evaluation [5] is a technique used to improve execution times of programs. This technique permits the partial evaluation of a program with respect to some partially specified input to the program, e.g., constants that appear in the source code. Our approach was initially inspired by this area of research.

7 Conclusion

We have proposed a novel approach to the problem of processing queries that cannot be completed for some reason. We have focused on the problem of processing queries in distributed heterogeneous databases with unavailable data sources. Our approach offers two aspects. First, in presence of unavailable data sources the query processing system returns a partial answer by partially evaluating the query. The partial answer may be resubmitted to the system to produce the final answer. Second, we define algorithms to extract information from a partial answer using additional queries. The additional queries are known as parachute queries.

We described a framework of various problems related to partial answers and parachute queries. Our description forms a taxonomy for work on the problem. In addition, the use of parachute queries provides a very flexible and familiar interface for application programs.

We have implemented our approach for one branch of the taxonomy in the case of unreachable data sources: a source is considered unavailable if it cannot be contacted. In this context, we have proposed algorithms for the evaluation of transparent partial answers, and for the extraction of information using parachute queries. We have used an application scenario to illustrate our approach.

The solution we have implemented is valid under certain working assumptions. First, once a source can be contacted it is assumed that it will produce its result completely. A source never returns half of an answer. Second, we have assumed that no updates relevant to a query are performed on the data sources between the

start of query processing and when the final result is returned. Third, we have implemented our algorithms in a mediator based architecture. Relaxing these assumptions is future work.

We have investigated a particular class of partial answers and parachute queries. In particular, we have concentrated on the case where query optimization is completely unchanged by partial evaluation. The relationships between the optimizer and the partial answers or the parachute queries are also a subject for future work.

Our approach based on partial evaluation generalizes to many applications. A query is evaluated until an interruption condition occurs, and then our algorithms are applied. We have concentrated on unavailable data sources because it is an important problem for heterogeneous database systems. We believe partial evaluation is important, in other domains, such as long lived query processing. In this context, the processing of a query last for example several day. Using partial evaluation, query processing could be easily interrupted, examined to determine its progress, and then resumed.

Acknowledgments

The authors wish to thank Louiqa Raschid and Michael Franklin for fruitful discussions, and Remy Amouroux, Olga Kapitskaia, Mauricio Lopez, Dan Smith and Pierre Yves Chevalier for comments on previous drafts of this paper.

References

[1] Sibel Adali, Kasim Selcuk Candan, Yannis Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *ACM SIGMOD Int. Conf. on Management of Data*, pages 137–148, Montreal, Canada, 1996.

[2] Laurent Amsaleg, Michael J. Franklin, Anthony Tomasic, and Tolga Urhan. Scrambling query plans to cope with unexpected delays. In *International Conference on Parallel and Distribution Information Systems (PDIS)*, Miami Beach, Florida, 1996.

[3] 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. To appear.

[4] Michael J. Carey et al. Towards heterogeneous multimedia information systems: The garlic approach. In *RIDE-DOM '95, Fifth Int. Workshop on Research Issues in Data Engineering - Distributed Object Management*, pages 124–131, Taipei, Taiwan, 1995.

[5] C. Consel and O. Danvy. Tutorial notes on partial evaluation. In *Conference record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages: papers presented at the symposium, Charleston, South Carolina, January 10–13, 1992*, pages 493–501, New York, NY, USA, 1993. ACM Press.

[6] Terry Gaasterland, Parke Godfrey, and Jack Minker. An overview of cooperative answering. *Journal of Intelligent Information Systems*, 1(2):123–157, 1992.

[7] H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and J. Widom. The TSIMMIS project - integration of heterogeneous information sources. In *Proc. of the 100th Anniversary Meeting of Information Processing Society of Japan*, pages 7–18, Tokyo, 1994.

[8] Gardarin et al. IRO-DB: a collection of selected papers. Technical Report RR-95/34, Laboratoire PRISM, Universite Versailles Saint-Quentin en Yvelines, CNRS, 1995.

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

[10] P. Kanellakis. Elements of relational database theory. In *Handbook of Theoretical Computer Science*, chapter 17. Elsevier Science Publishers B.V., j. van leeuwen edition, 1990.

[11] Donald Kossmann, Laura M. Hass, Edward L. Wimmers, and Jun Yang. I can do that! using wrapper input for query optimization in heterogeneous middleware systems. submitted to publication, 1996.

[12] Alon Y. Levy, Anand Rajaraman, and Joann J. Ordille. Querying heterogeneous information sources using source descriptions. In *VLDB'96, Proc. of 22th Int. Conf. on Very Large Data Bases*, pages 251–262, Mumbai (Bombay), India, 1996.

[13] Ling Liu and Calton Pu. Distributed interoperable object model and its application to large-scale interoperable database systems. In *Fourth International Conference on Information and Knowledge Management (CIKM'95)*, Baltimore, Maryland, 1995.

[14] Amihai Motro. Cooperative database systems. In *Proceedings of the 1994 Workshop on Flexible Query-Answering Systems (FQAS '94)*, pages 1–16. Department of Computer Science, Roskilde University, Denmark, 1994. Datalogiske Skrifter - Writings on Computer Science - Report Number 58.

[15] M. Rusinkiewicz et al. Semantic integration of information in open and dynamic environments. Technical report, Microelectronics and Computer Technology Corporation (MCC), 1996.

[16] Timos Sellis and Subrata Ghosh. On the multiple-query optimization problem. *Transactions on Knowledge and Data Engineering*, 2(2):262–266, June 1990.

[17] Ming-Chien Shan, Rafi Ahmen, Jim Davis, Weimin Du, and Kent Kent, William. *Modern Database Systems: The Object Model, Interoperability, and Beyond*, chapter Pegasus: A Heterogeneous Information Management System, pages 664–682. ACM Press, 1995.

[18] Anthony Tomasic, Rmy Amouroux, Philippe Bonnet, Olga Kapitskaia, Hubert Naacke, and Louiqa Raschid. The distributed information search component (DISCO) and the world-wide web. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona, 1997. To appear.

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

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

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

[22] Gio 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 Schemas

In this appendix we give the schemas for our scenerio in ODL-like syntax. The additional information in the interface tells DISCO the address of the name service that returns a remote object handle for access to the corresponding data source.

```
interface remote "//amber.inria.fr/bids" Bids (extent bids) {
  attribute string market;
  attribute int district;
  attribute string parution;
  attribute string validity;
}

interface remote "//amber.inria.fr/companies" Companies (extent companies) {
  attribute string name;
  attribute string activity;
  attribute int district;
  attribute string address;
  attribute string tel;
  attribute string fax;
};
```

```
interface local Region (extent regions) {
  attribute string region;
  attribute int district;
};
```

B Partial Evaluation Algorithm

We present, in this section, detailed algorithms for the `eval` and `construct` functions. We also introduce the `partialEval` function that combines both functions and return either the final result or a partial answer. We first define the abstract types that are used in the detailed algorithms.

abstractType Answer
    func isComplete() **returns** boolean

abstractType CompleteAnswer **subtype from** Answer
    func CompleteAnswer(Set of Values)
    func getValues() **returns** Set of Values

abstractType PartialAnswer **subtype from** Answer
    func PartialAnswer(String, PhysicalOperator)
    func extractInformation(String) **returns** Set of Values
    func extractInformation() **returns** Set of pairs associating a Set of values and a List of Sources
    func listOfParticipatingSources() **returns** List of Sources
    func listOfAvailableSources() **returns** List of Sources
    func listOfUnavailableSources() **returns** List of Sources

abstractType PhysicalOperator
    func kind() **returns** the kind of this operator
    func produceResult()
    func getResult() **returns** Set of Values
    func formatResult() **returns** String
    func markAvailable() **returns** Set of Values
    func markUnavailable() **returns** Set of Values
    func isAvailable() **returns** Set of Values

abstractType Filter **subtype from** PhysicalOperator
    func Filter(PhysicalOperator)
    func kind() **returns** SELECT
    func operand() **returns** PhysicalOperator
    func predicate() **returns** Predicate

abstractType Slice **subtype from** PhysicalOperator
    func Slice(PhysicalOperator)
    func kind() **returns** PROJECT
    func operand() **returns** PhysicalOperator
    func projectionList() **returns** ProjList

abstractType JoinNestedLoop **subtype from** PhysicalOperator
    func JoinNestedLoop(PhysicalOperator, PhysicalOperator)
    func kind() **returns** JOIN
    func rightOperand() **returns** PhysicalOperator
    func leftOperand() **returns** PhysicalOperator
    func predicate() **returns** Predicate
    func projectionList() **returns** Predicate

```
abstractType ScanRel subtype from PhysicalOperator
    func ScanRel(Relation)
    func kind() returns SCAN

abstractType Exec subtype from PhysicalOperator
    func Exec(PhysicalOperator)
    func kind() returns EXEC


abstractType EvalException
    func EvalException(PhysicalOperator)
    func getAnnotatedExecutionPlan() returns the root of the annotated execution plan

abstractType TimerException

func partialEval(root_of_execution_plan) returns Answer
try {
     values := eval(root_of_execution_plan);
     return new CompleteAnswer(values));
}
catch (EvalException ee) {
     root_of_annotated_execution_plan := ee.getAnnotatedExecutionPlan();
     residualQuery := constructQuery(root_of_annotated_execution_plan);
     return new PartialAnswer(residualQuery, annotated_execution_plan);
}
endfunc

func eval(physical_operator) returns Set of Values
                             throws EvalException
switch physical_operator.kind()
   case EXEC:
     try {
      physical_operator.produceResult();
      physical_operator.markAvailable();
      return physical_operator.getResult();
     } catch (TimerException te) {
      physical_operator.markUnavailable();
      raise EvalException(physical_operator);
     }
   case SCAN:
     physical_operator.produceResult();
     physical_operator.markAvailable();
     return physical_operator.getResult();
   case SELECT:
     try {
      eval(physical_operator.operand());
      physical_operator.produceResult();
      physical_operator.markAvailable();
      return physical_operator.getResult();
     } catch (EvalException ee) {
      physical_operator.markUnavailable();
      raise EvalException(physical_operator);
     }
   case PROJECT:
     try {
      eval(physical_operator.operand());
      physical_operator.produceResult();
```

```
      physical_operator.markAvailable();
      return physical_operator.getResult();
    } catch (EvalException ee) {
      physical_operator.markUnavailable();
      raise EvalException(physical_operator);
    }
  case JOIN:
    boolean leftAvailable;
    try {
      eval(physical_operator.leftOperand());
      leftAvailable := true;
    } catch (EvalException ee) {
      leftAvailable := false;
    }
    boolean rightAvailable;
    try {
      eval(physical_operator.rightOperand());
      rightAvailable := true;
    } catch (EvalException ee) {
      rightAvailable := false;
    }
    if (leftAvailable and rightAvailable) {
      physical_operator.produceResult();
      physical_operator.markAvailable();
      return physical_operator.getResult();
    } else {
      physical_operator.markUnavailable();
      raise EvalException(physical_operator);
    }
  }
}


func construct(physical_operator) returns Partially Evaluated Query
  if (physical_operator.isAvailable()) {
    return = "select " x
             "from " + physical_operator.formatValue();
      } else {
       switch (physical_operator.kind()) {
        case PROJECT
         return = "select " physical_operator.projectionList()
               "from " x + " in " construct(physical_operator.operand());
              case SELECT
               return = "select " + x +
               "from " + x + " in " + construct(physical_operator.operand()) +
                    "where " + physical_operator.predicate();
              case JOIN
               return = "select " + physical_operator.projectionList()+
               "from " + left + " in " + construct(physical_operator.leftOperand()) +", " +
            right + " in " + construct(physical_operator.rightOperand()) +
                    "where " + physical_operator.predicate();
              case SCAN
               return = "select " + x +
               "from " + x + " in " + physical_operator.relationName()
              case EXEC
              return = "select " + x +
                    "from " + x + " in " + construct(operator.subQuery())
```

```
    }
}
```

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 655, avenue de l'Europe, 38330 MONTBONNOT ST MARTIN
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité 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